

1
2 SDMX Technical Working Group

3 VTL Task Force

4

5

6

7

8

9 **VTL - version 2.0**

10 **(Validation & Transformation Language)**

11 **Part 2 - Reference Manual**

12
13
14
15
16
17
18
19
20
21
22
23
24 *July 2018*

25
26

27

Foreword

- 28 The Task force for the Validation and Transformation Language (VTL), created in 2012-2013 under the initiative
 29 of the SDMX Secretariat, is pleased to present the draft version of VTL 2.0.
- 30 The SDMX Secretariat launched the VTL work at the end of 2012, moving on from the consideration that SDMX
 31 already had a package for transformations and expressions in its information model, while a specific
 32 implementation language was missing. To make this framework operational, a standard language for defining
 33 validation and transformation rules (operators, their syntax and semantics) had to be adopted, while
 34 appropriate SDMX formats for storing and exchanging rules, and web services to retrieve them, had to be
 35 designed. The present VTL 2.0 package is only concerned with the first element, i.e., a formal definition of each
 36 operator, together with a general description of VTL, its core assumptions and the information model it is based
 37 on.
- 38 The VTL task force was set up early in 2013, composed of members of SDMX, DDI and GSIM communities and the
 39 work started in summer 2013. The intention was to provide a language usable by statisticians to express logical
 40 validation rules and transformations on data, described as either dimensional tables or unit-record data. The
 41 assumption is that this logical formalization of validation and transformation rules could be converted into
 42 specific programming languages for execution (SAS, R, Java, SQL, etc.), and would provide at the same time, a
 43 "neutral" business-level expression of the processing taking place, against which various implementations can be
 44 mapped. Experience with existing examples suggests that this goal would be attainable.
- 45 An important point that emerged is that several standards are interested in such a kind of language. However,
 46 each standard operates on its model artefacts and produces artefacts within the same model (property of
 47 closure). To cope with this, VTL has been built upon a very basic information model (VTL IM), taking the
 48 common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM 1.1, somewhat simplified and with
 49 some additional detail. In this way, existing standards (GSIM, SDMX, DDI, others) would be allowed to adopt VTL
 50 by mapping their information model against the VTL IM. Therefore, although a work-product of SDMX, the VTL
 51 language in itself is independent of SDMX and will be usable with other standards as well. Thanks to the
 52 possibility of being mapped with the basic part of the IM of other standards, the VTL IM also makes it possible to
 53 collect and manage the basic definitions of data represented in different standards.
- 54 For the reason described above, the VTL specifications are designed at logical level, independently of any other
 55 standard, including SDMX. The VTL specifications, therefore, are self-standing and can be implemented either on
 56 their own or by other standards (including SDMX). In particular, the work for the SDMX implementation of VTL
 57 is going in parallel with the work for designing this VTL version, and will entail a future update of the SDMX
 58 documentation.
- 59 The first public consultation on VTL (version 1.0) was held in 2014. Many comments were incorporated in the
 60 VTL 1.0 version, published in March 2015. Other suggestions for improving the language, received afterwards,
 61 fed the discussion for building the draft version 1.1, which contained many new features, was completed in the
 62 second half of 2016 and provided for public consultation until the beginning of 2017.
- 63 The high number and wide impact of comments and suggestions induced a high workload on the VTL TF, which
 64 agreed to proceed in two steps for the publication of the final documentation, taking also into consideration that
 65 some first VTL implementation initiatives had already been launched. The first step, the current one, is
 66 dedicated to fixing some high-priority features and making them as much stable as possible. A second step,
 67 scheduled for the next period, is aimed at acknowledging and fixing other features considered of minor impact
 68 and priority, which will be added hopefully without affecting neither the features already published in this
 69 documentation, nor the possible relevant implementations. Moreover, taking into account the number of very
 70 important new features that have been introduced in this version in respect to the VTL 1.0, it was agreed that the
 71 current VTL version should be considered as a major one and thus named VTL 2.0.
- 72 The VTL 2.0 package contains the general VTL specifications, independently of the possible implementations of
 73 other standards; in its final release, it will include:
- 74 a) Part 1 – the user manual, highlighting the main characteristics of VTL, its core assumptions and the
 information model the language is based on;
 - 76 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.
 - 78 c) eBNF notation (extended Backus-Naur Form) which is the technical notation to be used as a test bed for
 all the examples.
- 80 The present document is the part 2.

81 The latest version of VTL is freely available online at 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 Gelserma, 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).

97

98 Table of contents

99	Foreword	2
100	Table of contents	4
101	Introduction	8
102	Overview of the language and conventions	9
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	VTL-DL - Rulesets.....	29
113	define datapoint ruleset	29
114	define hierarchical ruleset	31
115	VTL-DL - User Defined Operators	39
116	define operator	39
117	Data type syntax	40
118	VTL-ML - Typical behaviours of the ML Operators.....	42
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	VTL-ML - General purpose operators.....	53

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

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

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

Introduction

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

257 | Overview of the language and conventions

258 Introduction

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

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

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

275 Examples of Transformations are:

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

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

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

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

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

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

```
302     (i)     DS_p <- if DS_np >= 0 then DS_np else DS_1 ;
303     (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 Tranformation 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 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 Typographical conventions

343

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

346

<i>Convention</i>	<i>Description</i>
<i>Italics Cambria</i>	<i>Basic scalar data types (in the text)</i> e.g. "...must have one Identifier of type time_period . If the Data Set...."
Bold Arial	<i>Keywords (in the description of the syntax and in the text)</i> e.g. Rule ::= { ruleName : } { when antecedentCondition then } consequentCondition { errorcode errorCode } { errorlevel errorLevel } e.g. ".....The rename operator allows to rename one or more Components..."
<i>Italics Arial</i>	<i>Optional Parameter (in the description of the syntax)</i> e.g. substr (op, start, length)
<u>Underlined Arial</u>	<i>Sub-expressions</i>
Normal font Arial	<ul style="list-style-type: none"><i>The operator's syntax (excluded the keywords, the optional Parameters and the sub-expressions)</i> e.g. length ("Hello, World!")<i>The examples of invocation of the operators</i> e.g. length ("Hello, World!")<i>Optional and Mandatory Parameters (in the text)</i> e.g. ".....If comp is a Measure in op, then in the result"

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>	359 <i>Compact name</i>	360 <i>Abbreviation</i>
360 <i>Data Set</i>	361 <i>Data Set</i>	362 <i>DS</i>
361 <i>Data Point</i>	362 <i>Data Point</i>	363 <i>DP</i>
362 <i>Identifier Component</i>	363 <i>Identifier</i>	364 <i>Id</i>
363 <i>Measure Component</i>	364 <i>Measure</i>	365 <i>Me</i>
364 <i>Attribute Component</i>	365 <i>Attribute</i>	366 <i>At</i>
365 <i>Data Set Component</i>	366 <i>Component</i>	367 <i>Comp</i>
366 <i>Value Domain Subset</i>	367 <i>Subset or Set</i>	368 <i>Set</i>
367 <i>Value Domain</i>	369 <i>Domain</i>	370 <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 style, 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)*
 - {optional} : *optional sub-expression (zero or 1 occurrence)*
 - {non-optional}¹ : *non-optional sub-expression (just 1 occurrence)*
 - {one-or-more}+ : *sub-expression repeatable from 1 to many occurrences*
 - {zero-or-more}* : *sub-expression repeatable from 0 to many occurrences*
 - { part1 | part2 | part3 } : *optional alternative sub-expressions (zero or 1 occurrence)*
 - { part1 | part2 | part3 }¹ : *alternative sub-expressions (just 1 occurrence)*
 - { part1 | part2 | part3 }+ : *alternative sub-expressions, from 1 to many occurrences*
 - { part1 | part2 | part3 }* : *alternative sub-expressions, from 0 to many occurrences*

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-expr₁-text sub-expr₂-name ... sub-expr_{N-1}-name sub-expr_N-text
422 sub-expr₂-name ::= sub-expr₂-text
423 ... possible others ...
424 sub-expr_{N-1}-name ::= sub-expr_{N-1}-text

425 In this representation of a meta-expression:

- 426
 - *The first line is the text of the meta-expression*
 - *sub-expr₁-text, sub-expr_N-text are sub-expressions directly written in the meta-expression*
 - *sub-expr₂-name, ... sub-expr_{N-1}-name are identifiers of sub-expressions.*
 - *sub-expr₂-text, ... sub-expr_{N-1}-text are subexpression written separately from the meta-expression.*
 - *The symbol ::= means "is defined as" and denotes the assignment of a sub-expression-text to a sub-expression-name.*

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 }¹ (op)

435 The meta-expression above synthesizes that:

- 436
 - **trim**, **ltrim**, **rtrim** are the operators' symbols (reserved keywords);
 - **()** are symbols of the operators syntax (reserved keywords);
 - **op** is the only operand of the three operators;
 - "**{ }**" and "**|**" are symbols of the meta-syntax; in particular "**|**" indicates that the three operators are alternative (a single invocation can contain only one of them) and "**{ }**" indicates that a single invocation 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.

448 Description of the data types of operands and result

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

Type conversion	cast	cast { op ,scalarType { , mask } }	converts to the specified data type	Func.	$\begin{aligned} \text{op} &:: \text{dataset}\{ \text{measure<scalar>} _ \} \\ &\quad \text{component}<\text{scalar}> \\ &\quad \text{scalar} \\ \text{scalarType} &:: \text{scalar type} \\ \text{mask} &:: \text{string} \end{aligned}$	dataset{ measure<scalar> _ } component<scalar> scalar	Changing data type
Join	inner_join , left_join , full_join , cross_join ,	<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 }¹ calcClause ::= { calcRole } calcComp := calcExpr {, { calcRole } calcComp := calcExpr }* calcRole ::= { identifier measure attribute viral attribute }¹ aggrClause ::= { aggrRole } aggrComp := aggrExpr {, { aggrRole } aggrComp := aggrExpr }* aggrRole ::= { measure attribute viral attribute }¹ groupingClause ::= { group by idList group except idList group all conversionExpr }¹ { having havingCondition } </pre>	Inner join, left outer join, full outer join, cross join,	Func.	$\begin{aligned} \text{ds} &:: \text{dataset} \\ \text{alias} &:: \text{name} \\ \text{usingId} &:: \text{name < component >} \\ \text{filterCondition} &:: \text{component<boolean>} \\ \text{applyExpr} &:: \text{dataset} \\ \text{calcComp} &:: \text{name<component>} \\ \text{calcExpr} &:: \text{component<scalar>} \\ \text{aggrComp} &:: \text{name<component>} \\ \text{aggrExpr} &:: \text{component<scalar>} \\ \text{groupingId} &:: \text{name < identifier >} \\ \text{conversionExpr} &:: \text{component<scalar>} \\ \text{havingCondition} &:: \text{component<boolean>} \\ \text{comp} &:: \text{name < component >} \\ \text{compFrom} &:: \text{component<scalar>} \\ \text{compTo} &:: \text{component<scalar>} \end{aligned}$	dataset	Specific
String concatenation	 	op1 op2	Concatenates two strings	Infix	$\begin{aligned} \text{op1, op2} &:: \text{dataset}\{ \text{measure<string>} _+ \} \\ &\quad \text{component}<\text{string}> \\ &\quad \text{string} \end{aligned}$	dataset { measure<string> _+ } component<string> string	On two scalars, DSS or DSCs

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

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

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

Natural logarithm	ln	ln (op)	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	power	power (base, exponent)	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	log	log (op, num)	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	sqrt	sqrt (op)	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	=	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	<>	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	>	left { > >= }¹ 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
	>=						
Less than	<	left { < <= }¹ 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
	<=						

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

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

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

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

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

462

463

464 VTL-ML - Evaluation order of the Operators

465 Within a single expression of the manipulation language, the operators are applied in sequence, according to the
 466 precedence order. Operators with the same precedence level are applied according to the default associativity
 467 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	Left-to-right
	/	Division	
VI	+	Addition	Left-to-right
	-	Subtraction	
		String concatenation	
VII	> >=	Greater than	Left-to-right
	< <=	Less than	
	=	Equal-to	
	<>	Not-equal-to	
	in	In a value list	
	not_in	Not in a value list	
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

469

470 Description of VTL Operators

471

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

- 473 • **Operator name**, which is also used to invoke the operator
- 474 • **Semantics**: a brief description of the purpose of the operator
- 475 • **Syntax**: the syntax of the Operator (this part follows the conventions described in the previous section
476 "Conventions for describing the operators' syntax")
- 477 • **Syntax description**: detailed explanation of the meaning of the various parts of the syntax
- 478 • **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

517 *Syntax*

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 }
```

531 *Syntax description*

532 rulesetName	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 Value Domains or the Represented Variables (see the information model) on which the Ruleset is defined. If valuedomain is specified then the Ruleset is applicable to the Data Sets having Components that take values on the specified Value Domains. If variable is specified then the Ruleset is applicable to Data Sets having the specified Variables as Components.
534 valueDomain	a Value Domain on which the Ruleset is defined.
535 vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Value Domain (parameter valueDomain) is used in the body of the rules.
536 variable	a Represented Variable on which the Ruleset is defined.
537 varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Variable (parameter valueDomain) is used in the body of the Rules.
538 <u>dpRule</u>	a Data Point Rule, as defined in the following parameters.
539 ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes.
540 antecedentCondition	a <i>boolean</i> expression to be evaluated for each single Data Point of the input Data Set. It can contain Values of the Value Domains or Variables specified in the Ruleset signature and constants; all the VTL-ML component level operators are allowed. If omitted then antecedentCondition is assumed to be TRUE.
541 consequentCondition	a <i>boolean</i> expression to be evaluated for each single Data Point of the input Data Set when the antecedentCondition 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
566	a literal denoting the error code associated to the rule, to be assigned to the possible non-
567	valid results in case the Rule is used for validation. If omitted then no error code is
568	assigned (NULL value). VTL assumes that a Value Domain errorcode_vd of error codes
569	exists in the Information Model and contains all possible error codes: the errorCode
570	literal must be one of the possible Values of such a Value Domain. VTL assumes also that a
571	Variable errorcode for describing the error codes exists in the IM and is a dependent
572	variable of the Data Sets which contain the results of the validation.
573	errorLevel
574	a literal denoting the error level (severity) associated to the rule, to be assigned to the
575	possible non-valid results in case the Rule is used for validation. If omitted then no error
576	level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of error
577	levels exists in the Information Model and contains all possible error levels: the
578	errorLevel literal must be one of the possible Values of such a Value Domain. VTL
	assumes also that a Variable errorlevel for describing the error levels exists in the IM and
	is a dependent variable of the Data Sets which contain the results of the validation.

Parameters

```
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
```

Constraints

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

Semantic specification

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

A Data Point Ruleset is a persistent object that contains Rules to be applied to the Data Points of a Data Set¹. The Data Point Rulesets can be invoked by the **check_datapoint** operator. The Rules are aimed at checking the combinations of values of the Data Set Components, assessing if these values fulfil the logical conditions expressed by the Rules themselves. The Rules are evaluated independently for each Data Point, returning a Boolean scalar value (i.e., TRUE for valid results and FALSE for non-valid results).

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

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

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

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

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

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

655 relationships between the relevant Code Items. These Rulesets are also called “vertical” taking into account the
656 tabular representation of a Data Set (considered as a mathematical function), in which each (vertical) column
657 represents a variable and each (horizontal) row represents a Data Point: these Rulesets are applied on variables
658 (columns), i.e., vertically on the tabular representation of a Data Set.

659 A main purpose of the hierarchical Rules is to express some more aggregated Code Items (e.g. the continents) in
660 terms of less aggregated ones (e.g., their countries) by using Code Item Relationships. This kind of relations can
661 be applied to aggregate data, for example to calculate an additive measure (e.g., the population) for the
662 aggregated Code Items (e.g., the continents) as the sum of the corresponding measures of the less aggregated
663 ones (e.g., their countries). These rules can be used also for validation, for example to check if the additive
664 measures relevant to the aggregated Code Items (e.g., the continents) match the sum of the corresponding
665 measures of their component Code Items (e.g., their countries), provided that the input Data Set contains all of
666 them, i.e. the more and the less aggregated Code Items.

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

672 The name “hierarchical” comes from the fact that this kind of Ruleset is able to express the hierarchical
673 relationships between Code Items at different levels of detail, in which each (aggregated) Code Item is expressed
674 as a partition of (disaggregated) ones. These relationships can be recursive, i.e., the aggregated Code Items can
675 be in their turn component of even more aggregated ones, without limitations about the number of recursions.

676 As a first simple example, the following Hierarchical Ruleset named “BeneluxCountriesHierarchy” contains a
677 single rule that asserts that, in the Value Domain “Geo_Area”, the Code Item BENELUX is the aggregation of the
678 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
```

683 *Syntax*

```
684
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 : } codelitemRelation { errorcode errorCode } { errorlevel errorLevel }
696
697     codelitemRelation ::=
698         { when leftCondition then }
699             leftCodelItem  { = | > | < | >= | <= }1
700             { + | - } rightCodelItem { [ rightCondition ] }
701             { { + | - }1 rightCodelItem { [ rightCondition ] } }*
```

702 *Syntax description*

704 rulesetName	the name of the Hierarchical Ruleset to be defined.
705 hrRulesetSignature	the signature of the Ruleset. It specifies the Value Domain or Variable on which the 706 Ruleset is defined, and the Conditioning Signature.
707 vdRulesetSignature	the signature of a Ruleset defined on Value Domains
708 varRulesetSignature	the signature of a Ruleset defined on Variables
709 hrRule	a single hierarchical rule, as described below.
710 vdConditioningSignature	specifies the Value Domains on which the conditions are defined. The Ruleset is meant 711 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 conditioning Value Domains (i.e., condValueDomain).
713		the Value Domain on which the Ruleset is defined
714	ruleValueDomain	a conditioning Value Domain of the Ruleset
715	condValueDomain	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Value Domain (i.e., condValueDomain) must be used.
716	vdAlias	
717		
718		
719		
720	<u>varConditioningSignature</u>	the signature of the (possible) conditions of the Ruleset defined on Variables. It specifies the Represented Variables (see the information model) on which these conditions are defined. The Ruleset is meant to be applicable to any Data Set having Components which are defined by the Variable on which the Ruleset is expressed (i.e., variable) and on the Conditioning Variables.
721		
722		
723		
724		
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 used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Variable (parameter condVariable) must be used.
728		
729		
730		
731	ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes. In addition, if the results of more than one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.
732		
733		
734		
735		
736		
737		
738		
739		
740	<u>codelitemRelation</u>	specifies a (possibly conditioned) Code Item Relation. It expresses a logical relation between Code Items belonging to the Value Domain of the hrRulesetSignature, possibly conditioned by the Values of the Value Domains or Variables of the Conditioning Signature. The relation is expressed by one of the symbols =, >, >=, <, <=, that in this context denote special logical relationships typical of Code Items. The first member of the relation is a single Code Item. The second member of the relationship is the composition of one or more Code Items combined using the symbols + or -, which in turn also denote special logical operators typical of Code Items. The meaning of these symbols is better explained below and in the User Manual.
741		
742		
743		
744		
745		
746		
747		
748		
749	errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error code is assigned (NULL value). VTL assumes that a Value Domain errorCode_vd of the error codes exists in the Information Model and contains all the possible error codes: the errorCode literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorCode for describing the error codes exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
750		
751		
752		
753		
754		
755		
756		
757	errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of the error levels exists in the Information Model and contains all the possible error levels: the errorLevel literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorlevel for describing the error levels exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
758		
759		
760		
761		
762		
763		
764		
765	leftCondition	a <i>boolean</i> expression which defines the pre-condition for evaluating the left member Code Item (i.e., it is evaluated only when the leftCondition is TRUE); It can contain references to the Value domains or the Variables of the conditioningSignature of the Ruleset and Constants; all the VTL-ML component level operators are allowed. The leftCondition is optional, if missing it is assumed to be TRUE and the Rule is always evaluated.
766		
767		
768		
769		
770		
771	leftCodelitem	a Code Item of the Value Domain specified in the hrRulesetSignature.

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

781 *Input parameters type*

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*

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

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" (>=)². 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

829 ² "Coincides" means "implies and is implied"

830 composition can be defined by means of Code Item Operators, whose goal is to compose some Code Items in
831 order to obtain another Code Item.

832 Please note that the symbols + and - do not denote the usual operations of sum and subtraction, but logical
833 operations between Code Items which are seen as events of the probability theory. In other words, two or more
834 Code Items cannot be summed or subtracted to obtain another Code Item, because they are events and not
835 numbers, however they can be manipulated through logical operations like "OR" and "Complement".

836 Note also that the + also acts as a declaration that all the Code Items denoted by + in the formula are mutually
837 exclusive one another (i.e., the corresponding events cannot happen at the same time), as well as the - acts as a
838 declaration that all the Code Items denoted by - in the formula are mutually exclusive one another and
839 furthermore that each one of them is a part of (implies) the result of the composition of all the Code Items having
840 the + sign.

841 At intuitive level, the symbol + means "*with*" (Benelux = Belgium *with* Luxembourg *with* Netherland) while the
842 symbol - means "*without*" (EU*without*UK = EuropeanUnion *without* UnitedKingdom).

843 When these relationships are applied to additive numeric measures (e.g., the population relevant to geographical
844 areas), they allow to obtain the measure values of the compound Code Items (i.e., the population of Benelux and
845 EU*without*UK) by summing or subtracting the measure values relevant to the component Code Items (i.e., the
846 population of Belgium, Luxembourg and Netherland). This is why these logical operations are denoted in VTL
847 through the same symbols as the usual sum and subtraction. Please note also that this property is valid
848 whichever is the Data Set and whichever is the additive measure (provided that the possible other Identifier
849 Components of the Data Set Structure have the same values), therefore the Rulesets of this kind are potentially
850 largely reusable.

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

857 The Hierarchical Ruleset may act on one or more Measures of the input Data Set provided that these measures
858 are additive (for example it cannot be applied on a measure containing a "mean" because it is not additive).

859 Within the Hierarchical Rulesets there can be dependencies between Rules, because the inputs of some Rules can
860 be the output of other Rules, so the former can be evaluated only after the latter. For example, the data relevant
861 to the Continents can be calculated only after the calculation of the data relevant to the Countries. As a
862 consequence, the order of calculation of the Rules is determined by their mutual dependencies and can be
863 different from the order in which the Rules are written in the Ruleset. The dependencies between the Rules form
864 a directed acyclic graph.

865 **The Hierarchical ruleset can be used for calculations** to calculate the upper levels of the hierarchy if the data
866 relevant to the leaves (or some other intermediate level) are available in the operand Data Set of the **hierarchy**
867 operator (for more information see also the "Hierarchy" operator). For example, having additive Measures
868 broken by region, it would be possible to calculate these Measures broken by countries, continents and the
869 world. Besides, having additive Measures broken by country, it would be possible to calculate the same Measures
870 broken by continents and the world.

871 When a Hierarchical Ruleset is used for calculation, only the Relations expressing coincidence (=) are evaluated
872 (provided that the leftCondition is TRUE, and taking into account only right-side Code Items whose
873 rightCondition is TRUE). The result Data Set will contain the compound Code Items (the left members of those
874 relations) calculated from the component Code Items (the right member of those Relations), which are taken
875 from the input Data Set (for more details about the evaluation options see the **hierarchy** operator). Moreover,
876 the clauses typical of the validation are ignored (e.g., ErrorCode, ErrorLevel).

877 The Hierarchical Ruleset can be also used to filter the input Data Points. In fact if some Code Items are defined
878 equal to themselves, the relevant Data Points are brought in the result unchanged. For example, the following
879 Ruleset will maintain in the result the Data Points of the input Data Set relevant to Belgium, Luxembourg and
880 Netherland and will add new Data Points containing the calculated value for Benelux:

```
882     define hierarchical ruleset BeneluxRuleset ( valuedomain rule GeoArea) is
883         Belgium = Belgium
884         ; Luxembourg = Luxembourg
885         ; Netherlands = Netherlands
886         ; Benelux = Belgium + Luxembourg + Netherlands
887     end hierarchical ruleset
```

889 **The Hierarchical Rulesets can be used for validation** in case various levels of detail are contained in the Data
890 Set to be validated (see also the **check_hierarchy** operator for more details). The Hierarchical Rulesets express

891 the coherency Rules between the different levels of detail. Because in the validation the various Rules can be
892 evaluated independently, their order is not significant.

893 If a Hierarchical Ruleset is used for validation, all the possible Relations (`=`, `>`, `>=`, `<`, `<=`) are evaluated (provided
894 that the leftCondition is TRUE and taking into account only right-side Code Items whose rightCondition is TRUE).
895 The Rules are evaluated independently. Both the Code Items of the left and right members of the Relations are
896 expected to belong to and taken from the input Data Set (for more details about the evaluation options see the
897 **check_hierarchy** operator). The Antecedent Condition is evaluated and, if TRUE, the operations specified in the
898 right member of the Relation are performed and the result is compared to the first member, according to the
899 specified type of Relation. The possible relations in which Code Items are defined as equal to themselves are
900 ignored. Further details are described in the **check_hierarchy** operator.

901 If the data to be validated are in different Data Sets, either they can be joined in advance using the proper VTL
902 operators or the validation can be done by comparing those Data Sets directly, without using a Hierarchical
903 Ruleset (see also the **check** operator).

904
905 **Through the right and left Conditions, the Hierarchical Rulesets allow to declare the time validity of**
906 **Rules and Relations.** In fact leftCondition and RightCondition can be defined in term of the time Value Domain,
907 expressing respectively when the left member Code Item has to be evaluated (i.e., when it is considered valid)
908 and when a right member Code Item participates in the relation.

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

911 Example 1 (for simplicity the time literals are written without the needed “cast” operation)

```
912 define hierarchical ruleset EuropeanUnionAreaCountries1
913   ( valuedomain condition ReferenceTime as Time rule GeoArea ) is
914     when between (Time, "1.1.1958", "31.12.1972")
915       then EU = BE + FR + DE + IT + LU + NL
916     ; when between (Time, "1.1.1973", "31.12.1980")
917       then EU = ... same as above ... + DK + IE + GB
918     ; when between (Time, "1.1.1981", "02.10.1985")
919       then EU = ... same as above ... + GR
920     ; when between (Time, "1.1.1986", "31.12.1994")
921       then EU = ... same as above ... + ES + PT
922     ; when between (Time, "1.1.1995", "30.04.2004")
923       then EU = ... same as above ... + AT + FI + SE
924     ; when between (Time, "1.5.2004", "31.12.2006")
925       then EU = ... same as above ... +CY+CZ+EE+HU+LT+LV+MT+PL+SI+SK
926     ; when between (Time, "1.1.2007", "30.06.2013")
927       then EU = ... same as above ... + BG + RO
928     ; when >= "1.7.2013"
929       then EU = ... same as above ... + HR
930 end hierarchical ruleset
```

931 Example 2 (for simplicity the time literals are written without the needed “cast” operation)

```
932 define hierarchical ruleset EuropeanUnionAreaCountries2
933   (valuedomain condition ReferenceTime as Time rule GeoArea ) is
934     EU =      AT [ Time >= "0101.1995" ]
935           + BE [ Time >= "01.01.1958" ]
936           + BG [ Time >= "01.01.2007" ]
937
938           +
939           + ...
940           + SE [ Time >= "01.01.1995" ]
941           + SI [ Time >= "01.05.2004" ]
942           + SK [ Time >= "01.05.2004" ]
943 end hierarchical ruleset
```

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

```
945 define hierarchical ruleset GeoHierarchy ( valuedomain rule Geo_Area) is
946   World = Africa + America + Asia + Europe + Oceania
947   ; 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

```

959 The Hierarchical Rulesets allow defining multiple relations for the same Code Item.

960 Multiple relations are often useful for validation. For example, the Balance of Payments item "Transport" can be
961 broken down both by type of carrier (Air transport, Sea transport, Land transport) and by type of objects
962 transported (Passengers and Freights) and both breakdowns must sum up to the whole "Transport" figure. In
963 the following example a RuleName is assigned to the different methods of breaking down the Transport.

```

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

```

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

```

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

```

980 In the calculation, basically, for each Rule, for all the input Data Points and provided that the conditions are
981 TRUE, the right Code Items are changed into the corresponding left Code Item, obtaining Data Points referred
982 only to the left Code Items. Then the outcomes of all the Rules of the Ruleset are aggregated together to obtain
983 the Data Points of the result Data Set.

984 As far as each left Code Item is calculated by means of a single Rule (i.e., a single calculation method), this
985 process cannot generate inconsistencies.

986 Instead if a left Code Item is calculated by means of more Rules (e.g., through more than one calculation method),
987 there is the risk of producing erroneous results (e.g., duplicated data), because the outcome of the multiple Rules
988 producing the same Code Item are aggregated together. Proper definition of the left or right conditions can avoid
989 this risk, ensuring that for each input Data Point just one Rule is applied.

990 If the Ruleset is aimed only at validation, there is no risk of producing erroneous results because in the validation
991 the rules are applied independently.

992 Examples

993 1) The Hierarchical Ruleset is defined on the Value Domain "sex": Total is defined as Male + Female.
994 No conditions are defined.

```

995 define hierarchical ruleset sex_hr (valuedomain rule sex) is
996     TOTAL = MALE + FEMALE
997 end hierarchical ruleset
998

```

999 2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS. No conditions
1000 are defined.

```

1001 define hierarchical ruleset BeneluxCountriesHierarchy (valuedomain rule GeoArea) is
1002     BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode "Bad value for Benelux"
1003

```

```
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
```

1030

VTL-DL - User Defined Operators

1031 define operator

1032 *Syntax*

```
1033     define operator operator_name ( { parameter { , parameter }* } )
1034     { returns outputType }
1035     is operatorBody
1036     end operator
```

1038 parameter::= parameterName parameterType { **default** parameterDefaultValue }

1039

1040 *Syntax description*

1041 operator_name	the name of the operator
1042 <u>parameter</u>	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 *Constraints*

- Each parameterName must be unique within the list of parameters
- parameterDefaultValue must be of the same data type as the corresponding parameter
- if outputType is specified then the type of operatorBody must be compatible with outputType
- If outputType is omitted then the type returned by the operatorBody expression is assumed
- 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
```

1075 *Example2:*

```
1076     define operator add (x integer default 0, y integer default 0)
1077         returns number is
1078             x+y
1079         end operator
```

1080 Data type syntax

1081 The VTL data types are described in the VTL User Manual. Types are used throughout this Reference Manual as
1082 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.

1090 `dataType ::= scalarType | scalarSetType | componentType | datasetType | operatorType | rulesetType`

1091 scalarType ::= { basicScalarType | valueDomainName | setName }¹ { scalarTypeConstraint } { { not } null }

1092 **basicScalarType** ::= **scalar | number | integer | string | boolean | time | date | time_period | duration**
1093

1094 **scalarTypeConstraint ::= [valueBooleanCondition] | { scalarLiteral { , scalarLiteral }* }**

1095 scalarSetType ::= **set** { < scalarType > }

1096 componentType ::= componentRole { < scalarType > }

1097 componentRole ::= component | identifier | measure | attribute | viral attribute

1098 **datasetType** ::= **dataset** { { **componentConstraint** { , **componentConstraint** }* } }

1099 **componentConstraint** ::= **componentType** { **componentName** | **multiplicityModifier** }¹

1100 multiplicityModifier ::= _ { + | * }

1101 operatorType ::= inputParameterType { * inputParameterType }* \rightarrow outputParameterType

1102 **inputParameterType** ::= scalarType | scalarSetType | componentType | datasetType | rulesetType

1103 outputParameterType ::= scalarType | componentType | datasetType

1104 rulesetType ::= ruleset | dpRuleset | hrRuleset

1105 dpRuleset ::= **datapoint**

1106 | datapoint_on_valuedomains { { valueDomainName { * valueDomainName }* } }

| datapoint_on_variables { { variableName { * variableName }* } }

1108 hrRuleset ::= **hierarchical**

1109 | hierarchical_on_valuedomains { { valueDomainName

1110 { (condValueDomainName { * condValueDomainName }*) } }

1111 | hierarchical_on_variables { { variableName

1112 { (condVariableName { * condVariableName }*) } } }

1113

1114 Note that the `valueBooleanCondition` in `scalarTypeConstraint` is expressed with reference to the fictitious
1115 variable "value" (see also the User Manual, section "Conventions for describing the Scalar Types"), which
1116 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”).

VTL-ML - Typical behaviours of the ML Operators

1126 In this section, the common behaviours of some class of VTL-ML operators are described, both for a better
1127 understanding of the characteristics of such classes and to factor out and not repeat the explanation for each
1128 operator of the class.

1129 Typical behaviour of most ML Operators

1130 Unless differently specified in the Operator description, the Operators can be applied to Scalar Values, to Data
1131 Sets and to Data Set Components.

1132 The operations on Scalar Values are primitive and are part of the core of the language. The other kind of
1133 operations can be typically be obtained by means of the scalar operations in conjunction with the Join operator,
1134 which is part of the core too.

1135 In the operations on Data Set, the Operators are meant to be applied by default only to the values of the
1136 Measures of the input Data Sets, leaving the Identifiers unchanged. The Attributes follow by default their specific
1137 propagation rules, which are described in the User Manual.

1138 In the operations on Components, the Operators are meant to be applied on the specified components of one
1139 input Data Set, in order to calculate a new component which becomes part of the resulting Data Set. In this case,
1140 the Attributes can be operated like the Measures.

1141 Operators applicable on one Scalar Value or Data Set or Data Set 1142 Component

1143

1144 *Operations on Scalar values*

1145 The operator is applied on a scalar value and returns a scalar value.

1146

1147 *Operations on Data Sets*

1148 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 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 This means that the operator is applied to the values of all the Measures of DS_1 in order to produce
1154 homonymous Measures in DS_r.

1155 The application of the operator is allowed only if all the Measures of the operand Data Set are of a data type
1156 compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the
1157 operand Data Sets are numeric). If the Measures of the operand Data Set are of different types, not all compatible
1158 with the operator to be applied, the membership or the keep clauses can be used to select only the proper
1159 Measures. No applicability constraints exist on Identifiers and Attributes, which can be any.

1160 As for the data content, for each Data Point (DP_1) of the operand Data Set, a result Data Point (DP_r) is returned,
1161 having for the Identifiers the same values as DP_1.

1162 For each Data Point DP_1 and for each Measure, the operator is applied on the Measure value of DP_1 and
1163 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 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1166 (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1167 of the operand Data Set (DS_r maintains the Attributes declared as "viral" in DS_1; these Attributes are
1168 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(...)`, this can be written as:

1174 `DS_r := DS_1 [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 [calc COMP_r := op COMP_1]`

1177 This means that the operator is applied on COMP_1 in order to calculate COMP_r.

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

1183 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(...)`, 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 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_1, DS_2) are joined to find the couples of Data Points (DP_1,
1225 DP_2), where DP_1 is from the first operand (DS_1) and DP_2 from the second operand (DS_2), 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_1, DP_2) a result Data Point (DP_r) is returned, having for the Identifiers the same values as
1229 DP_1 and DP_2.

1230 For each Measure and for each couple (DP_1, DP_2), the Measure values of DP_1 and DP_2 are composed through
1231 the operator so returning the Measure value of DP_r. An operand Scalar value is composed with all the Measures
1232 of the other operand.

1233 For each couple (DP_1, DP_2) and for each Attribute that propagates in DP_r, the Attribute value is calculated by
1234 applying the proper Attribute propagation algorithm on the values of the Attributes of DP_1 and DP_2.

1235 As for the data structure, the result Data Set (DS_r) 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_r maintains the Attributes
1238 declared as “viral” for the operand Data Sets; these Attributes are considered as “viral” also in DS_r, the “non-
1239 viral” Attributes of the operand Data Sets are not kept in DS_r).

1240 1241 *Operation on Data Set Components*

1242 The operator is applied either on two Data Set Components (COMP_1, COMP_2) belonging to the same Data Set
1243 (DS_1) or on a Component and a Scalar value, and returns another Component (COMP_r) which alters the
1244 structure of DS_1 in order to produce the result Data Set (DS_r). 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 [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 [calc COMP_r := COMP_1 op COMP_2]**

1250 This means that the operator is applied on COMP_1 and COMP_2 in order to calculate COMP_r.

- 1251 • If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components
1252 of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- 1253 • If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of
1254 the operator f (...) replace the DS_1 original values for such a Measure or Attribute in order to produce
1255 DS_r.
- 1256 • If COMP_r is one of the original Identifiers of DS_1, 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 Set³.
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_r cannot have the same
1262 name of an Identifier of DS_1.

1263 As for the data content, for each Data Point of DS_1, the values of COMP_1 and COMP_2 are composed through
1264 the operator so returning the value of COMP_r.

1265 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1266 (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1267 of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are
1268 considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). 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_r can be added
1271 to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a

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

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

1280

1281 *Operation on Data Set Components*

1282 The operator is applied either on a combination of more than two Data Set Components (COMP_1, COMP_2)
1283 belonging to the same Data Set (DS_1) or Scalar values, and returns another Component (COMP_r) which alters
1284 the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a
1285 Component is not allowed (it makes no sense).

1286 For example, using a functional style and denoting the operator with **f(...)**, this can be written as:

1287
$$DS_r := DS_1 [\text{substr} COMP_r := f(COMP_1, COMP_2, COMP_3)]$$

1288 This means that the operator is applied on COMP_1, COMP_2 and COMP_3 in order to calculate COMP_r.

- 1289 • If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components
1290 of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- 1291 • If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of
1292 the operator $f(...)$ replace the DS_1 original values for such a Measure or Attribute in order to produce
1293 DS_r.
- 1294 • If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can
1295 become inconsistent.

1296 In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

1297 The composition of more Data Set Components is allowed provided that they belong to the same Data Set⁴.
1298 Moreover, the input Components must belong to data types compatible with the operator (for example, a
1299 numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same
1300 name of an Identifier of DS_1.

1301 As for the data content, for each Data Point of DS_1, the values of COMP_1, COMP_2 and COMP_3 are composed
1302 through the operator so returning the value of COMP_r.

1303 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1304 (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1305 of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are
1306 considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is
1307 explicitly calculated, the attribute propagation rule is overridden.

1308 Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added
1309 to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a
1310 (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the
1311 result. For the alteration of role and virality see also the **calc** clause.

1312

1313 Behaviour of Boolean operators

1314 The Boolean operators are allowed only on operand Data Sets that have a single measure of type *boolean*. As for
1315 the other aspects, the behaviour is the same as the operators applicable on one or two Data Sets described above.

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

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 Attributes⁵.

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 31st 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:

	<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).

⁵ 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 Sets⁶.

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.

1396 *Operation on Data Set Components*

1397 For the same reasons above, the result Component cannot be the same as one of the operand Components and
1398 must be of the appropriate Scalar data type.

1400

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
<i>string</i>	<i>string_var</i>	<i>string_vd</i>

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

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

1407 Type Conversion and Formatting Mask

1408 The conversions between *scalar* types is provided by the operator **cast**, described in the section of the general
 1409 purpose operators. Some particular types of conversion require the specification of a formatting mask, which
 1410 specifies which format the source or the destination of the conversion should assume. The formatting masks for
 1411 the various scalar types are explained here.

1412 If needed, the formatting Masks can be personalized in the VTL implementations. If VTL rules are exchanged, the
 1413 personalised masks need to be shared with the partners of the exchange.

1414 The Numbers Formatting Mask

1415 The **number formatting mask** can be defined as a combination of characters whose meaning is the following:

- 1416 ○ “D” one numeric digit (if the scientific notation is adopted, D is only for the mantissa)
- 1417 ○ “E” one numeric digit (for the exponent of the scientific notation)
- 1418 ○ “*” an arbitrary number of digits
- 1419 ○ “+” at least one digit
- 1420 ○ “.” (dot) can be used as a separator between the integer and the decimal parts.
- 1421 ○ “,” (comma) can be used as a separator between the integer and the decimal parts.

1422

1423 Examples of valid masks are:

1424 DD.DDDDD, DD.D, D, D/DDDD, D*.D*, D+.D+, DD.DDDEEEE

1425 The Time Formatting Mask

1426 The format of the values of the types *time*, *date* and *time_period* can be specified through specific formatting
 1427 masks. A mask related to *time*, *date* and *time_period* is formed by a sequence of symbols which denote:

- 1428 - the time units that are used, for example years, months, days
- 1429 - the format in which they are represented, for example 4 digits for the year (2018), 2 digits for the month
 within the year (04 for April) and 2 digits for the day within the year and the month (05 for the 5th)
- 1430 - 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
- 1431 - 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).

1435 The time formatting masks follows the general rules below.

1436 For a numerical representations of the time units:

- 1437 - 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”
- 1438 - 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 {YYYY}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 ond 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 5th)
- 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 5th) 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 \PMM\MM 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:

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

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 *Code* *Duration*

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\DT\Thh\H\mm\MS\

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

1559 VTL Mask: \P\WW\W

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

1561 VTL Mask: \P\pp\P

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 "+"

1569 "-"

1570 designator of offset from UTC zone

1571 "/ 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

1598

VTL-ML - General purpose operators

1599 Parentheses : ()

1600

1601 *Syntax*

(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*

re <- op

1647

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 obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).
1653
1654

1655 *Examples of valid syntaxes*

1656 DS_r <- DS_1
1657 DS_r <- DS_1 - DS_2

1658
1659 *Semantics for scalar operations*

1660 empty

1661
1662 *Input parameters type*

1663 re :: name
1664 op :: dataset
1665

1666 *Result type*

1667 empty

1668
1669 *Additional constraints*

1670 The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set Component. When operations at Component level are invoked, the result is the Data Set which the output Components belongs to.
1671
1672
1673

1674 *Behaviour*

1675 The input operand op is assigned to the **persistent** result re, which assumes the same value as op. As mentioned,
1676 the operand op can be obtained through an expression as complex as needed (for example op can be the
1677 expression DS_1 - DS_2).

1678 The result re is a persistent Data Set that has the same data structure as the Operand. For example in DS_r <-
1679 DS_1 the data structure of DS_r is the same as the one of DS_1.

1680 If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value.
1681 For example, income <- 3 assigns the value 3 to the persistent Data Set named income.

1682

1683 *Examples*

1684

1685 Given the operand Data Set DS_1:
1686

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

1688 Example 1: DS_r <- DS_1 results in:

1689

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

1694 *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 obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2)).

1697

1698

1699

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

1705

1706 *Semantic for scalar operations*

1707 empty

1708

1709 *Input parameters type*

1710 re :: name

1711 op :: dataset | scalar

1712

1713 *Result type*

1714 empty

1715

1716 *Additional constraints*

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

1718

1719

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

1721

1722

1723

1724

1725

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 mentioned, the operand op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).

1728

1729

1730 The result re is a non-persistent Data Set that has the same data structure as the Operand. For example in DS_r := DS_1 the data structure of DS_r is the same as the one of DS_1.

1731

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.

1734

1735 *Examples*

1736

1737 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
1772 result and a new Measure is added. The Data Points' values for the new Measure are the same as the values of
1773 comp in ds. A default conventional name is assigned to the new Measure depending on its type: for example
1774 num_var if the Measure is *numeric*, string_var if it is *string* and so on (the default name can be renamed through
1775 the **rename** operator if needed).

1776 The Attributes follow the Attribute propagation rule as usual (viral Attributes of ds are maintained in the result
1777 as viral, non-viral ones are dropped). If comp is an Attribute, it follows the Attribute propagation rule too.

1778 The same symbol denoting the membership operator (#) is also used inside other operations at Component level
1779 (for example in **join**, **calc**, **aggr**) in order to identify the Components to be operated: please note that in these
1780 cases the symbol # does not denote the membership operator (i.e., this operator, which does not operate at
1781 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
1789
1790
1791

Example 1: DS_r := DS_1#Me_1 results in:
(assuming that At_1 is not viral in DS_1)

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

1792
1793
1794

(assuming that At_1 is viral in DS_1)

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

1795
1796
1797

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

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

1798
1799
1800

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

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

1801

1802 User-defined operator call

1803
1804
1805
1806

Syntax

operatorName ({ argument { , argument }* })

1807	<i>Input parameters</i>	
1808	operatorName	the name of an existing user-defined operator
1809	argument	argument passed to the operator
1810		
1811	<i>Examples of valid syntaxes</i>	
1812	max1 (2, 3)	
1813		
1814	<i>Semantic for scalar operations</i>	
1815	It depends on the specific user-defined operator that is invoked.	
1816		
1817	<i>Input parameters type</i>	
1818	operatorName ::	name
1819	argument ::	A data type compatible with the type of the parameter of the user-defined operator that is invoked (see also the “Type syntax” section).
1820		
1821		
1822		
1823	<i>Result type</i>	
1824	result ::	The data type of the result of the user-defined operator that is invoked (see also the “Type syntax” section).
1825		
1826		
1827	<i>Additional constraints</i>	
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 user defined operator (the correspondence is in the positional order).	
1830		
1831		
1832	<i>Behaviour</i>	
1833	The invoked user-defined operator is evaluated. The arguments passed to the operator in the invocation are associated to the corresponding parameters in positional order, the first argument as the value of the first parameter, the second argument as the value of the second parameter, and so on. An underscore (“_”) can be used to denote that the value for an optional operand is omitted. One or more optional operands in the last positions can be simply omitted.	
1834		
1835		
1836		
1837		
1838		
1839	<i>Examples</i>	
1840	<i>Example 1:</i>	
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	
1855	<i>Syntax</i>
1856	eval (externalRoutineName ({ argument } { , argument }*) language languageName returns outputType)
1857	
1858	<i>Input parameters</i>
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 type syntax)
1863	

```

1864
1865 Examples of valid syntaxes
1866 eval ( routine1 ( "eabcdefg" ) language "PL/SQL" returns string )
1867
1868 Semantics for scalar operations:
1869 This is not a scalar operation.
1870
1871 Input parameters type
1872 externalRoutineName :: name
1873 argument :: any data type
1874 language :: string
1875 outputType :: any data type restricting Data Set or scalar
1876
1877 Result Type
1878 result :: dataset
1879
1880 Additional constraints
1881 • The eval is the only VTL Operator that does not allow nesting and therefore a Transformation can contain
1882 just one invocation of eval and no other invocations. In other words, eval cannot be nested as the operand
1883 of another operation as well as another operator cannot be nested as an operand of eval
1884 • The result of an expression containing eval must be persistent
1885 • externalRoutineName is the conventional name of a non-VTL routine
1886 • the invoked external routine must be consistent with the VTL principles, first of all its behaviour must be
1887 functional, so having in input and providing in output first-order functions
1888 • argument is an argument passed to the external routine, it can be a name or a value of a VTL artefacts or
1889 some other parameter required by the routine
1890 • the arguments passed to the routine correspond to the parameters of the invoked external routine in
1891 positional order; as usual the optional parameters are substituted by the underscore if missing. The
1892 conversion of the VTL input/output data types from and to the external routine processor is left to the
1893 implementation.
1894
1895 Behaviour
1896 The eval operator invokes an external, non-VTL routine, and returns its result as a Data Set or a scalar. The
1897 specific data type can be given in the invocation. The routine specified in the eval operator can perform any
1898 internal logic.
1899
1900 Examples
1901 Assuming that SQL3 is an SQL statement which produces DS_r starting from DS_1:
1902
1903 DS_r := eval( SQL3( DS_1 ) language "PL/SQL"
1904     returns dataset { identifier<geo_area> ref_area,
1905         identifier<date> time,
1906         measure<number> obs_value,
1907         attribute<string> obs_status } )
1908
1909 Assuming that f is an externally defined Java method:
1910
1911 DS_r := DS_1 [calc Me := eval ( f (Me) language "Java" returns integer )]
1912

```

1913 Type conversion : **cast**

```

1914 Syntax
1915 cast ( op , scalarType { , mask} )
1916
1917 Input parameters
1918 op          the operand to be cast
1919 scalarType   the name of the scalar type into which op has to be converted
1920 mask        a character literal that specifies the format of op

```


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

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 a
2040 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 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 }  
2129     calcClause ::= { calcRole } calcComp := calcExpr  
2130             { , { calcRole } calcComp := calcExpr }*  
2131     calcRole ::= { identifier | measure | attribute | viral attribute }  
2132     aggrClause ::= { aggrRole } aggrComp := aggrExpr  
2133             { , { aggrRole } aggrComp := aggrExpr }*  
2134     aggrRole ::= { measure | attribute | viral attribute }  
2135     groupingClause ::= { group by groupId { , groupId }*  
2136             | group except groupId { , groupId }*  
2137             | group all conversionExpr }  
2138             { having havingCondition }
```

2141 *Input parameters*

2142 joinOperator	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 using clause is allowed for the left_join only under certain constraints described below and is not allowed at all for the full_join and cross_join)
2149 filterCondition	a condition (<i>boolean expression</i>) 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 <i>ds1</i> and <i>ds2</i> have the <i>numeric</i> measures <i>m1</i> and <i>m2</i> , the clause <i>apply ds1 + ds2</i> would result in calculating <i>m1 := ds1#m1 + ds2#m1</i> and <i>m2 := ds1#m2 + ds2#m2</i>
2157 calcClause	clause that specifies the Components to be calculated, their roles and their calculation algorithms, to be applied on the joined and filtered Data Points.
2159 calcRole	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
 2162 operands, used to calculate a Component
 2163 aggrClause clause that specifies the required aggregations, i.e., the aggregated Components to be
 2164 calculated, their roles and their calculation algorithm, to be applied on the joined and
 2165 filtered Data Points
 2166 aggrRole the role of the aggregated Component to be calculated; if omitted, the Measure role is
 2167 assumed
 2168 aggrComp the name of the aggregated Component to be calculated; this is a dependent Component
 2169 of the result (Measure or Attribute, not Identifier)
 2170 aggrExpr expression at component level, having only Components of the input Data Sets as
 2171 operands, which invokes an aggregate operator (e.g. **avg**, **count**, **max** ... , see also the
 2172 corresponding sections) to perform the desired aggregation. Note that the **count**
 2173 operator is used in an aggrClause without parameters, e.g.:
 2174

$$\text{DS_1} [\text{aggr Me_1 := count () group by Id_1 }]$$
 2175 groupingClause the following alternative grouping options:
 2176 **group by** the Data Points are grouped by the values of the specified Identifiers
 2177 (groupingId). The Identifiers not specified are dropped in the result.
 2178 **group except** the Data Points are grouped by the values of the Identifiers not
 2179 specified as groupingId. The specified Identifiers are dropped in the
 2180 result.
 2181 **group all** converts the values of an Identifier Component using conversionExpr
 2182 and keeps all the resulting Identifiers.
 2183 groupingId Identifier Component to be kept (in the **group by** clause) or dropped (in the **group
 2184 except** clause).
 2185 conversionExpr specifies a conversion operator (e.g. **time_agg**) to convert an Identifier from finer to
 2186 coarser granularity. The conversion operator is applied on an Identifier of the operand
 2187 Data Set.
 2188 havingCondition a condition (*boolean expression*) at component level, having only Components of the
 2189 input Data Sets as operands (and possibly constants), to be fulfilled by the groups of
 2190 Data Points: only groups for which havingCondition evaluates to TRUE appear in the
 2191 result. The havingCondition refers to the groups specified through the groupingClause,
 2192 therefore it must invoke aggregate operators (e.g. avg, count, max, ..., see also the
 2193 section Aggregate invocation). A correct example of havingCondition is
 2194 max(obs_value) < 1000, while the condition obs_value < 1000 is not a right
 2195 havingCondition, because it refers to the values of single Data Points and not to the
 2196 groups. The count operator is used in a havingCondition without parameters, e.g.:
 2197

$$\text{sum (ds group by id1 having count () >= 10)}$$
 2198 comp dependent Component (Measure or Attribute, not Identifier) to be kept (in the **keep**
 2199 clause) or dropped (in the **drop** clause)
 2200 compFrom the original name of the Component to be renamed
 2201 compTo the new name of the Component after the renaming
 2202
 2203 *Examples of valid syntaxes*
 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

```

Semantics for scalar operations

The join operator does not perform scalar operations.

Input parameters type

```

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

```

Result type

```

2245 result ::      dataset
2246
2247

```

Additional constraints

The aliases must be all distinct and different from the Data Set names. Aliases are mandatory for Data Sets which appear more than once in the Join (self-join) and for non-named Data Set obtained as result of a sub-expression. The **using** clause is not allowed for the **full_join** and for the **cross_join**, because otherwise a non-functional result could be obtained.

If the using clause is not specified (we will label this case as “Case A”), calling $\text{Id}(ds_i)$ the set of Identifier Components of operand ds_i , the following group of constraints must hold⁷:

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

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

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

The join operators must fulfil also other constraints:

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

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

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

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_1).
- The filterCondition, if present, is applied on VDS_1 , producing the Virtual Data Set VDS_2 .
- The specified calculation algorithms (**apply**, **calc** or **aggr**), if present, are applied on VDS_2 . For the Attributes that have not been explicitly calculated in these clauses, the Attribute propagation rule is applied (see the User Manual), so producing the Virtual Data Set VDS_3 .
- The **keep** or **drop** clause, if present, is applied on VDS_3 , producing the Virtual Data Set VDS_4 .
- The **rename** clause, if present, is applied on VDS_4 , producing the Virtual Data Set VDS_5 .
- The final automatic alias removal is performed in order to obtain the output Data Set.

2285 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.

2286 The **structure of the virtual Data Set** VDS_1 which is the output of the relational join is the following.

2287 For the **inner_join**, the **left_join** and the **full_join**, the virtual Data Set contains the following Components:

- The Components used as join keys, which appear once and maintain their original names and roles. In the cases A and B1, all of them are Identifiers. In the sub-case B2, the result takes the roles from the reference Data Set.
- In the sub-case B2: the Identifiers of the reference Data Set, which appear once and maintain their original name and role.
- The other Components coming from exactly one input Data Set, which appear once and maintain their original name
- The other Components coming from more than one input Data Set, which appears as many times as the Data Set they come from; to distinguish them, their names are prefixed with the alias (or the name) of the Data Set they come from, separated by the “#” symbol (e.g., $ds_i\#cmp_j$). For example, if the Component “population” appears in two input Data Sets “ ds_1 ” and “ ds_2 ” 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., “ $ds_1\#population$ ” and “ $ds_2\#population$ ”). 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.

2291 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.

2292 The **semantics of the relational join** is the following.

2293 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.

2294 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.

2295 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.

2296 The different kinds of relational joins behave as follows.

- **inner_join**: the Data Points of ds_1 , ..., ds_N are joined if they have the same values for the common Identifier Components or, if the **using** clause is present, for the specified Components. A (joined) virtual Data Point is generated in the virtual Data Set VDS_1 when a matching Data Point is found for each one of the input Data Sets. In this case, the Values of the Components of a virtual Data Point are taken from the

2335 corresponding Components of the matching Data Points. If there is no match for one or more input Data Sets,
2336 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 $i+1^{\text{th}}$ 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 $i+1^{\text{th}}$ 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 $i+1^{\text{th}}$ 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.

2373 The **semantics of the clauses** is the following.

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

2395 Components of the input Virtual Data Set and cannot use Components generated, for example, by other
2396 `calcExpr`. If the calculated Component is a new Component, it is added to the output virtual Data Set. If
2397 the Calculated component is a Measure or an Attribute that already exists in the input virtual Data Set,
2398 the calculated values overwrite the original values. If the Calculated component is an Identifier that
2399 already exists in the input virtual Data Set, an exception is raised because overwriting an Identifier
2400 Component is forbidden for preserving the functional behaviour. Analytic operators can be used in the
2401 `calc` clause.

- 2402 • **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of sub-
2403 expressions at Component level. Each Component is calculated through an independent sub-expression.
2404 It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or
2405 **viral attribute**. The substring **viral** allows to control the virality of Attributes, if the Attribute
2406 propagation rule is adopted (see the User Manual). The **aggr** sub-expressions are independent of one
2407 another, they can only reference Components of the input Virtual Data Set and cannot use Components
2408 generated, for example, by other **aggr** sub-expressions. The **aggr** computed Measures and Attributes
2409 are the only Measures and Attributes returned in the output virtual Data Set (plus the possible viral
2410 Attributes, see below **Attribute propagation**). The sub-expressions must contain only Aggregate
2411 operators, which are able to compute an aggregated Value relevant to a group of Data Points. The groups
2412 of Data Points to be aggregated are specified through the `groupingClause`, which allows the following
2413 alternative options.

- 2414 **group by** the Data Points are grouped by the values of the specified Identifier. The Identifiers not
2415 specified are dropped in the result.
2416 **group except** the Data Points are grouped by the values of the Identifiers not specified in the clause.
2417 The specified Identifiers are dropped in the result.
2418 **group all** converts an Identifier Component using `conversionExpr` and keeps all the resulting
2419 Identifiers.

2420 The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on
2421 the single groups, for example the minimum number of rows in the group.

2422 If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the
2423 clause returns a Data Set that contains a single Data Point and has no Identifier Components.

- 2424 • **keep** maintains in the output only the specified dependent Components (Measures and Attributes) of
2425 the input virtual Data Set and drops the non-specified ones. It has the role of a projection in the usual
2426 relational semantics (specifying which columns have to be projected in). Only one **keep** clause is
2427 allowed. If **keep** is used, **drop** must be omitted.
2428 • **drop** maintains in the output only the non-specified dependent Components (Measures and Attributes)
2429 of the input virtual Data Set (`component<scalar>`) and drops the specified ones. It has the role of a
2430 projection in the usual relational join semantics (specifying which columns will be projected out). Only
2431 one **drop** clause is allowed. If **drop** is used, **keep** must be omitted.
2432 • **rename** assigns new names to one or more Components (Identifier, Measure or Attribute Components).
2433 The resulting Data Set, after renaming all the specified Components, must have unique names of all its
2434 Components (otherwise a runtime error is raised). Only the Component name is changed and not the
2435 Component Values, therefore the new Component must be defined on the same Value Domain and Value
2436 Domain Subset as the original Component (see also the IM in the User Manual). If the name of a
2437 Component defined on a different Value Domain or Set is assigned, an error is raised. In other words,
2438 rename is a transformation of the variable without any change in its values.

2439 The semantics of the **Attribute propagation** in the join is the following. The Attributes calculated through the
2440 `calc` or **aggr** clauses are maintained unchanged. For all the other Attributes that are defined as **viral**, the
2441 Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User
2442 Manual). This is done before the application of the **drop**, **keep** and **rename** clauses, which acts also on the
2443 Attributes resulting from the propagation.

2444 The semantics of the **final automatic aliases** removal is the following. After the application of all the clauses, the
2445 structure of the final virtual Data Set is further modified. All the Components of the form
2446 “alias#component_name” (or “dataset_name#component_name”) are implicitly renamed into
2447 “component_name”. This means that the prefixes in the Component names are automatically removed. It is
2448 responsibility of the user to guarantee the absence of duplicated Component names once the prefixes are
2449 removed. In other words, the user must ensure that there are no pairs of Components whose names are of the
2450 form “alias1#c1” and “alias2#c1” in the structure of the virtual Data Point, since the removal of “alias1” and
2451 “alias2” would cause the clash. If, after the aliases removal two Components have the same name, an error is
2452 raised. In particular, name conflicts may derive if the using clause is present and some homonym Identifier
2453 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 Given the operand Data Sets DS_1 and DS_2:
2459

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

2478

2479 Example 4:

```
2480 DS_r := cross_join (DS_1 as d1, DS_2 as d2
2481     rename d1#Id_1 to Id11, d1#Id_2 to Id12, d2#Id1 to Id21, d2#Id2 to Id22, d1#Me_2
2482     to Me12 )
```

2483 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

2485

2486

2487 Example 5:

```
2488 DS_r := inner_join (DS_1 as d1, DS_2 as d2
2489     filter Me_1 = "A"
2490     calc Me_4 = Me_1 || Me_1A
2491     drop d1#Me_2)
```

2492

2493

2494

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

2495

2496

2497

2498 Example 6:

```
2499 DS_r := inner_join ( DS_1
2500     calc Me_2 := Me_2 || "_NEW"
2501     filter Id_2 ="B"
2502     keep Me_1, Me_2)
```

2503

2504

2505

where || is the string concatenation,

results in:

DS_r			
Id_1	Id_2	Me_1	Me_2
1	B	C	D_NEW

2506

2507

2508 Example 7:

2509 Given the operand Data Sets DS_1 and DS_2:

2510

2511

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

2512

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

2513
2514 DS_r := inner_join (DS_1 as d1, DS_2 as d2
2515 apply d1 || d2)
2516

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 Components” (see the section “Typical behaviours of the ML Operators”).
2551
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"

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"

2561

2562 Example 2 (on component): DS_r := DS_1[calc Me_2:= Me_1 || " world"] results in:

2563

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

2564

Whitespace removal : trim, rtrim, ltrim

2565

Syntax

{trim|rtrim|rtrim}¹(op)

2566

2567

Input parameters

2568

op the operand

2569

2570

Examples of valid syntaxes

2571

trim("Hello ")

2572

trim(ds_1)

2573

2574

Semantics for scalar operations

2575

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

2576

2577

Input parameters type

2578

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

2579

2580

Result type

2581

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

2582

2583

Additional constraints

2584

None.

2585

2586

Behaviour

2587

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").

2588

2589

Examples

2590

2591

Given the Data Set DS_1:

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

2592

2600 Example 1: DS_r := rtrim(DS_1) results in:

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

2641

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

2642

2643 Example 2 (on component): DS_r := DS_1[calc Me_2:= upper(Me_1)] results in:

2644

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 *Syntax*

2648 **substr (op, start, length)**

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 *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 *Semantics for scalar operations*

2662 The operator extracts a substring from op, which must be *string* type. The substring starts from the startth character of the input string and has a number of characters equal to the length parameter.

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

2669 For example:

2671 substr ("abcdefghijklmnoprstuvwxyz", 5 , 10) gives: "efghijklmn".
 2672 substr ("abcdefghijklmnoprstuvwxyz", 25 , 10) gives: "yz".
 2673 substr ("abcdefghijklmnoprstuvwxyz", 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

2684 length :: component < integer [value >= 0] >
2685 | integer [value >= 0]

2686
2687
2688

2689 *Result type*
2690 result :: dataset { measure<string> _+ }
2691 | component<string>
2692 | string

2693

2694 *Additional constraints*

2695 None.

2696

2697 *Behaviour*

2698 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
2699 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
2700 the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the
2701 section “Typical behaviours of the ML Operators”).

2702

2703 *Examples*

2704

2705 Given the operand Data Set DS_1:

2706

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

2707

2708 Example 1: DS_r:= substr (DS_1 , 7) results in:

2709

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"

2710

2711 Example 2: DS_r:= substr (DS_1 , 1 , 5) results in:

2712

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

2713

2714 Example3(on Components): DS_r:= DS_1 [calc Me_2:= substr (Me_2 , 1 , 5)] results in:

2715

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

Examples of valid syntaxes

2781

instr (DS_1, "ab", 2 , 3)

2782

instr (DS_1, "ab", 2)

2783

instr (DS_1, "ab", _, 2)

2784

instr (DS_1, "ab")

2785

instr (DS_1, "ab")

2786

Semantics for scalar operations

2787

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

2788

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

2789

- If nthoccurrence is omitted, the value is 1.

2790

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

2791

2792

For example:

2793

instr ("abcde", "c") gives 3

2794

instr ("abcdeffrxcwsd", "c", _, 3) gives 10

2795

instr ("abcdeffrxcwsd", "c", 5 , 3) gives 0

2796

Input parameters type

2797

2798

2799

2800

2801

```

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 Given the Data Set DS_2:
2839

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

2840
2841 Example 3 (applying the **instr** operator at component level to a multi Measure Data Set):
2842
2843 DS_r := DS_2 [calc Me_10:= instr(Me_1, "o"), Me_20:=instr(Me_2, "o")] results in:
2844

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 Example 4 (applying the **instr** operator at Data Set level to a multi Measure Data Set):
2848
2849 DS_r := instr(DS_2, "o") would give error because DS_2 has more Measures.
2850

2851 String length : length

2852 *Syntax*
2853 **length** (op)

2854
2855 *Input Parameters*
2856 op the operand

2857
2858 *Examples of valid syntaxes*
2859 length("Hello, World!")
2860 length(DS_1)

2861
2862 *Semantics for scalar operations*
2863 Returns the length of a string. For example, length("Hello, World!") gives 13
2864 For the empty string "" the value 0 is returned

2865
2866 *Input Parameters type*
2867 op :: dataset { measure<string> _ }
2868 | component<string>
2869 | string

2870
2871 *Result type*
2872 result :: dataset { measure<integer[value >= 0]> int_var }
2873 | component<integer[value >= 0]>
2874 | integer[value >= 0]

2875
2876 *Additional constraints*
2877 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 *Examples*

2884 Given the Data Set DS_1

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

2885

2886

2887 Example 1: DS_r := length(DS_1) results in:

DS_r		
Id_1	Id_2	int_var
1	A	5
2	B	null

2888

2889

2890 Example 2 (on component): DS_r:= DS_1[calc Me_2:=length(Me_1)] results in

2891

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

2892

2893 Given the Data Set DS_2:

2894

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

2895

2896

2897 Example 3 (applying the **length** operator at component level to a multi Measure Data Set):

2898

2899 DS_r := DS_2 [calc Me_10:= length(Me_1), Me_20:=length(Me_2)] results in:

2900

2901

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

2905 Example 4 (**length** operator applied at Data Set level to a multi Measure Data Set):

2906

2907 DS_r := length(DS_2) would give error because DS_2 has more Measures.

2907 VTL-ML - Numeric operators

2908 Unary plus : +

2909 *Syntax*

2910 + op

2911

2912 *Input parameters*

2913 op the operand

2914

2915 *Examples of valid syntaxes*

2916 + DS_1

2917 + 3

2918

2919 *Semantics for scalar operations*

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

2921 + 3 gives 3

2922 + (- 5) gives - 5

2923

2924 *Input Parameters type*

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

2928

2929 *Result type*

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

2933

2934 *Additional constraints*

2935 None.

2936

2937 *Behaviour*

2938 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”).

2939
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*.

2943

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
2952

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

results in:

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

2953 Unary minus: $-$

2954 *Syntax*

2955 $- op$

2956

2957 *Input parameters*

2958 op the operand

2959

2960 *Examples of valid syntaxes*

2961 $- DS_1$

2962 $- 3$

2963

2964 *Semantics for scalar operations*

2965 The operator $-$ inverts the sign of op . For example:

2966 $- 3$ gives $- 3$

2967 $- (- 5)$ gives 5

2968

2969 *Input Parameters type*

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

2971 $| component<number>$

2972 $| number$

2973

2974 *Result type*

2975 $result :: dataset \{ measure<number> _+ \}$

2976 $| component<number>$

2977 $| number$

2978

2979 *Additional constraints*

2980 None.

2981

2982 *Behaviour*

2983 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”).

2984 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is
2985 the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is
2986 *number* then the result has type *number*.

2987

2988 *Examples*

2989 Given the operand Data Set DS_1:

2990

2991

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 Example 1: DS_r := - DS_1 results in:

2994

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

2997

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

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

3012 Semantics for scalar operations

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> _+ }
| component<number>
| number

3019

3020

3021 Result type

3022

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

3023

3024 Additional constraints

3025

None.

3026

3027

3028

3029 ***Behaviour***
 3030 The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set
 3031 Components” (see the section “Typical behaviours of the ML Operators”).
 3032 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is
 3033 the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is
 3034 of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.
 3035

3036 ***Examples***
 3037 Given the operand Data Sets DS_1 and DS_2:
 3038

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

3039

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

3040

3041 *Example 1:* DS_r := DS_1 + DS_2 results in:

3042

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

3043

3044 *Example 2:* DS_r := DS_1 + 3 results in:

3045

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

3046

3047 *Example 3 (on components):* DS_r := DS_1 [calc Me_3 := Me_1 + 3.0] results in:

3048

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 Input Parameters

3053 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 Components” (see the section “Typical behaviours of the ML Operators”).

3080 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*.

3081

3082 Examples

3083 Given the operand Data Sets DS_1 and DS_2:

3084

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

3085

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

3086

3087 Example 1: DS_r := DS_1 - DS_2 results in:

3088

3089

3090

3091

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

3092
3093
3094

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

3095
3096 Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_1 - 3] results in:
3097

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

3098

Multiplication : *

3100 *Syntax*
3101 op1 * op2
3102
3103 *Input parameters*
3104 op1 the multiplicand
3105 op2 the multiplier
3106
3107 *Examples of valid syntaxes*
3108 DS_1 * DS_2
3109 3 * 5
3110

Semantics for scalar operations

3112 The operator multiplication returns the product of two numbers. For example:

3 * 5 gives 15

3114
3115 *Input parameters type*
3116 op1, op2 :: dataset { measure<number> _+ }
3117 | component<number>
3118 | number
3119

3120 *Result type*
3121 result :: dataset { measure<number> _+ }
3122 | component<number>
3123 | number
3124

3125 *Additional constraints*

3126 None.

3127 *Behaviour*

3129 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”).

3130
3131 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is
3132 the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is
3133 of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

3134

3135 *Examples*

3136 Given the operand Data Sets DS_1 and DS_2:

3137

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

3138

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

3139

3140 Example 1: DS_r := DS_1 * DS_2 results in:

3141

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

3142

3143 Example 2: DS_r := DS_1 * -3 results in:

3144

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

3145

3146
3147 Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_1 * Me_2] results in:

3148

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 *Examples*

3185 Given the operand Data Sets DS_1 and DS_2:

3186

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

3189

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

3190

3191 Example 1: DS_r := DS_1 / DS_2 results in:

3192

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

3193

3194 Example 2: DS_r := DS_1 / 10 results in:

3195

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

3196

3197 Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_2 / Me_1] results in:

3198

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

3199

3200 Modulo : mod

3201 Syntax

3202 **mod** (op1 , op2)

3203

3204 Input parameters

3205 op1 the dividend

3206 op2 the divisor

3207

3208 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 *Input Parameters type*

```
3221 op1, op2 :: dataset { measure<number> _+ }
3222 | component<number>
3223 | number
3224 divisor :: number
```

3225 *Result type*

```
3226 result :: dataset { measure<number> _+ }
3227 | component<number>
3228 | number
```

3229 *Additional constraints*

3230 None.

3231

3232 *Behaviour*

3233 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”).

3234 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*.

3235 *Examples*

3236 Given the operand Data Sets DS_1 and DS_2:

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

3245

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

3250
3251 Example 2: $DS_r := \text{mod} (DS_1, 15)$ results in:
3252

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

3253
3254 Example 3 (on components): $DS_r := DS_1[\text{calc} Me_3 := \text{mod}(DS_1\#Me_1, 3.0)]$ results in:
3255

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

3256

3257 Rounding : **round**

3258 *Syntax*
3259 **round** (*op* , *numDigit*)

3260
3261 *Input parameters*
3262 *op* the operand
3263 *numDigit* the number of positions to round to
3264

3265 *Examples of valid syntaxes*

3266 **round** (*DS_1* , 2)
3267 **round** (*DS_2*)
3268 **round** (3.14159 , 2)
3269 **round** (3.14159 , _)
3270

3271 *Semantics for scalar operations*

3272 The operator **round** rounds the operand to a number of positions at the right of the decimal point equal to the
3273 *numDigit* parameter. The decimal point is assumed to be at position 0. If *numDigit* is negative, the rounding
3274 happens at the left of the decimal point. The rounding operation leaves the *numDigit* position unchanged if the
3275 *numDigit+1* position is between 0 and 4, otherwise it adds 1 to the number that is in the *numDigit* position. All
3276 the positions greater than *numDigit* are set to 0. The basic scalar type of the result is *integer* if *numDigit* is
3277 omitted, *number* otherwise.
3278

For example:

3279 **round** (3.14159, 2) gives 3.14
3280 **round** (3.14159, 4) gives 3.1416
3281 **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

```

Input parameters type

```

3286 op1 ::      dataset { measure<number> _+ }
3287          | component<number>
3288          | number
3289 numDigit:: component < integer >
3290          | integer
3291
3292

```

Result type

```

3293 result ::      dataset { measure<number> _+ }
3294          | component<number>
3295          | number
3296
3297

```

Additional constraints

3299 None.

3300

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

Examples

3307 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)

3312

results in:

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

3313

3314 Example 2 (on components): DS_r := DS_1 [calc Me_10:= round(Me_1)]

3315

results in:

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

11	B	44.5	24.3	45
----	---	------	------	----

3316
3317 Example 3 (on components) : DS_r := DS_1 [calc Me_20:= round(Me_1 , -1)]
3318

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

3319

3320 Truncation : **trunc**

3321 *Syntax*
3322 **trunc** (op , numDigit)

3323
3324 *Input Parameters*
3325 op the operand
3326 numDigit the number of position from which to trunc
3327

3328 *Examples of valid syntaxes*

3329 trunc (DS_1 , 2)
3330 trunc (DS_1)
3331 trunc (3.14159 , 2)
3332 trunc (3.14159 , _)
3333

3334 *Semantics for scalar operations*

3335 The operator **trunc** truncates the operand to a number of positions at the right of the decimal point equal to the
3336 numDigit parameter. The decimal point is assumed to be at position 0. If numDigit is negative, the truncation
3337 happens at the left of the decimal point. The truncation operation leaves the numDigit position unchanged. All
3338 the positions greater than numDigit are eliminated. The basic scalar type of the result is *integer* if numDigit is
3339 omitted, *number* otherwise.

3340 For example:

3341 trunc (3.14159 , 2)	gives 3.14
3342 trunc (3.14159 , 4)	gives 3.1415
3343 trunc (12345.6 , 0)	gives 12345.0
3344 trunc (12345.6)	gives 12345
3345 trunc (12345.6 , _)	gives 12345
3346 trunc (12345.6 , -1)	gives 12340.0

3347

3348 *Input parameters type*

3349 op ::	dataset { measure<number> _+ }
	component<number>
	number
3352 numDigit ::	component < integer >
	integer

3354 *Result type*

3356 result ::	dataset { measure<number> _+ }
	component<number>
	number

3359

3360 *Additional constraints*

3361 None.

3362

3363 *Behaviour*
3364 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
3365 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
3366 the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the
3367 section “Typical behaviours of the ML Operators”).
3368

3369 *Examples*
3370

3371 Given the operand Data Set DS_1:
3372

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

3373
3374 Example 1: DS_r := trunc(DS_1, 0) results in:
3375

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

3376
3377 Example 2 (on components): DS_r := DS_1[calc Me_10:= trunc(Me_1)] results in:
3378

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

3379
3380 Example 3 (on components): DS_r := DS_1[calc Me_20:= trunc(Me_1 , -1)] results in:
3381

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

3382

3383 **Ceiling :** **ceil**

3384 *Syntax*

3385 **ceil (op)**

3386

3387 *Input parameters*

3388 op the operand

3389

3390 *Examples of valid syntaxes*

3391 **ceil (DS_1)**

3392 **ceil (3.14159)**

3393

3394 *Semantics for scalar operations*

3395 The operator **ceil** returns the smallest integer greater than or equal to op.

3396 For example:

3397 **ceil(3.14159)** gives 4

3398 **ceil(15)** gives 15

3399 **ceil(-3.1415)** gives -3

3400 **ceil(-0.1415)** gives 0

3401

3402 *Input parameters type*

3403 op :: dataset { measure<number> _+ }
3404 | component<number>
3405 | number

3406

3407 *Result type*

3408 result :: dataset { measure<integer> _+ }
3409 | component<integer>
3410 | integer

3411

3412 *Additional constraints*

3413 None.

3414

3415 *Behaviour*

3416 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

3419 *Examples*

3420 Given the operand Data Set DS_1:

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

3423 **Example 1:** **DS_r := ceil (DS_1)** results in:

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 Component” (see the section “Typical behaviours of the ML Operators”).

3463

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
3470

Example 1: $DS_r := \text{floor}(DS_1)$

results in:

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

3471
3472
3473

Example 2 (on components): $DS_r := DS_1 [\text{calc} Me_10 := \text{floor}(Me_1)]$ results in:

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

3474

Absolute value : **abs**

3475

Syntax

abs (op)

3477

Input parameters

3479

op the operand

3480

Examples of valid syntaxes

3482

abs (DS_1)

3483

abs (-5)

3484

Semantics for scalar operations

3486

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

3487

For example:

3488

abs (-5.49) gives 5.49

3489

abs (5.49) gives 5.49

3490

Input parameters type

3492

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

3496

Result type

3498

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

3500

3501 | number [value >= 0]

3502 *Additional constraints*

3504 None.

3505 *Behaviour*

3507 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set
3508 Component” (see the section “Typical behaviours of the ML Operators”).

3509 *Examples*

3511 Given the operand Data Set DS_1:

3512

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

3513

3514 Example 1: DS_r := abs (DS_1) results in:

3515

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

3516

3517 Example 2 (on components): DS_r := DS_1 [calc Me_10 := abs(Me_1)] results in:

3518

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

3519

3520 Exponential : exp

3521 *Syntax*

3522 exp (op)

3523

3524 *Input parameters*

3525 op the operand

3526

3527 *Examples of valid syntaxes*

3528 exp (DS_1)

3529 exp (5)

3530

3531 *Semantics for scalar operations*

3532 The operator **exp** returns e (base of the natural logarithm) raised to the op-th power.

3533 For example;

3534 exp (5) gives 148.41315...

3535 exp (1) gives 2.71828... (the number e)

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

3539 *Input parameters type*

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

3543 *Result type*

```
3544 Result type  
3545 result :: dataset { measure<number[value > 0]> _+ }  
3546 | component<number [value > 0]>  
3547 | number[value > 0]
```

3548 *Additional constraints*

3549 Additi
3550 N

3550
2551

3551 3552 *Bahuvrihi*

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 MI Operators”)

3556 Examples

Given the operand Data Set DS_1:

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

3559

3559

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)] 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 *Syntax*

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:

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

3659

3660

3661 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

3663

3664 Example 2 (on components): DS_r := DS_1[calc Me_1 := power(Me_1, 2)] results in:
3665

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

3666

3667 **Logarithm :** **log**3668 *Syntax*3669 **log** (op , num)

3670

3671 *Input parameters*

3672 op the base of the logarithm

3673 num the number to which the logarithm is applied

3674

3675 *Examples of valid syntaxes*

3676 log (DS_1, 2)

3677 log (1024, 2)

3678

3679 *Semantics for scalar operations*3680 The operator **log** calculates the logarithm of num base op.

3681 For example:

3682 log (1024, 2) gives 10

3683 log (1024, 10) gives 3.01

3684

3685 *Input parameters type*

3686 op :: dataset { measure<number [value > 1] > _+ }

3687 | component<number [value > 1] >

3688 | number [value > 1]

3689 num :: component<integer [value > 0] >

3690 | integer [value > 0]

3691

3692 *Result type*

3693 result :: dataset { measure<number> _+ }

3694 | component<number>

3695 | number

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 Component” (see the section “Typical behaviours of the ML Operators”).
 3747
 3748
 3749 *Examples*
 3750 Given the operand Data Set DS_1:
 3751

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] results in:
3809

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 *Syntax*
3813 left <> right
3814

3815
3816 *Input parameters*
3817 left the left operand
3818 right the right operand
3819

3820 *Examples of valid syntaxes*
3821 DS_1 <> DS_2
3822

3823 *Semantics for scalar operations*
3824 The operator returns FALSE if the left is equal to right, TRUE otherwise.
3825 For example:

3826 5 <> 9 gives: TRUE
3827 5 <> 5 gives: FALSE
3828 "hello" <> "hi" gives: TRUE
3829

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

3843 *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”).

3847 *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

3867 left { > | >= }¹ right

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
 3900 behaviours of the ML Operators”).
 3901

3902 *Examples*
 3903 Given the operand Data Set DS_1:

DS_1					
Id_1	Id_2	Id_3	Id_4	Id_5	Me_1
2	G	2011	Total	Percentage	NULL
2	R	2011	Total	Percentage	12.2
2	F	2011	Total	Percentage	29.5

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

3911

3912 Given the left operand Data Set:

3913

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	42.5

3914

3915 and the right operand Data Set:

3916

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	33.7

3917

3918 Example 3: DS_r:= DS_1 > DS_2 results in:

3919

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

3920

3921 If the Me_1 column for Germany in the DS_2 Data Set had a NULL value the result would be:

3922

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

3923

3924 Less than : < <=

3925

3926 Syntax

left { < | <= }¹ right

3927

3928 Input parameters

3929 left the left operand

3930 right the right operand

3931

3932 Examples of valid syntaxes

3933 DS_1 < DS_2

3934 DS_1 <= DS_2

3935

3936 Semantics for scalar operations

3937 The operator < returns TRUE if left is smaller than right, FALSE otherwise.

3938 The operator <= returns TRUE if left is smaller than or equal to right, FALSE otherwise.

3939

3940	For example:	
3941	$5 < 4$	gives: FALSE
3942	$5 \leq 5$	gives: TRUE
3943	"hello" < "hi"	gives: TRUE

Input parameters type

```
3946    left, right ::      dataset {measure<scalar> _ }
3947                                | component<scalar>
3948                                | scalar
```

Result type

```
3950     Result of p  
3951 result :: dataset { measure<boolean> bool_var }  
3952 | component<boolean>  
3953 | boolean
```

Additional constraints

3956 Operands left and right must be of the same scalar type

Behaviour

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

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	46818219
2012	M	Total	Total	NULL
2012	F	Total	Total	5401267
2012	W	Total	Total	7954662

3966 Example 1: DS_r := DS_1 < 15000000 results in:

DS_r				
Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	TRUE
2012	G	Total	Total	TRUE
2012	S	Total	Total	FALSE
2012	M	Total	Total	NULL
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

3968

3969 Between : between

Syntax

between (op, from, to)

3974 *Input parameters*
 3975 op the Data Set to be checked
 3976 from the left delimiter
 3977 to the right delimiter
 3978
 3979 *Examples of valid syntaxes*
 3980 ds2 := between(ds1, 5,10)
 3981 ds2 := ds1 [calc m1 := between(me2, 5, 10)]
 3982
 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
 3985 is a shortcut for the following:
 3986
 3987 op >= from and op <= to
 3988
 3989 The types of op, from and to must be compatible scalar types.
 3990
 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>
 3998
 3999 *Result type*
 4000 result :: dataset { measure<boolelan> bool_var }
 4001 | component<boolean>
 4002 | boolean
 4003
 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
 4006 same as that of from and to.
 4007
 4008 *Behaviour*
 4009 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical
 4010 behaviours of the ML Operators”).
 4011
 4012 *Examples*
 4013
 4014 Given the following Data Set DS_1:
 4015

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	6
R	Total	Percentage	Total	-2

4016 Example 1: DS_r:= between(ds1, 5,10) results in:
 4017
 4018

DS_1				
Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	FALSE

4019

4020 Element of: **in** / **not_in**

4021 *Syntax*

```
4022     op in collection
4023     op not_in collection
4024
4025
4026         collection ::= set | valueDomainName
```

4028 *Input parameters*

```
4029     op           the operand to be tested
4030     collection   the the Set or the Value Domain which contains the values
4031     set          the Set which contains the values (it can be a Set name or a Set literal)
4032     valueDomainName  the name of the Value Domain which contains the values
4033
```

4034 *Examples of valid syntaxes*

```
4035     ds := ds_2 in {1,4,6}      as usual, here the braces denote a set literal (it contains the values 1, 4 and 6)
4036     ds := ds_3 in mySet
4037     ds := ds_3 in myValueDomain
```

4039 *Semantics for scalar operations*

4040 The **in** operator returns TRUE if op belongs to the collection, FALSE otherwise.

4041 The **not_in** operator returns FALSE if op belongs to the collection, TRUE otherwise.

4042 For example:

4043 1 in { 1, 2, 3 }	returns	TRUE
4044 "a" in { "c", "ab", "bb", "bc" }	returns	FALSE
4045 "b" not_in { "b", "hello", "c" }	returns	FALSE
4046 "b" not_in { "a", "hello", "c" }	returns	TRUE

4048 *Input parameters type*

```
4049     op ::    dataset {measure<scalar> _}
4050             | component<scalar>
4051             | scalar
4052     collection :: set<scalar> | name<value_domain>
```

4054 *Result type*

```
4055     result ::   dataset { measure<boolean> bool_var }
4056             | component<boolean>
4057             | boolean
```

4059 *Additional constraints*

4060 The operand must be of a basic scalar data type compatible with the basic scalar type of the collection.

4061 *Behaviour*

4063 *Semantics*

4064 The **in** operator evaluates to TRUE if the operand is an element of the specified collection and FALSE otherwise,
4065 the **not_in** the opposite.

4066 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical
4067 behaviours of the ML Operators”).

4068 The collection can be either a *set* of values defined in line or a name that references an externally defined Value
4069 Domain or Set.

4071 *Examples*

4072 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

4074
4075
4076
4077
4078

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

4079
4080
4081
4082
4083

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

4084
4085
4086
4087

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

4088
4089
4090
4091
4092
4093

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

4094
4095

4096 **match_characters**

match_characters

4097 *Syntax*

4098 **match_characters** (op , pattern)

4099

4100 *Input parameters*

4101

op the dataset to be checked

4102

pattern the regular expression to check the Data Set or the Component against

4103

4104 *Examples of valid syntaxes*

4105

4106 **match_characters**(ds1, “[abc]+\\d\\d”)

4107 ds1 [**calc** m1 := **match_characters**(ds1, “[abc]+\\d\\d”)]

4108

4109 *Semantics for scalar operations*

4110

4111 **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.

4112

4113 *Input parameters type*

4114

4115 op :: dataset {measure<string> _}

4116

| component<string>

4117

| string

4118

pattern :: string | component<string>

4119

4120 *Result type*

4121

4122 result :: dataset { measure<boolelan> bool_var }

4123

| component<boolean>

4124

| boolean

4125

4126 *Additional constraints*

4127

4128 If op is a Data Set then it has exactly one measure.

4129

4130

4131

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:

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	<i>Examples of valid syntaxes</i>																																									
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	<i>Semantics for scalar operations</i>																																									
4209	This operator cannot be applied to scalar values.																																									
4210																																										
4211	<i>Input parameters type</i>																																									
4212	op1,																																									
4213	op2 :: dataset																																									
4214																																										
4215	<i>Result type</i>																																									
4216	result :: dataset { measure<boolean> bool_var }																																									
4217																																										
4218	<i>Additional constraints</i>																																									
4219	op1 has at least all the identifier components of op2 or op2 has at least all the identifier components of op1.																																									
4220																																										
4221	<i>Behaviour</i>																																									
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.																																									
4223																																										
4224	The result has the same Identifiers as op1 and a <i>boolean</i> 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.																																									
4225																																										
4226																																										
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.																																									
4228																																										
4229																																										
4230	The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").																																									
4231																																										
4232																																										
4233	<i>Examples</i>																																									
4234	Given the operand Data Sets DS_1 and DS_2:																																									
4235																																										
	<table border="1"> <thead> <tr><th colspan="5">DS_1</th></tr> <tr><th>Id_1</th><th>Id_2</th><th>Id_3</th><th>Id_4</th><th>Me_1</th></tr> </thead> <tbody> <tr><td>2012</td><td>B</td><td>Total</td><td>Total</td><td>11094850</td></tr> <tr><td>2012</td><td>G</td><td>Total</td><td>Total</td><td>11123034</td></tr> <tr><td>2012</td><td>S</td><td>Total</td><td>Total</td><td>46818219</td></tr> <tr><td>2012</td><td>M</td><td>Total</td><td>Total</td><td>417546</td></tr> <tr><td>2012</td><td>F</td><td>Total</td><td>Total</td><td>5401267</td></tr> <tr><td>2012</td><td>W</td><td>Total</td><td>Total</td><td>7954662</td></tr> </tbody> </table>		DS_1					Id_1	Id_2	Id_3	Id_4	Me_1	2012	B	Total	Total	11094850	2012	G	Total	Total	11123034	2012	S	Total	Total	46818219	2012	M	Total	Total	417546	2012	F	Total	Total	5401267	2012	W	Total	Total	7954662
DS_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																																										
	<table border="1"> <thead> <tr><th colspan="5">DS_2</th></tr> <tr><th>Id_1</th><th>Id_2</th><th>Id_3</th><th>Id_4</th><th>Me_1</th></tr> </thead> <tbody> <tr><td>2012</td><td>B</td><td>Total</td><td>Total</td><td>0.023</td></tr> <tr><td>2012</td><td>G</td><td>Total</td><td>M</td><td>0.286</td></tr> <tr><td>2012</td><td>S</td><td>Total</td><td>Total</td><td>0.064</td></tr> <tr><td>2012</td><td>M</td><td>Total</td><td>M</td><td>0.043</td></tr> </tbody> </table>		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										
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

Example 1: DS_r := exists_in (DS_1, DS_2, all)

results in:

4241

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

Example 2: DS_r := exists_in (DS_1, DS_2, true)

results in:

4244

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

Example 3: DS_r := exists_in (DS_1, DS_2, false)

results in:

4247

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
4264 boolean type.

4265 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
4290 behaviours of the ML Operators”).

4291

4292 *Examples*

4293 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

4295
4296

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

4297
4298
4299 Example 1: DS_r:= DS_1 and DS_2 results in:
4300

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

4301
4302 Example 2 (on Components): DS_r := DS_1 [calc Me_2:= Me_1 and true] results in:
4303

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

4304 Logical disjunction : or

4305 *Syntax*
4306 op1 or op2

4307
4308 *Input parameters*
4309 op1 the first operand
4310 op2 the second operand

4311
4312 *Examples of valid syntaxes*
4313 DS_1 or DS_2

4314
 4315 *Semantics for scalar operations*
 4316 The **or** operator returns TRUE if at least one of the operands is TRUE, otherwise FALSE. The two operands must
 4317 be of *boolean* type.
 4318 For example:

4319	FALSE or FALSE	gives FALSE
4320	FALSE or TRUE	gives TRUE
4321	FALSE or NULL	gives NULL
4322	TRUE or FALSE	gives TRUE
4323	TRUE or TRUE	gives TRUE
4324	TRUE or NULL	gives TRUE
4325	NULL or NULL	gives NULL

4326
 4327 *Input parameters type*

4328 op1,
 4329 op2 :: dataset {measure<boolean> _ }
 4330 | component<boolean>
 4331 | boolean

4332 *Result type*

4333 result :: dataset { measure<boolean> _ }
 4334 | component<boolean>
 4335 | boolean

4336
 4337 *Additional constraints*

4338 None.

4339

4340 *Behaviour*

4341 The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical
 4342 behaviours of the ML Operators”).

4343

4344 *Examples*

4345 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

4347

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

4349

4350 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 Example 2 (on Components): DS_r:= DS_1 [calc Me_2:= Me_1 or true]

4354

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
4370 operands must be of boolean type.

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

4380 Input parameters type

4381 op1,

4382 op2 :: dataset {measure<boolean> _ }
| component<boolean>
| boolean

4383

4384

4385
4386 *Result type*
4387 result :: dataset { measure<boolean> _ }
4388 | component<boolean>
4389 | boolean
4390
4391 *Additional constraints*
4392 None.
4393
4394 *Behaviour*
4395 The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical
4396 behaviours of the ML Operators”).
4397
4398 *Examples*
4399 Given the operand Data Sets DS_1 and DS_2:
4400

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

4401
4402

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

4403
4404 Example 1: DS_r:=DS_1 xor DS_2 results in:
4405

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

4406
4407 Example 2 (on Components): DS_r:= DS_1 [calc Me_2:= Me_1 xor true] results in:
4408

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:

not FALSE	gives TRUE
not TRUE	gives FALSE
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 Example 1: DS_r:= not DS_1 results in:

4450

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 Example 2 (on Components): DS_r:= DS_1 [calc Me_2 := not Me_1] results in:

4453

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

4455 VTL-ML - Time operators

4456 This chapter describes the **time** operators, which are the operators dealing with **time**, **date** and **time_period**
4457 basic scalar types. The general aspects of the behaviour of these operators is described in the section “Behaviour
4458 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.

4467 The time, date and time period formats used here are explained in the User Manual in the section “External
4468 representations and literals used in the VTL Manuals”.

4469 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.

4480 For a fully-detailed explanation, please refer to the User Manual.

Period indicator : period_indicator

4484 The operator **period_indicator** extracts the period indicator from a *time_period* value.

4485 *Syntax*

period_indicator ({ op })

4488 *Input parameters*

4489 op the operand

4491 Examples of valid syntaxes

4492 period indicator (ds 1)

4493 period_indicator
(if used in a clause the operand op can be omitted)

4495 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 ...).

4499 *Input parameters type*

```
4500 op :: dataset { identifier <time_period> _ , identifier *_ }  
4501 | component<time_period>  
4502 | time period
```

4504 *Result type*

```
4505 result :: dataset { measure<duration> duration_var }
4506          | component <duration>
4507          | duration
```

4509 *Additional constraints*

4510 If op is a Data Set then it has exactly an Identifier of type *time_period* and may have other Identifiers. If the
4511 operator is used in a clause and op is omitted, then the Data Set to which the clause is applied has exactly an
4512 Identifier of type *time_period* and may have other Identifiers.

4513 *Behaviour*

4515 The operator extracts the period indicator part of the *time_period* value. The period indicator is computed for
4516 each Data Point. When the operator is used in a clause, it extracts the period indicator from the *time_period*
4517 value the Data Set to which the clause is applied.

4518 The operator returns a Data Set with the same Identifiers of op and one Measure of type *duration* named
4519 duration_var. As for all the Variables, a proper Value Domain must be defined to contain the possible values of
4520 the period indicator and duration_var. The values used in the examples are listed at the beginning of this chapter
4521 "VTL-ML Time operators".

4522 *Examples*

4523 Given the Data Set DS_1:

DS_r			
Id_1	Id_2	Id_3	Me_1
A	1	2010	10
A	1	2013Q1	50

4524

4525 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

4527

4528 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

4530

4531

4532

4533 Fill time series : **fill_time_series**

4534

4535 *Syntax*

4536 **fill_time_series** (op { , limitsMethod })

4537

4538 limitsMethod ::= **single** | **all**

4539

4540 *Input parameters*

4541 op the operand

4542 limitsMethod method for determining the limits of the time interval to be filled (default: **all**)

4543

4544 *Examples of valid syntaxes*

4545 **fill_time_series** (ds)

4546 **fill_time_series** (ds, all)

4547
 4548 *Semantics for scalar operations*
 4549 The fill_time_series operator does not perform scalar operations.
 4550
 4551 *Input parameters type:*
 4552 op :: dataset { identifier <time> _, identifier _* }
 4553
 4554 *Result type:*
 4555 result :: dataset { identifier <time> _, identifier _* }
 4556
 4557
 4558 *Additional constraints*
 4559 The operand op has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.
 4560
 4561 *Behaviour*
 4562 This operator can be applied only on Data Sets of time series and returns a Data Set of time series.
 4563 The operator fills the possibly missing Data Points of all the time series belonging to the operand op within the
 4564 time limits automatically determined by applying the limit_method.
 4565 If limitsMethod is **all**, the time limits are determined with reference to all the time_series of the Data Set: the
 4566 limits are the minimum and the maximum values of the reference time Identifier Component of the Data Set.
 4567 If limitsMethod is **single**, the time limits are determined with reference to each single time_series of the Data
 4568 Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the time
 4569 series.
 4570 The expected Data Points are determined, for each time series, by considering the limits above and the period
 4571 (frequency) of the time series: all the Identifiers are kept unchanged except the reference time Identifier, which is
 4572 increased of one period at a time (e.g. day, week, month, quarter, year) from the lower to the upper time limit.
 4573 For each increase, an expected Data Point is identified.
 4574 If this expected Data Points is missing, it is added to the Data Set. For the added Data Points, Measures and
 4575 Attributes assume the NULL value.
 4576 The output Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set. The
 4577 output Data Set contains the same time series as the operand, because the time series Identifiers (all the
 4578 Identifiers except the reference time Identifier) are not changed.
 4579 As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the
 4580 reference time Identifier as well as the period of each time series.
 4581

4582 *Examples*
 4583 As described in the User Manual, the *time* data type is the intervening time between two time points and using the
 4584 ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In
 4585 the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-
 4586 MM/YYYY-MM is used.
 4587
 4588 Given the Data Set DS_1, which contains *annual* time series, where Id_2 is the reference time Identifier of *time*
 4589 type.:
 4590

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!"

4591
 4592 Example 1: DS_r := fill_time_series (DS_1, single) results in:
 4593

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 Example 2: DS_r := fill_time_series (DS_1, all)

4596

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 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:

4599

4600

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 Example 3: DS_r := fill_time_series (DS_2, single)

4603

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 Example 4: DS_r := fill_time_series (DS_2, all)

4606

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 Given the Data Set DS_3, which contains *annual* time series, where Id_2 is the reference time Identifier of
4610 *time_period* type:
4611

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 Example 5: DS_r := fill_time_series (DS_3, single)

4614

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 Example 6: DS_r := fill_time_series (DS_3, all)

4617

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

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,:
4621

4622

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 Example 7: DS_r := fill_time_series (DS_4, single)

4625

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.

4658

The *flow_to_stock* operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4659

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.

4660

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.

4661

4662

4671 Examples

4672

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.

4673

Given the Data Set DS_1, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time* type:

4674

DS_1		
Id_1	Id_2	Me_1
A	2010-01/2010-12	2
A	2011-01/2011-12	5

4675

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

4683

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 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):

4687

4688

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 Example 2: DS_r := flow_to_stock (DS_2) results in:

4691

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 Given the Data Set DS_3, which contains *annual* time series, where Id_2 is the reference time Identifier of
4694 *time_period* type:
4695

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 Example 3: DS_r := flow_to_stock (DS_3) results in:
4698

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 Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same
4702 phenomenon "A", where Id_2 is the reference time Identifier of *time_period* type:
4703

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

4704
4705
4706

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

4707
4708

4709 Stock to flow : stock_to_flow

4710
4711 *Syntax*
4712 **stock_to_flow (op)**
4713

4714 *Input parameters*

4715 op the operand

4716

4717

4718 *Examples of valid syntaxes*

4719 stock_to_flow (ds_1)

4720

4721 *Semantics for scalar operations*

4722 This operator does not perform scalar operations.

4723

4724 *Input parameters type:*

4725 op :: dataset { identifier < time > _, identifier _*, measure<number> _+ }

4726

4727 *Result type:*

4728 result :: dataset { identifier < time > _, identifier _*, measure<number> _+ }

4729

4730 *Additional constraints*

4731 The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

4732

4733 *Behaviour*

4734 The statistical data that describe the “state” of a phenomenon on a given moment (e.g. resident population on a
4735 given moment) are often referred to as “stock data”.

4736 On the contrary, the statistical data that describe “events” which can happen continuously (e.g. changes in the
4737 resident population, such as births, deaths, immigration, emigration), are often referred to as “flow data”.

4738 This operator takes in input a Data Set of time series which is interpreted as stock data and, for each time series,
4739 calculates the corresponding flow data by subtracting from the measure values of each regular period the
4740 corresponding measure values of the previous one.

4741 The **stock_to_flow** operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4742 The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and
4743 contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the
4744 reference time Identifier) are not changed.

4745 The Attribute propagation rule is not applied.
4746 As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the
4747 *time* Identifier as well as the *period* of each time series.

4748
4749
4750 *Examples*
4751

4752 As described in the User Manual, the *time* data type is the intervening time between two time points and using the
4753 ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In
4754 the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-
4755 MM/YYYY-MM is used.

4756 Given the Data Set DS_1, which contains *annual* time series, where Id_2 is the reference time Identifier of *time*
4757 type:
4758

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

4760
4761 Example 1: DS_r := stock_to_flow (DS_1) results in:
4762

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

4763
4764 Given the Data Set DS_2, which contains *annual* time series, where Id_2 is the reference time Identifier of *date*
4765 type (conventionally each period is identified by its last day):
4766

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 Example 2: DS_r := stock_to_flow (DS_2) results in:
4770

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 Given the Data Set DS_3, which contains *annual* time series, where Id_2 is the reference time Identifier of
4774 *time_period* type:
4775

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 Example 3: DS_r := stock_to_flow (DS_3) results in:
4778

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

4779

4780

4781

4782 Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same
 4783 phenomenon "A", where Id_2 is the *time* Identifier of *time_period* type:
 4784

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

4785

4786 Example 4: DS_r := stock_to_flow (DS_4) results in:
 4787

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

4788

4789 Time shift : timeshift

4790 *Syntax*

4791 **timeshift** (op , shiftNumber)

4792

4793 *Input parameters*

4794 op the operand

4795 shiftNumber the number of periods to be shifted

4796

4797 *Examples of valid syntaxes*

4798 timeshift (DS_1, 2)

4799 timeshift (DS_1, 1)

4800

4801 *Semantics for scalar operations*

4802 This operator does not perform scalar operations.

4803

4804 *Input parameters type:*
 4805 op :: dataset { identifier < time > _, identifier _* }
 4806 shiftNumber :: integer
 4807
 4808 *Result type:*
 4809 result :: dataset { identifier < time > _, identifier _* }
 4810
 4811 *Additional constraints*
 4812 The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.
 4813
 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.
 4826
 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
 4833 Given the Data Set DS_1, which contains *yearly* time series, where *Id_2* is the reference time Identifier of *time*
 4834 type:
 4835

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!"

4839

4840

4841 Given the Data Set DS_2, which contains *annual* time series, where Id_2 is the reference time Identifier of *date*
 4842 type (conventionally each period is identified by its last day):
 4843

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!"

4844

4845 Example 2: DS_r := timeshift (DS_2 , 2) results in:

4846

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!"

4847

4848

4849 Given the Data Set DS_3, which contains *annual* time series, where Id_2 is the reference time Identifier of
 4850 *time_period* type:
 4851

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: $DS_r := \text{timeshift} (DS_3 , 1)$ 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

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:

4857
4858
4859

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

Example 4: $DS_r := \text{time_shift} (DS_3 , -1)$ results in:

4862

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 *Examples*

4925 Given the Data Set DS_1

4926

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 Example 1: DS_r := sum (DS_1) group all time_agg ("A" , _ , Me_1) results in:

4929

DS_r		
Id_1	Id_2	Me_1
2010	A	60
2011	B	100
2010	C	20

4930

4931

4932 Example 2: DS_r := time_agg ("Q", cast ("2012M01", time_period, "YYYY\MM"))

4933

4934 Returns: "2012Q1".

4935

4936 Example 3: The following example maps a *date* to quarter level, 2012 (end of the period).

4937

4938 time_agg("Q", cast("20120213", date, "YYYYMMDD"), _ , last)

4939

4940 and produces a *date* value corresponding to the *string* "20120331"

4941

4942 Example 4: The following example maps a *date* to year level, 2012 (beginning of the period).

4943

4944 time_agg(cast("A", "2012M1", date, "YYYYMMDD"), _ , first)

4945

4946 and produces a *date* value corresponding to the *string* "20120101".

4947

4948 Actual time : **current_date**

4949

4950 *Syntax*

4951 **current_date ()**

4952

4953 *Input parameters*

4954 None

4955

4956 *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

4977

VTL-ML - Set operators

4978 Union: **union**

4979

Syntax

4980 **union (dsList)**

4982

4983 dsList ::= ds { , ds }*

4984

Input parameters

4986 dsList the list of Data Sets in the union

4987

Examples of valid syntaxes

4989 **union (ds2, ds3)**

4990

Semantics for scalar operations

4992 This operator does not perform scalar operations.

4993

Input parameters type

4995 ds :: dataset

4996

Result type

4998 result :: dataset

4999

Additional constraints

5001 All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

5002

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 The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of
5006 Data Points in the resulting Data Set, those Data Points are filtered by choosing the Data Point belonging to the left
5007 most operand Data Set. For instance, let's assume that in **union (ds1, ds2)** the operand ds1 contains a Data
5008 Point dp1 and the operand ds2 contains a Data Point dp2 such that dp1 has the same Identifiers values of dp2,
5009 then the resulting Data Set contains dp1 only.5010 The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical
5011 behaviours of the ML Operators").

5012 The automatic Attribute propagation is not applied.

5013

Examples

5017

5018 Given the operand Data Sets DS_1 and DS_2:

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 := 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 := 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
5036
5037
5038
5039
5040
5041
5042
5043

Intersection : intersect

Syntax

intersect (dsList)

dsList ::= ds { , ds }*

Input parameters

dsList the list of Data Sets in the intersection

5044	<i>Examples of valid syntaxes</i>
5045	intersect (ds2, ds3)
5046	
5047	<i>Semantics for scalar operations</i>
5048	This operator cannot be applied to scalar values.
5049	
5050	<i>Input parameters type</i>
5051	ds :: dataset
5052	
5053	<i>Return type</i>
5054	result :: dataset
5055	
5056	<i>Additional constraints</i>
5057	All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.
5058	
5059	<i>Behaviour</i>
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.
5061	
5062	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.
5063	
5064	The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").
5065	
5066	The automatic Attribute propagation is not applied.
5067	
5068	
5069	
5070	
5071	
5072	<i>Examples</i>
5073	Given the operand Data Sets DS_1 and DS_2:
5074	
5075	
5076	
5077	
5078	Example 1: DS_r := intersect(DS_1,DS_2) results in:
5079	
5080	
5081	Set difference : setdiff

5082																																									
5083	<i>Syntax</i>																																								
5084	setdiff (ds1, ds2)																																								
5085																																									
5086	<i>Input parameters</i>																																								
5087	ds1 the first Data Set in the difference (the minuend)																																								
5088	ds2 the second Data Set in the difference (the subtrahend)																																								
5089																																									
5090	<i>Examples of valid syntaxes</i>																																								
5091	setdiff (ds2, ds3)																																								
5092																																									
5093	<i>Semantics for scalar operations</i>																																								
5094	This operator cannot be applied to scalar values.																																								
5095																																									
5096	<i>Input parameters type</i>																																								
5097	ds1, ds2 :: dataset																																								
5098																																									
5099	<i>Result type</i>																																								
5100	result :: dataset																																								
5101																																									
5102	<i>Additional constraints</i>																																								
5103	The operand Data Sets have the same Identifier, Measure and Attribute Components.																																								
5104																																									
5105	<i>Behaviour</i>																																								
5106	The operator implements the set difference of functions (i.e. Data Sets), interpreting the Data Points of the input																																								
5107	Data Sets as the elements belonging to the operand sets, the minuend and the subtrahend, respectively. The																																								
5108	operator returns one single Data Set, with the same Identifier, Measure and Attribute Components as the																																								
5109	operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second. In other																																								
5110	words, for setdiff (ds1, ds2), the resulting Dataset contains all the data points Data Point dp1 of the operand ds1																																								
5111	such that there is no Data Point dp2 of ds2 having the same values for homonym Identifier Components.																																								
5112	The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical																																								
5113	behaviours of the ML Operators").																																								
5114	The automatic Attribute propagation is not applied.																																								
5115																																									
5116	<i>Examples</i>																																								
5117	Given the operand Data Sets DS_1 and DS_2:																																								
5118																																									
5119																																									
5120																																									
5121																																									
<table border="1"><thead><tr><th colspan="5">DS_1</th></tr><tr><th>Id_1</th><th>Id_2</th><th>Id_3</th><th>Id_4</th><th>Me_1</th></tr></thead><tbody><tr><td>2012</td><td>B</td><td>Total</td><td>Total</td><td>10</td></tr><tr><td>2012</td><td>G</td><td>Total</td><td>Total</td><td>20</td></tr><tr><td>2012</td><td>F</td><td>Total</td><td>Total</td><td>30</td></tr><tr><td>2012</td><td>M</td><td>Total</td><td>Total</td><td>40</td></tr><tr><td>2012</td><td>I</td><td>Total</td><td>Total</td><td>50</td></tr><tr><td>2012</td><td>S</td><td>Total</td><td>Total</td><td>60</td></tr></tbody></table>		DS_1					Id_1	Id_2	Id_3	Id_4	Me_1	2012	B	Total	Total	10	2012	G	Total	Total	20	2012	F	Total	Total	30	2012	M	Total	Total	40	2012	I	Total	Total	50	2012	S	Total	Total	60
DS_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																																					
<table border="1"><thead><tr><th colspan="5">DS_2</th></tr><tr><th>Id_1</th><th>Id_2</th><th>Id_3</th><th>Id_4</th><th>Me_1</th></tr></thead><tbody><tr><td>2011</td><td>B</td><td>Total</td><td>Total</td><td>10</td></tr><tr><td>2012</td><td>G</td><td>Total</td><td>Total</td><td>20</td></tr><tr><td>2012</td><td>F</td><td>Total</td><td>Total</td><td>30</td></tr><tr><td>2012</td><td>M</td><td>Total</td><td>Total</td><td>40</td></tr></tbody></table>		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										
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 Example 1: DS_r := setdiff (DS_1, DS_2) results in:

5124

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	10

5125

5126 Given the operand Data Sets DS_1 and DS_2 :

5127

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
R	M	2011		7
R	F	2011		10
R	T	2011		12

5128

5129

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1
R	M	2011		7
R	F	2011		10

5130

5131 Example 2: DS_r := setdiff (DS_1 , DS_2) results in:

5132

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
R	T	2011		12

5133

5134

5135 Simmetric difference : **symdiff**

5136

5137 *Syntax*5138 **symdiff (ds1, ds2)**

5139

5140 *Input parameters*

5141 ds1 the first Data Set in the difference

5142 ds2 the second Data Set in the difference

5143

5144 *Examples of valid syntaxes*

5145 symdiff (ds_2, ds_3)

5146

5147 *Semantics for scalar operations*

5148 This operator cannot be applied to scalar values.

5149

5150 *Input parameters type*

5151 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 `sympdiff (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 := `sympdiff (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

5182 VTL-ML - Hierarchical aggregation

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).
5194 ruleComp	ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined in hr.
5195 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.
5196 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.
5197 output	this parameter specifies the content of the resulting Data Set. The meaning of the possible values of the parameter is explained below.

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<number> _ }  
5214 hr :: name < hierarchical >  
5215 condComp :: name < component >  
5216 ruleComp :: name < identifier >
```

5217

5218 *Result type*

```
5219 result :: dataset {measure<number> _ }
```

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.
5223 Moreover, the Components specified as condComp and ruleComp must belong to the operand op and must take
5224 values on the Value Domains corresponding, in positional order, to the ones specified in the condition and rule
5225 parameter of hr.

5226 If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be
5227 specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in
5228 this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in
5229 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
5233 Set with the same Identifiers and the same Measure as op. The Attribute propagation rule is applied on the
5234 groups of Data Points which contribute to the same Data Points of the result.

5235 The behaviours relevant 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
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
Non_zero	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)

Partial_null	NULL	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Partial_zero	Zero	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

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
```

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 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 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 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

5327

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

5336 *in a Data Set expression:*5337 aggregateOperator (firstOperand { , additionalOperand }* { groupingClause })

5338

5339 *in a Component expression within an aggr clause*5340 aggregateOperator (firstOperand { , additionalOperand }*) { groupingClause }

5341

5342

5343 aggregateOperator ::= avg | count | max | median | min | stddev_pop
5344 | stddev_samp | sum | var_pop | var_samp5345 groupingClause ::= { **group by** groupId {, groupId}* }| **group except** groupId {, groupId}* }| **group all** conversionExpr }¹{ **having** havingCondition }

5346

5347

5348

5349

5350

5351

*Input Parameters*5352 aggregateOperator the keyword of the aggregate operator to invoke (e.g., **avg**, **count**, **max** ...)5353 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)

5354 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

5355 the following alternative grouping options:

5356 **group by** the Data Points are grouped by the values of the specified Identifiers (groupId). The Identifiers not specified are dropped in the result.5357 **group except** the Data Points are grouped by the values of the Identifiers not specified as groupId. The Identifiers specified as groupId are dropped in the result.5358 **group all** converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.5359 groupId Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).5360 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.5361 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:

5362 max(obs_value) < 1000

5363 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.:

5364 sum (ds group by id1 having count () >= 10)

5365

5366

5367

Examples of valid syntaxes

5368 avg (DS_1)

5369 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 *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 groupId :: 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 fo 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 Note: the example above can be rewritten equivalently in the following forms:
5458

DS_r := avg (DS_1 group except Id_2, Id_3)
DS_r := avg (DS_1#Me_1 group by Id_1)

5461
5462 *Example2:* DS_r := sum (DS_1 group by Id_1, Id_3) provided that At_1 is non viral, results in:
5463

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 *Example3:* DS_r := avg (DS_1) provided that At_1 is non viral results in:
5466

DS_r
Me_1
15.5

5467

5468 Example4: DS_r := DS_1 [aggr Me_2 := max (Me_1) , Me_3 := min (Me_1) group by Id_1]

5469

5470

5471

5472

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

5473

Analytic invocation

5474

Syntax

analyticOperator (firstOperand { , additionalOperand }* **over** (analyticClause))

5476

analyticOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**

5478

| **stddev_samp** | **sum** | **var_pop** | **var_samp**

5479

| **first_value** | **lag** | **last_value** | **lead** | **rank** | **ratio_to_report**

5480

analyticClause ::= { partitionClause } { orderClause } { windowClause }

5481

partitionClause ::= **partition** **by** identifier { , identifier }*

5482

orderClause ::= **order** **by** component { **asc** | **desc** } { , component { **asc** | **desc** } }*

5483

windowClause ::= { **data points** | **range** }¹ **between** limitClause **and** limitClause

5484

limitClause ::= { **num preceding** | **num following** | **current data point** | **unbounded preceding** | **unbounded following** }¹

5485

Parameters

5487

analyticOperator the keyword of the analytic operator to invoke (e.g., **avg**, **count**, **max** ...)

5488

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)

5489

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

5490

clause that specifies the analytic behaviour

5491

analyticClause 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.

5492

partitionClause 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.

5493

orderClause 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.

5494

5495

5496

5497

5498

5499

5500

5501

5502

5503

5504

5505

5506

5507

5508

clause that can specify either the lower or the upper boundaries of the sliding window. Each boundary is specified in relationship either to the whole partition or to the current data point under analysis by using the following keywords:

- **unbounded preceding** means that the sliding window starts at the first Data Point of the partition (it makes sense only as the first limit of the window)
 - **unbounded following** indicates that the sliding window ends at the last Data Point of the partition (it makes sense only as the second limit of the window)
 - **current data point** specifies that the window starts or ends at the current Data Point.
 - num **preceding** specifies either the number of **data points** to consider preceding the current data point in the order given by the orderClause (when **data points** is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the value of the current Data Point and the generic other Data Point (when **range** is specified in the windows clause).
 - num **following** specifies either the number of data points to consider following the current data point in the order given by the orderClause (when **data points** is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the values of the generic other Data Point and the current Data Point (when **range** is specified in the windows clause).

If the whole windowClause is omitted then the default is **data points between unbounded preceding and current data point**.

identifier	an Identifier Component of the input Data Set
component	a Component of the input Data Set
num	a scalar <i>number</i>

Examples of valid syntaxes

```
sum ( DS_1 over ( partition by Id_1 order by Id_2 ) )
sum ( DS_1 over ( order by Id_2 ) )
avg ( DS_1 over ( order by Id_1 data points between 1 preceding and 1 following ) )
DS_1 [ calc M1 := sum ( Me_1 over ( order by Id_1 ) ) ]
```

Semantics for scalar operations

The analytic operators cannot be applied to scalar values.

Input parameters type

firstOperand :: dataset

| component

additionalOperand :: see the type of the additional parameter (if any) of the invoked operator. The operators and their parameters are described in the following sections.

identifier :: name < identifier >

component :: name < component >

num :: **integer**

Result type

result :: dataset

component

Additional constraints

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

The analytic operations at component level can be invoked within the **calc** clause, both as part of a Join operator and the **calc** operator (see the parameter `calcExpr` of those operators).

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

5566 *Behaviour*

5567 The analytic Operator is applied as usual to all the Measures of the input Data Set (if invoked at Data Set level) or
5568 to the specified Component of the input Data Set (if invoked at Component level). In both cases, the operator
5569 calculates the desired output values for each Data Point of the input Data Set.

5570 The behaviour of the analytic operations can be procedurally described as follows:

- 5571 • The Data Points of the input Data Set are first partitioned (according to `partitionBy`) and then ordered
5572 (according to `orderBy`).
5573 • The operation is performed for each Data Point (named “current Data Point”) of the input Data Set. For each
5574 input Data Point, one output Data Point is returned, having the same values of the Identifiers. The analytic
5575 operator is applied to a “window” which includes a set of Data Points of the input Data Set and returns the
5576 values of the Measure(s) of the output Data Point.
5577 • If `windowClause` is not specified, then the set of Data Points which contribute to the analytic operation is
5578 the whole partition which the current Data Point belongs to
5579 • If `windowClause` is specified, then the set of Data Points is the one specified by `windowClause` (see
5580 `windowsClause` and `LimitClause` explained above).

5581 For the invocation at Data Set level, the resulting Data Set has the same Measures as the input Data Set
5582 `firstOperand`. For the invocation at Component level, the resulting Data Set has the Measures of the input Data
5583 Set plus the Measures explicitly calculated through the **calc** clause.

5584 For the invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level,
5585 the Attributes calculated within the `calc` clause are maintained in the result; for all the other Attributes that are
5586 defined as viral, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule
5587 section in the User Manual).

5588 As mentioned, the Analytic invocation at component level can be done within the **calc** clause, both as part of a
5589 Join operator and the **calc** operator (see the parameter `aggrCalc` of those operators), therefore, for a better
5590 comprehension fo the behaviour at Component level, see also those operators.

5591

5592 *Examples*

5593

5594 Given the Data Set DS_1:

5595

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

5596

5597 *Example1:*

5598

5599 DS_r := sum (DS_1 over (order by Id_1, Id_2, Id_3 data points between 1 preceding and 1 following))

5600

 results in:

5601

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 Example 1: DS_r := count (DS_1 group by Id_1) results in:

5645

DS_r	
Id_1	Int_var
2011	3
2012	2

5646

5647

Example 1: use of count in a **having** clause:

5648

DS_r := sum (DS_1 group by Id_1 having count() > 2) results in:

5649

5650

DS_r	
Id_1	Int_var
2011	3

5651

5652 Minimum value : min

5653 Aggregate syntax

min (dataset { groupingClause })

(in a Data Set expression)

5654

min (component) { groupingClause }

(in a Component expression within an aggr clause)

5655

5656 Analytic syntax

min (dataset over (analyticClause))

(in a Data Set expression)

5657

min (component over (analyticClause))

(in a Component expression within a calc clause)

5658

5659 Input parameters

dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

5660

5661 Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

5662

5663 Semantics for scalar operations

This operator cannot be applied to scalar values.

5664

5665

5666

5667

5668

5669

5670

5671

5672

5677 *Result type*
5678 result :: dataset
5679 | component

5681 *Additional constraints*

5682 None.

5684 *Behaviour*

5685 The operator returns the minimum value of the input values.
5686 For other details, see Aggregate and Analytic invocations.

5688 Examples

5689 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

5691
5692 Example 1: DS_r := min (DS_1 group by Id_1) results in:

DS_r	
Id_1	Me_1
2011	3
2012	2

5694 Maximum value : max

5695 Aggregate syntax

5696 max (dataset { groupingClause })

5697 **max** (component) { groupingClause }

(in a Data Set expression)

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

5699 *Analytic syntax*

5700 max (dataset over (analyticClause))

5701 max (component over (analyticClause))

(in a Data Set expression)

(in a Component expression within a **calc** clause)

5703 Input parameters

5705 component the operand Component

5706 groupingClause see Aggregate invocation

5707 analyticClause see Analytic invocation
5708

5709 *Examples of valid syntaxes*
5710 See Aggregate and Analytic invocations above, at the beginning of the section.

5712 *Semantics for scalar operations*
5713 This operator cannot be applied to scalar values.

5715 *Input parameters type*
5716 dataset :: dataset
5717 component :: component

5718
5719 *Result type*
5720 result :: dataset
5721 | component

5723 Additional constraints

5724 None.

5726 *Behaviour*

5727 The operator returns the maximum of the input values.
5728 For other details, see Aggregate and Analytic invocations.

5730 Examples

5731 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 Example 1: DS r := max (DS 1 group by Id 1) results in:

DS_r	
Id_1	Me_1
2011	7
2012	4

5736 Median value : median

5737	<i>Aggregate syntax</i>	
5738	median (dataset { <u>groupingClause</u> })	(in a Data Set expression)
5739	median (component) { groupingClause }	(in a Component expression within an aogr clause)

5740	
5741	<i>Analytic syntax</i>
5742	median (dataset over (<u>partitionClause</u>)) <i>(in a Data Set expression)</i>
5743	median (component over (<u>partitionClause</u>)) <i>(in a Component expression within a calc clause)</i>

5784	<i>Analytic syntax</i>	
5785	sum (dataset over (<u>analyticClause</u>))	<i>(in a Data Set expression)</i>
5786	sum (component over (<u>analyticClause</u>))	<i>(in a Component expression within a calc clause)</i>

5788	<i>Input parameters</i>	
5789	dataset	the operand Data Set
5790	component	the operand Component
5791	<u>groupingClause</u>	see Aggregate invocation
5792	analyticClause	see Analytic invocation

5794 *Examples of valid syntaxes*
5795 See Aggregate and Analytic invocations above, at the beginning of the section.

5797 *Semantics for scalar operations*

5798 This operator cannot be applied to scalar values.

5800 *Input parameters type*

5801 dataset :: dataset { measure<number> _+ }
5802 component :: component<number>

5804 *Result type*

```
5805    result :: dataset { measure<number> _+ }
5806          | component<number>
```

5808 Additional constraints

5809 None.

5811 *Behaviour*

5812 The operator returns the sum of the input values

For other details, see Aggregate and Analytic invocations.

5815 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

5819 Example 1: DS_r := sum (DS_1 group by Id_1) results in:

DS_r	
Id_1	Me_1
2011	15
2012	6

5821

5822	Average value :	avg																												
5823	<i>Aggregate syntax</i>																													
5824	avg (dataset { <u>groupingClause</u> })	(in a Data Set expression)																												
5825	avg (component) { <u>groupingClause</u> }	(in a Component expression within an aggr clause)																												
5826																														
5827	<i>Analytic syntax</i>																													
5828	avg (dataset over (<u>analyticClause</u>))	(in a Data Set expression)																												
5829	avg (component over (<u>analyticClause</u>))	(in a Component expression within a calc clause)																												
5830																														
5831	<i>Input parameters</i>																													
5832	dataset	the operand Data Set																												
5833	component	the operand Component																												
5834	<u>groupingClause</u>	see Aggregate invocation																												
5835	<u>analyticClause</u>	see Analytic invocation																												
5836																														
5837	<i>Examples of valid syntaxes</i>																													
5838	See Aggregate and Analytic invocations above, at the beginning of the section.																													
5839																														
5840	<i>Semantics for scalar operations</i>																													
5841	This operator cannot be applied to scalar values.																													
5842																														
5843	<i>Input parameters type</i>																													
5844	dataset ::	dataset {measure<number> _+}																												
5845	component ::	component<number>																												
5846																														
5847	<i>Result type</i>																													
5848	result ::	dataset { measure<number> _+ }																												
5849		component<number>																												
5850	<i>Additional constraints</i>																													
5851	None.																													
5852																														
5853	<i>Behaviour</i>																													
5854	The operator returns the average of the input values.																													
5855	For other details, see Aggregate and Analytic invocations.																													
5856																														
5857	<i>Examples</i>																													
5858	Given the Data Set DS_1:																													
5859																														
<table border="1"> <thead> <tr><th colspan="4">DS_1</th></tr> <tr><th>Id_1</th><th>Id_2</th><th>Id_3</th><th>Me_1</th></tr> </thead> <tbody> <tr><td>2011</td><td>A</td><td>XX</td><td>3</td></tr> <tr><td>2011</td><td>A</td><td>YY</td><td>5</td></tr> <tr><td>2011</td><td>B</td><td>YY</td><td>7</td></tr> <tr><td>2012</td><td>A</td><td>XX</td><td>2</td></tr> <tr><td>2012</td><td>B</td><td>YY</td><td>4</td></tr> </tbody> </table>			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
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																														
<table border="1"> <thead> <tr><th colspan="2">DS_r</th></tr> <tr><th>Id_1</th><th>Me_1</th></tr> </thead> <tbody> <tr><td>2011</td><td>5</td></tr> </tbody> </table>			DS_r		Id_1	Me_1	2011	5																						
DS_r																														
Id_1	Me_1																													
2011	5																													

2012	3
------	---

5863

Population standard deviation : **stddev_pop**

Aggregate syntax

stddev_pop (dataset { groupingClause })

(in a Data Set expression)

stddev_pop (component){ groupingClause }

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

5868

Analytic syntax

stddev_pop (dataset **over** (analyticClause))

(in a Data Set expression)

stddev_pop (component **over** (analyticClause))

(in a Component expression within a **calc** clause)

5872

Input parameters

dataset the operand Data Set

component the operand Component

groupingClause see Aggregate invocation

analyticClause see Analytic invocation

5878

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

5881

Semantics for scalar operations

This operator cannot be applied to scalar values.

5884

Input parameters type

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

component :: component<number>

5888

Result type

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

| component<number>

5892

Additional constraints

None.

5895

Behaviour

The operator returns the “population standard deviation” of the input values.

For other details, see Aggregate and Analytic invocations.

5899

Examples

5901

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

5904

Example 1: DS_r := stddev_pop (DS_1 group by Id_1) results in:

5905

5906

DS_r	
Id_1	Me_1
2011	1.633
2012	1

5907

5908 Sample standard deviation : **stddev_samp***Aggregate syntax*

5910 **stddev_samp** (dataset { groupingClause })
 5911 **stddev_samp** (component) { groupingClause }

*(in a Data Set expression)**(in a Component expr. within an aggr clause)*

5912

Analytic syntax

5913 **stddev_samp** (dataset over (analyticClause))
 5914 **stddev_samp** (component over (analyticClause))

*(in a Data Set expression)**(in a Component expr. within a calc clause)*

5916

Input parameters

5917 dataset the operand Data Set
 5918 component the operand Component
 5919 groupingClause see Aggregate invocation
 5920 analyticClause see Analytic invocation

5922

Semantics for scalar operations

5923 This operator cannot be applied to scalar values.

5925

Examples of valid syntaxes

5926 See Aggregate and Analytic invocations above, at the beginning of the section.

5928

Input parameters type

5929 dataset :: dataset { measure<number> _+ }
 5930 component :: component<number>

5932

Result type

5933 result :: dataset { measure<number> _+ }
 5934 | component<number>

5936

Additional constraints

5937 None.

5939

Behaviour

5940 The operator returns the “sample standard deviation” of the input values.

5941 For other details, see Aggregate and Analytic invocations.

5943

Examples

5944 Given the Data Set DS_1:

5946

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 Example 1: DS_r := stddev_samp (DS_1 group by Id_1) results in:

5949

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

6029 *Examples*

6030

6031 Given the Data Set DS_1

6032

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

6033

6034 Example 1: DS_r := var_samp (DS_1 group by Id_1) results in:

6035

DS_r	
Id_1	Me_1
2011	4
2012	2

6036

6037 First value : **first_value**

6038 *Syntax*

6039 **first_value** (dataset **over** (analyticClause)) *(in a Data Set expression)*

6040 **first_value** (component **over** (analyticClause)) *(in a Component expression within a calc clause)*

6041

6042 *Input parameters*

6043 dataset the operand Data Set

6044 component the operand Component

6045 analyticClause see Analytic invocation

6046

6047 *Examples of valid syntaxes*

6048 See Analytic invocation above, at the beginning of the section.

6049

6050 *Semantics for scalar operations*

6051 This operator cannot be applied to scalar values.

6052

6053 *Input parameters type*

6054 dataset :: dataset { measure<scalar> _+ }

6055 component :: component<scalar>

6056

6057 *Result type*

6058 result :: dataset
6059 | component<scalar>

6060

6061 *Additional constraints*

6062 The Aggregate invocation is not allowed.

6063

6064 *Behaviour*

6065 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.

6066

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 :

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 DS_r := first_value (DS_1 over (partition by Id_1, Id_2 order by Id_3 data points between 1 preceding and
6077 1 following))

6080 results in:

DS_r				
Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	3	1
A	XX	1995	4	5
A	XX	1996	6	5
A	YY	1993	5	3
A	YY	1994	5	2
A	YY	1995	2	2
A	YY	1996	2	2

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

6089 *Input parameters*

6090 dataset the operand Data Set
6091 component the operand Component
6092 analyticClause see Analytic invocation

6093

6094	<i>Examples of valid syntaxes</i>				
6095	See Analytic invocation above, at the beginning of the section.				
6096					
6097	<i>Semantics for scalar operations</i>				
6098	This operator cannot be applied to scalar values.				
6099					
6100	<i>Input parameters type</i>				
6101	dataset :: dataset {measure<scalar> _+}				
6102	component :: component<scalar>				
6103					
6104	<i>Result type</i>				
6105	result :: dataset				
6106	component<scalar>				
6107					
6108	<i>Additional constraints</i>				
6109	The Aggregate invocation is not allowed.				
6110					
6111	<i>Behaviour</i>				
6112	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.				
6113					
6114	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.				
6115					
6116	When invoked at Component level, it returns the last value of the specified Component.				
6117	For other details, see Analytic invocation.				
6118					
6119	<i>Examples</i>				
6120					
6121	Given the Data Set DS_1:				
6122					
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
6123					
6124					
6125	<i>Example 1:</i>				
6126					
6127	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))				
6128					
6129					
6130	results in:				
6131					
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 defaultValue :: 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 *Examples*

6176 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

6179

6180

6181 Example 1: DS_r := lag (DS_1 , 1 over (partition by Id_1 , Id_2 order by Id_3)) results in:

6182

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

6183

6184 lead : lead

6185 *Syntax*

6186

6187 *in a Data Set expression:*6188 **lead** (dataset , {offset {, defaultValue} } **over** ({ partitionClause } orderClause))

6189

6190 *in a Component expression within a calc clause:*6191 **lead** (component , {offset {, defaultValue} } **over** ({ partitionClause } orderClause))

6192

6193 *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

6200

6201 *Examples of valid syntaxes*

6202 See Analytic invocation above, at the beginning of the section.

6203

6204 *Semantics for scalar operations*

6205 This operator cannot be applied to scalar values.

6206

6207 *Input parameters type*

dataset ::	dataset
component ::	component

6210 offset :: integer [value > 0]
6211 default value :: scalar

6212 *Result type*

6214 result :: dataset
6215 | component

6216 *Additional constraints*

6218 The Aggregate invocation is not allowed.

6219 The windowClause of the Analytic invocation syntax is not allowed.

6220

6221 *Behaviour*

6222 In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data
6223 Point at the specified physical offset beyond the current Data Point.

6224 If defaultValue is not specified, then the value returned when the offset goes outside the partition is NULL.

6225 For other details, see Analytic invocation.

6226

6227 *Examples*

6228 Given the Data Set DS_1

6229

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

6231 Example 1: DS_r := lead (DS_1 , 1 over (partition by Id_1 , Id_2 order by Id_3)) results in:

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 specified in the orderClause. If some Data Points are in the same order according to the specified orderClause, the same order number (rank) is assigned and a gap appears in the sequence of the assigned ranks (for example, if four Data Points have the same rank 5, the following assigned rank would be 9).

6263 For other details, see Analytic invocation.

6264

6265 *Examples*

6266 Given the Data Set DS_1:

6267

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

6268

6269 *Example 1:*

6270

DS_r := DS_1 [calc Me2 := rank (over (partition by Id_1 , Id_2 order by Me_1))] results in:

6271

DS_r				
Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	2

6272

6273

6274

6275

6276

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
 6308 partition which the current Data Point belongs to.

6309 For other details, see Analytic invocation.

6310

6311 *Examples*

6312 Given the Data Set DS_1:

DS_1				
Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	3
A	XX	2002	7	5
A	XX	2003	6	1
A	YY	2000	12	0

A	YY	2001	8	8
A	YY	2002	6	5
A	YY	2003	14	-3

6314

6315

6316

6317

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

6318

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 associated (in positional order) to the conditioning Value Domains defined in dpr. If dpr is defined on Variables then listComp is the list of Components of op to be associated (in positional order) to the conditioning Variables defined in dpr (for documentation purposes).

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 each Rule in dpr that evaluates to FALSE on that Data Point. The resulting Data Set has the Measures of op.

6335 **all** the resulting Data Set contains a data point for each Data Point of op and each Rule in dpr. The resulting Data Set has the *boolean* Measure *bool_var*.

6336 **all_measures** the resulting Data Set contains a Data Point for each Data Point of op and each Rule in dpr. The resulting dataset has the Measures of op and the *boolean* Measure *bool_var*.

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.

6349

6350 *Input parameters type:*

```
6351 op ::      dataset  
6352 dpr ::      name < datapoint >  
6353 comp ::     name < component >
```

6355 *Result type:*

```
6356 result ::    dataset
```

6357

6358 *Additional constraints*

6359 If dpr is defined on Value Domains then it is mandatory to specify listComp. The Components specified in listComp must belong to the operand op and be defined on the Value Domains specified in the signature of dpr.
6360 If dpr is defined on Variables then the Components specified in the signature of dpr must belong to the operand op.

6363 If dpr is defined on Variables and listComp is specified then the Components specified in listComp are the same, in the same order, as those specified in op (they are provided for documentation purposes).

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
- 6370 Point (it contains at least the Rule name specified in dpr ⁸)
- 6371 • if the output parameter is **invalid**: the original Measures of op (no *boolean* measure)
- 6372 • if the output parameter is **all**: the *boolean* Measure *bool_var* whose value is the result of the evaluation
- 6373 of a rule on a Data Point (TRUE, FALSE or NULL).
- 6374 • if the output parameter is **all_measures**: the original measures of op and the *boolean* Measure *bool_var*
- 6375 whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- 6376 • the Measure errorcode that contains the errorcode specified in the rule
- 6377 • the Measure errorlevel that contains the errorlevel specified in the rule

6379 A Data Point of op can produce several Data Points in the resulting Data Set, each of them with a different value
 6380 of ruleid. If output is **invalid** then the resulting Data Set contains a Data Point for each Data Point of op and each
 6381 rule of dpr that evaluates to FALSE. If output is **all** or **all_measures** then the resulting Data Set contains a Data
 6382 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 Given the Data Set DS_1:

DS_1			
Id_1	Id_2	Id_3	Me_1
2011	I	CREDIT	10
2011	I	DEBIT	-2
2012	I	CREDIT	10
2012	I	DEBIT	2

6391 DS_r := check_datapoint (DS_1, dpr1) results in:

DS_r						
Id_1	Id_2	Id_3	ruleid	obs_value	errorcode	errorlevel
2011	I	DEBIT	dpr1_2	-2	Bad debit	

6394 DS_r := check_datapoint (DS_1, dpr1 all) results in:

DS_r						
Id_1	Id_2	Id_3	ruleid	bool_var	errorcode	errorlevel
2011	I	CREDIT	dpr1_1	true		
2011	I	CREDIT	dpr1_2	true		
2011	I	DEBIT	dpr1_1	true		

⁸ The content of ruleid maybe personalised in the implementation

2011	I	DEBIT	dpr1_2	false	Bad debit	
2012	I	CREDIT	dpr1_1	true		
2012	I	CREDIT	dpr1_2	true		
2012	I	DEBIT	dpr1_1	true		
2012	I	DEBIT	dpr1_2	true		

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 <u>op</u>	the Data Set to be checked
6410 <u>hr</u>	the hierarchical Ruleset to be used
6411 <u>condComp</u>	condComp is a Component of op to be associated (in positional order) to the conditioning Value Domains or Variables defined in hr (if any).
6413 <u>ruleComp</u>	ruleComp is the Identifier Component of op to be associated to the rule Value Domain or Variable defined in hr.
6415 <u>mode</u>	this parameter specifies how to treat the possible missing Data Points corresponding to the Code Items in the left and right sides of the rules and which Data Points are produced in output. The meaning of the possible values of the parameter is explained below.
6418 <u>input</u>	this parameter specifies the source of the values used as input of the comparisons. The meaning of the possible values of the parameter is explained below.
6420 <u>output</u>	this parameter specifies the structure and the content of the resulting dataset. The meaning of 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 <u>op</u> ::	dataset { measure<number> _ }
6429 <u>hr</u> ::	name < hierarchical >
6430 <u>condComp</u> ::	name < component >
6431 <u>ruleComp</u> ::	name < identifier >

6432

6433 *Result type*

6434 <u>result</u> ::	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 rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand

6438

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

6449 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

6459 First, the parameter input is used to determine the source of the Data Points used as input of the
6460 check_hierarchy. The possible options of the parameter input and the corresponding behaviours are the
following:

6461

6462 dataset this option addresses the case where all the input Data Points of all the Rules of the Ruleset are
6463 expected to be taken from the input Data Set (the operand op).

6464 For each Rule of the Ruleset and for each item on the left and right sides of the Rule, the
operator takes the input Data Points exclusively from the operand op.

6465

6466 dataset_priority this option addresses the case where the input Data Points of all the Rules of the Ruleset are
6467 preferably taken from the input Data Set (the operand op), however if a valid Measure value
6468 for an expected Data Point is not found in op, the attempt is made to take it from the computed
6469 output of a (possible) other Rule.

6470 For each Rule of the Ruleset and for each item on the left and right sides of the Rule:

- 6471 • if the item is not defined as the result (left side) of another Rule that applies the Code Item
relation “is equal to” (=), the current Rule takes the input Data Points from the operand
op.
- 6472 • if the item is defined as result of another Rule R that applies the Code Item relation “is
equal to” (=), then:
 - 6473 ○ if an expected input Data Point exists in op and its Measure is not NULL, then the
current Rule takes such Data Point from op;
 - 6474 ○ if an expected input Data Point does not exist in op or its measure is NULL, then
the current Rule takes the Data Point (if any) that has the same Identifiers’ values
from the computed output of the other Rule R;

6475

6476 if the parameter input is not specified then it is assumed to be dataset.

6477

6478 Then the parameter mode is considered, to determine the behaviour for missing Data Points and for the Data
6479 Points to be produced in the output. The possible options of the parameter mode and the corresponding
behaviours are the following:

6480

6481 non_null the result Data Point is produced when all the items involved in the comparison exist and have
6482 not NULL Measure value (i.e., when no Data Point corresponding to the Code Items of the left
6483 and right sides of the rule is missing or has NULL Measure value); under this option, in
6484 evaluating the comparison, the possible missing Data Points corresponding to the Code Items
6485 of the left and right sides of the rule are considered existing and having a NULL Measure value;
6486

6487

6488 non_zero the result Data Point is produced when at least one of the items involved in the comparison
6489 exist and have Measure not equal to 0 (zero); the possible missing Data Points corresponding
6490 to the Code Items of the left and right sides of the rule are considered existing and having a
6491 Measure value equal to 0;

6492

6493 partial_null the result Data Point is produced if at least one Data Point corresponding to the Code Items of
6494 the left and right sides of the rule is found (whichever is its Measure value); the possible
6495 missing Data Points corresponding to the Code Items of the left and right sides of the rule are
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 the left and right sides of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);
 6498
 6499
 6500
 6501 always_null the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to NULL;
 6502
 6503
 6504 always_zero the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);
 6505
 6506
 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
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
Non_zero	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)
Partial_null	NULL	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Partial_zero	Zero	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

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:

- all the Identifier Components of op
- the additional Identifier Component ruleid, whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in hr⁹)
- if the output parameter is all: the boolean Measure bool_var whose values are the result of the evaluation of the Rules (TRUE, FALSE or NULL).
- if the output parameter is invalid: the original Measure of op, whose values are taken from the Measure values of the Data Points of the left side of the Rule
- if the output parameter is all_measures: the boolean Measure bool_var, whose value is the result of the evaluation of a Rule on a Data Point (TRUE, FALSE or NULL), and the original Measure of op, whose values are taken from the Measure values of the Data Points of the left side of the Rule
- the Measure imbalance, which contains the difference between the Measure values of the Data Points on the left side of the Rule and the Measure values of the corresponding calculated Data Points on the right side of the Rule
- the Measure errorcode, which contains the errorcode value specified in the Rule
- the Measure errorlevel, which contains the errorlevel value specified in the Rule

6544 Note that a generic Data Point of op can produce several Data Points in the resulting Data Set, one for each Rule
6545 in which the Data Point appears as the left member of the comparison.

6548 Examples

6549 See also the examples in **define hierarchical ruleset**.

6551 Given the following hierarchical ruleset:

```
6553 define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
6554     R010 : A = J + K + L           errorlevel 5
6555     ; R020 : B = M + N + O           errorlevel 5
6556     ; R030 : C = P + Q      errorcode XX  errorlevel 5
6557     ; R040 : D = R + S           errorlevel 1
6558     ; R060 : F = Y + W + Z           errorlevel 7
6559     ; R070 : G = B + C
6560     ; R080 : H = D + E           errorlevel 0
6561     ; R090 : I = D + G      errorcode YY  errorlevel 0
6562     ; R100 : M >= N           errorlevel 5
6563     ; R110 : M <= G           errorlevel 5
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
6566 order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is
6567 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

⁹ 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 Example 1: DS_r := check_hierarchy (DS_1, HR_1 rule Id_2 partial_null all) results in:

6572

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:

- the Identifier Components of op
- a *boolean* Measure named **bool_var** that contains the result of the evaluation of the *boolean* dataset op
- the Measure imbalance that contains the specified imbalance
- the Measure errorcode that contains the specified errorcode
- the Measure errorlevel that contains the specified errorlevel

6612 If output is **all** then all data points are returned. If output is **invalid** then only the Data Points where **bool_var** is FALSE are returned.

6614

6615 *Examples*

6616

6617 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

2015	D	75
------	---	----

6619

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

6624

VTL-ML - Conditional operators

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 > 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,
6641 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 <boolean> _ }
6648 | component<Boolean>
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<
6660 | scalar

6661

6662 *Additional constraints*

- The operands thenOperand and elseOperand must be of the same scalar type.
- If the operation is at scalar level, thenOperand and elseOperand are scalar then condition must be scalar too (a *boolean* scalar).
- If the operation is at Component level, at least one of thenOperand and elseOperand is a Component (the other one can be scalar) and condition must be a Component too (a *boolean* Component); thenOperand, elseOperand and the other Components referenced in condition must belong to the same Data Set.
- If the operation is at Data Set level, at least one of thenOperand and elseOperand is a Data Set (the other one can be scalar) and condition must be a Data Set too (having a unique *boolean* Measure) and must have the same Identifiers as thenOperand or/and ElseOperand
 - If thenOperand and elseOperand are both Data Sets then they must have the same Components in the same roles
 - If one of thenOperand and elseOperand is a Data Set and the other one is a scalar, the Measures of the operand Data Set must be all of the same scalar type as the scalar operand.

6673

6674

6675

6676

6677

6678

6679 ***Behaviour***
 6680 For operations at Component level, the operation is applied for each Data Point of the unique input Data Set, the
 6681 **if-then-else** operator returns the value from the thenOperand Component when condition evaluates to **true**,
 6682 otherwise it returns the value from the elseOperand Component. If one of the operands thenOperand or
 6683 elseOperand is scalar, such a scalar value can be returned depending on the outcome of the condition.
 6684 For operations at Data Set level, the **if-then-else** operator returns the Data Point from thenOperand when the
 6685 Data Point of condition having the same Identifiers' values evaluates to **true**, and returns the Data Point from
 6686 elseOperand otherwise. If one of the operands thenOperand or elseOperand is scalar, such a scalar value can
 6687 be returned (depending on the outcome of the condition) and in this case it feeds the values of all the Measures
 6688 of the result Data Point.
 6689 The behaviour for two Data Sets can be procedurally explained as follows. First the condition Data Set is
 6690 evaluated, then its true Data Points are inner joined with thenOperand and its false Data Points are inner
 6691 joined with elseOperand, finally the union is made of these two partial results (the condition ensures that there
 6692 cannot be conflicts in the union).
 6693

Examples

6694 Example 1: given the operand Data Sets DS_cond, DS_1, DS_2:
 6695
 6696

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

6698

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

6699

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

6700

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 : nvl

Syntax

nvl(op1 , op2)

Input parameters

6708 op1 the first operand
6709 op2 the second operand

Examples of valid syntaxes

6711 Examples of varfa syntaxes
6712 nvl (ds1#m1, 0)
6713

`nvl (5, 0)` returns 5
`nvl (null, 0)` returns 0

Input Parameters type

```
6719 Input Parameters type  
6720 op1 :: dataset  
6721 | component<scalar>  
6722 | scalar
```

```
6722 | scalar  
6723  
6724 op2 :: dataset  
6725 | component  
6726 | <scalar>
```

Results

6728 *Result type*
6729 result :: dataset
6730 | component
6731 | scalar

Additional constraints

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.

Behaviour

The operator `nvl` returns the value from `op2` when the value from `op1` is null, otherwise it returns the value from `op1`.

The operator has the typical behaviour of the operators applicable on two scalar values or Data Sets or Data Set Components.

Also the following statement gives the same result: if isnull (op1) then op2 else op1

Examples

6747 Example 1: Given the input Data Set DS_1

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	NULL
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	NULL

6749

6750

6751

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

6752 VTL-ML - Clause operators

6753 Filtering Data Points : **filter**

6754

6755 *Syntax*

6756 op [**filter** filterCondition]

6757

6758 *Input parameters*

6759 op the operand

6760 filterCondition the filter condition

6761

6762 *Examples of valid syntaxes*

6763 DS_1 [filter Me_3 > 0]

6764 DS_1 [filter Me_3 + Me_2 <= 0]

6765

6766 *Semantics for scalar operations*

6767 This operator cannot be applied to scalar values.

6768

6769 *Input parameters type:*

6770 op :: dataset

6771 filterCondition :: component<boolean>

6772

6773 *Result type:*

6774 result :: dataset

6775

6776 *Additional constraints:*

6777 None.

6778

6779 *Behaviour*

6780 The operator takes as input a Data Set (op) and a *boolean* Component expression (filterCondition) and filters the
6781 input Data Points according to the evaluation of the condition. When the expression is TRUE the Data Point is
6782 kept in the result, otherwise it is not kept (in other words, it filters out the Data Points of the operand Data Set
6783 for which filterCondition condition evaluates to FALSE or NULL).

6784

6785 *Examples*

6786

6787 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:

6790

DS_r				
Id_1	Id_2	Id_3	Me_1	At_1
1	A	YY	2	F

1	A	XX	2	E
1	A	YY	2	F
1	B	YY	1	F

6791 Calculation of a Component : calc

6792

6793 Syntax

6794 op [**calc** { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }*]

6795

6796 calcRole ::= **identifier** | **measure** | **attribute** | **viral attribute**

6797

6798 Input parameters

6799 op the operand

6800 calcRole the role to be assigned to a Component to be calculated

6801 calcComp the name of a Component to be calculated

6802 calcExpr expression at component level, having only Components of the input Data Sets as operands,
6803 used to calculate a Component

6804

6805 Examples of valid syntaxes

6806 DS_1 [calc Me_3 := Me_1 + Me_2]

6807

6808 Semantics for scalar operations

6809 This operator cannot be applied to scalar values.

6810

6811 Input parameters type:

6812 op :: dataset

6813 calcComp :: name < component >

6814 calcExpr :: component<scalar>

6815

6816 Result type:

6817 result :: dataset

6818

6819 Additional constraints

6820 The calcComp parameter cannot be the name of an Identifier component.

6821 All the components used in calcComp must belong to the operand Data Set op.

6822

6823 Behaviour

6824 The operator calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at
6825 Component level. Each Component is calculated through an independent sub-expression. It is possible to specify
6826 the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**, therefore the calc
6827 clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling
6828 the virality of the calculated Attributes (for the attribute propagation rule see the User Manual). When the role is
6829 omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if
6830 the component does not exist in the operand Data Set then its role is Measure.

6831 The calcExpr sub-expressions are independent one another, they can only reference Components of the input
6832 Data Set and cannot use Components generated, for example, by other calcExpr. If the calculated Component is a
6833 new Component, it is added to the output Data Set. If the Calculated component is a Measure or an Attribute that
6834 already exists in the input Data Set, the calculated values overwrite the original values. If the calculated
6835 Component is an Identifier that already exists in the input Data Set, an exception is raised because overwriting
6836 an Identifier Component is forbidden for preserving the functional behaviour. Analytic invocations can be used
6837 in the **calc** clause.

6838

6839

6840 Examples

6841

6842

6843 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

6844

6845 Example1: DS_r := DS_1 [calc Me_1:= Me_1 * 2] results in:

6846

DS_r			
Id_1	Id_2	Id_3	Me_1
1	A	CA	40
1	B	CA	4
2	A	CA	4

6847

6848 Example2: DS_r := DS_1 [calc attribute At_1:= "EP"] results in:

6849

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

6850

6851 Aggregation : **aggr**

6852

6853 *Syntax*6854 op [**aggr** aggrClause { groupingClause }]

6855

6856 aggrClause ::= { aggrRole } aggrComp := aggrExpr
6857 { , { aggrRole } aggrComp := aggrExpr }*

6858

6859 groupingClause ::= { **group** by groupingId {, groupingId }*
6860 | **group except** groupingId {, groupingId }*
6861 | **group all** conversionExpr }
6862 { **having** havingCondition }

6863

6864 aggrRole ::= **measure** | **attribute** | **viral attribute**

6865

6866

6867 *Input Parameters*

6868

op the operand

6869

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

6870

aggrRole the role of the aggregated Component to be calculated

6871

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

6872

6873

6874

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 $DS_1 [\text{aggr } Me_1 := \text{count}() \text{ group by } Id_1]$
 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 $\text{max(obs_value)} < 1000$
 6900 instead the condition $\text{obs_value} < 1000$ 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 $\text{sum}(DS_1 \text{ group by } id1 \text{ having count } () \geq 10)$
 6904
 6905 *Examples of valid syntaxes*
 6906 $DS_1 [\text{aggr } M1 := \text{min} (Me_1) \text{ group by } Id_1, Id_2]$
 6907 $DS_1 [\text{aggr } M1 := \text{min} (Me_1) \text{ group except } Id_1, Id_2]$
 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

6930 *Behaviour*

6931 The operator **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of
6932 sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is
6933 possible to specify the role of the calculated Component among **measure attribute**, or **viral attribute**. The
6934 substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the
6935 User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data
6936 Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.
6937 The aggrExpr sub-expressions are independent of one another, they can only reference Components of the input
6938 Data Set and cannot use Components generated, for example, by other aggrExpr sub-expressions. The **aggr**
6939 computed Measures and Attributes are the only Measures and Attributes returned in the output Data Set (plus
6940 the possible viral Attributes). The sub-expressions must contain only Aggregate operators, which are able to
6941 compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are
6942 specified through the groupingClause, which allows the following alternative options.

6943 **group by** the Data Points are grouped by the values of the specified Identifiers. The Identifiers not
6944 specified are dropped in the result.

6945 **group except** the Data Points are grouped by the values of the Identifiers not specified in the clause. The
6946 specified Identifiers are dropped in the result.

6947 **group all** converts an Identifier Component using conversionExpr and keeps all the other Identifiers.

6948
6949 The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the
6950 single groups (for example the minimum number of Data Points in the group).

6951 If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause
6952 returns a Data Set that contains a single Data Point and has no Identifiers.

6953 The Attributes calculated through the **aggr** clauses are maintained in the result. For all the other Attributes that
6954 are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation
6955 Rule section in the User Manual).

6956
6957 *Examples*

6958
6959 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

6961

6962 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

6964

6965 Example2: DS_r := DS_1 [aggr Me_3:= min(Me_1) group except Id_3] results in:

DS_r		

6966

Id_1	Id_2	Me_3
1	A	0
1	B	3
2	A	2

6967
6968
6969
6970
6971
6972

Example3: DS_r := DS_1 [aggr Me_1:= sum(Me_1), Me_2 := max(Me_1)
group by Id_1 , Id_2
having avg (Me_1) > 2]

results in:

DS_r			
Id_1	Id_2	Me_1	Me_2
1	B	8	5
2	A	9	7

6973

6974 Maintaining Components: **keep**

6975

6976 *Syntax*

6977 op [**keep** comp {, comp }*]

6978

6979 *Input parameters*

6980

op the operand

6981

comp a component to keep

6982

6983 *Examples of valid syntaxes*

6984

DS_1 [keep Me_2, Me_3]

6985

6986 *Semantics for scalar operations*

6987

This operator cannot be applied to scalar values.

6988

6989 *Input parameters type:*

6990

op :: dataset

6991

comp :: name < component >

6992

6993 *Result type:*

6994

result :: dataset

6995

6996 *Additional constraints:*

6997

All the Components comp must belong to the input Data Set op.

6998

The Components comp cannot be Identifiers in op.

6999

7000 *Behaviour*

7001

The operator takes as input a Data Set (op) and some Component names of such a Data Set (comp). These Components can be Measures or Attributes of op but not Identifiers. The operator maintains the specified Components, drops all the other dependent Components of the Data Set (Measures and Attributes) and maintains the independent Components (Identifiers) unchanged. This operation corresponds to a projection in the usual relational join semantics (specifying which columns will be projected in among Measures and Attributes).

7007

7008

7009

7010 *Examples*

7011

Given the Data Set DS_1:

DS_1					
Id_1	Id_2	Id_3	Me_1	Me_2	At_1
2010	A	XX	20	36	E
2010	A	YY	4	9	F
2010	B	XX	9	10	F

7012
7013
7014

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

7015

7016 Removal of Components: drop

7017

7018 Syntax

7019 op [drop comp { , comp }*]

7020

7021 Input parameters

7022

7023 op the operand
comp a Component to drop

7024

7025 Examples of valid syntaxes

7026

DS_1 [drop Me_2, Me_3]

7027

7028 Semantics for scalar operations

7029

This operator cannot be applied to scalar values.

7030

7031 Input parameters type:

7032

op :: dataset
comp :: name < component >

7034

7035 Result type:

7036

result :: dataset

7037

7038 Additional constraints:

7039

All the Components comp must belong to the input Data Set op.

7040

The Components comp cannot be Identifiers in op.

7041

7042 Behaviour

7043

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).

7047

7048 Examples

7049

Given the Data Set DS_1:

7051

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

7052
7053 *Example1:* DS_r := DS_1 [drop At_1]
7054

results in:

DS_r			
Id_1	Id_2	Id_3	Me_1
2010	A	XX	20
2010	A	YY	4
2010	B	XX	9

7055 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 ::

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
7095
7096
7097

Examples

Given the Data Set DS_1:

DS_1				
Id_1	Id_2	Id_3	Me_1	At_1
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

7098
7099
7100

Example1: DS_r := DS_1 [rename Me_1 to Me_2, At_1 to At_2] results in:

DS_r				
Id_1	Id_2	Id_3	Me_2	At_2
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

7101 Pivoting : pivot

7102

Syntax

op [pivot identifier , measure]

7104

Input parameters

7105

op the operand

7106

identifier the Identifier Component of op to pivot

7107

measure the Measure Component of op to pivot

7108

7109

7110

7111

Examples of valid syntaxes

DS_1 [pivot Id_2, Me_1]

7112

Semantics for scalar operations

7113

This operator cannot be applied to scalar values.

7114

7115

Input Parameters type

7116

op :: dataset

7117

identifier :: name < identifier >

7118

measure :: name < measure >

7119

Result type

7120

result :: dataset

7121

7122

Additional constraints

7123

The Measures created by the operator according to the behaviour described below must be defined on the same

7124

Value Domain as the input Measure.

7125

7126

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
1. It creates a virtual Data Set VDS as a copy of op
 2. It drops the Identifier Component identifier and all the Measure Components from VDS.
 3. It groups VDS by the values of the remaining Identifiers.
 4. For each distinct value of identifier in op, it adds a corresponding measure to VDS, named as the value of identifier. These Measures are initialized with the NULL value.
 5. For each Data Point of op, it finds the Data Point of VDS having the same values as for the common Identifiers and assigns the value of measure (taken from the current Data Point of op) to the Measure of VDS having the same name as the value of identifier (taken from the Data Point of op).

7142 The result of the last step is the output of the operation.

7143 Note that **pivot** may create Measures whose names are non-regular (i.e. they may contain special characters, reserved keywords, etc.) according to the rules about the artefact names described in the User Manual (see the section “The artefact names” in the chapter “VTL Transformations”). As said in the User Manual, those names must be quoted to be referenced within an expression.

7149 Examples

7150 Given the Data Set DS_1:

DS_1			
Id_1	Id_2	Me_1	At_1
1	A	5	E
1	B	2	F
1	C	7	F
2	A	3	E
2	B	4	E
2	C	9	F

7154 Example1: DS_r := Ds_1 [pivot Id_2, Me_1] results in:

DS_r			
Id_1	A	B	C
1	5	2	7
2	3	4	9

7157

7158 Unpivoting : unpivot

7159 Syntax

7160 op [unpivot identifier , measure]

7162 Input parameters

7163 op the dataset operand
7164 identifier the Identifier Component to be created
7165 measure the Measure Component to be created

7166 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 described as follows:

7246

7247

1. It creates a virtual Data Set VDS as a copy of op
2. It maintains the Data Points of VDS for which identifier = value (for all the specified identifier) and eliminates all the Data Points for which identifier <> value (even for only one specified identifier)
3. It projects out ("drops", in VTL terms) all the identifier(s)

7252

The result of the last step is the output of the operation.

7254

The resulting Data Set has the Identifier Components that are not specified as identifier(s) and has the same Measure and Attribute Components of the input Data Set.

7257

7258

The result Data Set does not violate the functional constraint because after the filter of the step 2, all the remaining identifier(s) do not contain the same Values for all the Data Points. In other words, given that the input Data Set is a 1st order function and therefore does not contain duplicates, the result Data Set is a 1st order function as well. To show this, let K₁,...,K_m,...,K_n be the Identifier components for the generic input Data Set DS. Let us suppose that K₁,...,K_m are assigned to fixed values by using the subspace operator. A duplicate could arise only if in the result there are two Data Points DP_{r1} and DP_{r2} having the same value for K_{m+1},...,K_n, but this is impossible since such Data Points had same K₁,...,K_m in the original Data Set DS, which did not contain duplicates.

7264

7265

7266

7267 If we consider the vector space of Data Points individuated by the n-uples of Identifier components of a Data Set
 7268 DS(K_1, \dots, K_n, \dots) (along, e.g., with the operators of sum and multiplication), we have that the subspace operator
 7269 actually performs a subsetting of such space into another space with fewer Identifiers. This can be also seen as
 7270 the equivalent of a *dice* operation performed on hyper-cubes in multi-dimensional data warehousing.

7271

7272

7273 *Examples*

7274

7275 Given the Data Set DS_1:

7276

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

7277

7278

7279

Example 1: DS_r := DS_1 [sub Id_1 = 1, Id_2 = "A"]

results in:

DS_r		
Id_3	Me_1	At_1
XX	20	F
YY	1	F

7280

7281

7282

Example 2: DS_r := DS_1 [sub Id_1 = 1, Id_2 = "B", Id_3 = "YY"] results in:

DS_r	
Me_1	At_1
9	F

7283

7284

7285

7286

7287

7288

Example 3: DS_r := DS_1 [sub Id_2 = "A"] + DS_1 [sub Id_2 = "B"] results in:

Assuming that At_1 is viral and that in the propagation rule the greater value prevails, results in:

DS_r			
Id_1	Id_3	Me_1	At_1
1	XX	24	F
1	YY	10	F
2	XX	19	F
2	YY	20	F

7289

7290

7291