

SDMX Technical Working Group

VTL Task Force

VTL - version 2.0
(Validation & Transformation Language)

Part 2 - Reference Manual

April 2018

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
75 information model the language is based on;
- 76 b) Part 2 – the reference manual, containing the full library of operators ordered by category, including
77 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
79 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 Gelsema, Luca Gramaglia, Arofan Gregory, Gyorgy Gyomai,
92 Edgardo Greising, Dragan Ivanovic, Angelo Linardi, Juan Munoz, Chris Nelson, Stratos Nikoloutsos, Stefano
93 Pambianco, Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Roberto Sannino, Angel Simon Delgado,
94 Daniel Suranyi, Olav ten Bosch, Laura Vignola, Fernando Wagener and Nikolaos Zisimos.

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

97

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	84
150	Unary plus : +	84
151	Unary minus: -	85
152	Addition : +	86
153	Subtraction : -	88
154	Multiplication : *	89
155	Division : /	91
156	Modulo : mod	92
157	Rounding : round	94
158	Truncation : trunc	96
159	Ceiling : ceil	98
160	Floor: floor	99
161	Absolute value : abs	100
162	Exponential : exp	101
163	Natural logarithm : ln	102
164	Power : power	104
165	Logarithm : log	105
166	Square root : sqrt	106

167	VTL-ML - Comparison operators.....	108
168	Equal to : =	108
169	Not equal to : <>	109
170	Greater than : > >=	110
171	Less than : < <=	112
172	Between : between	114
173	Element of: in / not_in	115
174	match_characters match_characters	117
175	IsNull: isnull	118
176	Exists in : exists_in	120
177	VTL-ML - Boolean operators.....	122
178	Logical conjunction: and	122
179	Logical disjunction : or	123
180	Exclusive disjunction : xor	125
181	Logical negation : not	127
182	VTL-ML - Time operators.....	129
183	Period indicator : period_indicator	129
184	Fill time series : fill_time_series	130
185	Flow to stock : flow_to_stock	136
186	Stock to flow : stock_to_flow	139
187	Time shift : timeshift	142
188	Time aggregation : time_agg	145
189	Actual time : current_date	147
190	VTL-ML - Set operators.....	149
191	Union: union	149
192	Intersection : intersect	150
193	Set difference : setdiff	152
194	Simmetric difference : syndiff	153
195	VTL-ML - Hierarchical aggregation.....	155
196	Hierarchical roll-up : hierarchy	155
197	VTL-ML - Aggregate and Analytic operators.....	159
198	Aggregate invocation.....	160
199	Analytic invocation.....	163
200	Counting the number of data points: count	166
201	Minimum value : min	167

202 Maximum value : **max**..... 168

203 Median value : **median**..... 169

204 Sum : **sum** 170

205 Average value : **avg** 172

206 Population standard deviation : **stddev_pop** 173

207 Sample standard deviation : **stddev_samp** 174

208 Population variance : **var_pop**..... 175

209 Sample variance : **var_samp**..... 176

210 First value : **first_value**..... 177

211 Last value : **last_value**..... 178

212 Lag : **lag**..... 180

213 lead : **lead** 181

214 Rank : **rank**..... 183

215 Ratio to report : **ratio_to_report** 184

216 **VTL-ML - Data validation operators..... 186**

217 check_datapoint 186

218 check_hierarchy..... 188

219 check 192

220 **VTL-ML - Conditional operators..... 195**

221 if-then-else : **if** 195

222 Nvl : **nvl**..... 197

223 **VTL-ML - Clause operators 199**

224 Filtering Data Points : **filter** 199

225 Calculation of a Component : **calc** 200

226 Aggregation : **aggr** 201

227 Maintaining Components: **keep** 204

228 Removal of Components: **drop**..... 205

229 Change of Component name : **rename** 206

230 Pivoting : **pivot** 207

231 Unpivoting : **unpivot**..... 208

232 Subspace : **sub**..... 210

233

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

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

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

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

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

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

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

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

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

256

Introduction

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

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

- A left-hand side, which specifies the Artefact which the outcome of the expression is assigned to (this is the result of the Transformation);

- An assignment operator, which specifies also the persistency of the left hand side. The assignment operators are two, the first one for the persistent assignment (\leftarrow) and the other one for the non-persistent assignment ($:=$).

- A right-hand side, which is the expression to be evaluated, whose inputs are the operands of the Transformation. An expression consists in the invocation of VTL Operators in a certain order. When an Operator is invoked, for each input Parameter, an actual argument (operand) is passed to the Operator, which returns an actual argument for the output Parameter. In the right hand side (the expression), the Operators can be nested (the output of an Operator invocation can be input of the invocation of another Operator). All the intermediate results in an expression are non-persistent.

Examples of Transformations are:

```
DS_np := ( DS_1 - DS_2 ) * 2 ;
```

```
DS_p <- if DS_np >= 0 then DS_np else DS_1 ;
```

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

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

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

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

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

```
(i) DS_p <- if DS_np >= 0 then DS_np else DS_1 ;
```

```
(ii) DS_np := ( DS_1 - DS_2 ) * 2 ;
```

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

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

310

311 Conventions for writing VTL Transformations

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

313 Transformations:

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

317 Comments:

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

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

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

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

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

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

327

328 Examples of valid comments:

329 *Example 1:*

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

332 *Example 2:*

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

334 *Example 3:*

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

337 *Example 4:*

```
338         DS_r := DS_1           // my comment  
339                * DS_2 ;  
340         (for the VTL this statement is the Transformation DS_r := DS_1 * DS_2; )
```

341

342 **Typographical conventions**

343
344
345
346

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

<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 <i>time_period</i> . 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 (<i>op</i> , <i>start</i> , <i>length</i>)
<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. <code>length ("Hello, World!")</code>• <i>Optional and Mandatory Parameters (in the text)</i> e.g. ".....If comp is a Measure in <i>op</i>, 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>	<i>Compact name</i>	<i>Abbreviation</i>
359	<i>Data Set</i>	<i>Data Set</i>	<i>DS</i>
360	<i>Data Point</i>	<i>Data Point</i>	<i>DP</i>
361	<i>Identifier Component</i>	<i>Identifier</i>	<i>Id</i>
362	<i>Measure Component</i>	<i>Measure</i>	<i>Me</i>
363	<i>Attribute Component</i>	<i>Attribute</i>	<i>At</i>
364	<i>Data Set Component</i>	<i>Component</i>	<i>Comp</i>
365	<i>Value Domain Subset</i>	<i>Subset or Set</i>	<i>Set</i>
366	<i>Value Domain</i>	<i>Domain</i>	<i>VD</i>

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

370 Conventions for describing the operators’ syntax

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

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

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

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

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

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

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

392 $op1 + op2$

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

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

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

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

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

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

403 **substr** (op, start)

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

405 **substr** (op)

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

- 410 • non-optional : *non-optional sub-expression (text without braces)*
- 411 • {optional} : *optional sub-expression (zero or 1 occurrence)*
- 412 • {non-optional}¹ : *non-optional sub-expression (just 1 occurrence)*
- 413 • {one-or-more}⁺ : *sub-expression repeatable from 1 to many occurrences*
- 414 • {zero-or-more}^{*} : *sub-expression repeatable from 0 to many occurrences*
- 415 • { part1 | part2 | part3 } : *optional alternative sub-expressions (zero or 1 occurrence)*
- 416 • { part1 | part2 | part3 }¹ : *alternative sub-expressions (just 1 occurrence)*
- 417 • { part1 | part2 | part3 }⁺ : *alternative sub-expressions, from 1 to many occurrences*
- 418 • { 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*
- 427 • sub-expr₁-text, sub-expr_N-text *are sub-expressions directly written in the meta-expression*
- 428 • sub-expr₂-name, ... sub-expr_{N-1}-name *are identifiers of sub-expressions.*
- 429 • sub-expr₂-text, ... sub-expr_{N-1}-text *are subexpression written separately from the meta-expression.*
- 430 • The symbol ::= *means “is defined as” and denotes the assignment of a sub-expression-text to a sub-*
431 *expression-name.*

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

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

435 The meta-expression above synthesizes that:

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

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

443 ltrim (DS_2)

444 rtrim (DS_3)

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

447

448

Description of the data types of operands and result

449

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

450

451

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 <i>geo_area</i>
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 contain only the name of an artefact of such a type but not a sub-expression. For example:

454

comp :: name < component < string > >

455

456

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

457

458

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

459

460 VTL-ML Operators

461

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

Type conversion	cast	cast (op , scalarType { , mask })	converts to the specified data type	Func.	op :: dataset{ measure<scalar> _ } component<scalar> scalar scalarType :: scalar type mask :: string	dataset{ measure<scalar> _ } component<scalar> scalar	Changing data type
Join	inner_join, left_join, full_join, cross_join,	<pre> joinOperator (ds1 { as alias1 }, ..., dsN { as aliasN } { using usingComp } { filter filterCondition } { apply applyExpr calc calcClause aggr aggrClause { groupingClause } } { keep comp { comp }* drop comp { comp }* } { rename compFrom to compTo { , compFrom to compTo }* }) joinOperator ::= { inner_join left_join full_join cross_join }¹ calcClause ::= { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }* calcRole ::= { identifier measure attribute viral attribute }¹ aggrClause ::= { aggrRole } aggrComp := aggrExpr { , { aggrRole } aggrComp := aggrExpr }* aggrRole ::= { measure attribute viral attribute }¹ groupingClause ::= { group by idList group except idList group all conversionExpr }¹ { having havingCondition } </pre>	Inner join, left outer join, full outer join, cross join,	Func.	<pre> ds1, ..., dsN :: dataset alias1, ..., aliasN :: name usingId :: name < component > filterCondition :: component<boolean> applyExpr :: dataset calcComp :: name<component> calcExpr :: component<scalar> aggrComp :: name<component > aggrExpr :: component<scalar> groupingId :: name < identifier > conversionExpr :: component<scalar> havingCondition :: component<boolean> comp :: name < component > compFrom :: component<scalar> compTo :: component<scalar> </pre>	dataset	Specific
String concatenation		op1 op2	Concatenates two strings	Infix	<pre> op1, op2 :: dataset { measure<string> _+ } component<string> string </pre>	dataset { measure<string> _+ } component<string> string	On two scalars, DSSs or DSCs

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

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

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

Actual time	current_date	current_date ()	returns the current date	Func.		date	Specific
Union	union	union (dsList) <i>dsList ::= ds { , ds }*</i>	Computes the union of N datasets	Func.	ds :: dataset	dataset	Set
Intersection	intersect	intersect (dsList) <i>dsList ::= ds { , ds }*</i>	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 }) <i>condComp ::= component { , component }*</i> <i>mode ::= non_null non_zero partial_null partial_zero always_null always_zero</i> <i>input ::= dataset rule rule_priority</i> <i>output ::= computed all</i>	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> aggregateOperator (firstOperand { , additionalOperand }* { groupingClause }) <i>in a Component expression within an aggr clause</i> aggregateOperator (firstOperand { , additionalOperand }*) { groupingClause } aggregateOperator ::= avg count max median min stddev_pop stddev_samp sum var_pop var_samp groupingClause ::= { group by groupingId { , groupingId }* group except groupingId { , groupingId }* group all conversionExpr }1 { having havingCondition }	Set of statistical functions used to aggregate data	Func.	firstOperand :: dataset component additionalOperand :: type of the (possible) additional parameter of the aggregate Operator groupingId :: name < identifier > conversionExpr :: identifier havingCondition :: component<boolean>	dataset component	Specific

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

If then else	if ...then else...	if condition then thenOperand else elseOperand	Makes alternative calculations according to a condition	Func.	condition :: dataset { measure <boolean> _ } component<boolean> boolean thenOperand :: dataset component scalar elseOperand :: dataset component scalar	dataset component scalar	Specific
Nvl	nvl	nvl (op1, op2)	Replaces the null value with a value.	Func.	op1, op2:: dataset component scalar	dataset component scalar	Specific
Filtering Data Points	filter	op [filter condition]	Filter data using a Boolean condition	Clause	op :: dataset filterCondition :: component<boolean>	dataset	Specific
Calculation of a Component	calc	op [calc { <u>calcRole</u> } calcComp := calcExpr { , { <u>calcRole</u> } calcComp := calcExpr }*]	Calculates the values of a Structure Component	Clause	op :: dataset calcComp :: name < component > calcExpr :: component<scalar>	dataset	Specific
Aggregation	aggr	op [aggr aggrClause { groupingClause }] aggrClause ::= { <u>aggrRole</u> } aggrComp := aggrExpr { , { <u>aggrRole</u> } aggrComp := aggrExpr }* <u>groupingClause</u> ::= { group by groupingId { , groupingId }* 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.

468

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

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 *Syntax*

517
 518
 519 **define datapoint ruleset** rulesetName (dpRulesetSignature) **is**

520 dpRule
 521 { ; dpRule }*

522 **end datapoint ruleset**

523

524 dpRulesetSignature ::= **valuedomain** listValueDomains | **variable** listVariables

525 listValueDomains ::= valueDomain { **as** vdAlias } { , valueDomain { **as** vdAlias } }*

526 listVariables ::= variable { **as** varAlias } { , variable { **as** varAlias } }*

527 dpRule ::= { ruleName : } { **when** antecedentCondition **then** } consequentCondition
 528 { **errorcode** errorCode }
 529 { **errorlevel** errorLevel }

530

531 *Syntax description*

532 rulesetName the name of the Data Point Ruleset to be defined.

533 dpRulesetSignature the Cartesian space of the Ruleset (signature of the Ruleset), which specifies either the
 534 Value Domains or the Represented Variables (see the information model) on which the
 535 Ruleset is defined. If **valuedomain** is specified then the Ruleset is applicable to the Data
 536 Sets having Components that take values on the specified Value Domains. If **variable** is
 537 specified then the Ruleset is applicable to Data Sets having the specified Variables as
 538 Components.

539 valueDomain a Value Domain on which the Ruleset is defined.

540 vdAlias an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can
 541 be used for the sake of compactness in writing the Rules. If an alias is not specified then
 542 the name of the Value Domain (parameter valueDomain) is used in the body of the rules.

543 variable a Represented Variable on which the Ruleset is defined.

544 varAlias an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be
 545 used for the sake of compactness in writing the Rules. If an alias is not specified then the
 546 name of the Variable (parameter valueDomain) is used in the body of the Rules.

547 dpRule a Data Point Rule, as defined in the following parameters.

548 ruleName the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for
 549 validation then the ruleName identifies the validation results of the various Rules of the
 550 Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive
 551 order number of the Rule in the Ruleset. However please note that, if ruleName is
 552 omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules
 553 are added or existing Rules are deleted, and therefore the users that interpret the
 554 validation results must be aware of these changes.

555 antecedentCondition a *boolean* expression to be evaluated for each single Data Point of the input Data Set. It
 556 can contain Values of the Value Domains or Variables specified in the Ruleset signature
 557 and constants; all the VTL-ML component level operators are allowed. If omitted then
 558 antecedentCondition is assumed to be TRUE.

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

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

580 *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`

593 *Constraints*

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

599 *Semantic specification*

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

602 A Data Point Ruleset is a persistent object that contains Rules to be applied to the Data Points of a Data Set¹. The
603 Data Point Rulesets can be invoked by the **check_datapoint** operator. The Rules are aimed at checking the
604 combinations of values of the Data Set Components, assessing if these values fulfil the logical conditions
605 expressed by the Rules themselves. The Rules are evaluated independently for each Data Point, returning a
606 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 apply 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*

642

```
643 define datapoint ruleset DPR_1 ( valuedomain flow_type A, numeric_value 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 F, obs_value 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
```

682 *Syntax*

```
683
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 : } codeItemRelation { errorcode errorCode } { errorlevel errorLevel }
696 codeItemRelation ::=
697     { when leftCondition then }
698     leftCodeItem { = | > | < | >= | <= }1
699     { + | - } rightCodeItem { [ rightCondition ] }
700     { { + | - }1 rightCodeItem { [ rightCondition ] } }*
```

701 *Syntax description*

702		
703		
704	<u>rulesetName</u>	the name of the Hierarchical Ruleset to be defined.
705	<u>hrRulesetSignature</u>	the signature of the Ruleset. It specifies the Value Domain or Variable on which the 706 Ruleset is defined, and the Conditioning Signature.
707	<u>vdRulesetSignature</u>	the signature of a Ruleset defined on Value Domains
708	<u>varRulesetSignature</u>	the signature of a Ruleset defined on Variables
709	<u>hrRule</u>	a single hierarchical rule, as described below.

710	<u>vdConditioningSignature</u>	specifies the Value Domains on which the conditions are defined. The Ruleset is meant to be applicable to the Data Sets having Components that take values on the Value Domain on which the ruleset is defined (i.e., ruleValueDomain) and on the conditioning Value Domains (i.e., condValueDomain).
711		
712		
713		
714	ruleValueDomain	the Value Domain on which the Ruleset is defined
715	condValueDomain	a conditioning Value Domain of the Ruleset
716	vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this 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.
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 Variableomain (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>codeItemRelation</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
766		
767		
768		

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

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

798 *Constraints*

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

803 *Semantic specification*

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

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

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

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

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

822 *UnitedKingdom < Europe*

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

825 *January2000 < year2000*

² “Coincides” means “implies and is implied”

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

828 The first member of a Relation is a single Code Item. The second member can be either a single Code Item, like in
829 the example above, or a **logical composition of Code Items** giving another Code Item as result. The logical
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” (EUwithoutUK = 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 EUwithoutUK) 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:

```
881  
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
```

888
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                + SE [ Time >= "01.01.1995" ]
940                + SI [ Time >= "01.05.2004" ]
941                + SK [ Time >= "01.05.2004" ]
942     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 BoPIItem ) is
965         transport_method1 : Transport = AirTransport + SeaTransport + LandTransport
966         ; transport_method2 : Transport = PassengersTransport + FreightsTransport
967     end hierarchical ruleset
```

968
969
970 Multiple relations can be useful even for calculation. For example, imagine that the input Data Set contains data
971 about resident units broken down by region and data about non-residents units broken down by country. In
972 order to calculate a homogeneous level of aggregation (e.g., by country), a possible Ruleset is the following:

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

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

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

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

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

994

995 *Examples*

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

```
998  
999     define hierarchical ruleset sex_hr ( valuedomain rule sex) is
1000         TOTAL = MALE + FEMALE
1001     end hierarchical ruleset
```

1002

1003 2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS. No conditions
1004 are defined.
1005
1006 define hierarchical ruleset BeneluxCountriesHierarchy (valuedomain rule GeoArea) is
1007 BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode "Bad value for Benelux"
1008 end hierarchical ruleset
1009

1010 3) American economic partners. The first rule states that the value for North America should be greater than the
1011 value reported for US. This type of validation is useful when the data communicated by the data provider do not
1012 cover the whole composition of the aggregate but only some elements. No conditions are defined.
1013
1014 define hierarchical ruleset american_partners_hr (variable rule PartnerArea) is
1015 NORTH_AMERICA > US
1016 ; SOUTH_AMERICA = BR + UY + AR + CL
1017 end hierarchical ruleset
1018

1019 4) Example of an aggregate Code Item having multiple definitions to be used for validation only. The Balance of
1020 Payments item "Transport" can be broken down by type of carrier (Air transport, Sea transport, Land transport)
1021 and by type of objects transported (Passengers and Freights) and both breakdowns must sum up to the total
1022 "Transport" figure.
1023
1024 define hierarchical ruleset validationruleset_bop (variable rule BoPItem) is
1025 transport_method1 : Transport = AirTransport + SeaTransport + LandTransport
1026 ; transport_method2 : Transport = PassengersTransport + FreightsTransport
1027 end hierarchical ruleset
1028
1029

1031 **define operator**1032 *Syntax*

```

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

```

```

1037
1038 parameter::= parameterName parameterType { default parameterDefaultValue }
1039

```

1040 *Syntax description*

1041 <u>operator_name</u>	the name of the operator
1042 <u>parameter</u>	the names of parameters, their data types and defaultvalues
1043 <u>outputType</u>	the data type of the artefact returned by the operator
1044 <u>operatorBody</u>	the expression which defines the operation
1045 <u>parameterName</u>	the name of the parameter
1046 <u>parameterType</u>	the data type of the parameter
1047 <u>parameterDefaultValue</u>	the default value for the parameter (optional)

1048

1049 *Parameters*

1050 <u>operator_name</u>	name
1051 <u>outputType</u>	a VTL data type (see the Data Type Syntax below)
1052 <u>operatorBody</u>	a VTL expression having the parameters (i.e., <u>parameterName</u>) as the operands
1053 <u>parameterName</u>	name
1054 <u>parameterType</u>	a VTL data type (see the Data Type Syntax below)
1055 <u>parameterDefaultValue</u>	a Value of the same type as the parameter

1056

1057 *Constraints*

- 1058 • Each parameterName must be unique within the list of parameters
- 1059 • parameterDefaultValue must be of the same data type as the corresponding parameter
- 1060 • outputType must be compatible with the type of operatorBody (it can also be a sub-type of the type returned by the operatorBody expression)
- 1061 • If outputType is omitted then the type returned by the operatorBody expression is assumed
- 1062 • If parameterDefaultValue is specified then the parameter is optional

1064

1065 *Semantic specification*

1066 This operator defines a user-defined Operator by means of a VTL expression, specifying also the parameters, their data types, whether they are mandatory or optional and their (possible) default values.

1068

1069 *Examples*1070 *Example1:*

```

1071     define operator max1 (x integer, y integer)
1072     returns boolean is
1073     if x > y then x else y
1074     end define operator
1075

```

1075

1076 *Example2:*

```

1077     define operator add (x integer default 0, y integer default 0)
1078     returns number is
1079     x+y
1080     end define operator

```

1080

1081 **Data type syntax**

1082 The VTL data types are described in the VTL User Manual. Types are used throughout this Reference Manual as
 1083 both meta-syntax and syntax.

1084 They are used as meta-syntax in order to define the types of input and output parameters in the descriptions of
 1085 VTL operators; they are used in the syntax, and thus are proper part of the VTL, in order to allow other operators
 1086 to refer to specific data types. For example, when defining a custom operator (see the **define operator** above),
 1087 one will need to declare the type of the input/output parameters.

1088 The syntax of the data types is described below (as for the meaning of these definitions, see the section VTL Data
 1089 Types in the User Manual). See also the section “Conventions for describing the operators’ syntax” in the chapter
 1090 “Overview of the language and conventions” above.

1091 dataType ::= scalarType | compoundType

1092

1093 scalarType ::= { basicScalarType | valueDomainName | setName }¹ { scalarTypeConstraint } { **null** |
 1094 **not null** }

1095 basicScalarType ::= { **scalar** | **number** | **integer** | **string** | **boolean** | **time** | **date** | **time_period** |
 1096 **duration** }¹

1097 valueDomainName :: name

1098 setName :: name

1099 scalarTypeConstraint ::= [valueBooleanCondition] | { scalarLiteral { , scalarLiteral }^{*} }

1100 compoundType ::= componentType | datasetType | operatorType | rulesetType | productType |
 1101 universalSetType

1102 componentType ::= componentRole { < scalar type > }

1103 componentRole ::= { **component** | **identifier** | **measure** | **attribute** | **viral attribute** }¹

1104 datasetType ::= **dataset** { componentConstraint { , componentConstraint }^{*} }

1105 componentConstraint ::= componentType { componentName | multiplicityModifier }¹

1106 componentName :: name

1107 multiplicityModifier ::= _ { + | * }

1108 productType ::= { dataType { * dataType }⁺ }¹

1109 operatorType ::= { dataType -> dataType }¹

1110 rulesetType ::= { **ruleset** | dpRuleset | hrRuleset }¹

1111 dpRuleset ::= **datapoint**

1112 | **datapoint_on_valuedomains** { (name { * name }^{*}) }

1113 | **datapoint_on_variables** { (name { * name }^{*}) }

1114 hrRuleset ::= **hierarchical**

1115 | **hierarchical_on_valuedomains** { valueDomainName { *
 1116 (prodValueDomains) } }

1117 | **hierarchical_on_variables** { variableName { * (prodVariables) } }

1118 universalSetType ::= **set** { < dataType > }

1119

1120 Note that the valueBooleanCondition in scalarTypeConstraint is expressed with reference to the fictitious
 1121 variable “value” (see also the User Manual, section “Conventions for describing the Scalar Types”), which
 1122 represents the generic value of the scalar type, for example:

1123 integer { 0, 1 } means an integer number whose value is 0 or 1

1124 number [value >= 0] means a number greater or equal than 0

1125 string { "A", "B", "C" } means a string whose value is A, B or C:

1126 string [length (value) <= 10] means a string whose length is lower or equal than 10:
1127

1128 General examples of the syntax for defining types can be found in the User Manual, section VTL Data Types and
1129 in the declaration of the data types of the VTL operators (sub-sections “input parameters type” and “result
1130 type”).

1131 VTL-ML - Typical behaviours of the ML Operators

1132 In this section, the common behaviours of some class of VTL-ML operators are described, both for a better
1133 understanding of the characteristics of such classes and to factor out and not repeat the explanation for each
1134 operator of the class.

1135 Typical behaviour of most ML Operators

1136 Unless differently specified in the Operator description, the Operators can be applied to Scalar Values, to Data
1137 Sets and to Data Set Components.

1138 The operations on Scalar Values are primitive and are part of the core of the language. The other kind of
1139 operations can be typically be obtained by means of the scalar operations in conjunction with the Join operator,
1140 which is part of the core too.

1141 In the operations on Data Set, the Operators are meant to be applied by default only to the values of the
1142 Measures of the input Data Sets, leaving the Identifiers unchanged. The Attributes follow by default their specific
1143 propagation rules, which are described in the User Manual.

1144 In the operations on Components, the Operators are meant to be applied on the specified components of one
1145 input Data Set, in order to calculate a new component which becomes part of the resulting Data Set. In this case,
1146 the Attributes can be operated like the Measures.

1147 Operators applicable on one Scalar Value or Data Set or Data Set 1148 Component

1149 *Operations on Scalar values*

1150 The operator is applied on a scalar value and returns a scalar value.

1152 *Operations on Data Sets*

1153 The operator is applied on a Data Set and returns a Data Set.

1154 For example, using a functional style and denoting the operator with $f(...)$, this can be written as:

1155 $DS_r := f(DS_1)$

1156 The same operation, using an infix style and denoting the operator as **op**, can be also written as

1157 $DS_r := op DS_1$

1158 This means that the operator is applied to the values of all the Measures of DS_1 in order to produce
1159 homonymous Measures in DS_r .

1160 The application of the operator is allowed only if all the Measures of the operand Data Set are of a data type
1161 compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the
1162 operand Data Sets are numeric). If the Measures of the operand Data Set are of different types, not all compatible
1163 with the operator to be applied, the membership or the keep clauses can be used to select only the proper
1164 Measures. No applicability constraints exist on Identifiers and Attributes, which can be any.

1165 As for the data content, for each Data Point (DP_1) of the operand Data Set, a result Data Point (DP_r) is returned,
1166 having for the Identifiers the same values as DP_1 .

1167 For each Data Point DP_1 and for each Measure, the operator is applied on the Measure value of DP_1 and
1168 returns the corresponding Measure value of DP_r .

1169 For each Data Point DP_1 and for each viral Attribute, the value of the Attribute propagates unchanged in DP_r .

1170 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1171 (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1172 of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1 ; these Attributes are
1173 considered as “viral” also in DS_r , the “non-viral” Attributes of DS_1 are not kept in DS_r).

1174

1176 *Operations on Data Set Components*

1177 The operator is applied on a Component (COMP_1) of a Data Set (DS_1) and returns another Component
1178 (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r).

1179 For example, using a functional style and denoting the operator with $f(\dots)$, this can be written as:

1180 $DS_r := DS_1 [\text{calc } COMP_r := f(COMP_1)]$

1181 The same operation, using an infix style and denoting the operator as **op**, can be written as:

1182 $DS_r := DS_1 [\text{calc } COMP_r := \text{op } COMP_1]$

1183 This means that the operator is applied on COMP_1 in order to calculate COMP_r.

- 1184 • If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components
1185 of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- 1186 • If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of
1187 the operator $f(\dots)$ replace the DS_1 original values for such a Measure or Attribute in order to produce
1188 DS_r.
- 1189 • If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can
1190 become inconsistent.

1191 In any case, an operation on the Components of a Data Set produces a new Data Set, as in the example above.

1192 The application of the operator is allowed only if the input Component belongs to a data type compatible with
1193 the operator (for example, a numeric operator is applicable only on numeric Components). As already said,
1194 COMP_r cannot have the same name of an Identifier of DS_1.

1195 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
1196 returning the value of COMP_r.

1197 As for the data structure, like for the operations on Data Sets above, the result Data Set (DS_r) has the Identifiers
1198 and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the
1199 attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as
1200 “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not
1201 kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

1202 Moreover, in the case of the operations on Data Set Components, the (possible) new Component DS_r can be
1203 added to the original structure, the role of a (possible) existing DS_1 Component can be altered, the virality of a
1204 (possibly) existing DS_r Attribute can be altered, a (possible) COMP_r non-viral Attribute can be kept in the
1205 result. For the alteration of role and virality see also the **calc** clause.

1206 **Operators applicable on two Scalar Values or Data Sets or Data Set**
1207 **Components**

1208 *Operation on Scalar values*
1209

1210 The operator is applied on two Scalar values and returns a Scalar value.

1211 *Operation on Data Sets*
1212

1213 The operator is applied either on two Data Sets or on one Data Set and one Scalar value and returns a Data Set.
1214 The composition of a Data Set and a Component is not allowed (it makes no sense).

1215 For example, using a functional style and denoting the operator with $f(\dots)$, this can be written as:

1216 $DS_r := f(DS_1, DS_2)$

1217 The same kind of operation, using an infix style and denoting the operator as **op**, can be also written as

1218 $DS_r := DS_1 \text{ op } DS_2$

1219 This means that the operator is applied to the values of all the couples of Measures of DS_1 and DS_2 having the
1220 same names in order to produce homonymous Measures in DS_r. DS_1 or DS_2 may be replaced by a Scalar
1221 value.

1222 The composition of two Data Sets (DS_1, DS_2) is allowed if the two operand Data Sets have exactly the same
1223 Measures and if all these Measures belong to a data type compatible with the operator (for example, a numeric
1224 operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the
1225 operand Data Sets are different or of different types not all compatible with the operator to be applied, the
1226 membership or the **keep** clauses can be used to select only the proper Measures. The composition is allowed if

1227 these operand Data Sets have the same Identifiers or if one of them has at least all the Identifiers of the other one
1228 (in other words, the Identifiers of one of the Data Sets must be a superset of the Identifiers of the other one). No
1229 applicability constraints exist on the Attributes, which can be any.

1230 As for the data content, the operand Data Sets (DS₁, DS₂) are joined to find the couples of Data Points (DP₁,
1231 DP₂), where DP₁ is from the first operand (DS₁) and DP₂ from the second operand (DS₂), which have the
1232 same values as for the common Identifiers. Data Points that are not coupled are left out (the inner join is used).
1233 An operand Scalar value is treated as a Data Point that couples with all the Data Points of the other operand. For
1234 each couple (DP₁, DP₂) a result Data Point (DP_r) is returned, having for the Identifiers the same values as
1235 DP₁ and DP₂.

1236 For each Measure and for each couple (DP₁, DP₂), the Measure values of DP₁ and DP₂ are composed through
1237 the operator so returning the Measure value of DP_r. An operand Scalar value is composed with all the Measures
1238 of the other operand.

1239 For each couple (DP₁, DP₂) and for each Attribute that propagates in DP_r, the Attribute value is calculated by
1240 applying the proper Attribute propagation algorithm on the values of the Attributes of DP₁ and DP₂.

1241 As for the data structure, the result Data Set (DS_r) has all the Identifiers (with no repetition of common
1242 Identifiers) and the Measures of both the operand Data Sets, and has the Attributes resulting from the
1243 application of the attribute propagation rules on the Attributes of the operands (DS_r maintains the Attributes
1244 declared as “viral” for the operand Data Sets; these Attributes are considered as “viral” also in DS_r, the “non-
1245 viral” Attributes of the operand Data Sets are not kept in DS_r).

1246

1247 *Operation on Data Set Components*

1248 The operator is applied either on two Data Set Components (COMP₁, COMP₂) belonging to the same Data Set
1249 (DS₁) or on a Component and a Scalar value, and returns another Component (COMP_r) which alters the
1250 structure of DS₁ in order to produce the result Data Set (DS_r). The composition of a Data Set and a Component
1251 is not allowed (it makes no sense).

1252 For example, using a functional style and denoting the operator with **f** (...), this can be written as:

1253
$$DS_r := DS_1 [\text{calc } COMP_r := f (COMP_1, COMP_2)]$$

1254 The same operation, using an infix style and denoting the operator as **op**, can be written as:

1255
$$DS_r := DS_1 [\text{calc } COMP_r := COMP_1 \text{ op } COMP_2]$$

1256 This means that the operator is applied on COMP₁ and COMP₂ in order to calculate COMP_r.

- 1257 • If COMP_r is a new Component which originally did not exist in DS₁, it is added to the original Components
1258 of DS₁, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- 1259 • If COMP_r is one of the original Measures or Attributes of DS₁, the values obtained from the application of
1260 the operator **f** (...) replace the DS₁ original values for such a Measure or Attribute in order to produce
1261 DS_r.
- 1262 • If COMP_r is one of the original Identifiers of DS₁, the operation is not allowed, because the result can
1263 become inconsistent.

1264 In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

1265 The composition of two Data Set Components is allowed provided that they belong to the same Data Set³.
1266 Moreover, the input Components must belong to data types compatible with the operator (for example, a
1267 numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same
1268 name of an Identifier of DS₁.

1269 As for the data content, for each Data Point of DS₁, the values of COMP₁ and COMP₂ are composed through
1270 the operator so returning the value of COMP_r.

1271 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1272 (DS₁), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1273 of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS₁; these Attributes are
1274 considered as “viral” also in DS_r, the “non-viral” Attributes of DS₁ are not kept in DS_r). If an Attribute is
1275 explicitly calculated, the attribute propagation rule is overridden.

1276 Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added
1277 to the original structure of DS₁, the role of a (possibly) existing DS₁ 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

1278 (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the
1279 result. For the alteration of role and virality see also the **calc** clause.

1280 Operators applicable on more than two Scalar Values or Data Set 1281 Components

1282 The cases in which an operator can be applied on more than two Data Sets (like the Join operators) are described
1283 in the relevant sections.

1284 *Operation on Scalar values*

1285 The operator is applied on more Scalar values and returns a Scalar value according to its semantics.
1286
1287

1288 *Operation on Data Set Components*

1289 The operator is applied either on a combination of more than two Data Set Components (COMP_1, COMP_2)
1290 belonging to the same Data Set (DS_1) or Scalar values, and returns another Component (COMP_r) which alters
1291 the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a
1292 Component is not allowed (it makes no sense).

1293 For example, using a functional style and denoting the operator with $f(\dots)$, this can be written as:

1294 $DS_r := DS_1 [\text{substr } COMP_r := f(COMP_1, COMP_2, COMP_3)]$

1295 This means that the operator is applied on COMP_1, COMP_2 and COMP_3 in order to calculate COMP_r.

- 1296 • If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components
1297 of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- 1298 • If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of
1299 the operator $f(\dots)$ replace the DS_1 original values for such a Measure or Attribute in order to produce
1300 DS_r.
- 1301 • If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can
1302 become inconsistent.

1303 In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

1304 The composition of more Data Set Components is allowed provided that they belong to the same Data Set⁴.
1305 Moreover, the input Components must belong to data types compatible with the operator (for example, a
1306 numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same
1307 name of an Identifier of DS_1.

1308 As for the data content, for each Data Point of DS_1, the values of COMP_1, COMP_2 and COMP_3 are composed
1309 through the operator so returning the value of COMP_r.

1310 As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set
1311 (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes
1312 of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are
1313 considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is
1314 explicitly calculated, the attribute propagation rule is overridden.

1315 Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added
1316 to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a
1317 (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the
1318 result. For the alteration of role and virality see also the **calc** clause.

1319

1320 Behaviour of Boolean operators

1321 The Boolean operators are allowed only on operand Data Sets that have a single measure of type *boolean*. As for
1322 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

1323 Behaviour of Set operators

1324 These operators apply the classical set operations (union, intersection, difference, symmetric differences) to the
1325 Data Sets, considering them as sets of Data Points. These operations are possible only if the Data Sets to be
1326 operated have the same data structure, and therefore the same Identifiers, Measures and Attributes⁵.

1327 Behaviour of Time operators

1328 The *time* operators are the operators dealing with *time*, *date* and *time_period* basic scalar types. These types are
1329 described in the User Manual in the sections “Basic Scalar Types” and “External representations and literals used
1330 in the VTL Manuals”.

1331 The time-related formats used for explaining the time operators are the following (they are described also in the
1332 User Manual).

1333 For the *time* values:

1334 $YYYY-MM-DD/YYYY-MM-DD$

1335 Where *YYYY* are 4 digits for the year, *MM* two digits for the month, *DD* two digits for the day. For
1336 example:

1337 $2000-01-01/2000-12-31$ the whole year 2000

1338 $2000-01-01/2009-12-31$ the first decade of the XXI century

1339 For the *date* values:

1340 $YYYY-MM-DD$

1341 The meaning of the symbols is the same as above. For example:

1342 $2000-12-31$ the 31st December of the year 2000

1343 $2010-01-01$ the first of January of the year 2010

1344 For the *time_period* values:

1345 $YYYY\{P\}\{NNN\}$

1346 Where *YYYY* are 4 digits for the year, *P* is one character for the period indicator of the regular period (it
1347 refers to the *duration* data type and can assume one of the possible values listed below), *NNN* are from
1348 zero to three digits which contain the progressive number of the period in the year. For annual data the
1349 *A* and the three digits *NNN* can be omitted. For example:

1350 $2000M12$ the month of December of the year 2000 (duration: M)

1351 $2010Q1$ the first quarter of the year 2010 (duration: Q)

1352 $2010A$ the whole year 2010 (duration: A)

1353 2010 the whole year 2010 (duration: A)

1354 For the *duration* values, which are the possible values of the period indicator of the regular periods above, it is
1355 used for simplicity just one character whose possible values are the following:

1356	<u>Code</u>	<u>Duration</u>
1357	D	Day
1358	W	Week
1359	M	Month
1360	Q	Quarter
1361	S	Semester
1362	A	Year

1363 As mentioned in the User Manual, these are only examples of possible time-related representations, each VTL
1364 system is free of adopting different ones. In fact no predefined representations are prescribed, VTL systems are
1365 free to using them preferred or already existing ones.

1366 Several time operators deal with the specific case of Data Sets of time series, having an Identifier component that
1367 acts as the reference time and can be of one of the scalar types *time*, *date* or *time_period*; moreover this Identifier
1368 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

1369 It is worthwhile to recall here that, in the case of Data Sets of time series, VTL assumes that the information
 1370 about which is the Identifier Components that acts as the reference time and which is the period (frequency) of
 1371 the time series exists and is available in some way in the VTL system. The VTL Operators are aware of which is
 1372 the reference time and the period (frequency) of the time series and use these information to perform correct
 1373 operations. VTL also assumes that a Value Domain representing the possible periods (e.g. the period indicator
 1374 Value Domain shown above) exists and refers to the *duration* scalar type. For the assumptions above, the users
 1375 do not need to specify which is the Identifier Component having the role of reference time.

1376 The operators for time series can be applied only on Data Sets of time series and returns a Data Set of time
 1377 series. The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set
 1378 and contains the same time series as the operand. The Attribute propagation rule is not applied.

1379 Operators changing the data type

1380 These Operators change the Scalar data type of the operands they are applied to (i.e. the type of the result is
 1381 different from the type of the operand). For example, the **length** operator is applied to a value of *string* type and
 1382 returns a value of *integer* type. Another example is the **cast** operator.

1383 *Operation on Scalar values*

1384 The operator is applied on (one or more) Scalar values and returns one Scalar value of a different data type.

1386 *Operation on Data Sets*

1387 If an Operator change the data type of the Variable it is applied to (e.g., from *string* to *number*), the result Data Set
 1388 cannot maintain this Variable as it happens in the previous cases, because a Variable cannot have different data
 1389 types in different Data Sets⁶.

1390 As a consequence, the converted variable cannot follow the same rules described in the sections above and must
 1391 be replaced, in the result Data Set, by another Variable of the proper data type.

1392 For sake of simplicity, the operators changing the data type are allowed only on mono-measure operand Data
 1393 Sets, so that the conversion happens on just one Measure. A default generic Measure is assigned by default to the
 1394 result Data Set, depending on the data type of the result (the default Measure Variables are reported in the table
 1395 below).

1396 Therefore, if the operands are originally multi-measure, just one Measure must be pre-emptively selected (for
 1397 example through the membership operator) in order to apply the changing-type operator. Moreover, if in the
 1398 result Data Set a different Measure Variable name is desired than the one assigned by default, it is possible to
 1399 change the Variable name (see the **rename** operator).

1400 As for the Identifiers and the Attributes, the behaviour of these operators is the same as the typical behaviour of
 1401 the unary or binary operators.

1403 *Operation on Data Set Components*

1404 For the same reasons above, the result Component cannot be the same as one of the operand Components and
 1405 must be of the appropriate Scalar data type.

1407 *Default Names for Variables and Value Domains used in this manual*

1408 The following table shows the default Variable names and the relevant default Value Domain. These are only the
 1409 names used in this manual for explanatory purposes and can be personalised in the implementations. If VTL
 1410 rules are exchanged, the personalised names need to be shared with the partners of the exchange.

1412

Scalar data type	Default Variable	Default Value Domain
string	string_var	string_vd

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

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

1413 Type Conversion and Formatting Mask

1414 The conversions between *scalar* types is provided by the operator **cast**, described in the section of the general
1415 purpose operators. Some particular types of conversion require the specification of a formatting mask, which
1416 specifies which format the source or the destination of the conversion should assume. The formatting masks for
1417 the various scalar types are explained here.

1418 If needed, the formatting Masks can be personalized in the VTL implementations. If VTL rules are exchanged, the
1419 personalised masks need to be shared with the partners of the exchange.

1420 The Numbers Formatting Mask

1421 The **number formatting mask** can be defined as a combination of characters whose meaning is the following:

- 1422 ○ “D” one numeric digit (if the scientific notation is adopted, D is only for the mantissa)
- 1423 ○ “E” one numeric digit (for the exponent of the scientific notation)
- 1424 ○ “*” an arbitrary number of digits
- 1425 ○ “+” at least one digit
- 1426 ○ “.” (dot) can be used as a separator between the integer and the decimal parts.
- 1427 ○ “,” (comma) can be used as a separator between the integer and the decimal parts.

1428

1429 Examples of valid masks are:

1430 DD.DDDDD, DD.D, D, D.DDDD, D*.D*, D+.D+ , DD.DDDEEEE

1431 The Time Formatting Mask

1432 The format of the values of the types *time*, *date* and *time_period* can be specified through specific formatting
1433 masks. A mask related to *time*, *date* and *time_period* is formed by a sequence of symbols which denote:

- 1434 - the time units that are used, for example years, months, days
- 1435 - the format in which they are represented, for example 4 digits for the year (2018), 2 digits for the month
1436 within the year (04 for April) and 2 digits for the day within the year and the month (05 for the 5th)
- 1437 - the order of these parts; for example, first the 4 digits for the year, then the 2 digits for the month and finally
1438 the 2 digits for the day
- 1439 - other (possible) typographical characters used in the representation; for example, a line between the year
1440 and the month and between the month and the day (e.g., 2018-04-05).

1441 The time formatting masks follows the general rules below.

1442 For a numerical representations of the time units:

- 1443 - A digit is denoted through the use of a **special character** which depends on the time unit. for example Y is
1444 for “year”, M is for “month” and D is for “day”
- 1445 - The special character is lowercase for the time units shorter than the day (for example h for “hour”, m for
1446 “minute”, s for “second”) and uppercase for time units equal to “day” or longer (for example W for “week”, Q
1447 for “quarter”, S for “semester”)

- 1448 - The number of letters matches the number of digits, for example YYYY means that the year is represented
 1449 with four digits and MM that the month is of 2 digits
- 1450 - The numerical representation is assumed to be padded by leading 0 by default, for example MM means that
 1451 April is represented as 04 and the year 33 AD as 0033
- 1452 - If the numerical representation is not padded, the optional digits that can be omitted (if equal to zero) are
 1453 enclosed within braces; for example {M}M means that April is represented by 4 and December by 12, while
 1454 {YYY}Y means that the 33 AD is represented by 33

1455 For textual representations of the time units:

- 1456 - **Special words** denote a textual localized representation of a certain unit, for example DAY means a textual
 1457 representation of the day (MONDAY, TUESDAY ...)
- 1458 - An optional number following the special word denote the maximum length, for example DAY3 is a textual
 1459 representation that uses three characters (MON, TUE ...)
- 1460 - The case of the special word correspond to the case of the value; for example day3 (lowercase) denotes the
 1461 values mon, tue ...
- 1462 - The case of the initial character of the special word correspond to the case of the initial character of the time
 1463 format; for example Day3 denotes the values Mon, Tue ...
- 1464 - The letter P denotes the period indicator, (i.e., day, week, month ...) and the letter p denotes one digit for the
 1465 number of periods

1466 Representation of more time units:

- 1467 - If more time units are used in the same mask (for example years, months, days), it is assumed that the more
 1468 detailed units (e.g., the day) are expressed through the order number that they assume within the less
 1469 detailed ones (e.g., the month and the year). For example, if years, weeks and days are used, the weeks are
 1470 within the year (from 1 to 53) and the days are within the year and the week (from 1 to 7).
- 1471 - The position of the digits in the mask denotes the position of the corresponding values; for example,
 1472 YYYYMMDD means four digits for the year followed by two digits for the month and then two digits for the
 1473 day (e.g., 20180405 means the year 2018, month April, day 5th)
- 1474 - Any other character can be used in the mask, meaning simply that it appears in the same position; for
 1475 example, YYYY-MM-DD means that the values of year, month and day are separated by a line (e.g., 2018-
 1476 04-05 means the year 2018, month April, day 5th) and \PMM denotes the letter "P" followed by two
 1477 characters for the month.
- 1478 - The special characters and the special words, if prefixed by the reverse slash (\) in the mask, appear in the
 1479 same position in the time format; for example \PMMM means the letter "P" followed by two characters for
 1480 the month and then the letter "M"; for example, P03M means a period of three months (this is an ISO 8601
 1481 standard representation for a period of MM months). The reverse slash can appear in the format if needed
 1482 by prefixing it with another reverse slash; for example YYYY\\MM means for digits for the year, a reverse
 1483 slash and two digits for the month.

1484 -
 1485 The **special characters** and the corresponding time units are the following:

1486	C	century
1487	Y	year
1488	S	semester
1489	Q	quarter
1490	M	month
1491	W	week
1492	D	day
1493	h	hour digit (by default on 24 hours)
1494	m	minute
1495	s	second
1496	d	decimal of second
1497	P	period indicator (see the "duration" codes below)
1498	p	number of periods

1499
 1500 The **special words** for textual representations are the following:

1501	AM/PM	indicator of AM / PM (e.g. am/pm for “am” or “pm”)
1502	MONTH	textual representation of the month (e.g., JANUARY for January)
1503	DAY	textual representation of the day (e.g., MONDAY for Monday)

1504

1505 **Examples of formatting masks for the *time* scalar type:**

1506 A Scalar Value of type *time* denotes time intervals of any duration and expressed with any precision, which are
 1507 the intervening time between two time points.

1508 These examples are about three possible ISO 8601 formats for expressing time intervals:

- 1509 • Start and end time points, such as "2015-03-03T09:30:45Z/2018-04-05T12:30:15Z"

1510 VTL Mask: YYYY-MM-DDThh:mm:ssZ/YYYY-MM-DDThh:mm:ssZ

- 1511 • Start and duration, such as "2015-03-03T09:30:45-01/P1Y2M10DT2H30M"

1512 VTL Mask: YYYY-MM-DDThh:mm:ss-01/PY\YM\MDD\DT{h}h\HmM\M

- 1513 • Duration and end, such as "P1Y2M10DT2H30M/2018-04-05T12:30:00+02"

1514 VTL Mask: PY\YM\MDD\DT{h}h\HmM\M/YYYY-MM-DDThh:mm:ssZ

1515 Example of other possible ISO formats having accuracy reduced to the day

- 1516 • Start and end, such as "20150303/20180405"

1517 VTL Mask: YYYY-MM-DD/YYYY-MM-DD

- 1518 • Start and duration, such as "2015-03-03/P1Y2M10D"

1519 VTL Mask: YYYY-MM-DD/PY\YM\MDD\D

- 1520 • Duration and end, such as "P1Y2M10D/2018-04-05"

1521 VTL Mask: PY\YM\MDD\DT/YYYY-MM-DD

1522

1523 **Examples of formatting masks for the *date* scalar type:**

1524 A *date* scalar type is a point in time, equivalent to an interval of time having coincident start and end duration
 1525 equal to zero.

1526 These examples about possible ISO 8601 formats for expressing dates:

- 1527 • Date and day time with separators: "2015-03-03T09:30:45Z"

1528 VTL Mask: YYYY-MM-DDThh:mm:ssZ

- 1529 • Date and day time without separators "20150303T093045-01 "

1530 VTL Mask: YYYYMMDDThhmmss-01

1531 Example of other possible ISO formats having accuracy reduced to the day

- 1532 • Date and day-time with separators "2015-03-03/2018-04-05"

1533 VTL Mask: YYYY-MM-DD/YYYY-MM-DD

- 1534 • Start and duration, such as "2015-03-03/P1Y2M10D"

1535 VTL Mask: YYYY-MM-DD/PY\YM\MDD\D

1536

1537 **Examples of formatting masks for the *time_period* scalar type:**

1538 A *time_period* denotes non-overlapping time intervals having a regular duration (for example the years, the
 1539 quarters of years, the months, the weeks and so on). The *time_period* values include the representation of the
 1540 duration of the period.

1541 These examples are about possible formats for expressing time-periods:

- 1542 • Generic time period within the year such as: "2015Q4", "2015M12""2015D365"

1543 VTL Mask: YYYY{ppp} where P is the period indicator and ppp three digits for the number of
 1544 periods, in the values, the period indicator may assume one of the values of the duration scalar type
 1545 listed below.

- 1546 • Monthly period: "2015M03"

1547 VTL Mask: YYYY\MMM

1548

1549 **Examples of formatting masks for the *duration* scalar type:**

1550 A Scalar Value of type *duration* denotes the length of a time interval expressed with any precision and without

1551 connection to any particular time point (for example one year, half month, one hour and fifteen minutes).

1552 These examples are about possible formats for expressing durations (period / frequency)

1553

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

1554	<i>Code</i>	<i>Duration</i>
1555	D	Day
1556	W	Week
1557	M	Month
1558	Q	Quarter
1559	S	Semester
1560	A	Year

1561 VTL Mask: P (period indicator)

1562

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

1563 VTL Mask: \PYY\YM\MDD\DThh\Hmm\Mss\S

1564

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

1565 VTL Mask: \PWWW\W

1566

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

1567 VTL Mask: \PppP

1569 Examples of fixed characters used in the ISO 8601 standard which can appear as fixed characters in the relevant

1570 masks:

1571	P	designator of duration
1572	T	designator of time
1573	Z	designator of UTC zone
1574	“+”	designator of offset from UTC zone
1575	“-“	designator of offset form UTC zone
1576	/	time interval separator

1578 Attribute propagation

1579 The VTL has different default behaviours for Attributes and for Measures, to comply as much as possible with the

1580 relevant manipulation needs. At the Data Set level, the VTL Operators manipulate by default only the Measures

1581 and not the Attributes. At the Component level, instead, Attributes are calculated like Measures, therefore the

1582 algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the Operators. This is

1583 the behaviour of clauses like **calc**, **keep**, **drop**, **rename** and so on, either inside or outside the join (see the

1584 detailed description of these operators in the Reference Manual).

1585 The users which want to automatize the propagation of the Attributes' Values can optionally enforce a

1586 mechanism, called Attribute Propagation rule, whose behaviour is explained in the User Manual (see the section

1587 "Behaviour for Attribute Components"). The adoption of this mechanism is optional, users are free to allow the

1588 attribute propagation rule or not. The users that do not want to allow Attribute propagation rules simply will not

1589 implement what follows.

1590 In short, the automatic propagation of an Attribute depends on a Boolean characteristic, called "virality", which

1591 can be assigned to any Attribute of a Data Set (a viral Attribute has virality = TRUE, a non-viral Attribute has

1592 virality=FALSE, if the virality is not defined, the Attribute is considered as non-viral).

1593 By default, an Attribute propagates from the operand Data Sets (DS_i) to the result Data Set (DS_r) if it is "viral"

1594 at least in one of the operand Data Sets. By default, an Attribute which is viral in one of the operands DS_i is

1595 considered as viral also in the result DS_r.

1596 The Attribute propagation rule does not apply for the time series operators.
1597 The Attribute propagation rule does not apply if the operations on the Attributes to be propagated are explicitly
1598 specified in the expression (for example through the **keep** and **calc** operators). This way it is possible to keep in
1599 the result also Attribute which are non-viral in all the operands, to drop viral Attributes, to override the
1600 (possible) default calculation algorithm of the Attribute, to change the virality of the resulting Attributes.
1601
1602
1603

1604 VTL-ML - General purpose operators

1605 **Parentheses :** **()**

1606

1607 *Syntax*

1608 **(op)**

1609

1610 *Input parameters*

1611 **op** the operand to be evaluated before performing other operations written outside the parentheses.
1612 According to the general VTL rule, operators can be nested, therefore any Data Set, Component or scalar
1613 **op** can be obtained through an expression as complex as needed (for example **op** can be written as the
1614 expression $2 + 3$).

1615

1616 *Examples of valid syntaxes*

1617 **(DS_1 + DS_2)**
1618 **(CMP_1 - CMP_2)**
1619 **(2 + DS_1)**
1620 **(DS_2 - 3 * DS_3)**

1621

1622 *Semantic for scalar operations*

1623 Parentheses override the default evaluation order of the operators that are described in the section “VTL-ML –
1624 Evaluation order of the Operators”. The operations enclosed in the parentheses are evaluated first. For example
1625 $(2+3)*4$ returns 20, instead $2+3*4$ returns 14 because the multiplication has higher precedence than the
1626 addition.

1627

1628 *Input parameters type*

1629 **op ::** dataset
1630 | component
1631 | scalar

1632

1633 *Result type*

1634 **result ::** dataset
1635 | component
1636 | scalar

1637

1638 *Additional constraints*

1639 None.

1640

1641 *Behaviour*

1642 As mentioned, the **op** of the parentheses can be obtained through an expression as complex as needed (for
1643 example **op** can be written as $DS_1 - DS_2$. The part of the expression inside the parentheses is evaluated
1644 before the part outside of the parentheses. If more parentheses are nested, the inner parentheses are evaluated
1645 first, for example $(20 - 10 / (2 + 3)) * 3$ would give 54.

1646

1647 *Examples*

1648 **(DS_1 + DS_2) * DS_3**
1649 **(CMP_1 - CMP_2 / (CMP_3 + CMP_4)) * CMP_5**

1650 **Persistent assignment :** **<-**

1651

1652 *Syntax*

1653 **re <- op**

1654

1655 *Input Parameters*
 1656 re the result
 1657 op the operand. According to the general VTL rule allowing the indentation of the operators, op can be
 1658 obtained through an expression as complex as needed (for example op can be the expression DS_1 -
 1659 DS_2).

1660
 1661 *Examples of valid syntaxes*
 1662 DS_r <- DS_1
 1663 DS_r <- DS_1 - DS_2
 1664

1665 *Semantics for scalar operations*
 1666 empty
 1667

1668 *Input parameters type*
 1669 op :: dataset
 1670

1671
 1672 *Result type*
 1673 result :: dataset
 1674

1675 *Additional constraints*

1676 The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set
 1677 Component. When operations at Component level are invoked, the result is the Data Set which the output
 1678 Components belongs to.

1679 *Behaviour*

1681 The input operand op is assigned to the **persistent** result re, which assumes the same value as op. As mentioned,
 1682 the operand op can be obtained through an expression as complex as needed (for example op can be the
 1683 expression DS_1 - DS_2).

1684 The result re is a persistent Data Set that has the same data structure as the Operand. For example in DS_r <-
 1685 DS_1 the data structure of DS_r is the same as the one of DS_1.

1686 If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value.
 1687 For example, income <- 3 assigns the value 3 to the persistent Data Set named income.

1688
 1689 *Examples*

1690
 1691 Given the operand Data Set DS_1:
 1692

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

1693
 1694 *Example 1:* DS_r <- DS_1 results in:
 1695

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

1696 Non-persistent assignment : :=

1697 *Syntax*

1698 re := op

1699

1700 *Input parameters*

1701 re the result

1702 op the operand (according to the general VTL rule allowing the indentation of the operators, op can be
1703 obtained through an expression as complex as needed (for example op can be the expression DS_1 -
1704 DS_2).

1705

1706 *Examples of valid syntaxes*

1707 DS_r := DS_1

1708 DS_r := 3

1709 DS_r := DS_1 - DS_2

1710 DS_r := 3 + 2

1711

1712 *Semantic for scalar operations*

1713 empty

1714

1715 *Input parameters type*

1716 op :: dataset

1717 | scalar

1718

1719 *Result type*

1720 result :: dataset

1721

1722 *Additional constraints*

1723 The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set
1724 Component. When operations at Component level are invoked, the result is the Data Set which the output
1725 Components belongs to.

1726 The same symbol denoting the non-persistent assignment Operator (:=) is also used inside other operations at
1727 Component level (for example in **calc** and **aggr**) in order to assign the result of the operation to the output
1728 Component: please note that in these cases the symbol := does not denote the non-persistent assignment (i.e.,
1729 this Operator), which cannot operate at Component level, but a special keyword of the syntax of the other
1730 Operator in which it is used.

1731

1732 *Behaviour*

1733 The value of the operand op is assigned to the result re, which is non-persistent and therefore is not stored. As
1734 mentioned, the operand op can be obtained through an expression as complex as needed (for example op can be
1735 the expression DS_1 - DS_2).

1736 The result re is a non-persistent Data Set that has the same data structure as the Operand. For example in DS_r
1737 := DS_1 the data structure of DS_r is the same as the one of DS_1.

1738 If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value.
1739 For example, income := 3 assigns the value 3 to the non-persistent Data Set named income.

1740

1741 *Examples*

1742

1743 Given the operand Data Sets DS_1:

1744

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

1745

1746 *Example 1:* DS_r := DS_1 results in:
1747

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

1748

1749 **Membership :** #

1750

1751

Syntax

1752

ds#comp

1753

1754

Input Parameters

1755

ds the Data Set

1756

comp the Data Set Component

1757

1758

Examples of valid syntaxes

1759

DS_1#COMP_3

1760

1761

Semantic for scalar operations

1762

This operator cannot be applied to scalar values.

1763

1764

Input parameters type

1765

ds :: dataset

1766

comp :: name < component >

1767

1768

Result type

1769

result :: dataset

1770

1771

Additional constraints

1772

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

1773

1774

Behaviour

1775

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

1776

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

1777

If comp is an Identifier or an Attribute Component in ds, then all the existing Measures of ds are dropped in the

1778

result and a new Measure is added. The Data Points' values for the new Measure are the same as the values of

1779

comp in ds. A default conventional name is assigned to the new Measure depending on its type: for example

1780

num_var if the Measure is *numeric*, string_var if it is *string* and so on (the default name can be renamed through

1781

the **rename** operator if needed).

1782

The Attributes follow the Attribute propagation rule as usual (viral Attributes of ds are maintained in the result

1783

as viral, non-viral ones are dropped). If comp is an Attribute, it follows the Attribute propagation rule too.

1784

The same symbol denoting the membership operator (#) is also used inside other operations at Component level

1785

(for example in **join**, **calc**, **aggr**) in order to identify the Components to be operated: please note that in these

1786

cases the symbol # does not denote the membership operator (i.e., this operator, which does not operate at

1787

Component level), but a special keyword of the syntax of the other operator in which it is used.

1788

1789

1790

Examples

1791

Given the operand Data Set DS_1:

1792

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	

1793
1794
1795
1796
1797

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

1798
1799
1800

(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	

1801
1802
1803

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	

1804
1805
1806

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		

1807

1808 User-defined operator call

1809
1810
1811
1812

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

1813 *Input parameters*
1814 operatorName the name of an existing user-defined operator
1815 argument argument passed to the operator

1816
1817 *Examples of valid syntaxes*
1818 max1 (2, 3)

1819
1820 *Semantic for scalar operations*
1821 It depends on the specific user-defined operator that is invoked.

1822
1823 *Input parameters type*
1824 operatorName :: name
1825 argument :: A data type compatible with the type of the parameter of the user-defined operator that
1826 is invoked (see also the “Type syntax” section).

1827
1828
1829 *Result type*
1830 result :: The data type of the result of the user-defined operator that is invoked (see also the
1831 “Type syntax” section).

1832
1833 *Additional constraints*
1834 • operatorName must refer to an operator created with the **define operator** statement.
1835 • The type of each argument value must be compliant with the type of the corresponding parameter of the
1836 user defined operator (the correspondence is in the positional order).

1837
1838 *Behaviour*
1839 The invoked user-defined operator is evaluated. The arguments passed to the operator in the invocation are
1840 associated to the corresponding parameters in positional order, the first argument as the value of the first
1841 parameter, the second argument as the value of the second parameter, and so on. An underscore (“_”) can be
1842 used to denote that the value for an optional operand is omitted. One or more optional operands in the last
1843 positions can be simply omitted.

1844
1845 *Examples*
1846 *Example 1:*
1847
1848 Definition of the max1 operator (see also “define operator” in the VTL-DL):

```
1849           define operator max1 (x integer, y integer)  
1850           returns boolean  
1851           is if x > y then x else y  
1852           end define operator
```

1853
1854
1855 User-defined operator call of the max1 operator:

```
1856           max1 ( 2, 3 )  
1857  
1858
```

1859 Evaluation of an external routine : eval

1860
1861 *Syntax*
1862 **eval** (externalRoutineName ({ argument } { , argument }*), language, **returns** outputType)

1863
1864 *Input parameters*
1865 externalRoutineName the name of an external routine
1866 argument the arguments passed to the external routine
1867 language the implementation language of the routine
1868 outputType the data type of the object returned by eval (see the section: Data type syntax)

1870 *Examples of valid syntaxes*

1871 `eval (routine1 (DS_1))`

1872

1873 *Semantics for scalar operations:*

1874 This is not a scalar operation.

1875

1876 *Input parameters type*

1877 `externalRoutineName :: name`

1878 `argument :: any data type`

1879 `language :: string`

1880 `outputType :: any data type restricting Data Set or scalar`

1881

1882 *Result Type*

1883 `result :: dataset`

1884

1885 *Additional constraints*

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

1899

1900 *Behaviour*

1901 The **eval** operator invokes an external, non-VTL routine, and returns its result as a Data Set or a scalar. The specific data type can be given in the invocation. The routine specified in the **eval** operator can perform any internal logic.

1902

1903

1904

1905

1906 *Examples*

1907 Assuming that SQL3 is an SQL statement which produces DS_r starting from DS_1:

1908

```
1909 DS_r := eval( SQL3( DS_1 ) , "SQL",
1910             returns dataset { identifier<geo_area> ref_area,
1911                             identifier<date> time,
1912                             measure<number> obs_value,
1913                             attribute<string> obs_status } )
```

1913

1914 Assuming that f is an externally defined Java method:

1915

```
1916 DS_r := DS_1[calc Me := eval( f(Me) + 1, "Java", integer) ]
```

1917

1918 Type conversion : **cast**

1919 *Syntax*

```
1920 cast ( op , scalarType { , mask } )
```

1921

1922 *Input parameters*

1923 `op` the operand to be cast

1924 `scalarType` the name of the scalar type into which op has to be converted

1925 `mask` a character literal that specifies the format of op

1926

1927 *Examples of valid syntaxes*

1928 See the examples below.

1929

1930 *Semantics for scalar operations:*

1931 This operator converts the scalar type of `op` to the scalar type specified by `scalarType`. It returns a copy of `op`
1932 converted to the specified `scalarType`.

1933

1934 *Input parameters type*

1935 `op :: dataset{ measure<scalar> _ }`
1936 `| component<scalar>`
1937 `| scalar`

1938 `scalarType :: scalar type` (see the section: Data type syntax)

1939 `mask :: string`

1940

1941 *Result type*

1942 `result :: dataset{ measure<scalar> _ }`
1943 `| component<scalar>`
1944 `| scalar`

1945

1946 *Additional constraints*

- 1947 • Not all the conversions are possible, the specified casting operation is allowed only according to the
1948 semantics described below.
- 1949 • The mask must adhere to one of the formats specified below.

1950

1951 *Behaviour*

1952 **Conversions between basic scalar types**

1953 The VTL assumes that a basic scalar type has a unique internal and more possible external representations
1954 (formats).

1955 The external representations are those of the Value Domains which refers to such a basic scalar types (more
1956 Value Domains can refer to the same basic scalar type, see the VTL Data Types in the User Manual). For example,
1957 there can exist a *boolean* Value Domain which uses the values TRUE and FALSE and another *boolean* Value
1958 Domain which uses the values 1 and 0. The external representations are the ones of the Data Point Values and
1959 are obviously known by users.

1960 The unique internal representation of a basic scalar type, instead, is used by the **cast** operator as a technical
1961 expedient to make the conversion between external representations easier: not necessarily users are aware of it.
1962 In a conversion, the **cast** converts the source external representation into the internal representation (of the
1963 corresponding scalar type), then this last one is converted into the target external representation (of the target
1964 type). As mentioned in the User Manual, VTL does not prescribe any specific internal representation for the
1965 various scalar types, leaving different organisations free of using their preferred or already existing ones.

1966 In some cases, depending on the type of `op`, the output `scalarType` and the invoked operator, an automatic
1967 conversion is made, that is, even without the explicit invocation of the **cast** operator: this kind of conversion is
1968 called **implicit casting**.

1969 In other cases, more than all when the implicit casting is not possible, the type conversion must be specified
1970 explicitly through the invocation of the **cast** operator: this kind of conversion is called **explicit casting**. If an
1971 explicit casting is specified, the (possible) implicit casting is overridden. There are two main categories of
1972 implicit casting:

- 1973 • **“Explicit with mask”**: the explicit conversion requires a formatting mask that specifies how the actual
1974 casting is performed;
- 1975 • **“Explicit w/o mask”**: the explicit conversion does not requires a formatting mask.

1976 The table below summarises the possible castings between the basic scalar types. In particular, the input type is
1977 specified in the first column (row headings) and the output type in the first row (column headings).

1978

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

1979

1980 The type of casting can be personalised in specific environments, provided that the personalisation is explicitly
1981 documented with reference to the table above. For example, assuming that an explicit **cast** with mask is
1982 required and that in a specific environment a definite **mask** is used for such a kind of conversions, the **cast** can
1983 also become implicit provided that the **mask** that will be applied is specified.

1984 The **implicit casting** is performed when a value of a certain type is provided when another type is expected. Its
1985 behaviour is described here:

- 1986 • From **integer** to **number**: an *integer* is provided when a *number* is expected (for example, an *integer* and a
1987 *number* are passed as inputs of a n-ary numeric operator); it returns a *number* having the integer part equal
1988 to the *integer* and the decimal part equal to zero;
- 1989 • From **integer** to **string**: an *integer* is provided when a *string* is expected (for example, an *integer* is passed
1990 as an input of a *string* operator); it returns a *string* having the literal value of the *integer*;
- 1991 • From **number** to **string**: a *number* is provided when a *string* is expected; it returns the *string* having the
1992 literal value of the *number*; the decimal separator is converted into the character "." (dot).
- 1993 • From **boolean** to **string**: a *boolean* is provided when a *string* is expected; the boolean value TRUE is
1994 converted into the *string* "TRUE" and FALSE into the *string* "FALSE";
- 1995 • From **date** to **time**: a *date* (point in time) is provided when a *time* is expected (interval of time): the
1996 conversion results in an interval having the same start and end, both equal to the original *date*;
- 1997 • From **time_period** to **time**: a *time_period* (a regular interval of *time*, like a month, a quarter, a year ...) is
1998 provided when a *time* (any interval of time) is expected; it returns a *time* value having the same start and
1999 end as the *time_period* value.

2000 An implicit cast is also performed from a **value domain type** or a **set type** to a **basic scalar type**: when a *scalar*
2001 value belonging to a Value Domains or a Set is involved in an operation (i.e., provided as input to an operator),
2002 the value is implicitly cast into the basic scalar type which the Value Domain refers to (for this relationship, see
2003 the description of Type System in the User Manual). For example, assuming that the Component *birth_country* is
2004 defined on the Value Domain *country*, which contains the ISO 3166-1 numeric codes and therefore refers to the
2005 basic scalar type *integer*, the (possible) invocation `length(birth_country)`, which calculates the length of the input
2006 string, automatically casts the values of *birth_country* into the corresponding string. If the basic scalar type of the
2007 Value Domain is not compatible with the expression where it is used, an error is raised. This VTL feature is
2008 particularly important as it provides a general behaviour for the Value Domains and relevant Sets, preventing
2009 from the need of defining specific behaviours (or methods or operations) for each one of them. In other words,
2010 all the Values inherit the operations that can be performed on them from the basic scalar types of the respective
2011 Value Domains.

2012 The **cast** operator can be invoked explicitly even for the conversions which allow an implicit cast and in this case
2013 the same behaviour as the implicit cast is applied.

2014 The behaviour of the **cast** operator for the conversions that require **explicit casting without mask** is the
2015 following:

- 2016 • From **integer** to **boolean**: if the *integer* is different from 0, then TRUE is returned, FALSE otherwise.
- 2017 • From **number** to **integer**: converts a *number* with no decimal part into an *integer*; if the decimal part is
2018 present, a runtime error is raised.
- 2019 • From **number** to **boolean**: if the *number* is different from 0.0, then TRUE is returned, FALSE otherwise.

- 2020 • From **boolean** to **integer**: TRUE is converted into 1; FALSE into 0.
 - 2021 • From **boolean** to **number**: TRUE is converted into 1.0; FALSE into 0.0.
 - 2022 • From **date** to **time_period**: it converts a *date* into the corresponding daily value of *time_period*.
 - 2023 • From **string** to **integer**: the *integer* having the literal value of the *string* is returned; if the *string* contains a
2024 literal that cannot be matched to an *integer*, a runtime error is raised.
 - 2025 • From **string** to **time_period**: it converts a *string* value to a *time_period* value.
- 2026 When an **explicit casting with mask** is required, the conversion is made by applying the formatting mask which
2027 specifies the meaning of the characters in the output *string*. The formatting Masks are described in the section
2028 “VTL-ML – Typical Behaviour of the ML Operators”, sub-section “Type Conversion and Formatting Mask.
- 2029 The behaviour of the **cast** operator for such conversions is the following:
- 2030 • From **time** to **string**: it is applied the *time* formatting mask.
 - 2031 • From **date** to **string**: it is applied the *time_period* formatting mask.
 - 2032 • From **time_period** to **date**: it is applied a formatting mask which accepts two possible values (“START”,
2033 “END”). If “START” is specified, then the *date* is set to the beginning of the *time_period*; if “END” is specified,
2034 then the *date* is set to the end of the *time_period*.
 - 2035 • From **time_period** to **string**: it is applied the *time_period* formatting mask.
 - 2036 • From **duration** to **string**: a *duration* (an absolute time interval) is provided when a *string* is expected; it
2037 returns the *string* having the default *string* representation for the *duration*.
 - 2038 • From **string** to **number**: the *number* having the literal value of the *string* is returned; if the *string* contains a
2039 literal that cannot be matched to a *number*, a runtime error is raised. The *number* is generated by using a
2040 *number* formatting mask.
 - 2041 • From **string** to **time**: the *time* having the literal value of the *string* is returned; if the *string* contains a literal
2042 that cannot be matched to a *date*, a runtime error is raised. The *time* value is generated by using a *time*
2043 formatting mask.
 - 2044 • From **string** to **duration**: the *duration* having the literal value of the *string* is returned; if the *string* contains
2045 a literal that cannot be matched to a *duration*, a runtime error is raised. The *duration* value is generated by
2046 using a time formatting mask.

2047 **Conversions between basic scalar types and Value Domains or Set types**

2048 A value of a basic *scalar* type can be converted into a value belonging to a Value Domain which refers to such a
2049 *scalar* type. The resulting *scalar* value must be one of the allowed values of the Value Domain or Set; otherwise, a
2050 runtime error is raised. This specific use of **cast** operators does not really correspond to a type conversion; in
2051 more formal terms, we would say that it acts as a constructor, i.e., it builds an instance of the output type. Yet,
2052 towards a homogeneous and possibly simple definition of VTL syntax, we blur the distinction between
2053 constructors and type conversions and opt for a unique formalism. An example is given below.

2054 **Conversions between different Value Domain types**

2055 As a result of the above definitions, conversions between values of different Value Domains are also possible.
2056 Since an element of a Value Domain is implicitly cast into its corresponding basic scalar type, we can build on it
2057 to turn the so obtained scalar type into another Value Domain type. Of course, this latter Value Domain type must
2058 use as a base type this scalar type.

2059

2060 *Examples*

2061

2062 Example 1: from *string* to *number*

```
2063 ds2 := ds1[calc m2 := cast(m1, number, "DD.DDD") + 2 ]
```

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

2066

2067 Example 2: from *string* to *date*

```
2068 ds2 := ds1[calc m2 := cast(m1, date, "YYYY-MM-DD") ]
```

2069 In this case we use explicit cast from *string* to *date*. The mask is used to specify how the *string* must be
2070 interpreted in the conversion.

2071

2072 Example 3: from *number* to *integer*

2073 `ds2 := ds1[calc m2 := cast(m1, integer) + 3]`

2074 In this case we cast a *number* into an *integer*, no mask is required.

2075

2076 Example 4: from *number* to *string*

2077 `ds2 := ds1[calc m2 := length(cast(m1, string))]`

2078 In this case we cast a *number* into a *string*, no mask is required.

2079

2080 Example 5: from *date* to *string*

2081 `ds2 := ds1[calc m2 := cast(m1, string, "YY-MON-DAY hh:mm:ss")]`

2082 In this example a *date* instant is turned into a *string*. The mask is used to specify the *string* layout.

2083

2084 Example 6: from *string* to *GEO_AREA*

2085 `ds2 := ds1[calc m2 := cast(GEO_STRING, GEO_AREA)]`

2086 In this example we suppose we have elements of Value Domain Subset for *GEO_AREA*. Let *GEO_STRING* be a

2087 string Component of Data Set *ds1* with string values compatible with the *GEO_AREA* Value Domain Subset.

2088 Thus, the following expression moves *ds1* data into *ds2*, explicitly casting strings to geographical areas.

2089

2090 Example 7: from *GEO_AREA* to *string*

2091 `ds2 := ds1[calc m2 := length(GEO_AREA)]`

2092 In this example we use a Component *GEO_AREA* in a *string* expression, which calculates the length of the

2093 corresponding *string*; this triggers the automatic cast.

2094

2095 Example 8: from *GEO_AREA2* to *GEO_AREA1*

2096 `ds2 := ds1 [calc m2 := cast (GEO, GEO_AREA1)]`

2097 In this example we suppose we have to compare elements two Value Domain Subsets, They are both defined on

2098 top of Strings. The following cast expressions performs the conversion.

2099 Now, Component *GEO* is of type *GEO_AREA2*, then we specify it has to be cast into *GEO_AREA1*. As both

2100 work on *strings* (and the values are compatible), the conversion is feasible. In other words, the cast of an

2101 operand into *GEO_AREA1* would expect a *string*. Then, as *GEO* is of type *GEO_AREA2*, defined on top of

2102 *strings*, it is implicitly cast to the respective *string*; this is compatible with what cast expects and it is then able to

2103 build a value of type *GEO_AREA1*.

2104

2105 Example 9: from *string* to *time_period*

2106 In the following examples we convert from *strings* to *time_periods*, by using appropriate masks.

2107 The first quarter of year 2000 can be expressed as follows (other examples are possible):

2108 `cast ("2000Q1", time_period, "YYYY\QQ")`

2109 `cast ("2000-Q1", time_period, "YYYY-\QQ")`

2110 `cast ("2000-1", time_period, "YYYY-Q")`

2111 `cast ("Q1-2000", time_period, "\QQ-YYYY")`

2112 `cast ("2000Q01", time_period, "YYYY\QQQ")`

2113 Examples of daily data:

2114 `cast ("2000M01D01", time_period, "YYYY\MMM\DDD")`

2115 `cast ("2000.01.01", time_period, "YYYY\.\MM\.\DD")`

2116

2117

VTL-ML - Join operators

2118 The Join operators are fundamental VTL operators. They are part of the core of the language and allow to obtain
 2119 the behaviour of the majority of the other non-core operators, plus many additional behaviours that cannot be
 2120 obtained through the other operators.

2121 The Join operators are four, namely the `inner_join`, the `left_join`, the `full_join` and the `cross_join`. Because their
 2122 syntax is similar, they are described together.

2123 **Join :** **inner_join, left_join, full_join, cross_join**

2124 *Syntax*

```
2125     joinOperator ( ds1 { as alias1 } { , dsN { as aliasN } }* { using usingComp { , usingComp }* }
2126         { filter filterCondition }
2127         { apply applyExpr
2128           | calc calcClause
2129           | aggr aggrClause { groupingClause } }
2130         { keep comp { , comp }* | drop comp { , comp }* }
2131         { rename compFrom to compTo { , compFrom to compTo }* }
2132     )
```

2133 joinOperator ::= { **inner_join** | **left_join** | **full_join** | **cross_join** }¹

2134 calcClause ::= { calcRole } calcComp := calcExpr
 2135 { , { calcRole } calcComp := calcExpr }*

2136 calcRole ::= { **identifier** | **measure** | **attribute** | **viral attribute** }¹

2137 aggrClause ::= { aggrRole } aggrComp := aggrExpr
 2138 { , { aggrRole } aggrComp := aggrExpr }*

2139 aggrRole ::= { **measure** | **attribute** | **viral attribute** }¹

2140 groupingClause ::= { **group by** groupingId { , groupingId }*
 2141 | **group except** groupingId { , groupingId }*
 2142 | **group all** conversionExpr }¹
 2143 { **having** havingCondition }

2144

2145

2146 *Input parameters*

2147 <u>joinOperator</u>	the Join operator to be applied
2148 ds1, ..., dsN	the Data Set operands (at least one must be present)
2149 alias1, ..., aliasN	optional aliases for the input Data Sets, valid only within the “join” operation to make it easier to refer to them. If omitted, the Data Set name must be used.
2150	
2151 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)
2152	
2153	
2154 filterCondition	a condition (<i>boolean</i> expression) at component level, having only Components of the input Data Sets as operands, which is evaluated for each joined Data Point and filters them (when TRUE the joined Data Point is kept, otherwise it is not kept)
2155	
2156	
2157 applyExpr	an expression, having the input Data Sets as operands, which is pairwise applied to all their homonym Measure Components and produces homonym Measure Components in the result; for example if both the Data Sets ds1 and ds2 have the <i>numeric</i> measures m1 and m2, the clause <code>apply ds1 + ds2</code> would result in calculating <code>m1 := ds1#m1 + ds2#m1</code> and <code>m2 := ds1#m2 + ds2#m2</code>
2158	
2159	
2160	
2161	
2162 <u>calcClause</u>	clause that specifies the Components to be calculated, their roles and their calculation algorithms, to be applied on the joined and filtered Data Points.
2163	
2164 <u>calcRole</u>	the role of the Component to be calculated
2165 calcComp	the name of the Component to be calculated

2166	calcExpr	expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component
2167		
2168	<u>aggrClause</u>	clause that specifies the required aggregations, i.e., the aggregated Components to be calculated, their roles and their calculation algorithm, to be applied on the joined and filtered Data Points
2169		
2170		
2171	<u>aggrRole</u>	the role of the aggregated Component to be calculated; if omitted, the Measure role is assumed
2172		
2173	aggrComp	the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)
2174		
2175	aggrExpr	expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. avg , count , max ... , see also the corresponding sections) to perform the desired aggregation. Note that the count operator is used in an <u>aggrClause</u> without parameters, e.g.:
2176		
2177		
2178		
2179		DS_1 [aggr Me_1 := count () group by Id_1]
2180	<u>groupingClause</u>	the following alternative grouping options:
2181		group by the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.
2182		
2183		group except the Data Points are grouped by the values of the Identifiers not specified as groupingId. The specified Identifiers are dropped in the result.
2184		
2185		
2186		group all converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.
2187		
2188	groupingId	Identifier Component to be kept (in the group by clause) or dropped (in the group except clause).
2189		
2190	conversionExpr	specifies a conversion operator (e.g. time_agg) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set op.
2191		
2192		
2193	havingCondition	a condition (<i>boolean</i> expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which havingCondition evaluates to TRUE appear in the result. The havingCondition refers to the groups specified through the groupingClause, therefore it must invoke aggregate operators (e.g. avg , count , max , ..., see also the section Aggregate invocation). A correct example of havingCondition is max(obs_value) < 1000 , while the condition obs_value < 1000 is not a right havingCondition, because it refers to the values of single Data Points and not to the groups. The count operator is used in a havingCondition without parameters, e.g.:
2194		
2195		
2196		
2197		
2198		
2199		
2200		
2201		
2202		sum (ds group by id1 having count () >= 10)
2203	comp	dependent Component (Measure or Attribute, not Identifier) to be kept (in the keep clause) or dropped (in the drop clause)
2204		
2205	compFrom	the original name of the Component to be renamed
2206	compTo	the new name of the Component after the renaming
2207		
2208		<i>Examples of valid syntaxes</i>
2209	inner_join (ds1 as d1, ds2 as d2 using Id1, Id2	
2210	filter d1#Me1 + d2#Me1 <10	
2211	apply d1 / d2	
2212	keep Me1, Me2, Me3	
2213	rename Id1 to Id10, Id2 to Id20	
2214)	
2215		
2216	left_join (ds1 as d1, ds2 as d2	
2217	filter d1#Me1 + d2#Me1 <10,	
2218	calc Me1 := d1#Me1 + d2#Me3,	
2219	keep Me1	
2220	rename Id1 to Id10, Me1 to Me10	
2221)	
2222		
2223	full_join (ds1 as d1, ds2 as d2	
2224	filter d1#Me1 + d2#Me1 <10,	

```

2225         aggr Me1 := sum(Me1), attribute At20 := avg(Me2)
2226         group by Id1, Id2
2227         having sum(Me3) > 0
2228     )
2229 
```

Semantics for scalar operations

The join operator does not perform scalar operations.

Input parameters type

```

2234 ds1, ..., dsN ::      dataset
2235 alias1, ..., aliasN :: name
2236 usingId ::           name < component >
2237 filterCondition ::   component<boolean>
2238 applyExpr ::         dataset
2239 calcComp ::          name < component >
2240 calcExpr ::          component<scalar>
2241 aggrComp ::          name < component >
2242 aggrExpr ::          component<scalar>
2243 groupingId ::        name < identifier >
2244 conversionExpr ::    component<scalar>
2245 havingCondition ::   component<boolean>
2246 comp ::              name < component >
2247 compFrom ::          component<scalar>
2248 compTo ::            component<scalar>
2249 
```

Result type

```

2251 result ::            dataset
2252 
```

Additional constraints

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

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

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

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

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

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

- 2281 • An Identifier included in the **group except** clause (if any) cannot be included in the **rename** clause. If the
- 2282 **aggr** clause is invoked and the grouping clause is omitted, no Identifier can be included in the **rename**
- 2283 clause
- 2284 • A dependent Component not included in the **keep** clause (if any) cannot be renamed
- 2285 • A dependent Component included in the **drop** clause (if any) cannot be renamed

2286

2287 *Behaviour*

2288 The **semantics of the join operators** can be procedurally described as follows.

- 2289 • A relational join of the input operands is performed, according to SQL inner (**inner_join**), left-outer
- 2290 (**left_join**), full-outer (**full_join**) and Cartesian product (**cross_join**) semantics (these semantics will be
- 2291 explained below), producing an intermediate internal result, that is a Data Set that we will call “virtual”
- 2292 (VDS₁).
- 2293 • The filterCondition, if present, is applied on VDS₁, producing the Virtual Data Set VDS₂.
- 2294 • The specified calculation algorithms (**apply**, **calc** or **aggr**), if present, are applied on VDS₂. For the
- 2295 Attributes that have not been explicitly calculated in these clauses, the Attribute propagation rule is applied
- 2296 (see the User Manual), so producing the Virtual Data Set VDS₃.
- 2297 • The **keep** or **drop** clause, if present, is applied on VDS₃, producing the Virtual Data Set VDS₄.
- 2298 • The **rename** clause, if present, is applied on VDS₄, producing the Virtual Data Set VDS₅.
- 2299 • The final automatic alias removal is performed in order to obtain the output Data Set.

2300 An alias can be optionally declared for each input Data Set. The aliases are valid only within the “join” operation,

2301 in particular to allow joining a dataset with itself (self join). If omitted, the input Data Sets are referenced only

2302 through their Data Set names. If the aliases are ambiguous (for example duplicated or equal to the name of

2303 another Data Set), an error is raised.

2304 The **structure of the virtual Data Set** VDS₁ which is the output of the relational join is the following.

2305 For the **inner_join**, the **left_join** and the **full_join**, the virtual Data Set contains the following Components:

- 2306 • The Components used as join keys, which appear once and maintain their original names and roles. In
- 2307 the cases A and B1, all of them are Identifiers. In the sub-case B2, the result takes the roles from the
- 2308 reference Data Set.
- 2309 • In the sub-case B2: the Identifiers of the reference Data Set, which appear once and maintain their
- 2310 original name and role.
- 2311 • The other Components coming from exactly one input Data Set, which appear once and maintain their
- 2312 original name
- 2313 • The other Components coming from more than one input Data Set, which appears as many times as the
- 2314 Data Set they come from; to distinguish them, their names are prefixed with the alias (or the name) of
- 2315 the Data Set they come from, separated by the “#” symbol (e.g., ds#cmpj). For example, if the
- 2316 Component “population” appears in two input Data Sets “ds1” and “ds2” that have the aliases “a” and
- 2317 “b” respectively, the Components “a#population” and “b#population” will appear in the virtual Data Set.
- 2318 If the aliases are not defined, the two Components are prefixed with the Data Set name (i.e.,
- 2319 “ds1#population” and “ds2#population”). In this context, the symbol “#” does not denote the
- 2320 membership operator but acts just as a separator between the the Data Set and the Component names.
- 2321 • If the same Data Set appears more times as operand of the join (self-join) and the aliases are not defined,
- 2322 an exception is raised because it is not allowed that two or more Components in the virtual Data Set
- 2323 have the same name. In the self-join the aliases are mandatory to disambiguate the Component names.
- 2324 • If a Data Set in the join list is the result of a sub-expression, then an alias is mandatory all the same
- 2325 because this Data Set has no name. If the alias is omitted, an exception is raised.

2326 As for the **cross_join**, the virtual Data Set contains all the Components from all the operands, possibly prefixed

2327 with the aliases to avoid ambiguities.

2328 The **semantics of the relational join** is the following.

2329 The join is performed on some join keys, which are the Components of the input Data Sets whose values are used

2330 to match the input Data Points and produce the joined output Data Points.

2331 By default (only for the **full_join** and the **cross_join**), the join is performed on the subset of homonym Identifier

2332 Components of the input Data Sets.

2333 The parameter **using** allows to specify different join keys than the default ones, and can be used only for the

2334 **inner_join** and the **left_join** in order to preserve the functional behaviour of the operations.

2335 The different kinds of relational joins behave as follows.

- 2336 • **inner_join**: the Data Points of ds1, ..., dsN are joined if they have the same values for the common
- 2337 Identifier Components or, if the **using** clause is present, for the specified Components. A (joined) virtual
- 2338 Data Point is generated in the virtual Data Set VDS₁ when a matching Data Point is found for each one of the
- 2339 input Data Sets. In this case, the Values of the Components of a virtual Data Point are taken from the

2340 corresponding Components of the matching Data Points. If there is no match for one or more input Data Sets,
2341 no virtual Data Point is generated.

- 2342 • **left_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from
2343 the left side and proceeding towards the right side. The Data Points are matched like in the **inner_join**, but a
2344 virtual Data Point is generated even if no Data Point of the right Data Set matches (in this case, the Measures
2345 and Attributes coming from the right Data Set take the NULL value in the virtual Data Set). Therefore, for
2346 each Data Points of the left Data Set a virtual Data Point is always generated. These stepwise operations are
2347 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
2348 first “i” operands and ds_{i+1} is the $i+1^{th}$ operand. For each pair $\langle ds_i, ds_{i+1} \rangle$, the joined Data Set is fed with all
2349 the Data Points that match in ds_i and ds_{i+1} or are only in ds_i . The constraints described above guarantee the
2350 absence of null values for the Identifier Components of the joined Data Set, whose values are always taken
2351 from the left Data Set. If the join succeeds for a Data Point in ds_i , the values for the Measures and the
2352 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
2353 the Data Point in ds_i , null values are given to Measures and Attributes coming only from ds_{i+1} .
- 2354 • **full_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from
2355 the left side and proceeding toward the right side. The Data Points are matched like in the **inner_join** and
2356 **left_join**, but the **using** clause is not allowed and a virtual Data Point is generated either if no Data Point of
2357 the right Data Set matches with the left Data Point or if no Data Point of the left Data Set matches with the
2358 right Data Point (in this case, Measures and Attributes coming from the non matching Data Set take the NULL
2359 value in the virtual Data Set). Therefore, for each Data Points of the left and the right Data Set, a virtual Data
2360 Point is always generated. These stepwise operations are associative. More formally, consider the generic
2361 pair $\langle ds_i, ds_{i+1} \rangle$, where ds_i is the result of the **full_join** of the first “i” operands and ds_{i+1} is the $i+1^{th}$ operand.
2362 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
2363 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
2364 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
2365 Data Point in ds_i , NULL values are given to Measures and Attributes coming only from ds_{i+1} . Symmetrically, if
2366 no Data Point in ds_i matches the Data Point in ds_{i+1} , NULL values are given to Measures and Attributes
2367 coming only from ds_i . The constraints described above guarantee the absence of NULL values on the
2368 Identifier Components. As mentioned, the **using** clause is not allowed in this case.
- 2369 • **cross_join**: the join is performed stepwise, between consecutive pairs of input Data Sets, starting from the
2370 left side and proceeding toward the right side. No match is performed but the Cartesian product of the input
2371 Data Points is generated in output. These stepwise operations are associative. More formally, consider the
2372 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
2373 $i+1$ -th operand. For each pair $\langle ds_i, ds_{i+1} \rangle$, the resulting Data Set is fed with the Data Points obtained as the
2374 Cartesian product between the Data Points of ds_i and ds_{i+1} . The resulting Data Set will have all the
2375 Components from ds_i and ds_{i+1} . For the Data Sets which have at least one Component in common, the alias
2376 parameter is mandatory. As mentioned, the **using** parameter is not allowed in this case.

2377

2378 The **semantics of the clauses** is the following.

- 2379 • **filter** takes as input a Boolean Component expression (having type *component<boolean>*). This clause
2380 filters in or out the input Data Points; when the expression is TRUE the Data Point is kept, otherwise it is
2381 not kept in the result. Only one **filter** clause is allowed.
- 2382 • **apply** combines the homonym Measures in the source operands whose type is compatible with the
2383 operators used in **applyExpr**, generating homonym Measures in the output. The expression **applyExpr**
2384 can use as input the names or aliases of the operand Data Sets. It applies the expression to all the n-uples
2385 of homonym Measures in the input Data Sets producing in the target a single homonym Measure for
2386 each n-uple. It can be thought of as the multi-measure version of the **calc**. For example, if the following
2387 aliases have been declared: d1, d2, d3, then the following expression $d1+d2+d3$, sums all the homonym
2388 Measures in the three input Data Sets, say M1 and M2, so as to obtain in the result: $M1 := d1\#M1 +$
2389 $d2\#M1 + d3\#M1$ and $M2 := d1\#M2 + d2\#M2 + d3\#M2$. It is not only a compact version of a multiple
2390 **calc**, but also essential when the number of Measures in the input operands is not known beforehand.
2391 Only one **apply** clause is allowed.
- 2392 • **calc** calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at
2393 Component level. Each Component is calculated through an independent sub-expression. It is possible
2394 to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral**
2395 **attribute**, therefore the **calc** clause can be used also to change the role of a Component when possible.
2396 The keyword **viral** allows controlling the virality of Attributes (for the Attribute propagation rule see the
2397 User Manual). The following rule is used when the role is omitted: if the component exists in the
2398 operand Data Set then it maintains that role; if the component does not exist in the operand Data Set
2399 then the role is **measure**. The **calcExpr** are independent one another, they can only reference

2400 Components of the input Virtual Data Set and cannot use Components generated, for example, by other
2401 `calcExpr`. If the calculated Component is a new Component, it is added to the output virtual Data Set. If
2402 the Calculated component is a Measure or an Attribute that already exists in the input virtual Data Set,
2403 the calculated values overwrite the original values. If the Calculated component is an Identifier that
2404 already exists in the input virtual Data Set, an exception is raised because overwriting an Identifier
2405 Component is forbidden for preserving the functional behaviour. Analytic operators can be used in the
2406 **calc** clause.

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

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

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

group all converts an Identifier Component using `conversionExpr` and keeps all the resulting Identifiers.

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

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

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

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

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

2459 input Data Set have homonym Measures and there is no apply clause which unifies them; these Measures can be
 2460 renamed or removed.

2461
 2462 *Examples*

2463
 2464 Given the operand Data Sets DS_1 and DS_2:
 2465

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

2466

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

2467

2468

2469 *Example 1:*

2470 DS_r := inner_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

2473

2474

2475 *Example 2:*

2476 DS_r := left_join (DS_1 as d1, DS_2 as d2,
 2477 keep Me_1, d2#Me_2, Me_1A) results in:

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

2478

2479

2480 *Example 3:*

2481 DS_r := full_join (DS_1 as d1, DS_2 as d2,
 2482 keep Me_1, d2#Me_2, Me_1A) results in:

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

3	A	null	M	Z
---	---	------	---	---

2483
2484
2485
2486
2487
2488
2489

Example 4:

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

results in:

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

2490
2491
2492
2493
2494
2495
2496
2497
2498
2499

Example 5:

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

where || is the string concatenation,

results in:

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

2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510

Example 6:

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

where || is the string concatenation,

results in:

DS_r			
Id_1	Id_2	Me_1	Me_2
1	B	C	D_NEW

2511
2512
2513
2514
2515

Example 7:

Given the operand Data Sets DS_1 and DS_2:

2516

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

2517
2518
2519
2520
2521

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

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

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

2522
2523
2524

2525 VTL-ML - String operators

2526 String concatenation : ||

2527

2528 *Syntax*

2529 op1 || op2

2530

2531 *Input Parameters*

2532 op1, op2 the operands

2533

2534 *Examples of valid syntaxes*

2535 "Hello" || ", world!"

2536 ds_1 || ds_2

2537

2538 *Semantics for scalar operations*

2539 Concatenates two strings. For example, "Hello" || ", world!" gives "Hello, world!"

2540

2541 *Input parameters type*

2542 op1, op2 :: dataset { measure<string> _+ }

2543 | component<string>

2544 | string

2545

2546 *Result type*

2547 result :: dataset { measure<string> _+ }

2548 | component<string>

2549 | string

2550

2551 *Additional constraints*

2552 None.

2553

2554 *Behaviour*

2555 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”).

2556

2557 *Examples*

2558 Given the Data_Sets DS_1 and DS_2:

2560

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

2561

2562

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

2563

2564 *Example 1:* DS_r := DS_1 || DS_2 results in:

2565

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

2566
 2567 *Example 2 (on component):* DS_r := DS_1[calc Me_2:= Me_1 || " world"] results in:
 2568

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

2569 **Whitespace removal : trim, rtrim, ltrim**

2570 *Syntax*
 2571 {trim|ltrim|rtrim}¹ (op)

2572 *Input parameters*
 2573 op the operand

2574 *Examples of valid syntaxes*
 2575 trim("Hello ")
 2576 trim(ds_1)

2577 *Semantics for scalar operations*
 2578 Removes trailing or/and leading whitespace from a string. For example, trim("Hello ") gives "Hello".

2579 *Input parameters type*
 2580 op :: dataset { measure<string> _+ }
 2581 | component<string>
 2582 | string

2583 *Result type*
 2584 result :: dataset { measure<string> _+ }
 2585 | component<string>
 2586 | string

2587 *Additional constraints*
 2588 None.

2589 *Behaviour*
 2590 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”).

2591 *Examples*
 2592 Given the Data Set DS_1:
 2593
 2594

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

2605 *Example 1:* DS_r := rtrim(DS_1) results in:
 2606

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

2607
 2608 *Example 2 (on component):* DS_r := DS_1[calc Me_2:= rtrim(Me_1)] results in:
 2609

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

2610 Character case conversion : upper/lower

2611 *Syntax*
 2612 {upper | lower}¹ (op)

2613
 2614 *Input Parameters*
 2615 op the operand

2616
 2617 *Examples of valid syntaxes*
 2618 upper("Hello")
 2619 lower(ds_1)

2620
 2621 *Semantics for scalar operations*
 2622 Converts the character case of a string in upper or lower case. For example, upper("Hello") gives "HELLO".

2623
 2624 *Input Parameters type*
 2625 op :: dataset { measure<string> _+ }
 2626 | component<string>
 2627 | string

2628
 2629 *Result type*
 2630 result :: dataset { measure<string> _+ }
 2631 | component<string>
 2632 | string

2633
 2634 *Additional constraints*
 2635 None.

2636
 2637 *Behaviour*
 2638 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set
 2639 Component” (see the section “Typical behaviours of the ML Operators”).

2640
 2641 *Examples*
 2642 Given the Data Set DS_1:
 2643

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

2644
 2645 *Example 1:* DS_r := upper(DS_1) results in:
 2646

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

2647
 2648 *Example 2 (on component):* DS_r := DS_1[calc Me_2:= upper(Me_1)] results in:
 2649

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

2650

2651 Sub-string extraction : **substr**

2652 *Syntax*
 2653 **substr (op, start, length)**

2654
 2655 *Input parameters*
 2656 op the operand
 2657 start the starting digit (first character) of the string to be extracted
 2658 length the length (number of characters) of the string to be extracted
 2659

2660
 2661 *Examples of valid syntaxes*
 2662 substr (DS_1, 2 , 3)
 2663 substr (DS_1, 2)
 2664 substr (DS_1, _ , 3)
 2665 substr (DS_1)

2666
 2667 *Semantics for scalar operations*
 2668 The operator extracts a substring from op, which must be *string* type. The substring starts from the startth
 2669 character of the input string and has a number of characters equal to the length parameter.
 2670

- 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.

2671
 2672
 2673
 2674
 2675 For example:
 2676 substr ("abcdefghijklmnopqrstuvwxy", start:= 5 , length:= 10) gives: "efghijklmn".
 2677 substr ("abcdefghijklmnopqrstuvwxy", start:= 25 , length:= 10) gives: "yz".
 2678 substr ("abcdefghijklmnopqrstuvwxy", start:= 30 , length:= 10) gives: "".

2679
 2680 *Input parameters type*
 2681 op :: dataset { measure <string> _+ }
 2682 | component <string>
 2683 | string
 2684
 2685 start :: component < integer [value >= 1] >
 2686 | integer [value >= 1]
 2687

2688
 2689 length :: component < integer [value >= 0] >
 2690 | integer [value >= 0]

2693 *Result type*

2694 result :: dataset { measure<string> _+ }
 2695 | component<string>
 2696 | string

2699 *Additional constraints*

2700 None.

2702 *Behaviour*

2703 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
 2704 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
 2705 the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the
 2706 section “Typical behaviours of the ML Operators”).

2708 *Examples*

2709 Given the operand Data Set DS_1:
 2710

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

2712 *Example 1:* DS_r:= substr (DS_1 , 7) results in:
 2713
 2714

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"

2715 *Example 2:* DS_r:= substr (DS_1 , 1 , 5) results in:
 2716
 2717

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

2718 *Example3(on Components):* DS_r:= DS_1 [calc Me_2:= substr (Me_2 , 1 , 5)] results in:
 2719
 2720

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

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

2721

2722 String pattern replacement: `replace`

2723 *Syntax*

2724 `replace (op , pattern1, pattern2)`

2725

2726 *Input parameters*

2727 `op` the operand

2728 `pattern1` the pattern to be replaced

2729 `pattern2` the replacing pattern

2730

2731 *Examples of valid syntaxes*

2732 `replace(DS_1, "Hello", "Hi")`

2733 `replace(DS_1, "Hello")`

2734

2735 *Semantics for scalar operations*

2736 Replaces all the occurrences of a specified string-pattern (`pattern1`) with another one (`pattern2`). If `pattern2` is
 2737 omitted then all occurrences of `pattern1` are removed. For example:

2738

2739 `replace("Hello world", "Hello", "Hi")` gives "Hi world"

2740 `replace("Hello world", "Hello")` gives " world"

2741 `replace ("Hello", "ello", "i")` gives "Hi"

2742

2743 *Input parameters type*

2744 `op ::` dataset { measure<string> _+ }

2745 | component<string>

2746 | string

2747 `pattern1, pattern2 ::` component<string>

2748 | string

2749

2750 *Result type*

2751 `result ::` dataset { measure<string> _+ }

2752 | component<string>

2753 | string

2754

2755 *Additional constraints*

2756 None.

2757

2758 *Behaviour*

2759 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
 2760 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
 2761 the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the
 2762 section “Typical behaviours of the ML Operators”).

2763

2764 *Examples*

2765 Given the Data_Set DS_1:

2766

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

4	A	"hello!"
---	---	----------

2767
2768
2769

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

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

2770
2771
2772

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

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

2773

2774 String pattern location : instr

2775
2776
2777

Syntax

instr (op, pattern, start, occurrence)

2778
2779

Input parameters

2780
2781
2782
2783
2784

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

2785

Examples of valid syntaxes

2786
2787
2788
2789
2790

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

2791

Semantics for scalar operations

2792
2793
2794
2795

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 nth occurrence of the pattern, returning the position of its first character.

2796

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

2797

- If nth occurrence is omitted, the value is 1.

2798

If the nth occurrence of the string-pattern after the startth character is not found in the input string, the returned

2799

value is 0.

2800

For example:

2801

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

2802

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

2803

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

2804

2805
 2806 *Input parameters type*
 2807 op :: dataset { measure<string> _ }
 2808 | component<string>
 2809 | string
 2810 pattern :: component<string>
 2811 | string
 2812 start :: component < integer [value >= 1] >
 2813 | integer [value >= 1]
 2814 occurrence :: component < integer [value >= 1] >
 2815 | integer [value >= 1]

2816
 2817 *Result type*
 2818 result :: dataset { measure<integer[value >= 0]> int_var }
 2819 | component<integer[value >= 0]>
 2820 | integer[value >= 0]

2821
 2822 *Additional constraints*
 2823 For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.
 2824

2825 *Behaviour*
 2826 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
 2827 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
 2828 the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the
 2829 section “Typical behaviours of the ML Operators”).
 2830 If op is a Data Set then **instr** returns a dataset with a single measure int_var of type *integer*.

2831
 2832 *Examples*
 2833 Given the Data Set DS_1:
 2834

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

2835
 2836 *Example 1:* DS_r:= instr(ds_1,"hello") results in
 2837

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

2838
 2839 *Example 2 (on component):* DS_r := DS_1[calc Me_2:=instr(Me_1,"hello")] results in:
 2840

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

2841
2842
2843
2844

Given the Data Set DS_2:

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

2845
2846
2847
2848
2849

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

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

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

2850
2851
2852
2853
2854
2855

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

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

2856

String length : length

2857
2858
2859

Syntax

length (op)

2860
2861
2862

Input Parameters

op the operand

2863
2864
2865
2866

Examples of valid syntaxes

length("Hello, World!")

length(DS_1)

2867
2868
2869

Semantics for scalar operations

Returns the length of a string. For example, length("Hello, World!") gives 13

For the empty string "" the value 0 is returned

2870
2871
2872
2873
2874
2875

Input Parameters type

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

2876
2877
2878
2879
2880

Result type

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

2881 *Additional constraints*
 2882 For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.
 2883

2884 *Behaviour*
 2885 The operator has the behaviour of the “Operators changing the data type” (see the section “Typical behaviours of
 2886 the ML Operators”).
 2887 If op is a Data Set then **length** returns a dataset with a single measure *int_var* of type *integer*.
 2888

2889 *Examples*

2890 Given the Data Set DS_1
 2891
 2892

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

2893
 2894 *Example 1:* DS_r := length(DS_1) results in:
 2895

DS_r		
Id_1	Id_2	int_var
1	A	5
2	B	null

2896
 2897
 2898 *Example 2 (on component):* DS_r:= DS_1[calc Me_2:=length(Me_1)] results in
 2899

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

2900
 2901 Given the Data Set DS_2:
 2902

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

2903
 2904 *Example 3 (applying the length operator at component level to a multi Measure Data Set):*
 2905
 2906 DS_r := DS_2 [calc Me_10:= length(Me_1), Me_20:=length(Me_2)] results in:
 2907

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

2908
2909
2910 *Example 4 (**length** operator applied at Data Set level to a multi Measure Data Set):*
2911
2912 DS_r := length(DS_2) would give error because DS_2 has more Measures.

2913

VTL-ML - Numeric operators

2914

Unary plus : **+**

2915

Syntax

2916

+ op

2917

2918

Input parameters

2919

op the operand

2920

2921

Examples of valid syntaxes

2922

+ DS_1

2923

+ 3

2924

2925

Semantics for scalar operations

2926

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

2927

 + 3 gives 3

2928

 + (- 5) gives - 5

2929

2930

Input Parameters type

2931

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

2932

 | component<number>

2933

 | number

2934

2935

Result type

2936

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

2937

 | component<number>

2938

 | number

2939

2940

Additional constraints

2941

None.

2942

2943

Behaviour

2944

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”).

2945

2946

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

2947

2948

2949

2950

Examples

2951

Given the operand Data Set DS_1:

2952

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

2953

2954

Example 1: DS_r := + DS_1 results in:

2955

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

2956
 2957 *Example 2 (on components):* DS_r := DS_1 [calc Me_3 := + Me_1] results in:
 2958

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

2959 **Unary minus:** -

2960 *Syntax*

2961 - op

2962 *Input parameters*

2963 op the operand

2964 *Examples of valid syntaxes*

2965 - DS_1

2966 - 3

2967 *Semantics for scalar operations*

2968 The operator - inverts the sign of op. For example:

2969 - 3 gives - 3

2970 - (- 5) gives 5

2971 *Input Parameters type*

2972 op :: dataset { measure<number> _+ }

2973 | component<number>

2974 | number

2975 *Result type*

2976 result :: dataset { measure<number> _+ }

2977 | component<number>

2978 | number

2979 *Additional constraints*

2980 None.

2981 *Behaviour*

2982 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”).

2983 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is *number* then the result has type *number*.

2984 *Examples*

2985 Given the operand Data Set DS_1:

DS_1			
Id_1	Id_2	Me_1	Me_2
10	A	1	5.0

10	B	2	10.0
11	A	3	12.0

2998
2999
3000

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

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

3001
3002
3003

Example 2 (on components): $DS_r := DS_1 [\text{calc } Me_3 := - Me_1]$ results in:

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

3004
3005

3006 **Addition :** **+**

3007 *Syntax*
3008 $op1 + op2$

3009 *Input parameters*
3010 $op1$ the first addendum
3011 $op2$ the second addendum

3012 *Examples of valid syntaxes*
3013 $DS_1 + DS_2$
3014 $3 + 5$

3015 *Semantics for scalar operations*
3016 The operator addition returns the sum of two numbers. For example:
3017 $3 + 5$ gives 8

3018 *Input parameters type*
3019 $op1, op2 ::$ dataset { measure<number> $_+$ }
3020 | component<number>
3021 | number

3022 *Result type*
3023 $result ::$ dataset { measure<number> $_+$ }
3024 | component<number>
3025 | number

3026 *Additional constraints*
3027 None.

3028
3029
3030
3031
3032
3033
3034

3035 *Behaviour*
 3036 The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set
 3037 Components” (see the section “Typical behaviours of the ML Operators”).
 3038 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is
 3039 the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is
 3040 of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

3042 *Examples*
 3043 Given the operand Data Sets DS_1 and DS_2:

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

3045

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

3046 *Example 1:* DS_r := DS_1 + DS_2 results in:

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

3049 *Example 2:* DS_r := DS_1 + 3 results in:

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

3052 *Example 3 (on components):* DS_r := DS_1 [calc Me_3 := Me_1 + 3.0] results in:

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

3055 Subtraction : -

3056 *Syntax*

3057 op1 - op2

3058

3059 *Input Parameters*

3060 op1 the minuend

3061 op2 the subtrahend

3062

3063 *Examples of valid syntaxes*

3064 DS_1 - DS_2

3065 3 - 5

3066

3067 *Semantics for scalar operations*

3068 The operator subtraction returns the difference of two numbers. For example:

3069 3 - 5 gives - 2

3070

3071 *Input Parameters type*

3072 op1, op2:: dataset { measure<number> _+ }

3073 | component<number>

3074 | number

3075

3076 *Result type*

3077 result :: dataset { measure<number> _+ }

3078 | component<number>

3079 | number

3080

3081 *Additional constraints*

3082 None.

3083

3084 *Behaviour*

3085 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”).

3086 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*.

3090

3091 *Examples*

3092 Given the operand Data Sets DS_1 and DS_2:

3093

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

3094

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

3095

3096 *Example 1:* DS_r := DS_1 - DS_2 results in:

3097

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

3098

3099

3100

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

3101

3102

3103

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

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

3104

3105

Multiplication : *

3106

Syntax

op1 * op2

3108

3109

Input parameters

op1 the multiplicand

op2 the multiplier

3112

3113

Examples of valid syntaxes

DS_1 * DS_2

3 * 5

3116

3117

Semantics for scalar operations

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

3 * 5 gives 15

3120

3121

Input parameters type

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

| component<number>

| number

3125

3126

Result type

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

| component<number>

| number

3129

3130
 3131
 3132
 3133
 3134
 3135
 3136
 3137
 3138
 3139
 3140
 3141
 3142
 3143

Additional constraints

None.

Behaviour

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

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

Examples

Given the operand Data Sets DS_1 and DS_2:

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

3144

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

3145
 3146
 3147

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

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

3148
 3149
 3150

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

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

3151
 3152
 3153
 3154

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

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

3155

3156 **Division :** /

3157 *Syntax*

3158 op1 / op2

3159

3160 *Input parameters*

3161 op1 the dividend

3162 op2 the divisor

3163

3164 *Examples of valid syntaxes*

3165 DS_1 / DS_2

3166 3 / 5

3167

3168 *Semantics for scalar operations*

3169 The operator **division** divides two numbers. For example:

3170 3 / 5 gives 0.6

3171

3172 *Input parameters type*

3173 op1, op2 :: dataset { measure<number> _+ }

3174 | component<number>

3175 | number

3176

3177 *Result type*

3178 result :: dataset { measure<number> _+ }

3179 | component<number>

3180 | number

3181

3182 *Additional constraints*

3183 None.

3184

3185 *Behaviour*

3186 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”).

3187 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*.

3188 If op2 is 0 then the operation generates a run-time error.

3189

3190 *Examples*

3191 Given the operand Data Sets DS_1 and DS_2:

3192

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

3195

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

3196

3197

3198

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

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

3199

3200

3201

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

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

3202

3203

3204

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

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

3205

3206

Modulo : mod

3207

Syntax

3208

mod (op1 , op2)

3209

3210

Input parameters

3211

op1 the dividend

3212

op2 the divisor

3213

3214

Examples of valid syntaxes

3215 mod (DS_1, DS_2)
 3216 mod (DS_1, 5)
 3217 mod (5, DS_2)
 3218 mod (5, 2)

3219 *Semantics for scalar operations*

3220 The operator **mod** returns the remainder of op1 divided by op2. It returns op1 if divisor op2 is 0. For example:

3221 mod (5, 2) gives 1
 3222 mod (5, -2) gives -1
 3223 mod (8, 2) gives 0
 3224 mod (9, 0) gives 9

3225 *Input Parameters type*

3226 op1, op2 :: dataset { measure<number> _+ }
 3227 | component<number>
 3228 | number
 3229 divisor :: number

3230 *Result type*

3231 result :: dataset { measure<number> _+ }
 3232 | component<number>
 3233 | number

3234 *Additional constraints*

3235 None.

3236 *Behaviour*

3237 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”).
 3238 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*.

3239 *Examples*

3240 Given the operand Data Sets DS_1 and DS_2:

3241

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

3242

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

3243
 3244
 3245
 3246
 3247
 3248
 3249 *Example 1:* DS_r := mod (DS_1, DS_2) results in:
 3250

3251
 3252
 3253
 3254
 3255

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

3256
3257
3258

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

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

3259
3260
3261

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

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

3262

Rounding : **round**

3263

Syntax

round (op , numDigit)

3264
3265
3266

Input parameters

op the operand
numDigit the number of positions to round to

3267
3268
3269
3270

Examples of valid syntaxes

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

3271
3272
3273
3274
3275
3276

Semantics for scalar operations

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

3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287

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

3288 round (12345.6) gives 12346
 3289 round (12345.6, _) gives 12346
 3290 round (12345.6, -1) gives 12350.0

3291

3292 *Input parameters type*

3293 op1 :: dataset { measure<number> _+ }
 3294 | component<number>
 3295 | number
 3296 numDigit:: component < integer >
 3297 | integer
 3298

3299 *Result type*

3300 result :: dataset { measure<number> _+ }
 3301 | component<number>
 3302 | number
 3303

3304 *Additional constraints*

3305 None.

3306

3307 *Behaviour*

3308 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
 3309 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
 3310 the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the
 3311 section “Typical behaviours of the ML Operators”).
 3312

3313 *Examples*

3314 Given the operand Data Set DS_1:

3315

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

3316

3317 *Example 1:* DS_r := round(DS_1, 0) results in:

3318

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

3319

3320 *Example 2 (on components):* DS_r := DS_1 [calc Me_10:= round(Me_1)] results in:

3321

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

3322
3323
3324

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

results in:

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

3325

3326 Truncation : **trunc**

3327 *Syntax*

3328 **trunc** (op , numDigit)

3329

3330 *Input Parameters*

3331 op the operand
3332 numDigit the number of position from which to trunc

3333

3334 *Examples of valid syntaxes*

3335 trunc (DS_1 , 2)

3336 trunc (DS_1)

3337 trunc (3.14159 , 2)

3338 trunc (3.14159 , _)

3339

3340 *Semantics for scalar operations*

3341 The operator **trunc** truncates the operand to a number of positions at the right of the decimal point equal to the
3342 numDigit parameter. The decimal point is assumed to be at position 0. If numDigit is negative, the truncation
3343 happens at the left of the decimal point. The truncation operation leaves the numDigit position unchanged. All
3344 the positions greater than numDigit are eliminated. The basic scalar type of the result is *integer* if numDigit is
3345 omitted, *number* otherwise.

3346 For example:

3347 trunc (3.14159, 2) gives 3.14

3348 trunc (3.14159, 4) gives 3.1415

3349 trunc (12345.6, 0) gives 12345.0

3350 trunc (12345.6) gives 12345

3351 trunc (12345.6, _) gives 12345

3352 trunc(12345.6, -1) gives 12340.0

3353

3354 *Input parameters type*

3355 op :: dataset { measure<number> _+ }

3356 | component<number>

3357 | number

3358 numDigit :: component < integer >

3359 | integer

3360

3361 *Result type*

3362 result :: dataset { measure<number> _+ }

3363 | component<number>

3364 | number

3365

3366 *Additional constraints*

3367 None.

3368

3369 *Behaviour*

3370 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar
3371 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has
3372 the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the
3373 section “Typical behaviours of the ML Operators”).

3374
3375 *Examples*

3376 Given the operand Data Set DS_1:
3377
3378

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

3379
3380 *Example 1:* DS_r := trunc(DS_1, 0) results in:
3381

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

3382
3383 *Example 2 (on components):* DS_r := DS_1[calc Me_10:= trunc(Me_1)] results in:
3384

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

3385
3386 *Example 3 (on components):* DS_r := DS_1[calc Me_20:= trunc(Me_1 , -1)] results in:
3387

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

3388

3389 **Ceiling : ceil**

3390 *Syntax*

3391 **ceil (op)**

3392

3393 *Input parameters*

3394 op the operand

3395

3396 *Examples of valid syntaxes*

3397 ceil (DS_1)

3398 ceil (3.14159)

3399

3400 *Semantics for scalar operations*

3401 The operator **ceil** returns the smallest integer greater than or equal to op.

3402 For example:

3403 ceil(3.14159) gives 4

3404 ceil(15) gives 15

3405 ceil(-3.1415) gives -3

3406 ceil(-0.1415) gives 0

3407

3408 *Input parameters type*

3409 op :: dataset { measure<number> _+ }

3410 | component<number>

3411 | number

3412

3413 *Result type*

3414 result :: dataset { measure<integer> _+ }

3415 | component< integer >

3416 | integer

3417

3418 *Additional constraints*

3419 None.

3420

3421 *Behaviour*

3422 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”).

3423

3424 *Examples*

3425 Given the operand Data Set DS_1:

3426

3427

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

3428

3429 *Example 1:* DS_r := ceil (DS_1) results in:

3430

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

3431
3432
3433

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

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

3434

Floor: floor

3435

Syntax

3436

floor (op)

3437

3438

Input parameters

3439

op the operand

3440

3441

Examples of valid syntaxes

3442

floor (DS_1)

3443

floor (3.14159)

3444

3445

Semantics for scalar operations

3446

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

3447

For example:

3448

floor(3.1415) gives 3

3449

floor(15) gives 15

3450

floor(-3.1415) gives -4

3451

floor(-0.1415) gives -1

3452

3453

Input parameters type

3454

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

3455

3456

3457

3458

Result type

3459

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

3460

3461

3462

3463

Additional constraints

3464

None.

3465

Behaviour

3466

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”).

3467

3468

3469

3470

Examples

3471

Given the operand Data Set DS_1:

3472

3473

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

3474
3475
3476

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

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

3477
3478
3479

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

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

3480

Absolute value : abs

3481

Syntax

3482

abs (op)

3483

3484

Input parameters

3485

op the operand

3486

3487

Examples of valid syntaxes

3488

abs (DS_1)

3489

abs (-5)

3490

3491

Semantics for scalar operations

3492

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

3493

For example:

3494

abs (-5.49) gives 5.49

3495

abs (5.49) gives 5.49

3496

3497

Input parameters type

3498

3499

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

3500

3501

3502

3503

Result type

3504

3505

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

3506

3507 | number [value >= 0]

3508

3509 *Additional constraints*

3510 None.

3511

3512 *Behaviour*

3513 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set
3514 Component” (see the section “Typical behaviours of the ML Operators”).

3515

3516 *Examples*

3517 Given the operand Data Set DS_1:

3518

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

3519

3520 *Example 1:* DS_r := abs (DS_1) results in:

3521

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

3522

3523 *Example 2 (on components):* DS_r := DS_1 [Me_10 := abs(Me_1)] results in:

3524

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

3525

3526 **Exponential :** exp

3527 *Syntax*

3528 exp (op)

3529

3530 *Input parameters*

3531 op the operand

3532

3533 *Examples of valid syntaxes*

3534 exp (DS_1)

3535 exp (5)

3536

3537 *Semantics for scalar operations*

3538 The operator **exp** returns e (base of the natural logarithm) raised to the op-th power.

3539 For example;

3540 exp (5) gives 148.41315...

3541 exp (1) gives 2.71828... (the number e)

3542 exp (0) gives 1.0
 3543 exp (-1) gives 0.36787... (the number 1/e)

3544
 3545 *Input parameters type*

3546 op:: dataset { measure<number> _+ }
 3547 | component<number>
 3548 | number

3549
 3550 *Result type*

3551 result :: dataset { measure<number[value > 0]> _+ }
 3552 | component<number [value > 0]>
 3553 | number[value > 0]

3554
 3555 *Additional constraints*

3556 None.

3557
 3558 *Behaviour*

3559 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set
 3560 Component” (see the section “Typical behaviours of the ML Operators”).

3561
 3562 *Examples*

3563 Given the operand Data Set DS_1:

3564

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

3565

3566

3567 *Example 1:*

DS_r := exp(DS_1)

results in:

3568

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

3569

3570

3571 *Example 2 (on components):*

DS_r := DS_1 [Me_1 := exp (Me_1)]

results in:

3571

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

3572

3573 **Natural logarithm : ln**

3574 *Syntax*

3575 ln (op)

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

Input parameters

op the operand

Examples of valid syntaxes

ln (DS_1)
 ln (148)

Semantics for scalar operations

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

ln (148) gives 4.997...
 ln (e) gives 1.0
 ln (1) gives 0.0
 ln (0,5) gives -0.693...

Input parameters type

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

Result type

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

Additional constraints

None.

Behaviour

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

Examples

Given the operand Data Set DS_1:

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

3612
 3613
 3614
 3615

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

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

3616
 3617
 3618

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

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

3619

3620 **Power :** **power**

3621 *Syntax*

3622 **power (base , exponent)**

3623

3624 *Input parameters*

3625 **base** the operand

3626 **exponent** the exponent of the power

3627

3628 *Examples of valid syntaxes*

3629 **power (DS_1, 2)**

3630 **power (5, 2)**

3631

3632 *Semantics for scalar operations*

3633 The operator **power** raises a number (the base) to another one (the exponent).

3634 For example:

3635 **power (5, 2)** gives 25

3636 **power (5, 1)** gives 5

3637 **power (5, 0)** gives 1

3638 **power (5, -1)** gives 0.2

3639 **power (-5, 3)** gives -125

3640

3641 *Input parameters type*

3642 **base ::** dataset { measure<number> _+ }

3643 | component<number>

3644 | number

3645 **exponent ::** component<number>

3646 | number

3647

3648 *Result type*

3649 **result ::** dataset { measure<number> _+ }

3650 | component<number>

3651 | number

3652

3653 *Additional constraints*

3654 None.

3655

3656 *Behaviour*

3657 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”).

3661

3662 *Examples*

3663 Given the operand Data Set DS_1:

3664

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

3665
3666
3667
3668

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

3669
3670
3671

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

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

3672

Logarithm : log

3673

Syntax

log (op , num)

3674
3675
3676

Input parameters

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

3677

Examples of valid syntaxes

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

3678
3679
3680

Semantics for scalar operations

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

For example:

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

3681
3682
3683
3684

Input parameters type

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

3685
3686
3687

Result type

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

3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701

3702
3703
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714

Additional constraints

None.

Behaviour

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

Examples

Given the operand Data Set DS_1:

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

3715
3716
3717
3718

Example 1:

DS_r := log (DS_1, 2)

results in:

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

3719
3720
3721

Example 2 (on components):

DS_r := DS_1 [calc Me_1 := log (Me_1, 2)]

results in:

DS_r			
Id_1	Id_2	Me_1	Me_2
10	A	10.0	0.7545
10	B	6.0	13.45
11	A	5.0	1.87

3722

3723

Square root : sqrt

3724

Syntax

sqrt (op)

3726

3727

Input parameters

3728

op the operand

3729

3730

Examples of valid syntaxes

3731

sqrt (DS_1)

3732

sqrt (5)

3733

3734

Semantics for scalar operations

3735

The operator **sqrt** calculates the square root of a number. For example:

3736

sqrt (25) gives 5

3737
 3738 *Input parameters type*
 3739 op :: dataset { measure<number [value >= 0] > _+ }
 3740 | component<number [value >= 0] >
 3741 | number [value >= 0]

3742
 3743 *Result type*
 3744 result :: dataset { measure<number [value >= 0] > _+ }
 3745 | component<number [value >= 0] >
 3746 | number [value >= 0]

3747
 3748 *Additional constraints*
 3749 None.

3750
 3751 *Behaviour*
 3752 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set
 3753 Component” (see the section “Typical behaviours of the ML Operators”).

3754
 3755 *Examples*
 3756 Given the operand Data Set DS_1:
 3757

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

3758
 3759
 3760 *Example 1:* DS_r := sqrt(DS_1) results in:
 3761

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

3762
 3763
 3764 *Example 2 (on components):* DS_r := DS_1 [calc Me_1 := sqrt (Me_1)] results in:
 3765

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

3766
 3767
 3768

3770 Equal to : =

3771

3772 *Syntax*

3773 left = right

3774

3775 *Input parameters*

3776 left the left operand

3777 right the right operand

3778

3779 *Examples of valid syntaxes*

3780 DS_1 = DS_2

3781

3782 *Semantics for scalar operations*

3783 The operator returns TRUE if the left is equal to right, FALSE otherwise.

3784 For example:

3785 5 = 9 gives: FALSE

3786 5 = 5 gives: TRUE

3787 "hello" = "hi" gives: FALSE

3788

3789 *Input parameters type*

3790 left,

3791 right :: dataset {measure<scalar> _ }

3792 | component<scalar>

3793 | scalar

3794

3795 *Result type*

3796 result :: dataset { measure<boolean> bool_var }

3797 | component<boolean>

3798 | boolean

3799

3800 *Additional constraints*

3801 Operands left and right must be of the same scalar type

3802

3803 *Behaviour*

3804 The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

3805

3806 *Examples*

3807 Given the operand Data Set DS_1:

3808

3809

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

3810

3811 *Example 1:* DS_r := DS_1 = 0.08 results in:
 3812

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

3813
 3814 *Example 2 (on Components):* DS_r := DS_1 [calc Me_2 := Me_1 = 0.08]
 3815

results in:

DS_r					
Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
2012	B	Total	Total	NULL	NULL
2012	G	Total	Total	0.286	FALSE
2012	S	Total	Total	0.064	FALSE
2012	M	Total	Total	0.043	FALSE
2012	F	Total	Total	0.08	TRUE
2012	W	Total	Total	0.08	TRUE

3816

3817 Not equal to : <>

3818
 3819 *Syntax*

3820 left <> right

3821
 3822 *Input parameters*

3823 left the left operand

3824 right the right operand

3825
 3826 *Examples of valid syntaxes*

3827 DS_1 <> DS_2

3828
 3829 *Semantics for scalar operations*

3830 The operator returns FALSE if the left is equal to right, TRUE otherwise.

3831 For example:

3832 5 <> 9 gives: TRUE

3833 5 <> 5 gives: FALSE

3834 "hello" <> "hi" gives: TRUE

3835
 3836 *Input parameters type*

3837 left,

3838 right :: dataset {measure<scalar> _}

3839 | component<scalar>

3840 | scalar

3841

3842 *Result type*
 3843 result :: dataset { measure<boolean> bool_var }
 3844 | component<boolean>
 3845 | boolean

3846
 3847 *Additional constraints*
 3848 Operands left and right must be of the same scalar type
 3849

3850 *Behaviour*
 3851 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical
 3852 behaviours of the ML Operators”).
 3853

3854 *Examples*
 3855 Given the operand Data Sets DS_1 and DS_2:
 3856

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

3857
 3858

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

3859 *Example 1:* DS_r := DS_1 <> DS_2 results in:
 3860
 3861

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

3862
 3863 Note that due to the behaviour for NULL values, if the value for Greece in the second operand had also been
 3864 NULL, then the result would still be NULL for Greece.
 3865

3866 *Example 2 (on Components):* DS_r := DS_1 [Me_2 := Me_1 <> 7.5] results in:
 3867

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

3868
 3869

3870 **Greater than :** > >=

3871 *Syntax*
 3872 left { > | >= }¹ right
 3873

3874 *Input parameters*
 3875 left the left operand part of the comparison
 3876 right the right operand part of the comparison

3877
 3878 *Examples of valid syntaxes*
 3879 DS_1 > DS_2
 3880 DS_1 >= DS_2

3881 *Semantics for scalar operations*
 3882 The operator > returns TRUE if left is greater than right, FALSE otherwise.
 3884 The operator >= returns TRUE if left is greater than or equal to right, FALSE otherwise.
 3885 For example:

3886 5 > 9 gives: FALSE
 3887 5 >= 5 gives: TRUE
 3888 "hello" > "hi" gives: FALSE

3889
 3890 *Input parameters type*
 3891 left,
 3892 right :: dataset {measure<scalar> _ }
 3893 | component<scalar>
 3894 | scalar

3895
 3896 *Result type*
 3897 result :: dataset { measure<boolean> bool_var }
 3898 | component<boolean>
 3899 | boolean

3900
 3901 *Additional constraints*
 3902 Operands left and right must be of the same scalar type

3903
 3904 *Behaviour*
 3905 The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

3906
 3907 *Examples*
 3908 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

3911
 3912 *Example 1:* DS_r := DS_1 > 20 results in:
 3913

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

3914
 3915 *Example 2 (on Components):* DS_r := DS_1 [Me_2 := Me_1 > 20] results in:
 3916

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

3917
3918
3919

Given the left operand Data Set:

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

3920
3921
3922

and the right operand Data Set:

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

3923
3924
3925

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

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

3926
3927
3928

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

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

3929

3930 Less than : < <=

3931
3932
3933
3934

Syntax

left { < | <= }¹ right

3935
3936
3937
3938

Input parameters

left the left operand
right the right operand

3939
3940
3941

Examples of valid syntaxes

DS_1 < DS_2
DS_1 <= DS_2

3942
 3943
 3944
 3945
 3946
 3947
 3948
 3949
 3950
 3951
 3952
 3953
 3954
 3955
 3956
 3957
 3958
 3959
 3960
 3961
 3962
 3963
 3964
 3965
 3966
 3967
 3968
 3969
 3970

Semantics for scalar operations

The operator **<** returns TRUE if left is smaller than right, FALSE otherwise.
 The operator **<=** returns TRUE if left is smaller than or equal to right, FALSE otherwise.
 For example:

```

5 < 4           gives: FALSE
5 <= 5          gives: TRUE
"hello" < "hi"  gives: TRUE
  
```

Input parameters type

```

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

Result type

```

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

Additional constraints

Operands left and right must be of the same scalar type

Behaviour

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

Examples

Given the operand Data Set DS_1:

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

3971
 3972
 3973

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

3974

3975 **Between :** **between**

3976

3977 *Syntax*

3978 **between** (op, from, to)

3979

3980 *Input parameters*

3981 op the Data Set to be checked

3982 from the left delimiter

3983 to the right delimiter

3984

3985 *Examples of valid syntaxes*

3986 ds2 := between(ds1, 5,10)

3987 ds2 := ds1 [calc m1 := between(me2, 5, 10)]

3988

3989 *Semantics for scalar operations*

3990 The operator returns TRUE if op is greater than or equal to from and lower than or equal to to. In other terms, it is a shortcut for the following:

3991

op >= from and op <= to

3992

3993 The types of op, from and to must be compatible scalar types.

3994

3997 *Input parameters type*

3998 op :: dataset {measure<scalar> _}

3999 | component<scalar>

4000 | scalar

4001

4002 from :: scalar | component<scalar>

4003 to :: scalar | component<scalar>

4004

4005 *Result type*

4006 result :: dataset { measure<boolean> bool_var }

4007 | component<boolean>

4008 | boolean

4009

4010 *Additional constraints*

4011 The type of the operand (i.e., the measure of the dataset, the type of the component, the scalar type) must be the same as that of from and to.

4012

4013 *Behaviour*

4014 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

4015

4016 *Examples*

4017

4018 Given the following Data Set DS_1:

4019

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

4022

4023 Example 1: DS_r:= between(ds1, 5,10) results in:

4024

DS_1				
Id_1	Id_2	Id_3	Id_4	bool_var

G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	FALSE

4025

4026 Element of: `in / not_in`

4027

4028

Syntax

4029

op **in** collection

4030

op **not_in** collection

4031

4032

collection ::= set | valueDomainName

4033

4034

Input parameters

4035

op the operand to be tested

4036

collection the the Set or the Value Domain which contains the values

4037

set the Set which contains the values (it can be a Set name or a Set literal)

4038

valueDomainName the name of the Value Domain which contains the values

4039

4040

Examples of valid syntaxes

4041

ds := ds_2 in {1,4,6} as usual, here the braces denote a set literal (it contains the values 1, 4 and 6)

4042

ds := ds_3 in mySet

4043

ds := ds_3 in myValueDomain

4044

4045

Semantics for scalar operations

4046

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

4047

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

4048

For example:

4049

1 in { 1, 2, 3 } returns TRUE

4050

"a" in { "c", "ab", "bb", "bc" } returns FALSE

4051

"b" not_in { "b", "hello", "c" } returns FALSE

4052

"b" not_in { "a", "hello", "c" } returns TRUE

4053

4054

Input parameters type

4055

op :: dataset {measure<scalar> _ }

4056

| component<scalar>

4057

| scalar

4058

collection :: set<scalar> | name<value_domain>

4059

4060

Result type

4061

result :: dataset { measure<boolean> bool_var }

4062

| component<boolean>

4063

| boolean

4064

4065

Additional constraints

4066

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

4067

4068

Behaviour

4069

Semantics

4070

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

4071

the **not_in** the opposite.

4072

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

4073

behaviours of the ML Operators").

4074

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

4075

Domain or Set.

4076

4077

Examples

4078

Given the operand Data Set DS_1:

4079

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

4080

4081

Example 1:

4082

DS_r := DS_1 in { "BS", "MO", "HH", "PP" }

results in:

4083

4084

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

4085

4086

Example 2 (on Components):

4087

DS_r := DS_1 [calc Me_2:= Me_1 in { "BS", "MO", "HH", "PP" }]

results in:

4088

4089

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

4090

4091

Given the previous Data Set DS_1 and the following Value Domain named myGeoValueDomain (which has the basic scalar type string) :

4092

4093

myGeoValueDomain	
Code	Meaning
AF	Afghanistan
BS	Bahamas
FJ	Fiji
GA	Gabon
KH	Cambodia

MO	Macao
PK	Pakistan
QA	Quatar
UG	Uganda

4094
4095
4096
4097
4098
4099

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

4100
4101

4102 match_characters match_characters

4103
4104

Syntax

4105
4106
4107

match_characters (op , pattern)

4108

Input parameters

4109
4110

op the dataset to be checked

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

4111

Examples of valid syntaxes

4112

4113

4114

match_characters(ds1, "[abc]+\d\d")

4115

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

4116

4117

Semantics for scalar operations

4118

4119

4120

4121

4122

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.

4123

Input parameters type

4124

4125

4126

4127

4128

4129

4130

4131

4132

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

pattern :: string | component<string>

Result type

result :: dataset { measure<boolelan> bool_var }

4133 | component<boolean>
 4134 | boolean

4135 *Additional constraints*

4137 If op is a Data Set then it has exactly one measure.
 4138 pattern is a POSIX regular expression.

4139 *Behaviour*

4141 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

4142 *Examples*

4143 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

4146
 4147
 4148
 4149

DS_r:=(ds1, “[:alpha:]{2}[:digit:]{3}”) results in:

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

4150
 4151

4152 **IsNull: isnull**

4153 *Syntax*

4154 **isnull (op)**

4155 *Input parameters*

4156 operand mandatory the operand

4157 *Examples of valid syntaxes*

4158 isnull(DS_1)

4159 *Semantics for scalar operations*

4160 The operator returns TRUE if the value of the operand is NULL, FALSE otherwise.

4161 *Examples*

4162 isnull(“Hello”) gives: FALSE
 4163 isnull(NULL) gives: TRUE

4164 *Input parameters type*

4165 op :: dataset {measure<scalar> _}
 4166 | component<scalar>
 4167 | scalar

4168 *Result type*

4169 result :: dataset { measure<boolean> bool_var }
 4170 | component<boolean>
 4171 | boolean

4178
 4179
 4180
 4181
 4182
 4183
 4184
 4185
 4186
 4187
 4188

Additional constraints

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

Behaviour

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

Examples

Given the operand Data Set DS_1:

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

4189
 4190
 4191

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

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

4192
 4193
 4194

Example 2 (on Components): DS_r := DS_1[Me_2 := is_null(Me_1)] results in:

DS_r					
Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
2012	B	Total	Total	11094850	FALSE
2012	G	Total	Total	11123034	FALSE
2012	S	Total	Total	NULL	TRUE
2012	M	Total	Total	417546	FALSE
2012	F	Total	Total	5401267	FALSE
2012	N	Total	Total	NULL	TRUE

4195
 4196

4197 **Exists in :** **exists_in**

4198

4199 *Syntax*

4200 **exists_in** (op1, op2 { , retain })

4201

4202 retain ::= **true** | **false** | **all**

4203

4204 *Input parameters*

4205 op1 the operand dataset

4206 op2 the operand dataset

4207 retain the optional parameter to specify the Data Points to be returned (default: **all**)

4208

4209 *Examples of valid syntaxes*

4210 exists_in (DS_1, DS_2, true)

4211 exists_in (DS_1, DS_2)

4212 exists_in (DS_1, DS_2, all)

4213

4214 *Semantics for scalar operations*

4215 This operator cannot be applied to scalar values.

4216

4217 *Input parameters type*

4218 op1,

4219 op2 :: dataset

4220

4221 *Result type*

4222 result :: dataset { measure<boolean> bool_var }

4223

4224 *Additional constraints*

4225 op2 has all the identifier components of op1.

4226

4227 *Behaviour*

4228 The operator checks if the combinations of values of the Identifiers existing in op1 also exist in op2.

4229 The result has the same Identifiers as op1 and a *boolean* Measure bool_var whose value, for each Data Point of op1, is TRUE if the combination of values of the Identifier Components existing in op1 is found in a Data Point of op2, FALSE otherwise. If retain is **all** then both the Data Points having bool_var = TRUE and bool_var = FALSE are returned.

4230 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.

4231 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

4232

4233 *Examples*

4234 Given the operand Data Sets DS_1 and DS_2:

4235

4236

4237

4238

4239

4240

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

4241

4242

4243

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

4244
4245
4246

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

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

4247
4248
4249

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

DS_r				
Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	TRUE
2012	S	Total	Total	TRUE
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

4250
4251
4252

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

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

4253

4254

VTL-ML - Boolean operators

4255

Logical conjunction: **and**

4256

4257

Syntax

4258

op1 **and** op2

4259

4260

Input parameters

4261

op1 the first operand

4262

op2 the second operand

4263

4264

Examples of valid syntaxes

4265

DS_1 **and** DS_2

4266

4267

Semantics for scalar operations

4268

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

4269

4270

For example:

4271

FALSE **and** FALSE gives FALSE

4272

FALSE **and** TRUE gives FALSE

4273

TRUE **and** FALSE gives FALSE

4274

TRUE **and** TRUE gives TRUE

4275

4276

Input parameters type

4277

op1,

4278

op2 :: dataset {measure<boolean> _}

4279

| component<boolean>

4280

| boolean

4281

4282

Result type

4283

result :: dataset { measure<boolean> _}

4284

| component<boolean>

4285

| boolean

4286

4287

Additional constraints

4288

None.

4289

4290

Behaviour

4291

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

4292

4293

4294

Examples

4295

Given the operand Data Sets DS_1 and DS_2:

4296

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

4297
4298

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

4299
4300
4301
4302

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

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	FALSE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

4303
4304
4305

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

DS_r					
Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	TRUE	TRUE
M	64	B	2013	FALSE	FALSE
M	65	B	2013	TRUE	TRUE
F	15	U	2013	FALSE	FALSE
F	64	U	2013	FALSE	FALSE
F	65	U	2013	TRUE	TRUE

4306

Logical disjunction : or

4307

Syntax

4308

op1 or op2

4309

Input parameters

4311

op1 the first operand

4312

op2 the second operand

4313

Examples of valid syntaxes

4314

DS_1 or DS_2

4316

4317

Semantics for scalar operations

4318 The **or** operator returns TRUE if at least one of the operands is TRUE, otherwise FALSE. The two operands must
 4319 be of *boolean* type.

4320 For example:

4321 FALSE or FALSE gives FALSE
 4322 FALSE or TRUE gives TRUE
 4323 TRUE or FALSE gives TRUE
 4324 TRUE or TRUE gives TRUE
 4325

4326 *Input parameters type*

4327 op1,
 4328 op2 :: dataset {measure<boolean> _}
 4329 | component<boolean>
 4330 | boolean

4331 *Result type*

4332 result :: dataset { measure<boolean> _}
 4333 | component<boolean>
 4334 | boolean
 4335

4336 *Additional constraints*

4337 None.

4338 *Behaviour*

4339 The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical
 4340 behaviours of the ML Operators”).

4341 *Examples*

4342 Given the operand Data Sets DS_1 and DS_2:

4343

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

4346

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

4348

4349

4350

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

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	TRUE
M	64	B	2013	TRUE
M	65	B	2013	TRUE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

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

4351
4352 *Example 2 (on Components):* DS_r:= DS_1 [Me_2:= Me_1 or true]
4353

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

4354

4355 **Exclusive disjunction :** **xor**

4356 *Syntax*
4357 op1 **xor** op2

4358 *Input parameters*
4360 op1 the first operand
4361 op2 the second operand

4364 *Examples of valid syntaxes*
4365 DS_1 xor DS_2

4367 *Semantics for scalar operations*
4368 The **xor** operator returns TRUE if only one of the operand is TRUE (but not both), FALSE otherwise. The two
4369 operands must be of *boolean* type.

4370 For example:
4371 FALSE xor FALSE gives FALSE
4372 FALSE xor TRUE gives TRUE
4373 TRUE xor FALSE gives TRUE
4374 TRUE xor TRUE gives FALSE

4376 *Input parameters type*
4377 op1,
4378 op2 :: dataset {measure<boolean> _ }
4379 | component<boolean>
4380 | boolean

4382 *Result type*
4383 result :: dataset { measure<boolean> _ }
4384 | component<boolean>
4385 | boolean

4386

4387 *Additional constraints*

4388 None.

4389 *Behaviour*

4391 The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

4393 *Examples*

4395 Given the operand Data Sets DS_1 and DS_2:

4396

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	TRUE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	trTRUEue

4397

4398

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

4399

4400

4401

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

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	TRUE
M	64	B	2013	TRUE
M	65	B	2013	FALSE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

4402

4403

4404

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

DS_r					
Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	TRUE	FALSE
M	64	B	2013	FALSE	TRUE

M	65	B	2013	TRUE	FALSE
F	15	U	2013	FALSE	TRUE
F	64	U	2013	FALSE	TRUE
F	65	U	2013	TRUE	FALSE

4405

4406

Logical negation : **not**

4407

Syntax

4408

4409

not op

4410

4411

Input parameters

4412

op the operand

4413

4414

Examples of valid syntaxes

4415

not DS_1

4416

4417

Semantics for scalar operations

4418

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

4419

For example:

4420

not FALSE gives TRUE

4421

not TRUE gives FALSE

4422

4423

Input parameters type

4424

op :: dataset {measure<boolean> _ }

4425

| component<boolean>

4426

| boolean

4427

4428

Result type

4429

result :: dataset { measure<boolean> _ }

4430

| component<boolean>

4431

| boolean

4432

4433

Additional constraints

4434

None.

4435

4436

Behaviour

4437

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

4438

behaviours of the ML Operators”).

4439

4440

Examples

4441

Given the operand Data Set DS_1:

4442

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

4443

4444 *Example 1:* DS_r:= not DS_1 results in:
4445

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

4446 *Example 2 (on Components):* DS_r:= DS_1 [calc Me_2 := not Me_1] results in:
4447
4448

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

4449

4451 This chapter describes the **time** operators, which are the operators dealing with **time**, **date** and **time_period**
 4452 basic scalar types. The general aspects of the behaviour of these operators is described in the section “Behaviour
 4453 of the Time Operators”.

4454 The *time* data type is the most general type and denotes a generic time interval, having start and end points in
 4455 time and therefore a duration, which is the time intervening between the start and end points. The *date* data type
 4456 denotes a generic time instant (a point in time), which is a time interval with zero duration. The *time_period* data
 4457 type denotes a regular time interval whose regular duration is explicitly represented inside each *time_period*
 4458 value and is named *period_indicator*. In some sense, we say that *date* and *time_period* are special cases of *time*,
 4459 the former with coinciding extremes and zero duration and the latter with regular duration. The *time* data type is
 4460 overarching in the sense that it comprises *date* and *time_period*. Finally, *duration* data type represents a generic
 4461 time span, independently of any specific start and end date.

4462 The time, date and time period formats used here are explained in the User Manual in the section “External
 4463 representations and literals used in the VTL Manuals”.

4464 The period indicator P id of the *duration* type and its possible values are:

4465	D	Day
4466	W	Week
4467	M	Month
4468	Q	Quarter
4469	S	Semester
4470	A	Year

4471
 4472 As already said, these representation are not prescribed by VTL and are not part of the VTL standard, each VTL system
 4473 can personalize the representation of time, date, *time_period* and duration as desired. The formats shown above are only
 4474 the ones used in the examples.

4475 For a fully-detailed explanation, please refer to the User Manual.

4476

4477 Period indicator : `period_indicator`

4478

4479 The operator **period_indicator** extracts the period indicator from a *time_period* value.

4480

Syntax

4481 **period_indicator** ({ op })

4482

Input parameters

4484 op the operand

4485

Examples of valid syntaxes

4487 `period_indicator (ds_1)`

4488 `period_indicator` (if used in a clause the operand `op` can be omitted)

4489

Semantics for scalar operations

4491 **period_indicator** returns the period indicator of a *time_period* value. The period indicator is the part of the
 4492 *time_period* value which denotes the duration of the time period (e.g. day, week, month ...).

4493

Input parameters type

4495 `op :: dataset { identifier <time_period> _ , identifier _* }`

4496 | `component<time_period>`

4497 | `time_period`

4498

Result type

4500 `result :: dataset { measure<duration> duration_var }`

4501 | `component <duration>`

4502 | `duration`

4503

4504 *Additional constraints*
 4505 If *op* is a Data Set then it has exactly an Identifier of type *time_period* and may have other Identifiers. If the
 4506 operator is used in a clause and *op* is omitted, then the Data Set to which the clause is applied has exactly an
 4507 Identifier of type *time_period*.

4508 *Behaviour*
 4509 The operator extracts the period indicator part of the *time_period* value. The period indicator is computed for
 4510 each Data Point. When the operator is used in a clause, it extracts the period indicator from the *time_period*
 4511 value the Data Set to which the clause is applied.
 4512 The operator returns a Data Set with the same Identifiers of *op* and one Measure of type *duration* named
 4513 *duration_var*. As for all the Variables, a proper Value Domain must be defined to contain the possible values of
 4514 the period indicator and *duration_var*. The values used in the examples are listed at the beginning of this chapter
 4515 "VTL-ML Time operators".
 4516

4517 *Examples*

4518 Given the Data Set DS_1:

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

4519 *Example 1:* DS_r := period_indicator (DS_1) results in:
 4520
 4521

DS_r			
Id_1	Id_2	Id_3	duration_var
A	1	2010	A
A	1	2013Q1	Q

4522 *Example 2 (on component):* DS_r := DS_1 [filter period_indicator (Id_3) = "A"] results in:
 4523
 4524

DS_r			
Id_1	Id_2	Id_3	Me_1
A	1	2010	10

4525
 4526
 4527

4528 **Fill time series : fill_time_series**

4529 *Syntax*
 4530 **fill_time_series** (op { , limitsMethod })
 4531
 4532 limitsMethod ::= **single** | **all**
 4533
 4534

4535 *Input parameters*
 4536 *op* the operand
 4537 *limitsMethod* method for determining the limits of the time interval to be filled (default: **all**)
 4538

4539 *Examples of valid syntaxes*
 4540 fill_time_series (ds)
 4541 fill_time_series (ds, all)

4542

4543 *Semantics for scalar operations*

4544 The fill_time_series operator does not perform scalar operations.

4545

4546 *Input parameters type:*

4547 op :: dataset { identifier <time > _, identifier_* }

4548

4549 *Result type:*

4550 result :: dataset { identifier <time > _, identifier_* }

4551

4552

4553 *Additional constraints*

4554 The operand op has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

4555

4556 *Behaviour*

4557 This operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4558 The operator fills the possibly missing Data Points of all the time series belonging to the operand op within the time limits automatically determined by applying the limit_method.

4559 If limitsMmethod is **all**, the time limits are determined with reference to all the time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the Data Set.

4560 If limitsMmethod is **single**, the time limits are determined with reference to each single time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the time series.

4561 The expected Data Points are determined, for each time series, by considering the limits above and the *period* (*frequency*) of the time series: all the Identifiers are kept unchanged except the reference time Identifier, which is increased of one *period* at a time (e.g. day, week, month, quarter, year) from the lower to the upper time limit. For each increase, an expected Data Point is identified.

4562 If this expected Data Points is missing, it is added to the Data Set. For the added Data Points, Measures and Attributes assume the NULL value.

4563 The output Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set. The output Data Set contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.

4564 As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the reference time Identifier as well as the *period* of each time series.

4565

4577 *Examples*

4578 Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

4579

4580

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!"

4581

4582

4583

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

DS_r		
Id_1	Id_2	Me_1
A	2010-01/2010-12	"hello world"

A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
B	2011-01/2011-12	"hi, hello! "
B	2012-01/2012-12	"hi"
B	2013-01/2013-12	NULL
B	2014-01/2014-12	"hello!"

4584
4585
4586

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

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010-01/2010-12	"hello world"
A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
A	2014-01/2014-12	NULL
B	2010-01/2010-12	NULL
B	2011-01/2011-12	"hi, hello! "
B	2012-01/2012-12	"hi"
B	2013-01/2013-12	NULL
B	2014-01/2014-12	"hello!"

4587
4588
4589
4590

Given the Data Set DS_2, which contains *yearly* time series, where Id_2 is the reference time Identifier of *date* type and conventionally each period is identified by its last day:

DS_2		
Id_1	Id_2	Me_1
A	2010-12-31	"hello world"
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2011-12-31	"hi, hello! "
B	2012-12-31	"hi"
B	2014-12-31	"hello!"

4591
4592
4593

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

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2011-12-31	"hi, hello! "

B	2012-12-31	"hi"
B	2013-12-31	NULL
B	2014-12-31	"hello!"

4594
4595
4596

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

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
A	2014-12-31	NULL
B	2010-12-31	NULL
B	2011-12-31	"hi, hello! "
B	2012-12-31	"hi"
B	2013-12-31	NULL
B	2014-12-31	"hello!"

4597
4598
4599
4600
4601

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

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

4602
4603
4604

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

results in:

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

4605
4606
4607

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

results in:

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

4608
4609
4610
4611
4612

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

DS_4		
Id_1	Id_2	Me_1
A	2010Y	"hello world"
A	2012Y	"say hello"
A	2010Q1	"he"
A	2010Q2	"hi, hello! "
A	2010Q4	"hi"
A	2011Q2	"hello!"

4613
4614
4615

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

results in:

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

4616
4617
4618

Example 8: DS_r := fill_time_series (DS_4, all)

results in:

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

4619

4620

4621 **Flow to stock :** **flow_to_stock**

4622

4623 *Syntax*

4624 **flow_to_stock** (op)

4625

4626 *Input Parameters*

4627 op the operand

4628

4629 *Examples of valid syntaxes*

4630 flow_to_stock (ds_1)

4631

4632 *Semantics for scalar operations*

4633 This operator does not perform scalar operations.

4634

4635 *Input parameters type:*

4636 op :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

4637

4638 *Result type:*

4639 result :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

4640

4641 *Additional constraints*

4642 The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

4643

4644 *Behaviour*

4645 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”.

4647 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”.

4649 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.

4653 The flow_to_stock operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4654 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.

4657 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.

4659

4660

4661 *Examples*

4662

4663 Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

4664

4665

DS_1		
Id_1	Id_2	Me_1
A	2010-01/2010-12	2
A	2011-01/2011-12	5
A	2012-01/2012-12	-3
A	2013-01/2013-12	9
B	2010-01/2010-12	4

B	2011-01/2011-12	-8
B	2012-01/2012-12	0
B	2013-01/2013-12	6

4666
4667
4668

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

DS_r		
Id_1	Id_2	Me_1
A	2010-01/2010-12	2
A	2011-01/2011-12	7
A	2012-01/2012-12	4
A	2013-01/2013-12	13
B	2010-01/2010-12	4
B	2011-01/2011-12	-4
B	2012-01/2012-12	-4
B	2013-01/2013-12	2

4669
4670
4671
4672
4673

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

DS_2		
Id_1	Id_2	Me_1
A	2010-12-31	2
A	2011-12-31	5
A	2012-12-31	-3
A	2013-12-31	9
B	2010-12-31	4
B	2011-12-31	-8
B	2012-12-31	0
B	2013-12-31	6

4674
4675
4676

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

DS_r		
Id_1	Id_2	Me_1
A	2010-12-31	2
A	2011-12-31	7
A	2012-12-31	4
A	2013-12-31	13
B	2010-12-31	4
B	2011-12-31	-4
B	2012-12-31	-4
B	2013-12-31	2

4677
4678
4679
4680

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

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

4681
4682
4683

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

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

4684
4685
4686
4687
4688

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

DS_4		
Id_1	Id_2	Me_1
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
A	2010Q1	2
A	2010Q2	-3
A	2010Q3	7
A	2010Q4	-4

4689
4690
4691

Example 4: DS_r := flow_to_stock (DS_3) results in:

DS_r		
Id_1	Id_2	Me_1
A	2010Y	2
A	2011Y	9
A	2012Y	13
A	2013Y	26
A	2010Q1	2
A	2010Q2	-1
A	2010Q3	6
A	2010Q4	2

4692
4693

4694 **Stock to flow :** **stock_to_flow**

4695

4696 *Syntax*

4697 **stock_to_flow (op)**

4698

4699 *Input parameters*

4700

4701 op the operand

4702

4703 *Examples of valid syntaxes*

4704 stock_to_flow (ds_1)

4705

4706 *Semantics for scalar operations*

4707 This operator does not perform scalar operations.

4708

4709 *Input parameters type:*

4710 op :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

4711

4712 *Result type:*

4713 result :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

4714

4715 *Additional constraints*

4716 The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

4717

4718 *Behaviour*

4719 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”.

4720 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”.

4721 This operator takes in input a Data Set of time series which is interpreted as stock data and, for each time series, calculates the corresponding flow data by subtracting from the measure values of each regular period the corresponding measure values of the previous one.

4722 The **stock_to_flow** operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4723 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.

4724 The Attribute propagation rule is not applied.

4725 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.

4726

4727
4728
4729
4730
4731
4732
4733

4734
4735
4736
4737
4738
4739

Examples

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

DS_1		
Id_1	Id_2	Me_1
A	2010-01/2010-12	2
A	2011-01/2011-12	7
A	2012-01/2012-12	4
A	2013-01/2013-12	13
B	2010-01/2010-12	4
B	2011-01/2011-12	-4
B	2012-01/2012-12	-4
B	2013-01/2013-12	2

4740
4741
4742

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

DS_r		
Id_1	Id_2	Me_1
A	2010-01/2010-12	2
A	2011-01/2011-12	5
A	2012-01/2012-12	-3
A	2013-01/2013-12	9
B	2010-01/2010-12	4
B	2011-01/2011-12	-8
B	2012-01/2012-12	0
B	2013-01/2013-12	6

4743
4744
4745
4746
4747

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

DS_2		
Id_1	Id_2	Me_1
A	2010-12-31	2
A	2011-12-31	7
A	2012-12-31	4
A	2013-12-31	13
B	2010-12-31	4
B	2011-12-31	-4
B	2012-12-31	-4
B	2013-12-31	2

4748
4749

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

4750

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

4751
4752
4753
4754
4755

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

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

4756
4757
4758

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

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

4759
4760
4761
4762
4763
4764

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

DS_4		
Id_1	Id_2	Me_1
A	2010Y	2
A	2011Y	9
A	2012Y	13
A	2013Y	26
A	2010Q1	2
A	2010Q2	-1
A	2010Q3	6
A	2010Q4	2

4765
4766
4767

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

DS_r		
Id_1	Id_2	Me_1
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
A	2010Q1	2
A	2010Q2	-3
A	2010Q3	7
A	2010Q4	-4

4768

4769 Time shift : **timeshift**

4770 *Syntax*

4771 **timeshift** (op , shiftNumber)

4772

4773 *Input parameters*

4774 op the operand

4775 shiftNumber the number of periods to be shifted

4776

4777 *Examples of valid syntaxes*

4778 timeshift (DS_1, 2)

4779 timeshift (DS_1)

4780

4781 *Semantics for scalar operations*

4782 This operator does not perform scalar operations.

4783

4784 *Input parameters type:*

4785 op :: dataset { identifier < time > _ , identifier _ * }

4786 shiftNumber :: integer

4787

4788 *Result type:*

4789 result :: dataset { identifier < time > _ , identifier _ * }

4790

4791 *Additional constraints*

4792 The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

4793 .
 4794
 4795
 4796
 4797
 4798
 4799
 4800
 4801
 4802
 4803
 4804
 4805
 4806
 4807
 4808
 4809
 4810
 4811

Behaviour

This operator takes in input a Data Set of time series and, for each time series of the Data Set, shifts the reference time Identifier of a number of periods (of the time series) equal to the `shift_number` parameter. If `shift_number` is negative, the shift is in the past, otherwise in the future. For example, if the period of the time series is month and `shift_number` is -1 the reference time Identifier is shifted of two months in the past.

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

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

The Attribute propagation rule is not applied.

As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the *time* Identifier as well as the *period* of each data point.

Examples

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

DS_1		
Id_1	Id_2	Me_1
A	2010-01/2010-12	"hello world"
A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
B	2010-01/2010-12	"hi, hello! "
B	2011-01/2011-12	"hi"
B	2012-01/2012-12	NULL
B	2013-01/2013-12	"hello!"

4812
 4813
 4814

Example 1: `DS_r := time_shift (DS_1 , -1)` results in:

DS_r		
Id_1	Id_2	Me_1
A	2009-01/2009-12	"hello world"
A	2010-01/2010-12	NULL
A	2011-01/2011-12	"say hello"
A	2012-01/2012-12	"he"
B	2009-01/2009-12	"hi, hello! "
B	2010-01/2010-12	"hi"
B	2011-01/2011-12	NULL
B	2012-01/2012-12	"hello!"

4815
 4816
 4817
 4818
 4819

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

DS_2		
Id_1	Id_2	Me_1
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2010-12-31	"hi, hello! "
B	2011-12-31	"hi"
B	2012-12-31	NULL
B	2013-12-31	"hello!"

4820
4821
4822

Example 2: DS_r := time_shift (DS_2 , 2) results in:

DS_r		
Id_1	Id_2	Me_1
A	2012-12-31	"hello world"
A	2013-12-31	NULL
A	2014-12-31	"say hello"
A	2015-12-31	"he"
B	2012-12-31	"hi, hello! "
B	2013-12-31	"hi"
B	2014-12-31	NULL
B	2015-12-31	"hello!"

4823
4824
4825
4826
4827

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

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

4828
4829
4830

Example 3: DS_r := time_shift (DS_3 , 1) results in:

DS_r		
Id_1	Id_2	Me_1
A	2011Y	"hello world"

A	2012Y	NULL
A	2013Y	"say hello"
A	2014Y	"he"
B	2011Y	"hi, hello! "
B	2012Y	"hi"
B	2013Y	NULL
B	2014Y	"hello!"

4831
4832
4833
4834
4835

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

DS_4		
Id_1	Id_2	Me_1
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2013Y	"he"
A	2010Q1	"hi, hello! "
A	2010Q2	"hi"
A	2010Q3	NULL
A	2010Q4	"hello!"

4836
4837
4838

Example 4: DS_r := time_shift (DS_3 , -1) results in:

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

4839

4840 **Time aggregation : time_agg**

4841 The operator **time_agg** converts *time*, *date* and *time_period* values from a smaller to a larger duration.

4842
4843 *Syntax*
4844 **time_agg (periodIndTo { , periodIndFrom } { , op } { , first | last })**

4845
4846 *Input parameters*

4847 op the scalar value, the Component or the Data Set to be converted. If not specified, then
 4848 **time_agg** is used in combination within an aggregation operator
 4849 periodIndFrom the source period indicator
 4850 periodIndTo the target period indicator

4851
 4852

Examples of valid syntaxes

4854 sum (DS group all time_agg (Me, "A"))
 4855 time_agg ("A", cast ("2012Q1", time_period , "YYYY\Qq"))
 4856 time_agg("M", cast ("2012-12-23", date, "YYYY-MM-DD"))
 4857 time_agg("M", DS1)
 4858 ds_2 := ds1[calc Me1 := time_agg("M",Me1)]

4859

Semantics for scalar operations

4861 The operator converts a *time*, *date* or *time_period* value from a smaller to a larger duration.

4862

Input parameters type

4864 op :: dataset { identifier < time > _ , identifier _* }
 4865 | component<time>
 4866 | time
 4867 periodIndFrom :: duration
 4868 periodIndTo :: duration

4869

Result type

4871 op :: dataset { identifier < time > _ , identifier _* }
 4872 | component<time>
 4873 | time

4874

Additional constraints

4876 If **op** is a Data Set then it has exactly an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.
 4877 It is only possible to convert smaller duration values to larger duration values (e.g. it is possible to convert
 4878 *monthly* data to *annual* data but the contrary is not allowed).

4879

Behaviour

4881 The scalar version of this operator takes as input a *time*, *date* or *time_period* value, converts it to **periodIndTo**
 4882 and returns a scalar of the corresponding type.

4883 The Data Set version acts on a single Measure Data Set of type *time*, *date* or *time_period* and returns a Data Set
 4884 having the same structure.

4885 Finally, VTL also provides a component version, for use in combination with an aggregation operator, because
 4886 the change of frequency requires an aggregation. In this case, the operator converts the **period_indicator** of the
 4887 data points (e.g., convert *monthly* data to *annual* data).

4888 On *time* type, the operator maps the input value into the comprising larger regular interval, whose duration is
 4889 the one specified by the **periodIndTo** parameter.

4890 On *date* type, the operator maps the input value into the comprising larger period, whose duration is the one
 4891 specified by the **periodIndTo** parameter, which is conventionally represented either by the start or by the end
 4892 date, according to the **first/last** parameter.

4893 On *time_period* type, the operator maps the input value into the comprising larger time period specified by the
 4894 **periodIndTo** parameter (the original period indicator is converted in the target one and the number of periods is
 4895 adjusted correspondingly).

4896 The input duration **periodIndFrom** is optional. In case of *time_period* Data Points, the input duration can be
 4897 inferred from the internal representation of the value. In case of *time* or *date* types, it is inferred by the
 4898 implementation. Filters on input time series can be obtained with the **filter** clause.

4899

4900

Examples

4901 Given the Data Set DS_1

4902
 4903

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

4904
4905
4906

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

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

4907
4908
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4920
4921
4922
4923
4924

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

Returns: "2012Q1".

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

time_agg("Q", cast("20120213", date, "YYYYMMDD"), _ , false)

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

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

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

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

4925 Actual time : current_date

4926
4927
4928
4929
4930
4931
4932

Syntax

current_date ()

Input parameters

None

Examples of valid syntax

current_date

4935
4936
4937

Semantics for scalar operations

The operator **current_date** returns the current time as a date type.

4938
4939 *Input parameters type*
4940 This operator has no input parameters.
4941
4942 *Result type*
4943 result :: date
4944
4945 *Additional constraints*
4946 None.
4947
4948 *Behaviour*
4949 The operator return the current date
4950
4951 *Examples*
4952 cast (current_date, string, "YYYY.MM.DD")
4953

VTL-ML - Set operators

4954

4955

Union: **union**

4956

4957

Syntax

4958

union (dsList)

4959

4960

dsList ::= ds { , ds }*

4961

4962

Input parameters

4963

dsList the list of Data Sets in the union

4964

4965

Examples of valid syntaxes

4966

union (ds2, ds3)

4967

4968

Semantics for scalar operations

4969

This operator does not perform scalar operations.

4970

4971

Input parameters type

4972

ds :: dataset

4973

4974

Result type

4975

result :: dataset

4976

4977

Additional constraints

4978

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

4979

4980

Behaviour

4981

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.

4982

4983

The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by choosing the Data Point belonging to the left most operand Data Set. For instance, let's assume that in **union (ds1, ds2)** the operand ds1 contains a Data Point dp1 and the operand ds2 contains a Data Point dp2 such that dp1 has the same Identifiers values of dp2, then the resulting Data Set contains dp1 only.

4984

4985

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

4986

The automatic Attribute propagation is not applied.

4987

4988

Examples

4989

Given the operand Data Sets DS_1 and DS_2:

4990

4991

4992

4993

4994

4995

4996

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

4997

4998

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1

2012	N	Total	Total	23
2012	S	Total	Total	5

4999
5000
5001
5002

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

5003
5004
5005

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

5006
5007

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

5008
5009
5010
5011

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

5012
5013
5014
5015
5016
5017
5018
5019

Intersection : intersect

Syntax

intersect (dsList)

dsList ::= ds { , ds }*

Input parameters

dsList the list of Data Sets in the intersection

5020
 5021
 5022
 5023
 5024
 5025
 5026
 5027
 5028
 5029
 5030
 5031
 5032
 5033
 5034
 5035
 5036
 5037
 5038
 5039
 5040
 5041
 5042
 5043
 5044
 5045
 5046
 5047
 5048
 5049
 5050
 5051

Examples of valid syntaxes

intersect (ds2, ds3)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds :: dataset

Return type

result :: dataset

Additional constraints

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

Behaviour

The **intersect** operator implements the intersection of functions (i.e., Data Sets). The resulting Data Set has the same Identifier, Measure and Attribute Components of the operand Data Sets specified in the dsList, and contains the Data Points belonging to all the operand Data Sets.

The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by 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.

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

The automatic Attribute propagation is not applied.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3

5052
 5053

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1
2011	B	Total	Total	10
2012	G	Total	Total	2
2011	M	Total	Total	40

5054
 5055
 5056

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

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	G	Total	Total	2

5057

5058 **Set difference :** **setdiff**

5059

5060 *Syntax*

5061 **setdiff (ds1, ds2)**

5062

5063 *Input parameters*

5064 ds1 the first Data Set in the difference (the minuend)

5065 ds2 the second Data Set in the difference (the subtrahend)

5066

5067 *Examples of valid syntaxes*

5068 setdiff (ds2, ds3)

5069

5070 *Semantics for scalar operations*

5071 This operator cannot be applied to scalar values.

5072

5073 *Input parameters type*

5074 ds1, ds2 :: dataset

5075

5076 *Result type*

5077 result :: dataset

5078

5079 *Additional constraints*

5080 The operand Data Sets have the same Identifier, Measure and Attribute Components.

5081

5082 *Behaviour*

5083 The operator implements the set difference of functions (i.e. Data Sets), interpreting the Data Points of the input
5084 Data Sets as the elements belonging to the operand sets, the minuend and the subtrahend, respectively. The
5085 operator returns one single Data Set, with the same Identifier, Measure and Attribute Components as the
5086 operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second. In other
5087 words, for setdiff (ds1, ds2), the resulting Dataset contains all the data points Data Point dp1 of the operand ds1
5088 such that there is no Data Point dp2 of ds2 having the same values for homonym Identifier Components.

5089 The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical
5090 behaviours of the ML Operators”).

5091 The automatic Attribute propagation is not applied.

5092

5093 *Examples*

5094 Given the operand Data Sets DS_1 and DS_2:

5095

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

5096

5097

5098

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

5099
5100
5101

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

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	10

5102
5103
5104

Given the operand Data Sets DS_1 and DS_2 :

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

5105
5106

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

5107
5108
5109

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

DS_r			
Id_1	Id_2	Id_3	Me_1
R	T	2011	12

5110
5111

5112 **Simmetric difference :** **symdiff**

5113
5114

Syntax

symdiff (ds1, ds2)

5115
5116

Input parameters

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

5117
5118

Examples of valid syntaxes

symdiff (ds_2, ds_3)

5119
5120

Semantics for scalar operations

This operator cannot be applied to scalar values.

5121
5122

Input parameters type

5128 ds1, ds2 :: dataset

5129

5130 *Result type*

5131 result :: dataset

5132

5133 *Additional constraints*

5134 The operand Data Sets have the same Identifier, Measure and Attribute Components.

5135

5136 *Behaviour*

5137 The operator implements the symmetric set difference between functions (i.e. Data Sets), interpreting the Data Points of the input Data Sets as the elements in the operand Sets. The operator returns one Data Set, with the same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second and the Data Points that appear in the second Data Set but not in the first one.

5142 Data Points are compared to one another by Identifier Components. For symdiff (ds1, ds2), the resulting Data Set contains all the Data Points dp1 contained in ds1 for which there is no Data Point dp2 in ds2 with the same values for homonym Identifier components and all the Data Points dp2 contained in ds2 for which there is no Data Point dp1 in ds1 with the same values for homonym Identifier Components.

5146 The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical behaviours of the ML Operators”).

5148 The automatic Attribute propagation is not applied.

5149

5150 *Examples*

5151 Given the operand Data Sets DS_1 and DS_2 :

5152

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

5153

5154

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

5155

5156 *Example 1:* DS_r := symdiff (DS_1, DS_2) results in:

5157

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

5158

5160 Hierarchical roll-up : **hierarchy**5161 *Syntax*5162 **hierarchy** (op , hr { **condition** condComp { , condComp }* } { **rule** ruleComp } { mode } { input } { output })5163 mode ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero**5164 input ::= **dataset | rule | rule_priority**5165 output ::= **computed | all**

5166

5167 *Input parameters*

5168 op the operand Data Set.

5169 hr the hierarchical Ruleset to be applied.

5170 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).

5171 ruleComp ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined in hr.

5174 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.5177 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.5179 output this parameter specifies the content of the resulting Data Set. The meaning of the possible values of the parameter is explained below.

5180

5181 *Examples of valid syntaxes*

5182 hierarchy (DS1, HR1 rule Id_1 non_null all)

5184 hierarchy (DS2, HR2 condition Comp_1, Comp_2 rule Id_3 non_zero rule computed)

5185

5186 *Semantics for scalar operations*

5187 This operator cannot be applied to scalar values.

5188

5189 *Input parameters type*

5190 op :: dataset { measure<number> _ }

5191 hr :: name < hierarchical >

5192 condComp :: name < component >

5193 ruleComp :: name < dentifier >

5194

5195 *Result type*

5196 result :: dataset {measure<number> _ }

5197

5198 *Additional constraints*

5199 If hr is defined on Value Domains then it is mandatory to specify the condition (if any) and the rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand op and must take values on the Value Domains corresponding, in positional order, to the ones specified in the condition and rule parameter of hr.

5203 If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in the condition and rule signatures of hr.

5206

5208 *Behaviour*5209 The **hierarchy** operator applies the rules of hr to op as specified in the parameters. The operator returns a Data Set with the same Identifiers and the same Measure as op. The Attribute propagation rule is applied on the groups of Data Points which contribute to the same Data Points of the result.

5212 The behaviours relevanto to the different options of the input parameters are the following.

5213 First, the parameter **input** is considered to determine the source of the Data Points used as input of the
5214 Hierarchy. The possible options of the parameter **input** and the corresponding behaviours are the following:

5215 **dataset** For each Rule of the Ruleset and for each item on the right hand side of the Rule, the operator
5216 takes the input Data Points exclusively from the operand **op**.

5217 **rule** For each Rule of the Ruleset and for each item on the right-hand side of the Rule:

- 5218 • if the item is not defined as the result (left-hand side) of another Rule, the current Rule
5219 takes the input Data Points from the operand **op**
- 5220 • if the item is defined as the result of another Rule, the current Rule takes the input Data
5221 Points from the computed output of such other Rule;

5222 **rule_priority** For each Rule of the Ruleset and for each item on the right-hand side of the Rule:

- 5223 • if the item is not defined as the result (left-hand side) of another rule, the current Rule
5224 takes the input Data Points from the operand **op**.
- 5225 • if the item is defined as the result of another Rule, then:
 - 5226 ○ if an expected input Data Point exists in the computed output of such other Rule
5227 and its Measure is not NULL, then the current Rule takes such Data Point;
 - 5228 ○ if an expected input Data Point does not exist in the computed output of such
5229 other Rule or its measure is NULL, then the current Rule takes the Data Point
5230 from **op** (if any) having the same values of the Identifiers;

5231 if the parameter **input** is not specified then it is assumed to be **rule**.

5232 Then the parameter **mode** is considered, to determine the behaviour for missing Data Points and for the Data
5233 Points to be produced in the output. The possible options of the parameter **mode** and the corresponding
5234 behaviours are the following:

5235 **non_null** the result Data Point is produced when its computed Measure value is not NULL (i.e., when no
5236 Data Point corresponding to the Code Items of the right side of the rule is missing or has NULL
5237 Measure value); in the calculation, the possible missing Data Points corresponding to the Code
5238 Items of the right side of the rule are considered existing and having a Measure value equal to
5239 NULL;

5240 **non_zero** the result Data Point is produced when its computed Measure value is not equal to 0 (zero);
5241 the possible missing Data Points corresponding to the Code Items of the right side of the rule
5242 are considered existing and having a Measure value equal to 0;

5243 **partial_null** the result Data Point is produced if at least one Data Point corresponding to the Code Items of
5244 the right side of the rule is found (whichever is its Measure value); the possible missing Data
5245 Points corresponding to the Code Items of the right side of the rule are considered existing and
5246 having a NULL Measure value;

5247 **partial_zero** the result Data Point is produced if at least one Data Point corresponding to the Code Items of
5248 the right side of the rule is found (whichever is its Measure value); the possible missing Data
5249 Points corresponding to the Code Items of the right side of the rule are considered existing and
5250 having a Measure value equal to 0 (zero);

5251 **always_null** the result Data Point is produced in any case; the possible missing Data Points corresponding
5252 to the Code Items of the right side of the rule are considered existing and having have a
5253 Measure value equal to NULL;

5254 **always_zero** the result Data Point is produced in any case; the possible missing Data Points corresponding
5255 to the Code Items of the right side of the rule are considered existing and having a Measure
5256 value equal to 0 (zero);

5257 If the parameter **mode** is not specified, then it is assumed to be **non_null**
5258

5259 The following table summarizes the behaviour of the options of the parameter “mode”

5260

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)

5261

5262 Finally the parameter output is considered, to determine the content of the resulting Data Set. The possible
5263 options of the parameter output and the corresponding behaviours are the following:

5264 **computed** the resulting Data Set contains only the set of Data Points computed according to the Ruleset
5265 **all** the resulting Data Set contains the union between the set of Data Points “R” computed
5266 according to the Ruleset and the set of Data Points of op that have different combinations of
5267 values for the Identifiers. In other words, the result is the outcome of the following (virtual)
5268 expression: union (setdiff (op , R) , R)

5269 If the parameter output is not specified then it is assumed to be computed.

5270

5271 *Examples*

5272 Given the following hierarchical ruleset:

```
5273
5274     define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
5275         A = J + K + L
5276         ; B = M + N + O
5277         ; C = P + Q
5278         ; D = R + S
5279         ; E = T + U + V
5280         ; F = Y + W + Z
5281         ; G = B + C
5282         ; H = D + E
5283         ; I = D + G
5284     end hierarchical ruleset
```

5285

5286 And given the operand Data Set DS_1 (where At_1 is viral and the propagation rule says that the alphabetic
5287 order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points
5288 is assumed as NULL):

5289

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

5290
5291
5292
5293

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

5294
5295
5296
5297

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

5298
5299
5300
5301

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

5302
5303

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

5309

5310 Aggregate invocation

5311 *Syntax*

5312

5313 *in a Data Set expression:*

5314 **aggregateOperator** (firstOperand { , additionalOperand }* { groupingClause })

5315

5316 *in a Component expression within an aggr clause)*

5317 **aggregateOperator** (firstOperand { , additionalOperand }*) { groupingClause }

5318

5319

5320 **aggregateOperator** ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**

5321 | **stddev_samp** | **sum** | **var_pop** | **var_samp**

5322 **groupingClause** ::= { **group by** groupingId { , groupingId }*

5323 | **group except** groupingId { , groupingId }*

5324 | **group all** conversionExpr }¹

5325 { **having** havingCondition_ }

5326

5327

5328 *Input Parameters*

5329 **aggregateOperator**

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

5330 **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)

5333 **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

5336 **groupingClause**

the following alternative grouping options:

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

group except the Data Points are grouped by the values of the Identifiers not specified as groupingId. The Identifiers specified as groupingId are dropped in the result.

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

5344 **groupingId**

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

5346 **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.

5349 **havingCondition**

a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which havingCondition evaluates to TRUE appear in the result. The havingCondition refers to the groups specified through the groupingClause, therefore it must invoke aggregate operators (e.g. **avg**, **count**, **max** ..., see also the corresponding sections). A correct example of havingCondition is:

max(obs_value) < 1000

while the condition obs_value < 1000 is not a right havingCondition, because it refers to the values of single Data Points and not to the groups. The count operator is used in a havingCondition without parameters, e.g.:

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

5361 *Examples of valid syntaxes*

5362 **avg** (DS_1)

5363 **avg** (DS_1 group by Id_1, Id_2)

5364 avg (DS_1 group except Id_1, Id_2)
5365 avg (DS_1 group all time_agg ("Q"))

5366

5367 *Semantics for scalar operations*

5368 The aggregate operators cannot be applied to scalar values.

5369

5370 *Input parameters type*

5371 firstOperand :: dataset
5372 | component
5373 additionalOperand :: see the type of the additional parameter (if any) of the invoked
5374 aggregateOperator. The aggregate operators and their parameters are
5375 described in the following sections.
5376 groupingId :: name < identifier >
5377 conversionExpr :: identifier
5378 havingCondition :: component<boolean>

5379

5380 *Result type:*

5381 result :: dataset
5382 | component

5383

5384 *Additional constraints*

5385 The Aggregate invocation cannot be nested in other Aggregate or Analytic invocations.

5386 The aggregate operations at component level can be invoked within the **aggr** clause, both as part of a join
5387 operator and the **aggr** operator (see the parameter **aggrExpr** of those operators).

5388 The basic scalar types of firstOperand and additionalOperand (if any) must be compliant with the specific basic
5389 scalar types required by the invoked operator (the required basic scalar types are described in the table at the
5390 beginning of this chapter and in the sections of the various operators below).

5391 The conversionExpr parameter applies just one conversion operator to just one Identifier belonging to the input
5392 Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion
5393 operator.

5394 If the grouping clause is omitted, then all the input Data Points are aggregated in a single group and the clause
5395 returns a Data Set that contains a single Data Point and has no Identifiers.

5396

5397 *Behaviour*

5398 The **aggregateOperator** is applied as usual to all the measures of the firstOperand Data Set (if invoked at Data
5399 Set level) or to the firstOperand Component of the input Data Set (if invoked at Component level). In both cases,
5400 the operator calculates the required aggregated values for groups of Data Points of the input Data Set. The
5401 groups of Data Points to be aggregated are specified through the groupingClause, which allows the following
5402 alternative options.

5403

5404 **group by** the Data Points are grouped by the values of the specified Identifiers. The Identifiers not
5405 specified are dropped in the result.

5406 **group except** the Data Points are grouped by the values of the Identifiers not specified in the clause. The
5407 specified Identifiers are dropped in the result.

5408 **group all** converts an Identifier Component using conversionExpr and keeps all the Identifiers.

5409

5410 The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the
5411 single groups (for example the minimum number of rows in the group).

5412 If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the operator
5413 returns a Data Set that contains a single Data Point and has no Identifiers.

5414 For the invocation at Data Set level, the resulting Data Set has the same Measures as the operand. For the
5415 invocation at Component level, the resulting Data Set has the Measures explicitly calculated (all the other
5416 Measures are dropped because no aggregation behaviour is specified for them).

5417 For invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level,
5418 the Attributes calculated within the **aggr** clause are maintained in the result; for all the other Attributes that are
5419 defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule
5420 section in the User Manual).

5421 As mentioned, the Aggregate invocation at component level can be done within the **aggr** clause, both as part of a
5422 Join operator and the **aggr** operator (see the parameter **aggrExpr** of those operators), therefore, for a better
5423 comprehension fo the behaviour at Component level, see also those operators.

5424
 5425
 5426
 5427
 5428
 5429

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

5430
 5431
 5432

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

5433
 5434
 5435
 5436
 5437
 5438
 5439
 5440

Note: the example above can be rewritten equivalently in the following forms:

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

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

DS_r		
Id_1	Id_3	Me_1
2010	XX	22
2010	YY	23
2011	XX	40
2011	ZZ	1
2011	YY	-1
2012	ZZ	40
2012	YY	30

5441
 5442
 5443

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

DS_r
Me_1
15.5

5444
5445
5446
5447
5448
5449

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

provided that At_1 is viral and the first letter in alphabetic order prevails and NULL prevails on all the other characters, results in:

DS_r			
Id_1	Me_2	Me_3	At_1
2010	23	1	
2011	20	-1	N
2012	40	30	P

5450

Analytic invocation

5451

Syntax

5452 analyticOperator (firstOperand { , additionalOperand }* **over** (analyticClause))

5453

5454 analyticOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**

5455 | **stddev_samp** | **sum** | **var_pop** | **var_samp**

5456 | **first_value** | **lag** | **last_value** | **lead** | **rank** | **ratio_to_report**

5457 analyticClause ::= { partitionClause } { orderClause } { windowClause }

5458 partitionClause ::= **partition by** identifier { , identifier }*

5459 orderClause ::= **order by** component { **asc** | **desc** } { , component { **asc** | **desc** } }*

5460 windowClause ::= { **data points** | **range** }¹ **between** limitClause **and** limitClause

5461 limitClause ::= { num **preceding** | num **following** | **current data point** | **unbounded preceding** | **unbounded following** }¹

5462

Parameters

5464

analyticOperator

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

5465

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)

5466

5467

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

5469

5470

analyticClause

clause that specifies the analytic behaviour

5472

partitionClause

clause that specifies how to partition Data Points in groups to be analysed separately. The input Data Set is partitioned according to the values of one or more Identifier Components. If the clause is omitted, then the Data Set is partitioned by the Identifier Components that are not specified in the orderClause.

5473

5474

5475

orderClause

clause that specifies how to order the Data Points. The input Data Set is ordered according to the values of one or more Components, in ascending order if **asc** is specified, in descending order if **desc** is specified, by default in ascending order if the **asc** and **desc** keywords are omitted.

5477

5478

5479

windowClause

clause that specifies how to apply a sliding window on the ordered Data Points. The keyword **data points** means that the sliding window includes a certain number of Data Points before and after the current Data Point in the order given by the orderClause. The keyword **range** means that the sliding windows includes all the Data Points whose values are in a certain range in respect to the value, for the current Data Point, of the Measure which the analytic is applied to.

5481

5482

5483

5484

5485

5486 limitClause clause that can specify either the lower or the upper boundaries of the sliding window.
5487 Each boundary is specified in relationship either to the whole partition or to the
5488 current data point under analysis by using the following keywords:
5489 • **unbounded preceding** means that the sliding window starts at the first Data Point
5490 of the partition (it make sense only as the first limit of the window)
5491 • **unbounded following** indicates that the sliding window ends at the last Data Point
5492 of the partition (it makes sense only as the second limit of the window)
5493 • **current data point** specifies that the window starts or ends at the current Data
5494 Point.
5495 • num **preceding** specifies either the number of **data points** to consider preceding
5496 the current data point in the order given by the orderClause (when **data points** is
5497 specified in the window clause), or the maximum difference to consider, as for the
5498 Measure which the analytic is applied to, between the value of the current Data
5499 Point and the generic other Data Point (when **range** is specified in the windows
5500 clause).
5501 • num **following** specifies either the number of data points to consider following the
5502 current data point in the order given by the orderClause (when **data points** is
5503 specified in the window clause), or the maximum difference to consider, as for the
5504 Measure which the analytic is applied to, between the values of the generic other
5505 Data Point and the current Data Point (when **range** is specified in the windows
5506 clause).
5507 If the whole windowClause is omitted then the default is **data points between**
5508 **unbounded preceding and current data point**.
5509 identifier an Identifier Component of the input Data Set
5510 component a Component of the input Data Set
5511 num a scalar *number*

5513 *Examples of valid syntaxes*

5514 sum (DS_1 over (partition by Id_1 order by Id_2))
5515 sum (DS_1 over (order by Id_2))
5516 avg (DS_1 over (order by Id_1 data points between 1 preceding and 1 following))
5517 DS_1 [calc M1 := sum (Me_1 over (order by Id_1))]

5519 *Semantics for scalar operations*

5520 The analytic operators cannot be applied to scalar values.

5522 *Input parameters type*

5523 firstOperand :: dataset
5524 | component
5525 additionalOperand :: see the type of the additional parameter (if any) of the invoked operator. The operators
5526 and their parameters are described in the following sections.
5527 identifier :: name < identifier >
5528 component :: name < component >
5529 num :: integer

5531 *Result type*

5532 result :: dataset
5533 | component

5535 *Additional constraints*

5536 The analytic invocation cannot be nested in other Aggregate or Analytic invocations.
5537 The analytic operations at component level can be invoked within the **calc** clause, both as part of a Join operator
5538 and the **calc** operator (see the parameter calcExpr of those operators).
5539 The basic scalar types of firstOperand and additionalOperand (if any) must be compliant with the specific basic
5540 scalar types required by the invoked operator (the required basic scalar types are described in the table at the
5541 beginning of this chapter and in the sections of the various operators below).
5542

5543 *Behaviour*

5544 The analytic Operator is applied as usual to all the Measures of the input Data Set (if invoked at Data Set level) or
5545 to the specified Component of the input Data Set (if invoked at Component level). In both cases, the operator
5546 calculates the desired output values for each Data Point of the input Data Set.

5547 The behaviour of the analytic operations can be procedurally described as follows:

- 5548 • The Data Points of the input Data Set are first partitioned (according to partitionBy) and then ordered
5549 (according to orderBy).
- 5550 • The operation is performed for each Data Point (named “current Data Point”) of the input Data Set. For each
5551 input Data Point, one output Data Point is returned, having the same values of the Identifiers. The analytic
5552 operator is applied to a “window” which includes a set of Data Points of the input Data Set and returns the
5553 values of the Measure(s) of the output Data Point.
 - 5554 • If windowClause is not specified, then the set of Data Points which contribute to the analytic operation is
5555 the whole partition which the current Data Point belongs to
 - 5556 • If windowClause is specified, then the set of Data Points is the one specified by windowClause (see
5557 windowClause and LimitClause explained above).

5558 For the invocation at Data Set level, the resulting Data Set has the same Measures as the input Data Set
5559 firstOperand. For the invocation at Component level, the resulting Data Set has the Measures of the input Data
5560 Set plus the Measures explicitly calculated through the **calc** clause.

5561 For the invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level,
5562 the Attributes calculated within the calc clause are maintained in the result; for all the other Attributes that are
5563 defined as viral, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule
5564 section in the User Manual).

5565 As mentioned, the Analytic invocation at component level can be done within the **calc** clause, both as part of a
5566 Join operator and the **calc** operator (see the parameter **aggrCalc** of those operators), therefore, for a better
5567 comprehension fo the behaviour at Component level, see also those operators.

5568 *Examples*

5571 Given the Data Set DS_1:

DS_r			
Id_1	Id_2	Id_3	Me_1
2010	E	XX	5
2010	B	XX	-3
2010	R	XX	9
2010	E	YY	13
2011	E	XX	11
2011	B	ZZ	7
2011	E	YY	-1
2011	F	XX	0
2012	L	ZZ	-2
2012	E	YY	3

5573 *Example1:*

5576 DS_r := sum (DS_1 over (order by Id_1, Id_2, Id_3 data points between 1 preceding and 1 following))

5577 results in:

DS_r			
Id_1	Id_2	Id_3	Me_1

2010	B	XX	2
2010	E	XX	15
2010	E	YY	27
2010	R	XX	29
2011	B	ZZ	27
2011	E	XX	17
2011	E	YY	10
2011	F	XX	2
2012	E	YY	1
2012	L	ZZ	1

5579 **Counting the number of data points:** **count**

5580 *Aggregate syntax*

- 5581 **count** (dataset { groupingClause }) *(in a Data Set expression)*
- 5582 **count** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*
- 5583 **count** () *(in an **having** clause)*

5584

5585 *Analytic syntax*

- 5586 **count** (dataset **over** (analyticClause)) *(in a Data Set expression)*
- 5587 **count** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5588

5589 *Input parameters*

- 5590 dataset the operand Data Set
- 5591 component the operand Component
- 5592 groupingClause see Aggregate invocation
- 5593 analyticClause see Analytic invocation

5594

5595 *Examples of valid syntaxes*

5596 See Aggregate and Analytic invocations above, at the beginning of the section.

5597

5598 *Semantics for scalar operations*

5599 This operator cannot be applied to scalar values.

5600

5601 *Input parameters type*

- 5602 dataset :: dataset
- 5603 component :: component

5604

5605 *Result type*

- 5606 result :: dataset { measure<integer> int_var }
| component<integer>

5608

5609 *Additional constraints*

5610 None.

5611

5612 *Behaviour*

5613 The operator returns the number of the input Data Points.
5614 For other details, see Aggregate and Analytic invocations.

5615

5616 *Examples*

5617 Given the Data Set DS_1:

5618

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

5619

5620

5621

5622

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

DS_r	
Id_1	Int_var
2011	3
2012	2

5623

5624

5625

5626

5627

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

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

DS_r	
Id_1	Int_var
2011	3

5628

5629

Minimum value : **min**

5630

Aggregate syntax

5631

min (dataset { groupingClause })

(in a Data Set expression)

5632

min (component) { groupingClause }

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

5633

5634

Analytic syntax

5635

min (dataset **over** (analyticClause))

(in a Data Set expression)

5636

min (component **over** (analyticClause))

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

5637

5638

Input parameters

5639

dataset the operand Data Set

5640

component the operand Component

5641

groupingClause see Aggregate invocation

5642

analyticClause see Analytic invocation

5643

5644

Examples of valid syntaxes

5645

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

5646

5647

Semantics for scalar operations

5648

This operator cannot be applied to scalar values.

5649

5650 *Input parameters type*
 5651 dataset :: dataset
 5652 component :: component

5653 *Result type*
 5654 result :: dataset
 5655 | component

5656 *Additional constraints*
 5657 None.

5658 *Behaviour*
 5659 The operator returns the minimum value of the input values.
 5660 For other details, see Aggregate and Analytic invocations.

5661 *Examples*
 5662 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

5668 *Example 1:* DS_r := min (DS_1 group by Id_1) results in:
 5669
 5670

DS_r	
Id_1	Me_1
2011	3
2012	2

5671 **Maximum value :** **max**

5672 *Aggregate syntax*
 5673 **max** (dataset { groupingClause }) *(in a Data Set expression)*
 5674 **max** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*

5675 *Analytic syntax*
 5676 **max** (dataset **over** (analyticClause)) *(in a Data Set expression)*
 5677 **max** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5678 *Input parameters*
 5679 dataset the operand Data Set
 5680 component the operand Component
 5681 groupingClause see Aggregate invocation
 5682 analyticClause see Analytic invocation

5686 *Examples of valid syntaxes*
 5687 See Aggregate and Analytic invocations above, at the beginning of the section.

5688 *Semantics for scalar operations*
 5689 This operator cannot be applied to scalar values.

5691 *Input parameters type*
 5692 dataset :: dataset
 5693 component :: component

5694 *Result type*
 5695 result :: dataset
 5696 | component

5697 *Additional constraints*
 5698 None.

5699 *Behaviour*
 5700 The operator returns the maximum of the input values.
 5701 For other details, see Aggregate and Analytic invocations.

5702 *Examples*
 5703 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

5710 *Example 1:* DS_r := max (DS_1 group by Id_1) results in:
 5711
 5712

DS_r	
Id_1	Me_1
2011	7
2012	4

5713 **Median value :** **median**

5714 *Aggregate syntax*
 5715 **median** (dataset { groupingClause }) *(in a Data Set expression)*
 5716 **median** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*

5717 *Analytic syntax*
 5718 **median** (dataset **over** (analyticClause)) *(in a Data Set expression)*
 5719 **median** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5720
 5721

5722 *Input parameters*
 5723 dataset the operand Data Set
 5724 component the operand Component
 5725 groupingClause see Aggregate invocation
 5726 analyticClause see Analytic invocation

5727
 5728 *Examples of valid syntaxes*
 5729 See Aggregate and Analytic invocations above, at the beginning of the section.

5730
 5731 *Semantics for scalar operations*
 5732 This operator cannot be applied to scalar values.

5733
 5734 *Input parameters type*
 5735 dataset :: dataset {measure<number>_+}
 5736 component :: component<number>

5737
 5738 *Result type*
 5739 result :: dataset { measure<number> _+ }
 5740 | component<number>

5741
 5742 *Additional constraints*
 5743 None.

5744
 5745 *Behaviour*
 5746 The operator returns the median value of the input values.
 5747 For other details, see Aggregate and Analytic invocations.

5748
 5749 *Examples*
 5750 Given the Data Set DS_1:
 5751

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

5752
 5753
 5754 *Example 1:* DS_r := median (DS_1 group by Id_1) results in:
 5755

DS_r	
Id_1	Me_1
2011	5
2012	3

5756 **Sum :** **sum**

5757 *Aggregate syntax*
 5758 **sum** (dataset { groupingClause }) *(in a Data Set expression)*
 5759 **sum** (component) { groupingClause } *(in a Component expression within an aggr clause)*

5760

5761 *Analytic syntax*
 5762 **sum (dataset over (analyticClause))** *(in a Data Set expression)*
 5763 **sum (component over (analyticClause))** *(in a Component expression within a **calc** clause)*
 5764

5765 *Input parameters*
 5766 dataset the operand Data Set
 5767 component the operand Component
 5768 groupingClause see Aggregate invocation
 5769 analyticClause see Analytic invocation
 5770

5771 *Examples of valid syntaxes*
 5772 See Aggregate and Analytic invocations above, at the beginning of the section.
 5773

5774 *Semantics for scalar operations*
 5775 This operator cannot be applied to scalar values.
 5776

5777 *Input parameters type*
 5778 dataset :: dataset { measure<number> _+ }
 5779 component :: component<number>
 5780

5781 *Result type*
 5782 result :: dataset { measure<number> _+ }
 5783 | component<number>
 5784

5785 *Additional constraints*
 5786 None.
 5787

5788 *Behaviour*
 5789 The operator returns the sum of the input values.
 5790 For other details, see Aggregate and Analytic invocations.
 5791

5792 *Examples*
 5793 Given the Data Set DS_1 :
 5794

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

5795
 5796 *Example 1:* DS_r := sum (DS_1 group by Id_1) results in:
 5797

DS_r	
Id_1	Me_1
2011	15
2012	6

5798

5799 **Average value :** **avg**

5800 *Aggregate syntax*

5801 **avg (dataset { groupingClause })** *(in a Data Set expression)*

5802 **avg (component) { groupingClause }** *(in a Component expression within an **aggr** clause)*

5803
5804 *Analytic syntax*

5805 **avg (dataset **over** (analyticClause))** *(in a Data Set expression)*

5806 **avg (component **over** (analyticClause))** *(in a Component expression within a **calc** clause)*

5807
5808 *Input parameters*

5809 **dataset** the operand Data Set
5810 **component** the operand Component
5811 **groupingClause** see Aggregate invocation
5812 **analyticClause** see Analytic invocation

5813
5814 *Examples of valid syntaxes*

5815 See Aggregate and Analytic invocations above, at the beginning of the section.

5816
5817 *Semantics for scalar operations*

5818 This operator cannot be applied to scalar values.

5819
5820 *Input parameters type*

5821 **dataset ::** dataset {measure<number> _+}
5822 **component ::** component<number>

5823
5824 *Result type*

5825 **result ::** dataset { measure<number> _+ }
5826 | component<number>

5827 *Additional constraints*

5828 None.

5829
5830 *Behaviour*

5831 The operator returns the mean of the input values.
5832 For other details, see Aggregate and Analytic invocations.

5833
5834 *Examples*

5835 Given the Data Set DS_1:

5836

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

5837

5838 *Example 1:* **DS_r := avg (DS_1 group by Id_1)**

results in:

5839

DS_r	
Id_1	Me_1
2011	5

2012	3
------	---

5840

5841 Population standard deviation : **stddev_pop**

5842 *Aggregate syntax*

5843 **stddev_pop** (dataset { groupingClause }) *(in a Data Set expression)*

5844 **stddev_pop** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*

5845

5846 *Analytic syntax*

5847 **stddev_pop** (dataset **over** (analyticClause)) *(in a Data Set expression)*

5848 **stddev_pop** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5849

5850 *Input parameters*

5851 dataset the operand Data Set
 5852 component the operand Component
 5853 groupingClause see Aggregate invocation
 5854 analyticClause see Analytic invocation

5855

5856 *Examples of valid syntaxes*

5857 See Aggregate and Analytic invocations above, at the beginning of the section.

5858

5859 *Semantics for scalar operations*

5860 This operator cannot be applied to scalar values.

5861

5862 *Input parameters type*

5863 dataset :: dataset { measure<number> _+ }
 5864 component :: component<number>

5865

5866 *Result type*

5867 result :: dataset { measure<number> _+ }
 5868 | component<number>

5869

5870 *Additional constraints*

5871 None.

5872

5873 *Behaviour*

5874 The operator returns the “population standard deviation” of the input values.

5875 For other details, see Aggregate and Analytic invocations.

5876

5877 *Examples*

5878

5879 Given the Data Set DS_1:

5880

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

5881

5882 *Example 1:* DS_r := stddev_pop (DS_1 group by Id_1) results in:

5883

DS_r	
Id_1	Me_1
2011	1.633
2012	1

5884

5885 Sample standard deviation : **stddev_samp**

5886 *Aggregate syntax*

5887 **stddev_samp** (dataset { groupingClause })

(in a Data Set expression)

5888 **stddev_samp** (component) { groupingClause }

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

5889

5890 *Analytic syntax*

5891 **stddev_samp** (dataset **over** (analyticClause))

(in a Data Set expression)

5892 **stddev_samp** (component **over** (analyticClause))

*(in a Component expr. within a **calc** clause)*

5893

5894 *Input parameters*

5895 dataset the operand Data Set

5896 component the operand Component

5897 groupingClause see Aggregate invocation

5898 analyticClause see Analytic invocation

5899

5900 *Semantics for scalar operations*

5901 This operator cannot be applied to scalar values.

5902

5903 *Examples of valid syntaxes*

5904 See Aggregate and Analytic invocations above, at the beginning of the section.

5905

5906 *Input parameters type*

5907 dataset :: dataset { measure<number> _+ }

5908 component :: component<number>

5909

5910 *Result type*

5911 result :: dataset { measure<number> _+ }

5912 | component<number>

5913

5914 *Additional constraints*

5915 None.

5916

5917 *Behaviour*

5918 The operator returns the “sample standard deviation” of the input values.

5919 For other details, see Aggregate and Analytic invocations.

5920

5921 *Examples*

5922 Given the Data Set DS_1:

5923

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

5924
5925 *Example 1:* DS_r := stddev_samp (DS_1 group by Id_1) results in:
5926

DS_r	
Id_1	Me_1
2011	2
2012	1.4142

5927

5928 **Population variance :** **var_pop**

5929 *Aggregate syntax*

5930 **var_pop** (dataset { groupingClause }) *(in a Data Set expression)*

5931 **var_pop** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*

5932

5933 *Analytic syntax*

5934 **var_pop** (dataset **over** (analyticClause)) *(in a Data Set expression)*

5935 **var_pop** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5936

5937 *Input parameters*

5938 dataset the operand Data Set
5939 component the operand Component
5940 groupingClause see Aggregate invocation
5941 analyticClause see Analytic invocation

5942

5943 *Examples of valid syntaxes*

5944 See Aggregate and Analytic invocations above, at the beginning of the section.

5945

5946 *Semantics for scalar operations*

5947 This operator cannot be applied to scalar values.

5948

5949 *Input parameters type*

5950 dataset :: dataset {measure<number>_+}

5951 component :: component<number>

5952

5953 *Result type*

5954 result :: dataset { measure<number> _+ }

5955 | component<number>

5956

5957 *Additional constraints*

5958 None.

5959

5960 *Behaviour*

5961 The operator returns the “population variance” of the input values.

5962 For other details, see Aggregate and Analytic invocations.

5963

5964 *Examples*

5965 Given the Data Set DS_1 :

5966

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

5967
5968
5969

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

DS_r	
Id_1	Me_1
2011	2,6667
2012	1

5970 **Sample variance : var_samp**

5971 *Aggregate syntax*
5972 **var_samp** (dataset { groupingClause }) *(in a Data Set expression)*
5973 **var_samp** (component) { groupingClause } *(in a Component expression within an **aggr** clause)*

5974 *Analytic syntax*
5975 **var_samp** (dataset **over** (analyticClause)) *(in a Data Set expression)*
5976 **var_samp** (component **over** (analyticClause)) *(in a Component expression within a **calc** clause)*

5978 *Input parameters*
5979 **dataset** the operand Data Set
5980 **component** the operand Component
5981 **groupingClause** see Aggregate invocation
5982 **analyticClause** see Analytic invocation

5984 *Examples of valid syntaxes*
5985 See Aggregate and Analytic invocations above, at the beginning of the section.

5987 *Semantics for scalar operations*
5988 This operator cannot be applied to scalar values.

5990 *Input parameters type*
5991 **dataset** :: dataset {measure<number>_+}
5992 **component** :: component<number>

5994 *Result type*
5995 **result** :: dataset { measure<number> _+ }
5996 | component<number>

5998 *Additional constraints*
6000 None.

6001 *Behaviour*
6002 The operator returns the sample variance of the input values.
6003

6004 For other details, see Aggregate and Analytic invocations.

6005

6006 *Examples*

6007

6008 Given the Data Set DS_1

6009

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

6010

6011 *Example 1:* DS_r := var_samp (DS_1 group by [Id_1]) results in:

6012

DS_r	
Id_1	Me_1
2011	4
2012	2

6013

6014 **First value :** `first_value`

6015 *Syntax*

6016 **first_value** (dataset **over** (analyticClause)) *(in a Data Set expression)*

6017 **first_value** (component **over** (analyticClause)) *(in a Component expression within a calc clause)*

6018

6019 *Input parameters*

6020 dataset the operand Data Set

6021 component the operand Component

6022 analyticClause see Analytic invocation

6023

6024 *Examples of valid syntaxes*

6025 See Analytic invocation above, at the beginning of the section.

6026

6027 *Semantics for scalar operations*

6028 This operator cannot be applied to scalar values.

6029

6030 *Input parameters type*

6031 dataset :: dataset { measure<scalar> _+ }

6032 component :: component<scalar>

6033

6034 *Result type*

6035 result :: dataset

6036 | component<scalar>

6037

6038 *Additional constraints*

6039 The Aggregate invocation is not allowed.

6040

6041 *Behaviour*
 6042 The operator returns the first value (in the value order) of the set of Data Points that belong to the same analytic
 6043 window as the current Data Point.
 6044 When invoked at Data Set level, it returns the first value for each Measure of the input Data Set. The first value of
 6045 different Measures can result from different Data Points.
 6046 When invoked at Component level, it returns the first value of the specified Component.
 6047 For other details, see Analytic invocation.

6048
 6049 *Examples*
 6050 Given the Data Set DS_1 :
 6051

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

6052
 6053 *Example 1:*
 6054
 6055 DS_r := first_value (DS_1 over (partition by Id_1, Id_2 order by Id_3 data points between 1 preceding and
 6056 1 following))
 6057
 6058 results in:
 6059

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

6060
 6061 **Last value :** **last_value**

6062 *Syntax*
 6063 **last_value** (dataset **over** (analyticClause)) *(in a Data Set expression)*
 6064 **last_value** (component **over** (analyticClause)) *(in a Component expression within a calc clause)*

6065
 6066 *Input parameters*
 6067 dataset the operand Data Set

6068 component the operand Component
 6069 analyticClause see Analytic invocation

6070
 6071 *Examples of valid syntaxes*

6072 See Analytic invocation above, at the beginning of the section.

6073
 6074 *Semantics for scalar operations*

6075 This operator cannot be applied to scalar values.

6076
 6077 *Input parameters type*

6078 dataset :: dataset {measure<scalar> _+}
 6079 component :: component<scalar>

6080
 6081 *Result type*

6082 result :: dataset
 6083 | component<scalar>

6084
 6085 *Additional constraints*

6086 The Aggregate invocation is not allowed.

6087
 6088 *Behaviour*

6089 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.

6090 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.

6091 When invoked at Component level, it returns the last value of the specified Component.

6092 For other details, see Analytic invocation.

6093
 6094
 6095
 6096 *Examples*

6097 Given the Data Set DS_1:

6099

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

6100

6101

6102 *Example 1:*

6103

6104 DS_r := last_value (DS_1 over (partition by Id_1, Id_2 order by Id_3 data points between 1 preceding and
 6105 1 following))

6106

6107 results in:

6108

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

6109

6110 **Lag :** **lag**

6111 *Syntax*

6112

6113 *in a Data Set expression:*

6114 **lag** (dataset {, offset {, default**Value** } } **over** ({ partitionClause } orderClause))

6115

6116 *In a Component expression within a **calc** clause:*

6117 **lag** (component {, offset {, default**Value** } } **over** ({ partitionClause } orderClause))

6118

6119 *Input parameters*

- 6120 dataset the operand Data Set
- 6121 component the operand Component
- 6122 offset the relative position prior to the current Data Point
- 6123 default**Value** the value returned when the offset goes outside of the partition.
- 6124 partitionClause see Analytic invocation
- 6125 orderClause see Analytic invocation

6126

6127 *Examples of valid syntaxes*

6128 See Analytic invocation above, at the beginning of the section.

6129

6130 *Semantics for scalar operations*

6131 This operator cannot be applied to scalar values.

6132

6133 *Input parameters type*

- 6134 dataset :: dataset
- 6135 component :: component
- 6136 offset :: integer [value > 0]
- 6137 default value :: scalar

6138

6139 *Result type*

- 6140 result :: dataset
- 6141 | component

6142

6143 *Additional constraints*

6144 The Aggregate invocation is not allowed.

6145 The windowClause of the Analytic invocation syntax is not allowed.

6146

6147 *Behaviour*

6148 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.

6149 If default**Value** is not specified then the value returned when the offset goes outside the partition is NULL.

6150

6151 For other details, see Analytic invocation.

6152

6153 *Examples*

6154 Given the Data Set DS_1 :

6155

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

6156
6157
6158
6159

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

results in:

DS_r				
Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	NULL	NULL
A	XX	1994	3	1
A	XX	1995	4	9
A	XX	1996	7	5
A	YY	1993	NULL	NULL
A	YY	1994	9	3
A	YY	1995	5	4
A	YY	1996	10	2

6160

6161 **lead :** **lead**

6162 *Syntax*

6163
6164 *in a Data Set expression:*

6165 **lead** (dataset , {offset {, defaultValue} } **over** ({ partitionClause } orderClause))

6166
6167 *in a Component expression within a calc clause:*

6168 **lead** (component , {offset {, defaultValue} } **over** ({ partitionClause } orderClause))

6169
6170 *Input parameters*

6171 **dataset** the operand Data Set
6172 **component** the operand Component
6173 **offset** the relative position beyond the current Data Point
6174 **defaultValue** the value returned when the offset goes outside the partition.
6175 **partitionClause** see Analytic invocation
6176 **orderClause** see Analytic invocation

6177
6178 *Examples of valid syntaxes*

6179 See Analytic invocation above, at the beginning of the section.

6180
6181 *Semantics for scalar operations*

6182 This operator cannot be applied to scalar values.

6183

6184 *Input parameters type*
 6185 dataset :: dataset
 6186 component :: component
 6187 offset :: integer [value > 0]
 6188 default value :: scalar

6189 *Result type*
 6191 result :: dataset
 6192 | component

6194 *Additional constraints*
 6195 The Aggregate invocation is not allowed.
 6196 The windowClause of the Analytic invocation syntax is not allowed.

6197 *Behaviour*
 6199 In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data
 6200 Point at the specified physical offset beyond the current Data Point.
 6201 If defaultValue is not specified, then the value returned when the offset goes outside the partition is NULL.
 6202 For other details, see Analytic invocation.

6204 *Examples*
 6205 Given the Data Set DS_1
 6206

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

6207 *Example 1:* DS_r := lead (DS_1 , 1 over (partition by Id_1 , Id_2 order by Id_3)) results in:
 6208
 6209

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

6210

6211 Rank : rank

6212 *Syntax*

6213 **rank (over ({ partitionClause } orderClause))** *(in a Component expression within a calc clause)*

6214

6215 *Input parameters*

6216 partitionClause see Analytic invocation

6217 orderClause see Analytic invocation

6218

6219 *Examples of valid syntaxes*

6220 See Analytic invocation above, at the beginning of the section.

6221

6222 *Semantics for scalar operations*

6223 This operator cannot be applied to scalar values.

6224

6225 *Input parameters type*

6226 dataset :: dataset

6227 component :: component

6228

6229 *Result type*

6230 result :: dataset { measure<integer> int_var }

6231 | component<integer>

6232

6233 *Additional constraints*

6234 The invocation at Data Set level is not allowed.

6235 The Aggregate invocation is not allowed.

6236 The windowClause of the Analytic invocation syntax is not allowed.

6237

6238 *Behaviour*

6239 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).

6243 For other details, see Analytic invocation.

6244

6245 *Examples*

6246 Given the Data Set DS_1:

6247

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

6248

6249

6250 *Example 1:*

6251

6252 DS_r := DS_1 [calc Me2 := rank (over (partition by Id_1 , Id_2 order by Me_1)) results in:

6253

DS_r				
Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	2
A	XX	2002	7	4
A	XX	2003	6	3
A	YY	2000	9	3
A	YY	2001	5	1
A	YY	2002	10	4
A	YY	2003	5	1

6254

6255 **Ratio to report :** **ratio_to_report**

6256 *Syntax*

6257 **ratio_to_report** (dataset **over** (partitionClause))

(in a Data Set expression)

6258 **ratio_to_report** (component **over** (partitionClause))

*(in a Component expr. within a **calc** clause)*

6259

6260 *Input parameters*

6261 **dataset** the operand Data Set
6262 **component** the operand Component
6263 **partitionClause** see Analytic invocation

6264

6265 *Examples of valid syntaxes*

6266 See Analytic invocation above, at the beginning of the section.

6267

6268 *Semantics for scalar operations*

6269 This operator cannot be applied to scalar values.

6270

6271 *Input parameters type*

6272 **dataset** :: dataset { measure<number>_+ }
6273 **component** :: component<number>

6274

6275 *Result type*

6276 **result** :: dataset { measure<number>_+ }
6277 | component<number>

6278

6279 *Additional constraints*

6280 The Aggregate invocation is not allowed.

6281 The orderClause and windowClause of the Analytic invocation syntax are not allowed.

6282

6283 *Behaviour*

6284 The operator returns the ratio between the value of the current Data Point and the sum of the values of the partition which the current Data Point belongs to.

6285 For other details, see Analytic invocation.

6286

6287 *Examples*

6288 Given the Data Set DS_1:

6289

6290

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

6291
6292
6293
6294

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

6295

6297 **check_datapoint**6298 *Syntax*6299 **check_datapoint** (op , dpr { **components** listComp } { **output** output })6300 listComp ::= comp { , comp }*6301 output ::= **invalid** | **all** | **all_measures**6302 *Input parameters*

6303 op the Data Set to check

6304 dpr the Data Point Ruleset to be used

6305 listComp if dpr is defined on Value Domains then listComp is the list of Components of op to be
 6306 associated (in positional order) to the conditioning Value Domains defined in dpr. If dpr is
 6307 defined on Variables then listComp is the list of Components of op to be associated (in
 6308 positional order) to the conditioning Variables defined in dpr (for documentation purposes).

6309 comp Component of op

6310 output specifies the Data Points and the Measures of the resulting Data Set:

6311 **invalid** the resulting Data Set contains a Data Point for each Data Point of op and
 6312 each Rule in dpr that evaluates to FALSE on that Data Point. The resulting
 6313 Data Set has the Measures of op.

6314 **all** the resulting Data Set contains a data point for each Data Point of op and
 6315 each Rule in dpr. The resulting Data Set has the *boolean* Measure *bool_var*.

6316 **all_measures** the resulting Data Set contains a Data Point for each Data Point of op and
 6317 each Rule in dpr. The resulting dataset has the Measures of op and the
 6318 *boolean* Measure *bool_var*.

6319 If not specified then output is assumed to be invalid. See the Behaviour for further details.

6320 *Examples of valid syntaxes*

6321 check_datapoint (DS1, DPR invalid)

6322 check_datapoint (DS1, DPR all_measures)

6323

6324 *Semantics for scalar operations*

6325 This operator cannot be applied to scalar values.

6326

6327 *Input parameters type:*

6328 op :: dataset

6329 dpr :: name < datapoint >

6330 comp :: name < component >

6331

6332 *Result type:*

6333 result :: dataset

6334

6335 *Additional constraints*6336 If dpr is defined on Value Domains then it is mandatory to specify listComp. The Components specified in
 6337 listComp must belong to the operand op and be defined on the Value Domains specified in the signature of dpr.6338 If dpr is defined on Variables then the Components specified in the signature of dpr must belong to the operand
 6339 op.6340 If dpr is defined on Variables and listComp is specified then the Components specified in listComp are the same,
 6341 in the same order, as those specified in op (they are provided for documentation purposes).

6342

6343 *Behaviour*

6344 It returns a Data Set having the following Components:

- 6345 • the Identifier Components of op
- 6346 • the Identifier Component ruleid whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in dpr ⁸)
- 6347 • if the output parameter is **invalid**: the original Measures of op (no *boolean* measure)
- 6348 • if the output parameter is **all**: the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- 6349 • if the output parameter is **all_measures**: the original measures of op and the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- 6350 • the Measure errorcode that contains the errorcode specified in the rule
- 6351 • the Measure errorlevel that contains the errorlevel specified in the rule

6352 A Data Point of op can produce several Data Points in the resulting Data Set, each of them with a different value
 6353 of ruleid. If output is **invalid** then the resulting Data Set contains a Data Point for each Data Point of op and each
 6354 rule of dpr that evaluates to FALSE. If output is **all** or **all_measures** then the resulting Data Set contains a Data
 6355 Point for each Data Point of op and each rule of dpr.

6360 *Examples*

```

6361 define datapoint ruleset dpr1 ( variable Id_3, Me_1 ) is
6362     when Id_3 = "CREDIT" then Me_1 >= 0 errorcode "Bad credit"
6363     ; when Id_3 = "DEBIT" then Me_1 >= 0 errorcode "Bad debit"
6364 end datapoint ruleset
  
```

6365 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

6368 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	

6371 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		

6375

6376 check_hierarchy

6377 *Syntax*

6378 **check_hierarchy** (op , hr { **condition** condComp { , condComp }* } { **rule** ruleComp }
6379 { mode } { input } { output })

6380 mode ::= non_null | non_zero | partial_null | partial_zero | always_null | always_zero

6381 input ::= dataset | dataset_priority

6382 output ::= invalid | all | all_measures

6383

6384

6385 *Input parameters*

6386 op the Data Set to be checked

6387 hr the hierarchical Ruleset to be used

6388 condComp condComp is a Component of op to be associated (in positional order) to the conditioning
6389 Value Domains or Variables defined in hr (if any).

6390 ruleComp ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined
6391 in hr.

6392 mode this parameter specifies how to treat the possible missing Data Points corresponding to the
6393 Code Items in the left and right sides of the rules and which Data Points are produced in
6394 output. The meaning of the possible values of the parameter is explained below.

6395 input this parameter specifies the source of the values used as input of the comparisons. The
6396 meaning of the possible values of the parameter is explained below.

6397 output this parameter specifies the structure and the content of the resulting dataset. The meaning of
6398 the possible values of the parameter is explained below.

6399

6400 *Examples of valid syntaxes*

6401 check_hierarchy (DS1, HR_2 non_null dataset invalid)

6402 check_hierarchy (DS1, HR_3 non_zero dataset_priority all)

6403

6404 *Input parameters type*

6405 op :: dataset { measure<number> _ }

6406 hr :: name < hierarchical >

6407 condComp :: name < component >

6408 ruleComp :: name < identifier >

6409

6410 *Result type*

6411 result :: dataset { measure<number> _ }

6412

6413 *Additional constraints*

6414 If hr is defined on Value Domains then it is mandatory to specify the condition (if any in the ruleset hr) and the
6415 rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand

6416 op and must take values on the Value Domains corresponding, in positional order, to the ones specified in the
6417 condition and rule parameter of hr.

6418 If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be
6419 specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in
6420 this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in
6421 the condition and rule signatures of hr.

6422

6423

6424 *Behaviour*

6425

6426 The **check_hierarchy** operator applies the Rules of the Ruleset hr to check the Code Items Relations between
6427 the Code Items present in op (as for the Code Items Relations, see the User Manual - section "Generic Model for
6428 Variables and Value Domains"). The operator checks if the relation between the left and the right member is
6429 fulfilled, giving TRUE in positive case and FALSE in negative case.

6430

6431 The Attribute propagation rule is applied on each group of Data Points which contributes to the same Data Point
6432 of the result.

6433

6434 The behaviours relevanto to the different options of the input parameters are the following.

6435 First, the parameter input is used to determine the source of the Data Points used as input of the
6436 check_hierarchy. The possible options of the parameter input and the corresponding behaviours are the
6437 following:

6438 **dataset** this option addresses the case where all the input Data Points of all the Rules of the Ruleset are
6439 expected to be taken from the input Data Set (the operand op).

6440 For each Rule of the Ruleset and for each item on the left and right sides of the Rule, the
6441 operator takes the input Data Points exclusively from the operand op.

6442 **dataset_priority** this option addresses the case where the input Data Points of all the Rules of the Ruleset are
6443 preferably taken from the input Data Set (the operand op), however if a valid Measure value
6444 for an expected Data Point is not found in op, the attempt is made to take it from the computed
6445 output of a (possible) other Rule.

6446 For each Rule of the Ruleset and for each item on the left and right sides of the Rule:

6447 • if the item is not defined as the result (left side) of another Rule that applies the Code Item
6448 relation "is equal to" (=), the current Rule takes the input Data Points from the operand
6449 op.

6450 • if the item is defined as result of another Rule R that applies the Code Item relation "is
6451 equal to" (=), then:

6452 ○ if an expected input Data Point exists in op and its Measure is not NULL, then the
6453 current Rule takes such Data Point from op;

6454 ○ if an expected input Data Point does not exist in op or its measure is NULL, then
6455 the current Rule takes the Data Point (if any) that has the same Identifiers' values
6456 from the computed output of the other Rule R;

6457 if the parameter input is not specified then it is assumed to be dataset.

6458 Then the parameter mode is considered, to determine the behaviour for missing Data Points and for the Data
6459 Points to be produced in the output. The possible options of the parameter mode and the corresponding
6460 behaviours are the following:

6461 **non_null** the result Data Point is produced when all the items involved in the comparison exist and have
6462 not NULL Measure value (i.e., when no Data Point corresponding to the Code Items of the left
6463 and right sides of the rule is missing or has NULL Measure value); under this option, in
6464 evaluating the comparison, the possible missing Data Points corresponding to the Code Items
6465 of the left and right sides of the rule are considered existing and having a NULL Measure value;

6466 **non_zero** the result Data Point is produced when at least one of the items involved in the comparison
6467 exist and have Measure not equal to 0 (zero); the possible missing Data Points corresponding
6468 to the Code Items of the left and right sides of the rule are considered existing and having a
6469 Measure value equal to 0;

6470 **partial_null** the result Data Point is produced if at least one Data Point corresponding to the Code Items of
6471 the left and right sides of the rule is found (whichever is its Measure value); the possible

6472 missing Data Points corresponding to the Code Items of the left and right sides of the rule are
6473 considered existing and having a NULL Measure value;
6474 **partial_zero** the result Data Point is produced if at least one Data Point corresponding to the Code Items of
6475 the left and right sides of the rule is found (whichever is its Measure value); the possible
6476 missing Data Points corresponding to the Code Items of the left and right sides of the rule are
6477 considered existing and having a Measure value equal to 0 (zero);
6478 **always_null** the result Data Point is produced in any case; the possible missing Data Points corresponding
6479 to the Code Items of the left and right sides of the rule are considered existing and having a
6480 Measure value equal to NULL;
6481 **always_zero** the result Data Point is produced in any case; the possible missing Data Points corresponding
6482 to the Code Items of the left and right sides of the rule are considered existing and having a
6483 Measure value equal to 0 (zero);

6484 If the parameter **mode** is not specified, then it is assumed to be **non_null**.

6485 The following table summarizes the behaviour of the options of the parameter “mode”

6486

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)

6487

6488 Finally the parameter **output** is considered, to determine the structure and content of the resulting Data Set. The
6489 possible options of the parameter **output** and the corresponding behaviours are the following:

6490 **all** all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and
6491 the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is
6492 returned in the *boolean* Measure **bool_var**. The original Measure Component of the Data Set **op**
6493 is not returned.

6494 **invalid** only the invalid (FALSE) Data Points produced by the comparison are returned. The result of
6495 the comparison (*boolean* Measure **bool_var**) is not returned. The original Measure Component
6496 of the Data Set **op** is returned and contains the Measure values taken from the Data Points on
6497 the left side of the rule.

6498 **all_measures** all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and
6499 the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is
6500 returned in the *boolean* Measure **bool_var**. The original Measure Component of the Data Set **op**
6501 is returned and contains the Measure values taken from the Data Points on the left side of the
6502 rule.

6503 If the parameter output is not specified then it is assumed to be invalid.

6504 In conclusion, the operator returns a Data Set having the following Components:

- 6505 • all the Identifier Components of op
- 6506 • the additional Identifier Component ruleid, whose aim is to identify the Rule that has generated the
- 6507 actual Data Point (it contains at least the Rule name specified in hr⁹)
- 6508 • if the output parameter is all: the *boolean* Measure bool_var whose values are the result of the
- 6509 evaluation of the Rules (TRUE, FALSE or NULL).
- 6510 • if the output parameter is invalid: the original Measure of op, whose values are taken from the Measure
- 6511 values of the Data Points of the left side of the Rule
- 6512 • if the output parameter is all_measures: the *boolean* Measure bool_var, whose value is the result of the
- 6513 evaluation of a Rule on a Data Point (TRUE, FALSE or NULL), and the original Measure of op, whose
- 6514 values are taken from the Measure values of the Data Points of the left side of the Rule
- 6515 • the Measure imbalance, which contains the difference between the Measure values of the Data Points on
- 6516 the left side of the Rule and the Measure values of the corresponding calculated Data Points on the right
- 6517 side of the Rule
- 6518 • the Measure errorcode, which contains the errorcode value specified in the Rule
- 6519 • the Measure errorlevel, which contains the errorlevel value specified in the Rule

6520
6521 Note that a generic Data Point of op can produce several Data Points in the resulting Data Set, one for each Rule
6522 in which the Data Point appears as the left member of the comparison.

6523
6524

6525 *Examples*

6526 See also the examples in **define hierarchical ruleset**.

6527

6528 Given the following hierarchical ruleset:

6529

```
6530     define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
6531           R010 :      A = J + K + L                errorlevel 5
6532           ; R020 :      B = M + N + O                errorlevel 5
6533           ; R030 :      C = P + Q      errorcode XX  errorlevel 5
6534           ; R040 :      D = R + S                    errorlevel 1
6535           ; R060 :      F = Y + W + Z                errorlevel 7
6536           ; R070 :      G = B + C
6537           ; R080 :      H = D + E                    errorlevel 0
6538           ; R090 :      I = D + G      errorcode YY  errorlevel 0
6539           ; R100 :      M >= N                      errorlevel 5
6540           ; R110 :      M <= G                      errorlevel 5
```

6541 end hierarchical ruleset

6542

6543 And given the operand Data Set DS_1 (where At_1 is viral and the propagation rule says that the alphabetic
6544 order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is
6545 assumed as NULL):

6546

DS_1		
Id_1	Id_2	Me_1
2010	A	5
2010	B	11
2010	C	0
2010	G	19
2010	H	NULL

⁹ The content of ruleid maybe personalised in the implementation

2010	I	14
2010	M	2
2010	N	5
2010	O	4
2010	P	7
2010	Q	-7
2010	S	3
2010	T	9
2010	U	NULL
2010	V	6

6547
6548
6549

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

DS_r						
Id_1	Id_2	ruleid	Bool_var	imbalance	errorcode	errorlevel
2010	A	R010	NULL	NULL	NULL	5
2010	B	R020	TRUE	0	NULL	5
2010	C	R030	TRUE	0	XX	5
2010	D	R040	NULL	NULL	NULL	1
2010	E	R050	NULL	NULL	NULL	0
2010	F	R060	NULL	NULL	NULL	7
2010	G	R070	FALSE	8	NULL	NULL
2010	H	R080	NULL	NULL	NULL	0
2010	I	R090	NULL	NULL	YY	0
2010	M	R100	FALSE	-3	NULL	5
2010	M	R110	TRUE	-17	NULL	5

6550
6551

6552 check

6553 Syntax

6554 **check** (op { **errorcode** errorcode } { **errorlevel** errorlevel } { **imbalance** imbalance } { output })

6555 output ::= **invalid** | **all**

6556 Input parameters

6557 **op** a *boolean* Data Set (a *boolean* condition expressed on one or more Data Sets)

6558 **errorcode** the error code to be produced when the condition evaluates to FALSE. It must be a valid value
6559 of the `errorcode_vd` Value Domain (or *string* if the `errorcode_vd` Value Domain is not found).
6560 It can be a Data Set or a *scalar*. If not specified then `errorcode` is NULL.

6561 **errorlevel** the error level to be produced when the condition evaluates to FALSE. It must be a valid value
6562 of the `errorlevel_vd` Value Domain (or *integer* if the `errorlevel_vd` Value Domain is not found).
6563 It can be a Data Set or a *scalar*. If not specified then `errorlevel` is NULL.

6564 imbalance the imbalance to be computed. imbalance is a *numeric* mono-measure Data Set with the same
6565 Identifiers of op. If not specified then imbalance is NULL.

6566 output specifies which Data Points are returned in the resulting Data Set:
6567 **invalid** returns the Data Points of op for which the condition evaluates to
6568 FALSE
6569 **all** returns all Data Points of op
6570 If not specified then output is **all**.

6571 *Examples of valid syntaxes*

6572 check (DS1 > DS2 errorcode myerrorcode errorlevel myerrorlevel imbalance DS1 - DS2 invalid)

6573 *Input parameters type:*

6574 op :: dataset
6575 errorcode :: errorcode_vd
6576 errorlevel :: errorlevel_vd
6577 imbalance :: number

6578 *Result type:*

6579 result :: dataset

6580 *Additional constraints*

6581 op has exactly a *boolean* Measure Component.

6582 *Behaviour*

6583 It returns a Data Set having the following components:

- 6584 • the Identifier Components of op
- 6585 • a *boolean* Measure named **bool_var** that contains the result of the evaluation of the *boolean* dataset op
- 6586 • the Measure imbalance that contains the specified imbalance
- 6587 • the Measure errorcode that contains the specified errorcode
- 6588 • the Measure errorlevel that contains the specified errorlevel

6589 If output is **all** then all data points are returned. If output is **invalid** then only the Data Points where bool_var is
6590 FALSE are returned.

6591 *Examples*

6592 Given the Data Sets DS_1 and DS_2 :

6595

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

6596

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

6597

6598

6599

Example 1: DS_r := check (DS1 >= DS2 imbalance DS1 - DS2) returns:

DS_r					
Id_1	Id_2	bool_var	imbalance	errorcode	errorlevel
2010	I	FALSE	-8	NULL	NULL
2011	I	TRUE	0	NULL	NULL
2012	I	TRUE	0	NULL	NULL
2013	I	FALSE	-3	NULL	NULL
2014	I	TRUE	0	NULL	NULL
2015	I	TRUE	0	NULL	NULL
2010	D	FALSE	-25	NULL	NULL
2011	D	TRUE	0	NULL	NULL
2012	D	TRUE	5	NULL	NULL
2013	D	TRUE	0	NULL	NULL
2014	D	FALSE	-15	NULL	NULL
2015	D	TRUE	0	NULL	NULL

6600

6602 **if-then-else** : **if**

6603

6604 *Syntax*6605 **if** condition **then** thenOperand **else** elseOperand

6606

6607 *Input parameters*

6608

6609 condition a Boolean condition (dataset, component or scalar)

6610 thenOperand the operand returned when condition evaluates to **true**6611 elseOperand the operand returned when condition evaluates to **false**

6612

6613 *Examples of valid syntaxes*

6614 if A > B then A else B

6615

6616 *Semantics for scalar operations*6617 The **if** operator returns thenOperand if condition evaluates to **true**, elseOperand otherwise. For example, considering the statement:

6618 if x1 > x2 then 2 else 5,

6619 for x1 = 3, x2 = 0 it returns 2

6620 for x1 = 0, x2 = 3 it returns 5

6621

6622 *Input Parameters type*

6623 condition :: dataset { measure <boolean> _ }

6624 | component<Boolean>

6625 | boolean

6626 thenOperand :: dataset

6627 | component

6628 | scalar

6629 elseOperand :: dataset

6630 | component

6631 | scalar

6632

6633 *Result type*

6634 result :: dataset

6635 | component<

6636 | scalar

6637

6638 *Additional constraints*

6639

- 6640 • The operands thenOperand and elseOperand must be of the same scalar type.
- 6641 • If the operation is at scalar level, thenOperand and elseOperand are scalar then condition must be scalar too (a *boolean* scalar).
- 6642 • 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.
- 6643 • 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
 - 6644 ○ If thenOperand and elseOperand are both Data Sets then they must have the same Components in the same roles
 - 6645 ○ 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.

6646

6647

6648

6649

6650

6651

6652

6653

6656 *Behaviour*

6657 For operations at Component level, the operation is applied for each Data Point of the unique input Data Set, the
6658 **if-then-else** operator returns the value from the thenOperand Component when condition evaluates to **true**,
6659 otherwise it returns the value from the elseOperand Component. If one of the operands thenOperand or
6660 elseOperand is scalar, such a scalar value can be returned depending on the outcome of the condition.

6661 For operations at Data Set level, the **if-then-else** operator returns the Data Point from thenOperand when the
6662 Data Point of condition having the same Identifiers' values evaluates to **true**, and returns the Data Point from
6663 elseOperand otherwise. If one of the operands thenOperand or elseOperand is scalar, such a scalar value can
6664 be returned (depending on the outcome of the condition) and in this case it feeds the values of all the Measures
6665 of the result Data Point.

6666 The behaviour for two Data Sets can be procedurally explained as follows. First the condition Data Set is
6667 evaluated, then its true Data Points are inner joined with thenOperand and its false Data Points are inner
6668 joined with elseOperand, finally the union is made of these two partial results (the condition ensures that there
6669 cannot be conflicts in the union).

6670
6671 *Examples*

6672 *Example 1:* given the operand Data Sets DS_cond, DS_1, DS_2:
6673
6674

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

6675

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

6676

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

6677

6678 DS_r := if (DS_cond#Id_4 = "F") then DS_1 else DS_2 returns:
 6679

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

6680 **Nvl :** **nvl**

6681 *Syntax*

6682 **nvl (op1 , op2)**

6683

6684 *Input parameters*

6685 op1 the first operand

6686 op2 the second operand

6687

6688 *Examples of valid syntaxes*

6689 nvl (ds1#m1, 0)

6690

6691 *Semantics for scalar operations*

6692 The operator nvl returns op2 when op1 is **null**, otherwise op1. For example:

6693 nvl (5, 0) returns 5

6694 nvl (null, 0) returns 0

6695

6696 *Input Parameters type*

6697 op1 :: dataset

6698 | component<scalar>

6699 | scalar

6700

6701 op2 :: dataset

6702 | component

6703 | <scalar>

6704

6705 *Result type*

6706 result :: dataset

6707 | component

6708 | scalar

6709

6710 *Additional constraints*

6711 If op1 and op2 are scalar values then they must be of the same type.

6712 If op1 and op2 are Components then they must be of the same type.

6713 If op1 and op2 are Data Sets then they must have the same Components.

6714

6715 *Behaviour*

6716 The operator nvl returns the value from op2 when the value from op1 is null, otherwise it returns the value from op1.

6717 The operator has the typical behaviour of the operators applicable on two scalar values or Data Sets or Data Set Components.

6718 Also the following statement gives the same result: if isnull (op1) then op2 else op1

6719

6720 *Examples*

6721

6722 *Example 1:* Given the input Data Set DS_1

6723

6724

6725

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

DS_r := nvl (DS_1, 0) returns:

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	0
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	0

6726
6727
6728

6730 **Filtering Data Points :** **filter**

6731

6732 *Syntax*6733 `op [filter filterCondition]`

6734

6735 *Input parameters*6736 `op` the operand6737 `filterCondition` the filter condition

6738

6739 *Examples of valid syntaxes*6740 `DS_1 [filter Me_3 > 0]`6741 `DS_1 [filter Me_3 + Me_2 <= 0]`

6742

6743 *Semantics for scalar operations*

6744 This operator cannot be applied to scalar values.

6745

6746 *Input parameters type:*6747 `op ::` dataset6748 `filterCondition ::` component<boolean>

6749

6750 *Result type:*6751 `result ::` dataset

6752

6753 *Additional constraints:*

6754 None.

6755

6756 *Behaviour*

6757 The operator takes as input a Data Set (`op`) and a *boolean* Component expression (`filterCondition`) and filters the
 6758 input Data Points according to the evaluation of the condition. When the expression is TRUE the Data Point is
 6759 kept in the result, otherwise it is not kept (in other words, it filters out the Data Points of the operand Data Set
 6760 for which `filterCondition` condition evaluates to FALSE or NULL).

6761

6762 *Examples*

6763

6764 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

6765

6766 *Example1:* `DS_r := DS_1 [filter Id_1 = 1 and Me_1 < 10]` results in:

6767

DS_r				
Id_1	Id_2	Id_3	Me_1	At_1
1	A	XX	2	E
1	A	YY	2	F

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

6768 Calculation of a Component : **calc**

6769

6770 *Syntax*

6771 op [**calc** { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }*]

6772

6773 calcRole ::= **identifier** | **measure** | **attribute** | **viral attribute**

6774

6775 *Input parameters*

6776 op the operand

6777 calcRole the role to be assigned to a Component to be calculated

6778 calcComp the name of a Component to be calculated

6779 calcExpr expression at component level, having only Components of the input Data Sets as operands,

6780 used to calculate a Component

6781

6782 *Examples of valid syntaxes*

6783 DS_1 [calc Me_3 := Me_1 + Me_2]

6784

6785 *Semantics for scalar operations*

6786 This operator cannot be applied to scalar values.

6787

6788 *Input parameters type:*

6789 op :: dataset

6790 calcComp :: name < component >

6791 calcExpr :: component<scalar>

6792

6793 *Result type:*

6794 result :: dataset

6795

6796 *Additional constraints*

6797 The calcComp parameter cannot be the name of an Identifier component.

6798 All the components used in calcComp must belong to the operand Data Set op.

6799

6800 *Behaviour*

6801 The operator calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at
6802 Component level. Each Component is calculated through an independent sub-expression. It is possible to specify
6803 the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**, therefore the calc
6804 clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling
6805 the virality of the calculated Attributes (for the attribute propagation rule see the User Manual). When the role is
6806 omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if
6807 the component does not exist in the operand Data Set then its role is Measure.

6808 The calcExpr sub-expressions are independent one another, they can only reference Components of the input
6809 Data Set and cannot use Components generated, for example, by other calcExpr. If the calculated Component is a
6810 new Component, it is added to the output Data Set. If the Calculated component is a Measure or an Attribute that
6811 already exists in the input Data Set, the calculated values overwrite the original values. If the calculated
6812 Component is an Identifier that already exists in the input Data Set, an exception is raised because overwriting
6813 an Identifier Component is forbidden for preserving the functional behaviour. Analytic invocations can be used
6814 in the **calc** clause.

6815

6816

6817 *Examples*

6818

6819

6820 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

6821
6822 *Example1:* DS_r := DS_1 [calc Me_1:= Me_1 * 2] results in:
6823

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

6824
6825 *Example2:* DS_r := DS_1 [calc attribute At_1:= "EP"] results in:
6826

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

6827

6828 Aggregation : **aggr**

6829

6830 *Syntax*

6831 op [**aggr** aggrClause { groupingClause }]

6832

6833 aggrClause ::= { aggrRole } aggrComp := aggrExpr
6834 { , { aggrRole } aggrComp := aggrExpr }*

6835

6836 groupingClause ::= { **group by** groupingId {, groupingId }*
6837 | **group except** groupingId {, groupingId }*
6838 | **group all** conversionExpr }¹
6839 { **having** havingCondition }

6840

6841 aggrRole ::= **measure** | **attribute** | **viral attribute**

6842

6843

6844 *Input Parameters*

6845 op the operand
6846 aggrClause clause that specifies the required aggregations, i.e., the aggregated Components to be
6847 calculated, their roles and their calculation algorithm, to be applied on the joined and
6848 filtered Data Points
6849 aggrRole the role of the aggregated Component to be calculated
6850 aggrComp the name of the aggregated Component to be calculated; this is a dependent Component
6851 of the result (Measure or Attribute, not Identifier)

6852 **aggrExpr** expression at component level, having only Components of the input Data Sets as
6853 operands, which invokes an aggregate operator (e.g. **avg, count, max ...**, see also the
6854 corresponding sections) to perform the desired aggregation. Note that the **count**
6855 operator is used in an **aggrClause** without parameters, e.g.:

6856 `DS_1 [aggr Me_1 := count () group by Id_1]`

6857 **groupingClause** the following alternative grouping options:

6858 **group by** the Data Points are grouped by the values of the specified Identifiers
6859 (**groupingId**). The Identifiers not specified are dropped in the result.

6860 **group except** the Data Points are grouped by the values of the Identifiers not
6861 specified as **groupingId**. The Identifiers specified as **groupingId** are
6862 dropped in the result.

6863 **group all** converts the values of an Identifier Component using **conversionExpr**
6864 and keeps all the resulting Identifiers.

6865 **groupingId** Identifier Component to be kept (in the **group by** clause) or dropped (in the **group**
6866 **except** clause).

6867 **conversionExpr** specifies a conversion operator (e.g., **time_agg**) to convert an Identifier from finer to
6868 coarser granularity. The conversion operator is applied on an Identifier of the operand
6869 Data Set **op**.

6870 **havingCondition** a condition (boolean expression) at component level, having only Components of the
6871 input Data Sets as operands (and possibly constants), to be fulfilled by the groups of
6872 Data Points: only groups for which **havingCondition** evaluates to **TRUE** appear in the
6873 result. The **havingCondition** refers to the groups specified through the **groupingClause**,
6874 therefore it must invoke aggregate operators (e.g. **avg, count, max ...**, see also the
6875 section **Aggregate invocation**). A correct example of **havingCondition** is:

6876 `max(obs_value) < 1000`

6877 instead the condition `obs_value < 1000` is not a right **havingCondition**, because it
6878 refers to the values of the single Data Points and not to the groups. The **count** operator
6879 is used in a **havingCondition** without parameters, e.g.:

6880 `sum (DS_1 group by id1 having count () >= 10)`

6881
6882 *Examples of valid syntaxes*

6883 `DS_1 [aggr M1 := min (Me_1) group by Id_1, Id_2]`

6884 `DS_1 [aggr M1 := min (Me_1) group except Id_1, Id_2]`

6885

6886 *Semantics for scalar operations*

6887 This operator cannot be applied to scalar values.

6888

6889 *Input parameters type:*

6890 **op** :: dataset

6891 **aggrComp** :: name < component >

6892 **aggrExpr** :: component<scalar>

6893 **groupingId** :: name <identifier >

6894 **conversionExpr** :: identifier<scalar>

6895 **havingCondition** :: component<boolean>

6896

6897 *Result type:*

6898 **result** :: dataset

6899

6900 *Additional constraints*

6901 The **aggrComp** parameter cannot be the name of an Identifier component.

6902 All the components used in **aggrExpr** must belong to the operand Data Set **op**.

6903 The **conversionExpr** parameter applies just one conversion operator to just one Identifier belonging to the input
6904 Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion
6905 operator.

6906

6907 *Behaviour*

6908 The operator **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of
6909 sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is
6910 possible to specify the role of the calculated Component among **measure attribute**, or **viral attribute**. The
6911 substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the
6912 User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data
6913 Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.

6914 The **aggrExpr** sub-expressions are independent of one another, they can only reference Components of the input
6915 Data Set and cannot use Components generated, for example, by other **aggrExpr** sub-expressions. The **aggr**
6916 computed Measures and Attributes are the only Measures and Attributes returned in the output Data Set (plus
6917 the possible viral Attributes). The sub-expressions must contain only Aggregate operators, which are able to
6918 compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are
6919 specified through the groupingClause, which allows the following alternative options.

6920 **group by** the Data Points are grouped by the values of the specified Identifiers. The Identifiers not
6921 specified are dropped in the result.

6922 **group except** the Data Points are grouped by the values of the Identifiers not specified in the clause. The
6923 specified Identifiers are dropped in the result.

6924 **group all** converts an Identifier Component using conversionExpr and keeps all the other Identifiers.

6925
6926 The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the
6927 single groups (for example the minimum number of Data Points in the group).

6928 If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause
6929 returns a Data Set that contains a single Data Point and has no Identifiers.

6930 The Attributes calculated through the **aggr** clauses are maintained in the result. For all the other Attributes that
6931 are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation
6932 Rule section in the User Manual).

6933
6934 *Examples*

6935
6936 Given the Data Set DS_1:
6937

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

6938
6939 *Example1:* DS_r := DS_1 [aggr Me_1:= sum(Me_1) group by Id_1 , Id_2] results in:
6940

DS_r		
Id_1	Id_2	Me_1
1	A	2
1	B	8
2	A	9

6941
6942 *Example2:* DS_r := DS_1 [aggr Me_3:= min(Me_1) group except Id_3] results in:
6943

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

6944
6945 *Example3:* DS_r := DS_1 [aggr Me_1:= sum(Me_1), Me_2 := max(Me_1)
6946 group by Id_1 , Id_2
6947 having mean (Me_1) > 2]
6948
6949

results in:

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

6950

6951 **Maintaining Components: keep**

6952
6953 *Syntax*
6954 op [**keep** comp {, comp }*]
6955

6956 *Input parameters*
6957 op the operand
6958 comp a component to keep
6959

6960 *Examples of valid syntaxes*
6961 DS_1 [keep Me_2, Me_3]
6962

6963 *Semantics for scalar operations*
6964 This operator cannot be applied to scalar values.
6965

6966 *Input parameters type:*
6967 op :: dataset
6968 comp :: name < component >
6969

6970 *Result type:*
6971 result :: dataset
6972

6973 *Additional constraints:*
6974 All the Components comp must belong to the input Data Set op.
6975 The Components comp cannot be Identifiers in op.
6976

6977 *Behaviour*
6978 The operator takes as input a Data Set (op) and some Component names of such a Data Set (comp). These
6979 Components can be Measures or Attributes of op but not Identifiers. The operator maintains the specified
6980 Components, drops all the other dependent Components of the Data Set (Measures and Attributes) and
6981 maintains the independent Components (Identifiers) unchanged. This operation corresponds to a projection in
6982 the usual relational join semantics (specifying which columns will be projected in among Measures and
6983 Attributes).
6984
6985

6986 *Examples*

6987

6988 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

6989

6990 *Example1:* DS_r := DS_1 [keep Me_1] results in:

6991

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

6992

6993 Removal of Components: **drop**

6994

6995 *Syntax*

6996 op [**drop** comp { , comp }*]

6997

6998 *Input parameters*

6999 op the operand

7000 comp a Component to drop

7001

7002 *Examples of valid syntaxes*

7003 DS_1 [drop Me_2, Me_3]

7004

7005 *Semantics for scalar operations*

7006 This operator cannot be applied to scalar values.

7007

7008 *Input parameters type:*

7009 op :: dataset

7010 comp :: name < component >

7011

7012 *Result type:*

7013 result :: dataset

7014

7015 *Additional constraints:*

7016 All the Components comp must belong to the input Data Set op.

7017 The Components comp cannot be Identifiers in op.

7018

7019 *Behaviour*

7020 The operator takes as input a Data Set (op) and some Component names of such a Data Set (comp). These

7021 Components can be Measures or Attributes of op but not Identifiers. The operator drops the specified

7022 Components and maintains all the other Components of the Data Set. This operation corresponds to a projection

7023 in the usual relational join semantics (specifying which columns will be projected out).

7024

7025 *Examples*

7026

7027 Given the Data Set DS_1:
7028

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

7029 *Example1:* DS_r := DS_1 [drop At_1] results in:
7030
7031

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

7032 Change of Component name : **rename**

7033 *Syntax*
7034 op [**rename** comp_from **to** comp_to { , comp_from **to** comp_to}*]
7035

7036 *Input Parameters*
7037 op the operand
7038 comp_from the original name of the Component to rename
7039 comp_to the new name of the Component after the renaming
7040

7041 *Examples of valid syntaxes*
7042 DS_1 [rename Me_2 to Me_3]
7043

7044 *Semantics for scalar operations*
7045 This operator cannot be applied to scalar values.
7046

7047 *Input Parameters type*
7048 op :: dataset
7049 comp_from :: name < component >
7050 comp_to :: name < component >
7051

7052 *Result type*
7053 result :: dataset
7054

7055 *Additional constraints*
7056 The corresponding pairs of Components before and after the renaming (dsc_from and dsc_to) must be defined
7057 on the same Value Domain and the same Value Domain Subset.
7058 The components used in dsc_from must belong to the input Data Set and the component used in the dsc_to
7059 cannot have the same names as other Components of the result Data Set.
7060

7061 *Behaviour*
7062 The operator assigns new names to one or more Components (Identifier, Measure or Attribute Components).
7063 The resulting Data Set, after renaming the specified Components, must have unique names of all its Components
7064 (otherwise a runtime error is raised). Only the Component name is changed and not the Component Values,
7065 therefore the new Component must be defined on the same Value Domain and Value Domain Subset as the
7066 original Component (see also the IM in the User Manual). If the name of a Component defined on a different

7067 Value Domain or Set is assigned, an error is raised. In other words, **rename** is a transformation of the variable
 7068 without any change in its values.

7069
 7070

7071 *Examples*

7072

7073 Given the Data Set DS_1:

7074

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

7075

7076 *Example1:* DS_r := DS_1 [rename Me_1 to Me_2, At_1 to At_2] results in:

7077

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

7078 **Pivoting : pivot**

7079

7080 *Syntax*

7081 op [**pivot** identifier , measure]

7082

7083 *Input parameters*

7084 op the operand
 7085 identifier the Identifier Component of op to pivot
 7086 measure the Measure Component of op to pivot

7087

7088

7089 *Examples of valid syntaxes*

7090 DS_1 [pivot Id_2, Me_1]

7091

7092 *Semantics for scalar operations*

7093 This operator cannot be applied to scalar values.

7094

7095 *Input Parameters type*

7096 op :: dataset
 7097 identifier :: name < identifier >
 7098 measure :: name < measure >

7099

7100 *Result type*

7101 result :: dataset

7102

7103 *Additional constraints*
 7104 The Measures created by the operator according to the behaviour described below must be defined on the same
 7105 Value Domain as the input Measure.
 7106

7107 *Behaviour*
 7108 The operator transposes several Data Points of the operand Data Set into a single Data Point of the resulting Data
 7109 Set. The semantics of **pivot** can be procedurally described as follows.
 7110

- 7111 1. It creates a virtual Data Set VDS as a copy of **op**
- 7112 2. It drops the Identifier Component **identifier** and all the Measure Components from VDS.
- 7113 3. It groups VDS by the values of the remaining Identifiers.
- 7114 4. For each distinct value of **identifier** in **op**, it adds a corresponding measure to VDS, named as the value of
 7115 **identifier**. These Measures are initialized with the NULL value.
- 7116 5. For each Data Point of **op**, it finds the Data Point of VDS having the same values as for the common
 7117 Identifiers and assigns the value of **measure** (taken from the current Data Point of **op**) to the Measure of
 7118 VDS having the same name as the value of **identifier** (taken from the Data Point of **op**).
 7119

7120 The result of the last step is the output of the operation.
 7121

7122 Note that **pivot** may create Measures whose names are non-regular (i.e. they may contain special characters,
 7123 reserved keywords, etc.) according to the rules about the artefact names described in the User Manual (see the
 7124 section “The artefact names” in the chapter “VTL Transformations”). As said in the User Manual, those names
 7125 must be quoted to be referenced within an expression.
 7126

7127 *Examples*

7128 Given the Data Set DS_1:
 7129
 7130

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

7131 *Example1:* DS_r := Ds_1 [pivot Id_2, Me_1] results in:
 7132
 7133

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

7134

7135 **Unpivoting : unpivot**

7136
 7137 *Syntax*

7138 op [**unpivot** identifier , measure]
 7139

7140 *Input parameters*
 7141 **op** the dataset operand
 7142 **identifier** the Identifier Component to be created
 7143 **measure** the Measure Component to be created

7144
 7145 *Examples of valid syntaxes*
 7146 DS [unpivot Id_5, Me_3]

7147
 7148 *Semantics for scalar operations*
 7149 This operator cannot be applied to *scalar* values.

7150
 7151 *Input Parameters type*
 7152 **op ::** dataset
 7153 **identifier ::** name < identifier >
 7154 **measure ::** name < measure >

7155
 7156 *Result type*
 7157 **result ::** dataset

7158
 7159 *Additional constraints*
 7160 All the measures of **op** must be defined on the same Value Domain.

7161
 7162 *Behaviour*
 7163 The **unpivot** operator transposes a single Data Point of the operand Data Set into several Data Points of the result Data set. Its semantics can be procedurally described as follows.

- 7165 1. It creates a virtual Data Set VDS as a copy of **op**
- 7166 2. It adds adds the Identifier Component **identifier** and the Measure Component **measure** to VDS.
- 7167 3. For each Data Point DP and for each Measure M of **op** whose value is not NULL, the operator inserts a Data Point into VDS whose values are assigned as specified in the following points
- 7168 4. The VDS Identifiers other than **identifier** are assigned the same values as the corresponding Identifiers of the **op** Data Point
- 7169 5. The VDS **identifier** is assigned a value equal to the **name** of the Measure M of **op**
- 7170 6. The VDS **measure** is assigned a value equal to the **value** of the Measure M of **op**

7171
 7172 The result of the last step is the output of the operation.

7173
 7174 When a Measure is NULL then **unpivot** does not create a Data Point for that Measure.
 7175 Note that in general pivoting and unpivoting are not exactly symmetric operations, i.e., in some cases the unpivot operation applied to the pivoted Data Set does not recreate exactly the original Data Set (before pivoting).

7176
 7177
 7178
 7179
 7180 *Examples*

7181
 7182 Given the Data Set DS_1:

DS_1			
Id_1	A	B	C
1	5	2	7
2	3	4	9

7183
 7184
 7185
 7186 *Example1:* DS_r := DS_1 [unpivot Id_2, Me_1] results in:

DS_r		
Id_1	Id_2	Me_1

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

7189

7190 **Subspace :** **sub**

7191

7192 *Syntax*

7193 op [**sub** identifier = value { , identifier = value }*]

7194

7195 *Input parameters*

7196 op dataset

7197 identifier Identifier Component of the input Data Set op

7198 value valid value for identifier

7199

7200 *Examples of valid syntaxes*

7201 DS_r := DS_1 [Id_2 = "A", Id_5 = 1]

7202

7203 *Semantics for scalar operations*

7204 This operator cannot be applied to scalar values.

7205

7206 *Input Parameters type*

7207 op :: dataset

7208 identifier :: name < identifier >

7209 value :: scalar

7210

7211 *Result type*

7212 result :: dataset

7213

7214 *Additional constraints*

7215 The specified Identifier Components identifier(s) must belong to the input Data Set op.

7216 Each Identifier Component can be specified only once.

7217 The specified value must be an allowed value for identifier.

7218

7219

7220 *Behaviour*

7221

7222 The operator returns a Data Set in a subspace of the one of the input Dataset. Its behaviour can be procedurally described as follows:

7223

7224

- 7225 1. It creates a virtual Data Set VDS as a copy of op
- 7226 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)
- 7227 3. It projects out ("drops", in VTL terms) all the identifier(s)

7228

7229 The result of the last step is the output of the operation.

7230

7231 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.

7232

7233 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

7234

7237 Data Set is a 1st order function and therefore does not contain duplicates, the result Data Set is a 1st order
 7238 function as well. To show this, let $K_1, \dots, K_m, \dots, K_n$ be the Identifier components for the generic input Data Set DS.
 7239 Let us suppose that K_1, \dots, K_m are assigned to fixed values by using the subspace operator. A duplicate could arise
 7240 only if in the result there are two Data Points DP_{r1} and DP_{r2} having the same value for K_{m+1}, \dots, K_n , but this is
 7241 impossible since such Data Points had same K_1, \dots, K_m in the original Data Set DS, which did not contain
 7242 duplicates.

7243
 7244 If we consider the vector space of Data Points individuated by the n-uples of Identifier components of a Data Set
 7245 DS(K_1, \dots, K_n, \dots) (along, e.g., with the operators of sum and multiplication), we have that the subspace operator
 7246 actually performs a subsetting of such space into another space with fewer Identifiers. This can be also seen as
 7247 the equivalent of a *dice* operation performed on hyper-cubes in multi-dimensional data warehousing.
 7248

7249
 7250 *Examples*

7251
 7252 Given the Data Set DS_1:
 7253

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

7254
 7255 *Example 1:* DS_r := DS_1 [sub Id_1 = 1, Id_2 = "A"] results in:
 7256

DS_r		
Id_3	Me_1	At_1
XX	20	F
YY	1	F

7257
 7258 *Example 2:* DS_r := DS_1 [sub Id_1 = 1, Id_2 = "B", Id_3 = "YY"] results in:
 7259

DS_r	
Me_1	At_1
9	F

7260
 7261 *Example 3:* DS_r := DS_1 [sub Id_2 = "A"] + DS_1 [sub Id_2 = "B"] results in:

7262
 7263
 7264 Assuming that At_1 is viral and that in the propagation rule the greater value prevails, results in:
 7265

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

7266
7267
7268