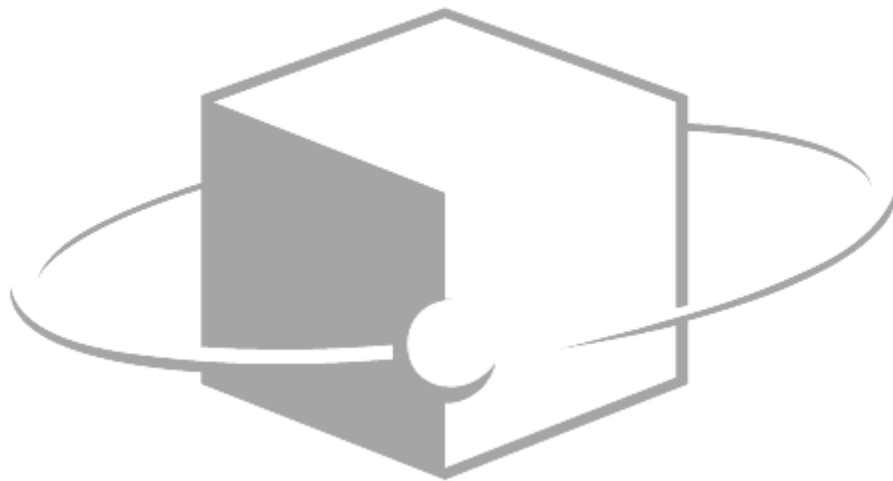


---

Reference

# instantOLAP

Version 2.6.1  
17.11.2009





# Copyright

---

*Copyright (C) 2002-2008 Thomas Behrends Softwareentwicklung e.K. All rights reserved.*

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Thomas Behrends. Thomas Behrends assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Thomas Behrends.

instantOLAP is a registered trademark of Thomas Behrends. Windows is a registered trademark of Microsoft Corporation. All other product and company names are trademarks or registered trademarks of their respective holders.

# Contents

---

- Copyright ..... 3
- Contents ..... 4
- Typographical Conventions ..... 20
  
- CHAPTER 1: Query properties ..... 21**
- Multiple Query ..... 22
  - Align Blocks ..... 22
  - Author ..... 22
  - Background ..... 23
  - CSS ..... 23
  - Date ..... 24
  - Description ..... 24
  - Enable CSV Export ..... 25
  - Enable PDF Export ..... 25
  - Enable XLS Export ..... 25
  - External Transformer ..... 26
  - External Transformer Name ..... 26
  - Filter ..... 27
  - Icon ..... 27
  - Logo ..... 28
  - Model ..... 28
  - Name ..... 29
  - Print Height ..... 30
  - Print Logo ..... 30
  - Print Width ..... 31
  - Refresh Time ..... 31
  - Title ..... 32
  - Visible ..... 32
- Selector ..... 34
  - Default ..... 34
  - Description ..... 34
  - Height ..... 35
  - Match ..... 35
  - Name ..... 36
  - Options ..... 36
  - Submit ..... 37
  - Text ..... 37
  - Title ..... 38
  - Type ..... 38
  - Use Filter ..... 40
  - Visible ..... 40
  - Width ..... 40
- Selector Group ..... 42
  - Background ..... 42
  - Border ..... 42
  - Height ..... 43
  - Popup ..... 43
  - Title ..... 44
  - Visible ..... 44
  - Width ..... 44
- Outer block ..... 46

Export .....	46
Filter .....	46
Iteration .....	47
Keep Together .....	47
Orientation .....	48
Pagebreak .....	49
Visible .....	49
Inner block .....	50
Automation Delay .....	50
Background .....	50
Border .....	51
Color .....	51
Export .....	52
Filter .....	52
Font .....	53
Font Size .....	53
Font Weight .....	54
Format .....	54
Height .....	56
Iteration .....	57
Keep Together .....	57
Link .....	58
Link Icon .....	58
Link Keys .....	59
Link Name .....	60
Link Target .....	60
Padding .....	61
Position X .....	61
Position Y .....	62
Orientation .....	62
Title .....	63
Title Align .....	63
Visible .....	64
Width .....	65
Query .....	66
Corner Align .....	66
Corner Background .....	66
Corner Bottom Color .....	67
Corner Font .....	67
Corner Font Size .....	68
Corner Font Weight .....	68
Corner Foreground .....	69
Corner Left Color .....	69
Corner Right Color .....	70
Corner Text .....	70
Corner Top Color .....	71
Corner VAlign .....	71
Filter .....	72
Fixed Columns .....	73
Fixed Rows .....	74
Highlight Color .....	74
Span Body .....	75
Span Headers .....	75
Subcube .....	76
Suppress Cols .....	77
Suppress Rows .....	78
Header .....	79
Align .....	79
Assertion .....	79
Background .....	80
Bottom Color .....	81
Bottom Padding .....	81

Cell Align .....	82
Cell Background .....	83
Cell Bottom Color .....	83
Cell Bottom Padding .....	84
Cell Color .....	84
Cell Font .....	85
Cell Font Size .....	85
Cell Font Weight .....	85
Cell Left Color .....	86
Cell Left Padding .....	87
Cell Link .....	87
Cell Link Icon .....	88
Cell Link Keys .....	88
Cell Link Name .....	89
Cell Link Target .....	89
Cell Right Color .....	90
Cell Right Padding .....	90
Cell Top Color .....	91
Cell Top Padding .....	91
Cell Vertical Align .....	92
Drilldown .....	93
Drilldown Encapsulate .....	93
Drilldown Iteration .....	94
Drilldown Prefetch .....	95
Filter .....	95
Font .....	96
Font Size .....	96
Font Weight .....	97
Foreground .....	97
Format .....	98
Formula .....	99
Group By .....	99
Height .....	100
Input .....	101
Iteration .....	101
Left Color .....	102
Left Padding .....	102
Link .....	103
Link Icon .....	104
Link Keys .....	104
Link Name .....	105
Link Target .....	105
Right Color .....	106
Right Padding .....	106
Rotate .....	107
Sort .....	107
Sort Descending .....	108
Text .....	108
Title .....	109
Top Color .....	109
Top Padding .....	110
Vertical Align .....	110
Visible .....	111
Width .....	112
Z-Order .....	112
Comment .....	113
Author .....	113
Copy To Result .....	113
Date .....	114
Export .....	114
Formula .....	115
Padding .....	115
Text .....	116
Text Indent .....	116

Visible .....	117
<b>CHAPTER 2: Chart properties .....</b>	<b>119</b>
Line-Chart properties .....	120
3D Depth .....	120
3D Mode On .....	120
Area_0 .....	121
Auto Label Spacing On .....	121
Chart Background .....	121
Chart Foreground .....	122
Chart Title .....	122
Connected Lines On .....	123
Default Grid Lines Color .....	123
Default Grid Lines On .....	123
Floating Label Font .....	124
Floating On Legend Off .....	124
Font .....	125
Foreground .....	125
Graph Insets .....	125
Grid Adjustment On .....	126
Grid Image .....	126
Grid Line Colors .....	127
Grid Lines .....	127
Grid Lines Color .....	127
Label_0 .....	128
Label Url_0 .....	128
Label URL Target_0 .....	129
Legend Colors .....	129
Legend Columns .....	130
Legend Font .....	130
Legend Image .....	130
Legend Labels .....	131
Legend On .....	131
Legend Position .....	132
Legend Reverse On .....	132
Line Stroke .....	132
Line Width .....	133
Locale .....	133
Lower Range .....	134
Max Value Line Count .....	134
Range .....	135
Range 2 .....	135
Range Adjusted .....	135
Range 2 Adjusted .....	136
Range Adjuster On .....	136
Range 2 Adjuster On .....	137
Range Adjuster Position .....	137
Range 2 Adjuster Position .....	137
Range Axis Label .....	138
Range 2 Axis Label .....	138
Range Axis Label Angle .....	139
Range 2 Axis Label Angle .....	139
Range Axis Label Font .....	139
Range 2 Axis Label Font .....	140
Range Color .....	140
Range 2 Color .....	140
Range Decimal Count .....	141
Range 2 Decimal Count .....	141
Range Label Font .....	142
Range 2 Label Font .....	142
Range Label Postfix .....	142
Range 2 Label Postfix .....	143

Range Label Prefix .....	143
Range 2 Label Prefix .....	143
Range Labels Off .....	144
Range 2 Labels Off .....	144
Range On .....	144
Range 2 On .....	145
Range Position .....	145
Range 2 Position .....	146
Range Step .....	146
Range 2 Step .....	147
Sample Axis Label .....	147
Range Axis Label Angle .....	147
Sample Axis Label Font .....	148
Sample Colors .....	148
Sample Decimal Count .....	148
Sample Highlight Image .....	149
Sample Highlight On .....	149
Sample Highlight Size .....	150
Sample Highlight Style .....	150
Sample Label Angle .....	150
Sample Label Colors .....	151
Sample Label Font .....	151
Sample Labels .....	152
Sample Labels On .....	152
Sample Label Style .....	152
Sample Scroller On .....	153
Series Label Colors .....	153
Series Labels On .....	154
Series Label Style .....	154
Series Line Off .....	154
Single Click URL On .....	155
Stacked On .....	155
Target Labels Position .....	156
Target Value Line_0 .....	156
Thousands Delimiter .....	157
Title Font .....	157
Url Target .....	157
Value Label Angle .....	158
Value Label Font .....	158
Value Label Postfix .....	159
Value Label Prefix .....	159
Value Labels On .....	159
Value Label Style .....	160
Value Lines Color .....	160
Value Lines On .....	161
Visible Samples .....	161
Zoom On .....	161
Bar-Chart properties .....	163
3D Depth .....	163
3D Mode On .....	163
Auto Label Spacing On .....	164
Bar Alignment .....	164
Bar Label Angle .....	164
Bar Label Colors .....	165
Bar Label Font .....	165
Bar Labels .....	165
Bar Labels On .....	166
Bar Label Style .....	166
Bar Outline Color .....	167
Bar Outline Off .....	167
Bar Type .....	167
Bar Width .....	168
Chart Background .....	168
Chart Foreground .....	169

Chart Title .....	169
Default Grid Lines Color .....	169
Default Grid Lines On .....	170
Floating Label Font .....	170
Floating On Legend Off .....	171
Font .....	171
Foreground .....	171
Graph Insets .....	172
Grid Adjustment On .....	172
Grid Image .....	173
Grid Line Colors .....	173
Grid Lines .....	174
Grid Lines Color .....	174
Label_0 .....	174
Label Url_0 .....	175
Label URL Target_0 .....	175
Legend Colors .....	176
Legend Columns .....	176
Legend Font .....	177
Legend Image .....	177
Legend Labels .....	177
Legend On .....	178
Legend Position .....	178
Legend Reverse On .....	179
Locale .....	179
Lower Range .....	179
Max Value Line Count .....	180
Multi Color On .....	180
Range .....	181
Range 2 .....	181
Range Adjusted .....	181
Range 2 Adjusted .....	182
Range Adjuster On .....	182
Range 2 Adjuster On .....	183
Range Adjuster Position .....	183
Range 2 Adjuster Position .....	183
Range Axis Label .....	184
Range 2 Axis Label .....	184
Range Axis Label Angle .....	185
Range 2 Axis Label Angle .....	185
Range Axis Label Font .....	185
Range 2 Axis Label Font .....	186
Range Color .....	186
Range 2 Color .....	186
Range Decimal Count .....	187
Range 2 Decimal Count .....	187
Range Label Font .....	188
Range 2 Label Font .....	188
Range Label Postfix .....	188
Range 2 Label Postfix .....	189
Range Label Prefix .....	189
Range 2 Label Prefix .....	189
Range Labels Off .....	190
Range 2 Labels Off .....	190
Range On .....	190
Range 2 On .....	191
Range Position .....	191
Range 2 Position .....	192
Range Step .....	192
Range 2 Step .....	193
Sample Axis Label .....	193
Range Axis Label Angle .....	193
Sample Axis Label Font .....	194
Sample Axis Range .....	194

Sample Colors .....	194
Sample Decimal Count .....	195
Sample Label Angle .....	195
Sample Label Colors .....	196
Sample Label Font .....	196
Sample Labels .....	196
Sample Label Selection Color .....	197
Sample Labels On .....	197
Sample Label Style .....	198
Sample Scroller On .....	198
Series Label Colors .....	198
Series Label Font .....	199
Series Labels On .....	199
Series Label Style .....	200
Single Click URL On .....	200
Target Labels Position .....	200
Target Value Line_0 .....	201
Thousands Delimiter .....	201
Title Font .....	202
Url Target .....	202
Value Label Angle .....	203
Value Label Font .....	203
Value Label Postfix .....	203
Value Label Prefix .....	204
Value Labels On .....	204
Value Label Style .....	204
Value Lines Color .....	205
Value Lines On .....	205
Visible Samples .....	206
Zoom On .....	206
Pie-Chart properties .....	207
3D Mode On .....	207
Angle .....	207
Chart Title .....	208
Depth .....	208
Detached Distance .....	208
Detached Slices .....	209
Floating Label Font .....	209
Floating On Legend Off .....	210
Font .....	210
Foreground .....	210
Graph Insets .....	211
Inside Label Color .....	211
Inside Label Font .....	212
Label_0 .....	212
Label Url_0 .....	213
Label URL Target_0 .....	213
Legend Colors .....	213
Legend Columns .....	214
Legend Font .....	214
Legend Image .....	215
Legend Labels .....	215
Legend On .....	215
Legend Position .....	216
Legend Reverse On .....	216
Locale .....	217
Outside Label Color .....	217
Outside Label Font .....	217
Percent Decimal Count .....	218
Percent Labels On .....	218
Percent Label Style .....	219
Pie Label Font .....	219
Pie Labels On .....	219
Pie Rotation On .....	220

Pointing Label Color .....	220
Pointing Label Font .....	221
Sample Colors .....	221
Sample Decimal Count .....	221
Sample Label Colors .....	222
Sample Labels .....	222
Sample Labels On .....	223
Sample Label Style .....	223
Selection Style .....	223
Series Label Colors .....	224
Series Labels On .....	224
Series Label Style .....	225
Single Click URL On .....	225
Slice Separator Color .....	226
Slice Separator On .....	226
Thousands Delimiter .....	226
Title Font .....	227
Url Target .....	227
ValueLabelPostfix .....	228
Value Label Prefix .....	228
Value Labels On .....	228
Value Label Style .....	229
<b>CHAPTER 3: Configuration properties .....</b>	<b>231</b>
Configuration .....	232
Allow ambiguous cube results for equal coordinates .....	232
Build Offline Stores immediatley .....	232
Enable Adhoc Queries .....	233
Load Timeout .....	233
Lock-On-Error Time .....	234
Max Cell Count .....	234
Max Result Size .....	234
Max Selector Size .....	235
Preprocessor .....	235
Query Timeout .....	236
Shutdown-On-Idle Time .....	236
Start model automatically .....	236
Database .....	238
Catalog .....	238
Charset .....	238
Column-end bracket .....	239
Column-start bracket .....	239
Concat-Operator .....	240
Datasource .....	240
Drop IN with NULL .....	240
Escape Character .....	241
Force Connection .....	242
Group By Index .....	242
Group By Subselect .....	243
Initial Query .....	244
JDBC-Driver .....	244
Limit Syntax .....	245
Load Links .....	245
Load Table-Sizes .....	246
Max connection age .....	246
Max connection count .....	247
Max Table-Count .....	247
Max IN-Count .....	248
Name .....	248
Order By Index .....	249
Password .....	249

Read Only .....	250
Schema .....	250
Schema Names .....	251
Single IN Operator .....	251
Table-end bracket .....	252
Table-start bracket .....	253
Table Names .....	253
Table Types .....	254
Timeout .....	254
URL .....	255
Use Aliases .....	255
Use Joins .....	256
User .....	257
Use Schema .....	257
Validation Query .....	258
Table .....	259
Name .....	259
Schema .....	259
Column .....	261
Name .....	261
Type .....	261
Alias .....	262
Name .....	262
SQL-Where .....	262
Table .....	263
Link .....	264
Direction .....	264
SQL-Where .....	265
Name .....	265
SQL-Where .....	265
Link-Expression .....	267
Operator .....	267
Source Expression .....	267
Target Expression .....	268
Table-Expression .....	269
Name .....	269
SQL-Expression .....	269
Table .....	270
Type .....	270
Type .....	270
Dimension .....	272
Cron Pattern .....	272
Default Text-Attribute .....	272
Description .....	273
Level-Names .....	273
Load Children .....	274
Max Size .....	274
Name .....	274
Storage .....	275
Visible .....	276
Grant .....	277
Action .....	277
See also .....	277
Expression .....	277
Key .....	279
Format .....	279
Description .....	279
ID .....	280

Level Name .....	280
Type .....	281
Unit .....	281
Unit .....	282
Key-Attribute .....	283
Description .....	283
Ignore Missing Targets .....	283
Name .....	283
Relink-Attribute .....	284
Target Attribute .....	284
Type .....	285
Unique .....	286
Value .....	286
Unit .....	287
SQL-Keyloader .....	288
Database .....	288
Distinct .....	288
Format .....	289
Ignore Hierarchy Violation .....	289
Level Name .....	290
Mode .....	290
Null-Value .....	291
Order-By-ID .....	291
Parent Attribute .....	292
Parent Format .....	293
Parent Null-Value .....	293
Parent SQL-Expression .....	294
Parent Trim .....	294
Prefix .....	295
Recursive .....	295
SQL-Check Expression .....	297
SQL-Expression .....	297
SQL-Order .....	298
SQL-Order Descending .....	298
SQL-Where .....	299
Trim .....	299
Unit .....	300
SQL-Attribute .....	301
Description .....	301
Format .....	301
Ignore Missing Targets .....	302
Load Distinct .....	302
Name .....	303
Null-Value .....	303
Relink-Attribute .....	303
SQL-Expression .....	304
Target Attribute .....	304
Trim .....	305
Type .....	305
Unique .....	306
Unit .....	307
Time-Keyloader .....	308
Exclude Pattern .....	308
Exclude Values .....	308
Language .....	309
Level Name .....	309
Locale Format .....	310
Mode .....	310
Parent .....	311
Pattern .....	312
Start .....	312
Startshift .....	313

Stop .....	314
Stopshift .....	314
Unit .....	315
Time-Attribute .....	316
Description .....	316
Ignore Missing Targets .....	316
Name .....	316
Pattern .....	317
Relink-Attribute .....	317
Target Attribute .....	318
Type .....	318
Unique .....	319
Unit .....	320
Number-Keyloader .....	321
Format .....	321
Level Name .....	321
Max .....	322
Min .....	322
Unit .....	323
SQL-Cube .....	324
Active .....	324
Database .....	324
Distinct .....	324
Distinct .....	325
Enable Load .....	325
Enable Store .....	326
Match .....	327
Match Expression .....	327
Match Mode .....	328
Name .....	329
SQL-Order .....	329
SQL-Order Descending .....	330
Table Order .....	330
SQL-Where .....	331
Unit .....	331
SQL-Dimension .....	333
Attribute .....	333
Dimension .....	334
Format .....	334
Key .....	335
Level .....	335
Mode .....	336
Null-ID .....	336
Omit-Percentage .....	336
Operator .....	338
Prefix .....	338
SQL-Expression .....	339
SQL Where .....	339
Swap Expression .....	340
Trim .....	340
SQL-Fact .....	342
Fact .....	342
Match .....	342
SQL-Expression .....	343
SQL-Where .....	343
Store .....	344
Active .....	344
Algorithm .....	344
Build Timeout .....	345
Cron Pattern .....	345

Granularity .....	346
Match .....	347
Name .....	347
Name .....	348
Store Dimension .....	349
Load Mode .....	349
Store Fact .....	350
Aggregation .....	350
Formula .....	351
Active .....	351
Description .....	351
Expression .....	351
Fact .....	352
Match .....	352
File-Cache .....	354
Active .....	354
Cache Line-Numbers .....	354
Cron-Pattern .....	355
Match-Expression .....	355
Max Age .....	356
Filename .....	356
Include .....	357
Model .....	357
<b>CHAPTER 4: Expressions .....</b>	<b>359</b>
The Type-system .....	360
All .....	360
Any .....	360
Boolean .....	361
Date .....	361
Double .....	361
Integer .....	361
Key .....	361
Number .....	361
String .....	361
Value .....	361
Syntax .....	362
Dimensions and selections .....	362
Dimension-Levels .....	362
Operators .....	363
Brackets .....	364
Accessing attributes .....	365
Accessing variables .....	366
Function calls .....	367
Level-Functions .....	368
Fact-Functions .....	368
Comments .....	369
Constants .....	370
Boolean constants .....	370
Integer numbers .....	370
Double numbers .....	371
Strings .....	371
Dimension-Keys .....	372
NULL .....	372
Functions .....	374
ABC .....	374
ABS .....	374

ADD .....	375
ALL .....	376
ANCESTORS .....	377
AND .....	378
ATTRIBUTENAMES .....	379
ATTRIBUTENV .....	379
ATTRIBUTES .....	380
AVAILABLEOFFLINE .....	381
AVG .....	381
AVGKEY .....	382
BEAUTIFY .....	383
BELONGSTO .....	384
CATCH .....	384
CASE .....	385
CEIL .....	386
CHILDREN .....	387
CLUSTER .....	387
COALESCE .....	388
COLSPAN .....	389
CONCAT .....	389
CONTAINS .....	390
CONTAINSTEXT .....	391
COUNT .....	392
COUNTRY .....	392
CUBE .....	393
DATEADD .....	395
DATEDIFF .....	395
DEBUG .....	396
DEFAULTTEXT .....	396
DEVIATION .....	397
DEPTH .....	398
DIMENSIONATTRIBUTENAMES .....	399
DIMENSIONNAME .....	399
DIMENSIONNAMES .....	400
DISTINCT .....	401
DIV .....	401
DLOOKUP .....	402
DRILLKEY .....	404
DRILLEVEL .....	404
DSORT .....	405
DURATIONTOSTRING .....	406
ELEMENT_AT .....	406
EMPTY .....	407
EMPTYASNULL .....	408
ENDSWITH .....	408
EQUAL .....	409
ERROR .....	410
EVAL .....	410
EXISTS .....	411
EXP .....	412
FACTROOT .....	412
FAMILY .....	413
FILTER .....	414
FILTERKEYS .....	414
FIND .....	415
FIRST .....	416
FKEY .....	417
FLOOR .....	417
FOREACH .....	418
FORECAST .....	419
FPOP .....	422
FPUSH .....	423
GETDAY .....	424
GETHOUR .....	425

GETSECOND .....	425
GETMINUTE .....	426
GETMONTH .....	426
GETSECOND .....	427
GETYEAR .....	427
GREATER .....	428
GREATER_OR_EQUAL .....	429
HASACCESS .....	430
HASCHILDREN .....	430
HASKEYS .....	431
HASLEVEL .....	432
HASPOSITION .....	432
HASROLES .....	433
HASUSER .....	434
IIF .....	434
IN .....	435
INTERSECT .....	436
ISCHILD OF .....	436
ISEMPTY .....	437
ISLEAF .....	438
ISNULL .....	439
ISPARENT OF .....	439
ITERATIONKEY .....	440
IVALUE .....	441
JOIN .....	441
KEYINDEX .....	442
LANGUAGE .....	443
LAST .....	443
LEAFS .....	444
LEFT .....	445
LESS .....	446
LESS_OR_EQUAL .....	447
LEVEL .....	448
LEVELNAMES .....	449
LEVEL OF .....	449
LIKE .....	450
LIMIT .....	451
LOCALE .....	452
LOOKUP .....	452
LTRIM .....	453
MATCH .....	454
MATRIX .....	455
MAX .....	456
MAXKEY .....	456
MAX_X .....	457
MAX_Y .....	457
MIN .....	458
MINKEY .....	459
MIN_X .....	459
MIN_Y .....	460
MIX .....	460
MOD .....	461
MUL .....	462
NEIGHBOURS .....	463
NEXT .....	464
NONFACTROOTS .....	465
NONLEAFS .....	465
NOT .....	466
NOW .....	467
OR .....	468
PARENT .....	469
PEDIGREE .....	470
PERCENTILE .....	470
POSITION OF .....	471

POW .....	472
PRED .....	472
PREV .....	473
RANGE .....	475
REGEXP .....	475
REGRESSION .....	476
REPLACE .....	477
RETURNTYPE .....	478
REVERSE .....	478
RIGHT .....	479
ROUND .....	480
ROWNUM .....	480
ROWSPAN .....	481
RTRIM .....	482
SORT .....	483
SPLINE .....	483
SPLIT .....	484
SQRT .....	485
STARTSWITH .....	485
STRLEN .....	486
SUB .....	487
SUBSTR .....	488
SUBSTRINGBEHIND .....	488
SUBTOTAL .....	489
SUBTOTALS .....	490
SUCC .....	490
SUM .....	492
SWITCH .....	492
SYSDATE .....	493
TEXTPOSITION .....	493
TIMESTAMP .....	494
TODATE .....	495
TOINTEGER .....	496
TOKEY .....	497
TOLOWER .....	497
TONUMBER .....	498
TOSTRING .....	499
TOUPPER .....	500
TRIM .....	501
TOUPPER .....	502
TYPE .....	502
UNEQUAL .....	503
UNIT .....	504
UPPERNEXT .....	504
UPPERPREV .....	505
USER .....	506
VARIANCE .....	507
WITHOUT .....	507
XML2ASCII .....	508
X .....	508
XHEADER .....	509
Y .....	510
YHEADER .....	510
YTD .....	511
ZERO .....	512

<b>CHAPTER 5: Formats .....</b>	<b>513</b>
Date formats .....	514
Number formats .....	516
Cron patterns .....	517

---

Colors .....	518
<b>CHAPTER 6: SQL-Expressions .....</b>	<b>519</b>
SQL-Expressions in instantOLAP .....	520
Syntax .....	522
Tables and columns .....	522
Constants .....	523
Operators .....	523
Functions .....	525
Aggregation functions .....	525
SQL Passthrough .....	527
Brackets .....	528

# Typographical Conventions

---

Before you start using this guide, it is important to understand the terms and typographical conventions used in the documentation. The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information
<b>Special Bold</b>	Items you must select, such as menu options, command buttons, or items in a list.
Brackets (<>)	Used for variable expressions such as parameters.
Monospace font	Marks code examples or an expression-syntax.
<i>Emphasis</i>	Use to emphasize the importance of a point.
CAPITALS	Names of keys on the keyboard. for example, SHIFT, CTRL, or ALT.
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another, for example, CTRL+P, or ALT+F4.

# CHAPTER 1:

## Query properties

### Contents of this chapter:

Multiple Query .....	22
Selector .....	34
Selector Group .....	42
Outer block .....	46
Inner block .....	50
Query .....	66
Header .....	79
Comment .....	113

# Multiple Query

---

## Align Blocks

### Type

Boolean

### Since

2.6.1

### Description

By default, all inner blocks in a report are stretched to a equal width and all blocks are displayed aligned at the right end.

By setting this property to "false" you can disable this automatic stretching. The default value for this property is "true".

### Examples

```
Align Blocks = "false"
```

## Author

### Type

Constant String

### Since

1.2

### Description

The name of the author of a query. This is a simple text- and can be set to any value you want. The author is set initially when a query is created with the query-wizard.

The author of a query will be displayed in the index and above the query when the query was executed.

### Examples

```
Author = "admin"
```

**See also**

Date

**Background****Type**

String

**Since**

2.6.1

**Description**

This property sets the background color for the complete report.

**Examples**

```
Background = "' #eeeeee' "
```

**CSS****Type**

String

**Since**

1.2

**Description**

When using the default stylesheet (determined by the property Stylesheet), all colors and fonts are set by the cascading-stylesheet (CSS) `"/iolap/stylesheet/iolap.css"`. You can change this CSS for a query by changing its CSS-property and use your own CSS-file.

**Examples**

```
CSS = "' /iolap/stylesheet/green.css' "
```

**See also**

Stylesheet

## Date

### Type

Constant String

### Since

1.2

### Description

The creation-date of a query. This is a simple text and can be set to any value you want. The date is set initially when a query is created with the query-wizard.

The date will be displayed in the index and above the query when the query was executed.

### Examples

```
Date = "01.04.2004 10:30:00"
```

### See also

Author

## Description

### Type

Constant String

### Since

1.2

### Description

This property allows to add a description to queries. The description of a query will be display in the index-page of the Web-Frontend under the report-title.

### Examples

```
Description = "Sales-Report for all products"
```

### See also

Date, Author

## Enable CSV Export

### Type

Boolean

### Since

2.6.0

### Description

By setting this property to "false" you can disable the CSV export for this report. The default value for this property is "true".

### Examples

```
Enable CSV Export = "false"
```

## Enable PDF Export

### Type

Boolean

### Since

2.6.0

### Description

By setting this property to "false" you can disable the PDF export for this report. The default value for this property is "true".

### Examples

```
Enable PDF Export = "false"
```

## Enable XLS Export

### Type

Boolean

### Since

2.6.0

## Description

By setting this property to "false" you can disable the Excel export for this report. The default value for this property is "true".

## Examples

```
Enable XLS Export = "false"
```

## External Transformer

### Type

String

### Since

2.6.1

### Description

Since version 2.6.1 new export formats can be plugged into a report by defining the used Java class (with its complete package) in this property. Whenever this property is used, the displayed export format name in the toolbar of the query viewer is defined by the property "External Transformer Name".

### Examples

```
External Transformator = "com.mycompany.MyTransformer"
```

### See also

External Transformer Name

## External Transformer Name

### Type

String

### Since

2.6.1

### Description

Since version 2.6.1 new export formats can be plugged into a report by defining the used Java class (with its complete package) in the property "External Transformer". This property defines the display name for the new export format in the query viewer toolbar.

## Examples

```
External Transformator Name = "My Export"
```

## See also

External Transformer

## Filter

### Type

Key

### Since

1.2

### Description

This property sets the global filter for a query. A filter is a set of keys (with no, one or more keys per dimension). Whenever later expressions refer to the current selection of a dimension, they will receive the keys set by this or later filters.

Filtering dimensions does not forbid embedded elements to refer other keys than the ones in the filter, it only set the current selection of them. E.g. if a filter "Product:ProductA" was used, the expression "Product" would return "Product:A" but the expression "NEXT( Product )" will still work and return a key (which is not part of the filter).

### Examples

```
Filter = "Product:ProductA"
```

Sets the selection to the key "Product:A"

## Icon

### Type

Constant String

### Since

2.0

### Description

Use this property to display an alternative symbol for this report in the index-page of the Web-Frontend.

The icon can be provided both as relative (e.g. "mylogo.gif") or absolute (e.g. "/iolap/mydemo/logo.gif") path. Note that relative paths may not work after saving queries or snapshots because the saved query or snapshot are placed in another folder.

In difference to the query-logo, this is a simple string and no stringexpression which means the icon of a query is constant and can't be evaluated (because the icon will be visible for the user before the query is being executed).

## Examples

```
Icon = "icon.gif"
```

## See also

Logo

## Logo

### Type

String

### Since

1.2

### Description

To use another logo with this query you have to change this property. The default-value for this property is `"/iolap/logo.gif"`.

The logo can be provided both as relative (e.g. "mylogo.gif") or absolute (e.g. "/iolap/mydemo/logo.gif") path. Note that relative paths may not work after saving queries or snapshots because the saved query or snapshot are placed in another folder.

## Examples

```
Logo = "' mylogo.gif' "
```

```
Logo = "IIF( HASROLE( 'manager' ), 'manager.gif', 'logo.gif' )"
```

## See also

Icon

## Model

### Type

Constant String

## Since

1.0

## Description

The mandatory property "Model" defines the used model for a query. All dimensions and facts a query can use are defined by the model, so this is the most important property.

Each model is defined by a configuration (a file with the extension ".config") and its location in the repository determines the exact model-name. E.g. a configuration called "demo.config" in a directory "demo" generates the model "/demo/demo".

You may refer models in a relative or absolute way. Absolute paths start with "/", relative path start with a ".." (for the parent folder) or a folder or model-name. E.g. the value "/demo/demo" would refer the model "demo" in the root-folder "demo" and the value "config/demo" would refer the model "demo" in a subfolder "config".

## Examples

```
Model = "demo"
```

Refers to the model "demo" in the same folder

```
Model = "/demo/demo"
```

Refers to the model "demo" in the root-folder "demo"

```
Model = "../demo"
```

Refers to the model "demo" in the parent-folder

```
Model = "config/demo"
```

Refers to the model "demo" in the sub-folder "config"

## Name

### Type

Constant String

### Since

1.0

### Description

The "Name" property sets the query-name under which it will be displayed in the navigation and index-page of the web-frontend. In difference to the title of a query, the name-property expects a plain string (and no formula), because the name is displayed before the execution of the query.

## Examples

```
Name = "Sales overview"
```

## Print Height

### Type

String

### Since

2.1

### Description

When a query is exported to a printable format (PDF), this property sets the height of the page. This property expects a string-expression resulting into string of the format <number><unit>. Valid units are "cm", "in", "mm" or "px" for pixels.

Whenever a page wouldn't fit onto a normal-sized page (e.g. A4) you can resize the page with this property. Then the user can resize the page to the desired print-size when printing the report.

### Examples

```
Print Height = "' 50cm' "
```

### See also

Print Logo, Print Width

## Print Logo

### Type

String

### Since

2.1

### Description

When a query is exported to a printable format (PDF), this property sets the logo for the printout. The logo can be provided both as relative (e.g. "mylogo.gif") or absolute (e.g. "/iolap/mydemo/logo.gif") path. Note that relative paths may not work after saving queries or snapshots because the saved query or snapshot are placed in another folder.

If no specific print-logo is defined, the default logo (defined by the property "Logo") is used for PDF-export.

## Examples

```
Print Logo = "' mylogo.gif' "
```

## See also

Logo, Print Height, Print Width

## Print Width

### Type

String

### Since

2.1

### Description

When a query is exported to a printable format (PDF), this property sets the width of the page. This property expects a string with both the size and the unit of the size, e.g. "21cm".

Whenever a page wouldn't fit onto a normal-sized page (e.g. A4) you can resize the page with this property. Then the user can resize the page to the desired print-size when printing the report.

### Examples

```
Print Width = "' 40cm' "
```

### See also

Print Height, Print Logo

## Refresh Time

### Type

Constant Integer

### Since

2.2

### Description

Queries can be automatically refreshed while the user watches them. This property allows to set the refresh time in seconds.

Note that the refresh uses Java-Script. So disabling Java-Script also disables the automatic refresh of a query.

## Examples

```
Refresh Time = "900"
```

Refreshes the report every 15 minutes

## Title

### Type

String

### Since

1.0

### Description

The "Title" is an expression for the generated title of the query. The title is generated at moment of the execution by evaluating the stringexpression.

The title can contain current filter-setting etc. and can become quite long. Whenever a query is exported or saved into a snapshot, the title is taken as the result-name, so you should make the title well speaking.

## Examples

```
Title = "'Sales report for ' + TOSTRING( Time )"
```

## See also

Name

## Visible

### Type

Boolean

### Since

1.2

### Description

With the property "Visible" you can determine if a query is visible for the end-user or not. By default every query is visible.

If the property is set to "false", the query will become invisible in the index but still be link able from other queries. Use this property e.g. to hide detail-queries which should only be reachable from other queries.

## Examples

```
Visible = "true"
```

```
Visible = "false"
```

## See also

Show Portal

# Selector

---

## Default

### Type

Value

### Since

1.2

### Description

This property defines the initially selected value(s) for a selector. The values returned by the expression hold in this property should be a subset of the options, otherwise the system will raise an exception.

If the selector is a single-selector, the default-value only should return one value. Otherwise multiple values can be returned and will be selected initially.

If you define a default for a interval-selector, your default-expression should return two keys. The first will set the from- and the second the to-value.

### Examples

```
Default = "Product: ProductA"
```

```
Default = "LAST( LEVEL( Time, 2 ) )"
```

```
Default = "5"
```

### See also

Match

## Description

### Type

Constant String

### Since

2.6.1

## Description

This property may contain a description of the selector. The description is not used or displayed at any other point of the system.

## Examples

```
Description = "Product selector"
```

## Height

### Type

Integer

### Since

2.1

## Description

This property sets the height of a selector in pixels (you must insert a formula here resulting to an integer-value). If no height is set, it will be set automatically by the frontend.

## Examples

```
Height = "30"
```

## See also

Width

## Match

### Type

Boolean

### Since

2.2

## Description

The optional match-expression defined with this property is used to filter the options and selections of a filter.

Especially for hierarchy-selectors and for input-fields, using this property is a much better and less resource-intensive way to reduce the possible selection, because the possible options of the selector (which could be a large number of keys) don't have to be calculated in ahead.

## Examples

```
Match = "HASLEVEL( Product, 2 )"
```

```
Match = "Product.State = 1"
```

## See also

Options

## Name

### Type

String

### Since

1.2

## Description

This property defines the name of a selector. The name of a selector will define the variable-name under which you later can access the selection in the query or which dimension (with the same name) will be filtered.

If you want to filter a dimension, you don't need to set this name because the selector's name is evaluated automatically by the system (the dimension-name of the first option-key is taken as the selectorname).

## Examples

```
Name = "Y-Dimension"
```

## See also

Options

## Options

### Type

Value

### Since

1.0

## Description

Every selector has a number of options which the user can choose from (or search in). Define the options with a Value-Expression in this property.

If the expression results to keys (if the expression is a Key-Expression), the "Name" of the selector will be set automatically to the dimension's name and the selector will filter the corresponding dimension. If the expression result to other values, you additionally have to set the name for this selector and can use the selection as variable later.

## Examples

```
Options = "LEVEL( Time, 2 )"
```

```
Options = "10 | 20 | 30"
```

## See also

Default, Match, Name

## Submit

### Type

Boolean

### Since

1.2

## Description

If this property is set to "true", the query will refresh immediately after the selectors selection has changed. Otherwise the user has to press the "reload" button to view the changed report.

## Examples

```
Submit = "true"
```

```
Submit = "false"
```

## Text

### Type

String

### Since

1.1

### Description

By default, the options of a selectors will be displayed with their default-text (e.g. the ID for keys or the simple string-conversion for numbers and other values). If you want to change the text being display for each option, you can define a Text-Expression with this property (if the options contain dimensions-keys, a filter will be passed to the expression and you can refer to the key by its dimension-name).

Note this property only changes the displayed text for each option but not the selected value!

### Examples

```
Text = "LIMIT( Product, 10 )"
```

```
Text = "Product.ProductID + ':' + TOSTRING( Product )"
```

### See also

Title

## Title

### Type

String

### Since

1.2

### Description

This property changes the title of a selector which is the text above the selector. If you don't specify any title, the "Name" of the selector (or the name of the influenced dimension) will be used asb title.

### Examples

```
Title = "' Prod. No' "
```

### See also

Name, Text

## Type

### Type

Constant String ('single', 'multiple', 'radio', 'checkbox', 'hierarchy', 'hierarchy-applet', 'calendar', 'button', 'input' or 'constant')

## Since

1.1

## Description

There are different types of selectors from which you can choose one for each selector by changing this property:

- **single:** The selector is a simple drop-down box known from Windows and browsers. The user only can select one option.
- **multiple:** The selector is displayed as a list where the user can select multiple options.
- **radio:** The options are displayed as radio-buttons and the user only can select one option.
- **checkbox:** The options are display as check boxes where the user can select multiple options.
- **input:** The user can input a match-pattern (with the wildcards \* and ?) and search within the options.
- **constant:** The selector is unchangeable.
- **hierarchy:** The user can navigate in a dimension-hierarchy and select one item.
- **hierarchy\_applet:** The user can navigate in a dimensionhierarchy and select one item (this is an interactive applet).
- **calendar:** The user can single pick a date from an interactive calendar (this is an interactive applet).
- **button:** The user can select a single item with a button.

## Examples

```
Type = "single"
```

```
Type = "multiple"
```

```
Type = "radio"
```

```
Type = "checkbox"
```

```
Type = "input"
```

```
Type = "constant"
```

```
Type = "button"
```

```
Type = "hierarchy"
```

```
Type = "hierarchy_applet"
```

```
Type = "calendar"
```

## See also

Name, Text

## Use Filter

### Type

Boolean

### Since

2.1

### Description

If the "Use Filter" property is set to "true", all formulas (e.g. the "Options" or "Default" property) will use the filter of the query (influenced by the previous selectors and queryparameters). Otherwise, the formulas will use the root-filter (then each dimension is set to its root-key).

By default, selectors don't use the query-filter.

### Examples

```
Use Filter = "true"
```

```
Use Filter = "false"
```

## Visible

### Type

Boolean

### Since

2.6.1

### Description

This property defines if the selector becomes visible inside the selector bar. By default this property is empty and the selector is visible.

### Examples

```
Visible = "$TYPE = 'A' "
```

## Width

### Type

Integer

## Since

2.1

## Description

This property sets the width of a selector in pixels (you must insert a formula here resulting to an integer-value). If no width is set, it will be set automatically by the frontend.

## Examples

```
width = "100"
```

## See also

Height

# Selector Group

---

## Background

### Type

String

### Since

2.2.2

### Description

The "Background" property sets the background-color for the selector-group. If no color is defined (this is the default-setting), the group becomes transparent.

### Examples

```
Background = "' red' "
```

```
Background = "NULL"
```

### See also

Border

## Border

### Type

String

### Since

2.2.2

### Description

The "Border" property defines the color for the border for the selector-group.

### Examples

```
Border = "' black' "
```

```
Border = "NULL"
```

**See also**

Background

**Height****Type**

Integer

**Since**

2.2.2

**Description**

The "Height" property defines the height of the selector-group in pixels. If no height is defined the height of the group will be automatically extended by the content of the group.

**Examples**

```
Height = "250"
```

**See also**

Width

**Popup****Type**

Boolean

**Since**

2.6.0

**Description**

Use this property to save space in your selector bar by converting the selector group in to a popup selector: If set to "true", only the title of the group will be displayed as a button and the rest of the group becomes invisible.

Whenever the user clicks onto the title button, the group becomes visible as a popup panel and the use can change its contained selectors. If the user clicks onto the button again or anywhere outside the popup panel, the panel will disappear.

**Examples**

```
Popup = "true"
```

## Title

### Type

String

### Since

2.2.2

### Description

With the "Title" property you can define a title for your selector-group which will be displayed above the group. If no title is defined, the place for the title will stay empty (there is no default-title for selector-groups).

### Examples

```
Title = "' ABC-Settings' "
```

## Visible

### Type

Boolean

### Since

2.6.1

### Description

This property defines if the selector group becomes visible inside the selector bar. By default this property is empty and the group is visible.

### Examples

```
Visible = "$TYPE = 'A' "
```

## Width

### Type

Integer

### Since

2.2.2

## Description

The "Width" property defines the width of the selector-group in pixels. If no width is defined the width of the group will be automatically extended by the content of the group.

## Examples

```
width = "250"
```

## See also

Height

# Outer block

---

## Export

### Type

Boolean

### Since

2.2

### Description

If this property contains "true" (the default value), this block will be copied to the result whenever the user exports a query (e.g. to PDF or Excel). Otherwise, this block will disappear.

### Examples

```
Export = "true"
```

```
Export = "false"
```

### See also

Visible

## Filter

### Type

Key

### Since

1.2

### Description

Each element of a query has a filter which influences all expressions in the encapsulated elements. A filter is a collection of keys with no, one or more keys per dimension. All encapsulated elements of this outer-block, which refer to the current filter, will retrieve the selected keys for the corresponding dimension.

With the Filter-Property of an outer block, you can change the filter and pass it over to the encapsulated elements. E.g. you could filter one or more products etc. The property expects a Key-Expression and the result of the Key-Expression will be applied to the filter. Applying

means, for all dimensions contained in the result the dimension in the filter will be changed to the keys of this dimension. All other dimensions will stay unaffected.

## Examples

```
Filter = "Product: ProductA"
```

Each expression referring to "Product" will now return "ProductA" as result

## Iteration

### Type

Key

### Since

1.1

### Description

Most elements in a report can be repeated automatically by specifying an iteration for the element. To repeat an element, insert a valid Key-Expression into the Iteration-property. Then the element will be repeated for each single key of the expression-result (e.g. if the iteration returns 5 elements, the vertical block will be visible 5 times). If the expression-result was NULL, no block will be displayed.

The iteration does not only repeat the element, it also influences the filter for each single copy: The filter for each copy is composed out of the surrounding filter (for vertical blocks, the surrounding filter is defined by the query-filter and the selectors) and the single key from the iteration- expression. The key is applied to the filter which means the selection for the one dimension of the filter is overridden with this key, all other dimensions stay unaffected.

Inside the copy of the element, you refer to the iterated key by using the dimension-name inside expressions. Just using the dimensionname returns the current selection for the dimension, which is the single key set by the iterator.

### Examples

```
Iteration = "CHILDREN( Product )"
```

Iterates the block for each Product. Sub elements can refer the product with "Product".

## Keep Together

### Type

Boolean

### Since

2.2

## Description

When the expression stored in this property returns "true", the PDF-output for this query will try to keep all sub-blocks, tables and elements inside this block together on a single page. This will not be possible if the block is larger than a page of the PDF file.

The default for this property is "false" and outer blocks are usually floated over page margins when generating a PDF export of a query.

## Examples

```
Keep Together = "true"
```

```
Keep Together = "false"
```

## See also

Pagebreak

## Orientation

### Type

Constant String ('horizontal', 'vertical', 'tabbed' or 'animated')

### Since

2.0

### Description

The Orientation-property of the outer block defines, how the inner blocks of this block will be arranged. There are four different possibilities to arrange the inner blocks:

- **horizontal:** all blocks will be lined up in a row (default setting)
- **vertical:** all blocks will be displayed under the previous one
- **tabbed:** each block becomes a tab in a tabbed display
- **animated:** the blocks will be displayed after each other, only one block will be visible at a given time.

When using the 'animated' orientation, you can control the display-time (the duration of being visible) of each block with its "Animation Delay" property.

*For animated charts it is better to display as generated images (using the "easychartservlet" format) because the chart-applet would flicker too much when being displayed in an animation.*

## Examples

```
Orientation = "horizontal"
```

```
Orientation = "vertical"
```

```
Orientation = "tabbed"
```

```
Orientation = "animated"
```

## Pagebreak

### Type

Boolean

### Since

2.2.2

### Description

If the "Pagebreak" property returns true, a pagebreak will be inserted before the block in the PDF-output.

### Examples

```
Pagebreak = "POSITIONOF( Product ) > 0"
```

### See also

Keep Together

## Visible

### Type

Boolean

### Since

2.2

### Description

If the expression stored in this optional property result to "false", the block will become invisible inside the result. Note that invisible blocks are not created and consume no CPU time.

### Examples

```
Visible = "true"
```

```
Visible = "$SHOW_DETAILS = 'true' "
```

### See also

Export

# Inner block

---

## Automation Delay

### Type

Integer

### Since

2.2.2

### Description

If the owner of this block (an outer block) uses an "Animation" for the orientation of its content, each contained inner block (including this) will be visible for a freely defined period of time until it becomes invisible and will be replaced on the screen by the following block.

By changing this property you can define (in milliseconds) how long the block will be visible. The default value for this property is "1000" which means that that block would be visible for one second.

### Examples

```
Automation Delay = "250":
```

## Background

### Type

String

### Since

2.1

### Description

This property sets the background-color of the inner block. The expression must result to a string containing the color-name or hexcode (with a leading #).

### Examples

```
Background = "' red' ":
```

Sales	Quantity	Amount
Desktop	258,280	524,724,788
Notebook	488,194	1,098,261,324
Pocket PC	383,930	122,267,368
Tablet PC	179,222	495,649,228

## See also

Border, Color

## Border

### Type

String

### Since

2.1

### Description

This property sets the border-color and title-background of the inner block. The expression must result to a string containing the colorname or hex-code (with a leading #).

### Examples

```
Border = "' #FF0000' ":
```

Sales	Quantity	Amount
Desktop	259.260	524.724.709
Notebook	409.194	1.098.981.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	485.949.726

## See also

Background, Color

## Color

### Type

String

### Since

2.1

### Description

This property sets the font color of the block title. The expression must result to a string containing the color-name or hex-code (with a leading #).

### Examples

```
Color = "' red' "
```

Sales		
	Quantity	Amount
Desktop	258.260	524.724.708
Notebook	409.184	1.098.361.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	495.949.726

## See also

Background, Border

## Export

### Type

Boolean

### Since

2.2

### Description

If this property contains "true" (the default value), this block will be copied to the result whenever the user exports a query (e.g. to PDF or Excel). Otherwise, this block will disappear.

### Examples

```
Export = "true"
```

```
Export = "false"
```

## See also

Visible

## Filter

### Type

Key

### Since

1.2

### Description

Each element of a query has a filter which influences the result of Key-Expressions in all encapsulated elements. A filter is a collection of keys with no, one or more keys per dimension. All encapsulated elements of this inner-block, which refer to the current filter, will retrieve the selected keys for the corresponding dimension.

With the Filter-Property of an inner block, you can change the filter and pass it over to the encapsulated elements. E.g. you could filter one or more products etc. The property expects a Key-Expression and the result of the Key-Expression will be applied to the filter. Applying

means, for all dimensions contained in the result the dimension in the filter will be changed to the keys of this dimension. All other dimensions will stay unaffected.

## Examples

```
Filter = "Product: ProductA"
```

Each expression referring to "Product" will now return "ProductA" as result

## Font

### Type

String

### Since

2.1

### Description

This property sets the font of the block-title. This property must contain a formula returning the font-name as string. If no font-name is set or the formula returns NULL, the system will use the default-font for the title.

### Examples

```
Font = "' Courier New' "
```

Sales	Quantity	Amount
Desktop	258.060	524.724.708
Notebook	409.184	1.098.961.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	455.949.726

### See also

Font Weight, Font Size

## Font Size

### Type

Integer

### Since

2.1

### Description

This property sets the font-size of the block-title in pixels. If no "Font Size" is set or the formula contained in this property returns NULL, the default-size of the system is used.

## Examples

```
Font Size = "20"
```

Set the title-font size to 20 pixels:

Sales	Quantity	Amount
Desktop	258.260	524.724.708
Notebook	408.184	1.098.361.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	495.949.726

## See also

Font, Font Weight

## Font Weight

### Type

String

### Since

2.1

### Description

This property sets the weight of the block-title. Possible weights are 'bold' and 'lighter'.

## Examples

```
Font Width = "'lighter' "
```

Display the title in plain-font

```
Font Width = "'bold' "
```

Display the title in bold-font:

## See also

Font, Font Size

## Format

### Type

Constant String ('table', 'easychart', 'easychartservlet' or 'newsticker')

### Since

1.1

## Description

The Format-property on an inner block sets the display-format for the query inside the block. The following formats are available:

- **table**: the results will be displayed as a pivot-table
- **easychart**: the result will be displayed as a chart (using the chart-applet)
- **easychartservlet**: the result will be displayed as a chart (as server-side generated gif)
- **newsticker**: the result will be display in a newsticker

### table

The table-format displays the data of the encapsulated queries in a pivot table-format. Each row of the query will become a row in the pivot-table and each column will become column in the pivot-table. This format doesn't need any sizing-properties for the inner block, because the table-size will be calculated automatically to the best fit.

Sales	Quantity	Amount
Desktop	258.260	524.724.705
Notebook	409.104	1.098.361.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	485.949.726

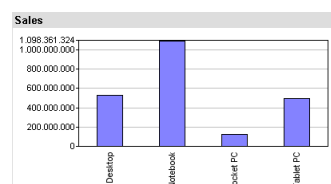
Pivot-Tables allow the following display-features of the queries:

- Texts and Values
- Drilldown
- Links and Cell-Links
- Traffic-Lighting (assertions)
- Images in cells
- Rotated Header-Texts

### easychart and easychartservlet

The chart-formats (easychart and easychartservlet) will display the result of the query as a business-diagram. The result will be mapped automatically to a chart (each column will become a category and each row will become a data-series in the chart). Because the diagram needs a defined size to be displayed, you must set the properties Width and Height of the inner block, otherwise no chart would be visible.

There are two different ways of embedding a chart into query: As a Java-Applet (easychart) and as a server-side generated PNG-image (easychartservlet). The server-side variant is a bit slower than the applet variant, because the images have to be generated and downloaded with separate connections from the server. The applet need Java 1.1 support in your browser (which the most browsers have).



Charts allow the following display-features of the queries:

- Texts and Values

- Links and Cell-Links (can also be used to drill-down into the chart)

### ***newsticker***

The Newsticker-format displays the result of the embedded query in form of a newsticker: Each row of the result (without the headers) will become one message in the ticker. With multiple rows in your result you can create multiple messages in your newsticker.

Sales		
Top:	524.724.708	Notebook: 1.098.361.324
		Pocket PC: 122.257.31

Newstickers allow the following display-features of the queries:

- Texts and Values
- Cell-Links

## **Examples**

```
Format = "table"
```

```
Format = "easychart"
```

```
Format = "easychartservlet"
```

```
Format = "Newsticker"
```

## **See also**

Height, Width

## **Height**

### **Type**

Integer

### **Since**

1.1

### **Description**

This property sets the height (in pixels) of the block. To set the height of a block is necessary for some block-formats (easychart, easychartservlet), all other formats will calculate their automatically if this property is left empty.

### **Examples**

```
Height = "300"
```

## See also

Format, Width

## Iteration

### Type

Key

### Since

1.1

### Description

Most elements in a report can be repeated automatically by specifying an iteration for the element. To repeat an element, insert a valid Key-Expression into the Iteration-property. Then the element will be repeated for each single key of the expression-result (e.g. if the iteration returns 5 elements, the vertical block will be visible 5 times). If the expression-result was NULL, no block will be displayed.

The iteration does not only repeat the element, it also influences the filter for each single copy: The filter for each copy is composed out of the surrounding filter (for inner blocks, the surrounding filter is defined by the outer block) and the single key from the iterationexpression. The key is applied to the filter which means the selection for the one dimension of the filter is overridden with this key, all other dimensions stay unaffected.

Inside the copy of the element, you refer to the iterated key by using the dimension-name inside expressions. Just using the dimensionname returns the current selection for the dimension, which is the single key set by the iterator.

### Examples

```
Iteration = "CHILDREN( Product )"
```

Iterates the block for each Product. Sub-elements can refer the product with "Product"

## Keep Together

### Type

Boolean

### Since

2.2

### Description

When the expression stored in this property returns "true", the PDF-output for this query will try to keep all tables and elements inside this block together on a single page. This will not be possible if the block is larger than a page of the PDF file.

The default for this property is "false" and inner blocks are usually floated over page margins when generating a PDF export of a query.

## Examples

```
Keep Together = "true"
```

```
Keep Together = "false"
```

## Link

### Type

String

### Since

2.1

### Description

This property sets a link for the inner block to the URL returned by the formula in this property. The returned link must be a valid URL (relative or absolute URLs are both allowed). You also may add some parameters (delimited by ? and &) to the URL.

The link will be automatically extended by all keys in the current filter of the block. If you don't want the system to append all keys from the current filter to the URL you can use the Link-Keys property to define exactly which keys to add.

You may also use javascript in links. Links starting with "javascript:" will be transferred to the "onClick" attribute of a link instead of the "href" attribute.

## Examples

```
Link = "' details.html' "
```

```
Link = "'javascript: alert( 'Hello world' )'"
```

## See also

Link-Icon, Link-Keys, Link-Name, Link-Target

## Link Icon

### Type

String

### Since

2.1


## Description

With this property you can specify a link-icon to be displayed next to the link or link-name of the inner block. The result of this property must be a valid URL for an icon (in JPEG, GIF or PNG format).

## Examples

```
Link-Icon = "'/iolap/icons/icon_detail.gif' "
```

Sales		
	Quantity	Amount
Desktop	258 280	524 724 708
Notebook	409 194	1 098 361 324
Pocket PC	203 920	122 257 350

SHOW DETAILS 

## See also

Link, Link-Keys, Link-Name, Link-Target

## Link Keys

### Type

Key

### Since

2.1

## Description

This property controls the generation of the inner-block links. If no "Link Keys" are defined, the link-generator will append the complete current filter dimension to the link, so the target-query will exactly have the same (input) filter as this block has. For example, a block showing the "Product:A" at "Time:2004" will append this filter to its generated links.

If you don't want the generator exactly to transfer the filter of a block and control the list of parameters yourself, you can use the Link-Keys properties to generate a list of keys, which will be appended as parameters to the generated link. The property expects a Key-Expression and each key of the expression-result will become one parameter appended to the link. The expression can refer to the current selection, but you also can leave out keys or use more complex expression.

## Examples

```
Link Keys = "Product | NEXT( Time )"
```

Link with the current product, but for the following year / month / day

## See also

Link, Link-Icon, Link-Name, Link-Target

## Link Name

### Type

String

### Since

2.1

### Description

This property sets a name for the link of the block. The link-name will be displayed instead of the link-url.

### Examples

```
Link Name = "' SHOW DETAILS' "
```

Sales	Quantity	Amount
Desktop	259.260	\$24.724.708
Notebook	409.184	1.098.561.524
Pocket PC	203.820	122.257.350

SHOW DETAILS

### See also

Link, Link-Icon, Link-Keys, Link-Target

## Link Target

### Type

String

### Since

2.1

### Description

By default, the target-report of a block-link appears in the same window as the source report. By setting this property, a new window (with the name defined in this property) will be opened, containing the target-report.

### Examples

```
Link Target = "' new_window' "
```

### See also

Link, Link-Icon, Link-Keys, Link-Name

## Padding

### Type

Integer

### Since

2.1

### Description

This property sets the padding (in pixels) of the inner block. The padding is the space between the block-border and the inner content or the title. If no padding is set or the expression returns NULL, the system will use the default-padding.

### Examples

```
Padding = "10"
```

Sets the padding to 10 pixels:

Sales		
	Quantity	Amount
Desktop	258 280	524 724 708
Notebook	409 184	1 098 361 324
Pocket PC	203 920	122 257 350
Tablet PC	179 222	495 949 726

## Position X

### Type

Integer

### Since

2.6.1

### Description

Together with the property "Position Y", you can use this property to give the block a fixed position on the screen. When using a fixed position, the block is no longer arranged in its parents "orientation" and does not influence the sibling blocks when they are arranged.

### Examples

```
Position X = "100"
```

### See also

Position Y

## Position Y

### Type

Integer

### Since

2.6.1

### Description

Together with the property "Position X", you can use this property to give the block a fixed position on the screen. When using a fixed position, the block is no longer arranged in its parents "orientation" and does not influence the sibling blocks when they are arranged.

### Examples

```
Position Y = "100"
```

### See also

Position X

## Orientation

### Type

Constant String ('horizontal', 'vertical', 'tabbed' or 'animated')

### Since

2.0

### Description

The Orientation-property of the inner block defines, how the content of this block will be arranged. There are four different possibilities to arrange the inner blocks:

- **horizontal:** all blocks will be lined up in a row (default setting)
- **vertical:** all blocks will be displayed under the previous one
- **tabbed:** each block becomes a tab in a tabbed display
- **animated:** the blocks will be displayed after each other, only one block will be visible at a given time.

*For animated charts it is better to display as generated images (using the "easychartservlet" format) because the chart-applet would flicker too much when being displayed in an animation.*

### Examples

Orientation = "horizontal"

Orientation = "vertical"

Orientation = "tabbed"

Orientation = "animated"

## Title

### Type

String

### Since

1.1

### Description

With this property you can define a subtitle for an inner block. If a title is defined and the expression doesn't result to NULL, the generated title will be displayed above the table or chart.

### Examples

Title = "' Sales for category ' + TOSTRING( Product )"

The screenshot shows a web interface with a dropdown menu labeled 'Product' set to 'Notebook'. Below it is a table with the following data:

Sales for category Notebook		
	Quantity	Amount
Acer TravelMate 233XV Notebook	51,436	136,610,138
Acer TravelMate 281XV Notebook		
Acer TravelMate TM233XVI Notebook PC		
Acer TravelMate TM234LCI Notebook PC		

### See also

Title Align

## Title Align

### Type

String ('left', 'center' or 'right')

### Since

2.2

## Description

This property sets the text-alignment for the title of the inner block. The expression stored in this property must return one of the following strings:

- **left:** left alignment (default value)
- **center:** center alignment
- **right:** right alignment

## Examples

```
Title Align = "'left' "
```

```
Title Align = "'center' "
```

```
Title Align = "'right' "
```

Sales for category Notebook		
	Quantity	Amount
Acer TravelMate 233XV Notebook	51.436	136.610.138
Acer TravelMate 261XV Notebook		
Acer TravelMate TM233XVI Notebook PC		
Acer TravelMate TM23-IL-CI Notebook PC		

## See also

Title

## Visible

### Type

Boolean

### Since

2.2

## Description

If the expression stored in this optional property result to "false", the block will become invisible inside the result. Note that invisible blocks are not created and consume no CPU time.

## Examples

```
Visible = "true"
```

```
Visible = "$SHOW_DETAILS = 'true' "
```

## See also

Export

## Width

### Type

Integer

### Since

1.1

### Description

This property sets the width (in pixels) of the block. To set the width of a block is necessary for some block-formats (easychart, easychartervlet), all other formats will ignore the width set with this property.

### Examples

```
width = "300"
```

### See also

Format, Height

# Query

---

## Corner Align

### Type

String ('left', 'center' or 'right')

### Since

1.2

### Description

This property sets the text-alignment for the upper left corner of the pivot-table. The expression stored in this property must return one of the following strings:

- **left:** left alignment
- **center:** center alignment
- **right:** right alignment

### Examples

```
Corner-Align = "' left' "
```

```
Corner-Align = "' center' "
```

```
Corner-Align = "' right' "
```

### See also

Corner VAlign

## Corner Background

### Type

String

### Since

1.2

## Description

The "Corner Background" property sets the background-color for the upper left corner of the pivot-table. The expression must result to a string containing the color-name or hex-code (with a leading #).

## Examples

```
Corner-Background = "' red' "
```

```
Corner-Background = "' #FFFF00' "
```

## See also

Corner Foreground

## Corner Bottom Color

### Type

String

### Since

1.2

## Description

This property set the color of the lower border for the table-corner. The expression must result to a string containing the color-name or hex-code (with a leading #).

## Examples

```
Corner-Bottom-Color = "' black' "
```

## See also

Corner Left Color, Corner Right Color, Corner Top Color

## Corner Font

### Type

String

### Since

1.2

## Description

This property sets the font for the table-corner.

## Examples

```
Corner-Font = "' Arial' "
```

## See also

Corner Font Size, Corner Font Weight

## Corner Font Size

### Type

Integer

### Since

1.2

## Description

This property sets the font-size (in pixels) for the table-corner.

## Examples

```
Corner-Font-Size = "10"
```

## See also

Corner Font, Corner Font Weight

## Corner Font Weight

### Type

String

### Since

1.2

## Description

This property sets the font-weight for the table-corner. The expression must have one of the following strings as result:

- **lighter**: Normal font
- **bold**: Bold font

## Examples

```
Corner Font Weight = "' lighter' "
```

```
Corner Font Weight = "' bold' "
```

## See also

Corner Font, Corner Font Size

## Corner Foreground

### Type

String

### Since

1.2

### Description

The "Corner-Foreground" property sets the text-color for the upper left corner of the pivot-table. The expression must result to a string containing the color-name or hex-code (with a leading #).

## Examples

```
Corner Foreground = "' red' "
```

```
Corner Foreground = "' #FFFF00' "
```

## See also

Corner-Background

## Corner Left Color

### Type

String

### Since

1.2

### Description

This property set the color of the left border for the table-corner. The expression must result to a string containing the color-name or hexcode (with a leading #).

## Examples

```
Corner-Left-Color = "' black' "
```

## See also

Corner Bottom Color, Corner Right Color, Corner Top Color

## Corner Right Color

### Type

String

### Since

1.2

### Description

The expression must result to a string containing the color-name or hex-code (with a leading #).

## Examples

```
Corner Right Color = "' black' "
```

## See also

Corner-Bottom-Color, Corner-Left-Color, Corner-Top-Color

## Corner Text

### Type

String

### Since

1.2

### Description

With this property you can display a text in the corner of a query. The result of the text-expression will be display in the upper-left corner of the table. If no corner-text was specified or the expression returns NULL, the corner will display the header-titles (if specified). If no header-title is specified neither, the corner will remain empty.

You also can display images in the corner. Simply return a valid URL endings with ".png", ".jpg" or ".gif". Note the image must be available in the instantOLAP-repository to be visible inside the Workbench-Preview.

## Examples

Corner Text = "NULL"

Empty corner


Corner Text = "' All Amounts in EURO' "

Display the Text "All amounts in EURO" in the corner:

Sales		
All amounts in EURO	Quantity	Amount
Desktop	258.260	524.724.708
Notebook	409.184	1.098.361.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	485.949.726

Corner Text = "' /iolap/logo.gif' "

Display the image logo.gif" in the corner:

Sales		
	Quantity	Amount
 InstantOLAP		
Desktop	258.260	524.724.708
Notebook	409.184	1.098.361.324
Pocket PC	203.920	122.257.350
Tablet PC	179.222	485.949.726

## Corner Top Color

### Type

String

### Since

1.2

### Description

This property sets the color of the upper-border for the table-corner. The expression must result to a string containing the color-name or hex-code (with a leading #).

### Examples

Corner Top Color = "' black' "

### See also

Corner-Bottom-Color, Corner-Left-Color, Corner-Right-Color

## Corner VAlign

### Type

String ('top', 'middle' or 'bottom')

## Since

2.2

## Description

This property sets the vertical text-alignment for the upper left corner of the pivot-table. The expression stored in this property must return one of the following strings:

- **top**: top alignment
- **middle**: middle alignment
- **bottom**: bottom alignment

## Examples

```
Corner-Align = "' top' "
```

```
Corner-Align = "' middle' "
```

```
Corner-Align = "' bottom' "
```

## See also

Corner Align

## Filter

### Type

Key

### Since

1.2

### Description

This property sets the filter for a query. A filter is a set of keys with no, one or more keys per dimension. Whenever later expressions refer to the current selection of a dimension, they will receive the keys set by this or later filters.

Filtering dimensions does not forbid embedded elements to refer other keys than the ones in the filter, it only set the current selection of them. E.g. if a filter "Product:ProductA" was used, the expression "Product" would return "Product:A" but the expression "NEXT( Product )" will still work and return a key (which is not part of the filter).

If you want to reduce dimensions in the way that every expression will only return keys being included in the "filter" and headers have no access to other key, you must use the Subcube property of the query.

## Examples

```
Filter = "Product: ProductA"
```

Sets the selection to the key "Product:A"

## See also

Subcube

## Fixed Columns

### Type

Integer

### Since

2.6.0

### Description

Since version 2.6, tables are exported to Excel with fixed row and column headers. By default, the number of fixed columns is defined by the depth of the x-header of a pivot table, but by setting this property, you can change the number of fixed columns (or disable the feature of by setting it to "0").

Since version 2.6.1, tables can also be displayed with fixed rows or columns in the HTML view. This property also affects the number of fixed columns in this case.

## Examples

```
Fixed Columns = 3
```

Fixes the first 3 columns

```
Fixed Columns = NULL
```

The number of fixed columns is automatically determined

```
Fixed Columns = 0
```

The table has no fixed columns

## See also

Fixed Rows

## Fixed Rows

### Type

Integer

### Since

2.6.0

### Description

Since version 2.6, tables are exported to Excel with fixed row and column headers. By default, the number of fixed rows is defined by the depth of the y-header of a pivot table, but by setting this property, you can change the number of fixed rows (or disable the feature of by setting it to "0").

Since version 2.6.1, tables can also be displayed with fixed rows or columns in the HTML view. This property also affects the number of fixed rows in this case.

### Examples

```
Fixed Rows = 2
```

Fixes the first 2 rows

```
Fixed Rows = NULL
```

The number of fixed rows is automatically determined

```
Fixed Rows = 0
```

The table has no fixed rows

### See also

Fixed Columns

## Highlight Color

### Type

String

### Since

1.2

## Description

In the HTML-view instantOLAP will highlight the row in a pivot-table on which the user holds the mouse on (unless all of the cells of the row have their own background color) by changing its background color.

The default color for the highlighted row is a light yellow, but this can be changed by using this property. Also, by setting the property to "NULL" the highlight will be switched off.

## Examples

```
Highlight Color = "' red' "
```

Changes the highlight-color to red.

```
Highlight Color = "NULL"
```

Switches off the highlight function.

## Span Body

### Type

Boolean

### Since

2.6.0

## Description

By default, the headers of a pivot-table are displayed as single cells and not merged together with other cells like the headers. By setting this property to true, you can group cells with equal content together. This can be very useful whenever you want to display a list report (which only contains body cells) in the same matter as normal pivot tables.

## Examples

```
Span Body = "true"
```

## See also

[Span Headers](#)

## Span Headers

### Type

Boolean

## Since

2.2

## Description

By default, the headers of a pivot-table are clustered whenever the content of two or more headers next to each other is equal. This result into an optical grouping of headers.

By setting this property to "false" you can switch of this behavior and display each header in its own box, regardless of their content and their neighbors content.

## Examples

Span Headers = "true"

Table with spanning headers

Sales		Quantity
Desktop	Apple eMac Desktop 17 M8951LL/A	\$1,028
	Apple iMac Desktop with 17 M8935LL/A	
	Apple Power Mac Desktop M8839LL/A	\$1,748
	Apple Power Mac Desktop M8831LL/A	
	Apple Power Mac Desktop M8832LL/A	
	Apple Power Mac Desktop M9145LL/A	
	Compaq Presario 6480NX Desktop PC	\$1,010
	Compaq Presario S3800NX Desktop PC	

Span Headers = "false"

Table without spanning headers, the category is repeated in every row

Sales		Quantity
Desktop	Apple eMac Desktop 17 M8951LL/A	\$1,028
Desktop	Apple iMac Desktop with 17 M8935LL/A	
Desktop	Apple Power Mac Desktop M8839LL/A	\$1,748
Desktop	Apple Power Mac Desktop M8831LL/A	
Desktop	Apple Power Mac Desktop M8832LL/A	
Desktop	Apple Power Mac Desktop M9145LL/A	
Desktop	Compaq Presario 6480NX Desktop PC	\$1,010
Desktop	Compaq Presario S3800NX Desktop PC	

## See also

Span Body

## Subcube

### Type

Key

### Since

2.0

### Description

The "Subcube" property allows the query to reduce dimensions to a subset of their keys. All embedded elements of the query (e.g. headers) referring to the reduced dimensions will only see the reduced versions of the dimensions and all expressions (e.g. the iteration-expression in the headers) will act as if there were only these keys in the dimensions.

The Subcube property expects a key-expressions. The result of the expression will build the new sub cube - all keys of each dimension are used to build the new dimension. Note that

dimensions will be empty if the expression returns no key for that dimension - also for the fact-dimension!

Sub cubing is good for performance reasons and reducing the complexity of a query, especially when drilldown is used. For example, the subcube expression could search for all keys of a dimension having values for a special fact and then reduce this dimension to these keys. After setting the sub cube, the drilldown inside the query will only show keys with data. This is easier and faster than searching for keys with values inside the drilldown. And, much more important, the sub cube-expression is only executed once per query, whether the drilldown-expression would be executed for each single key.

Don't confuse sub cubing with filtering. The filter (set with the Filter property) only sets the current selections of the filter but still allows the reference of keys not being listed in the filter. Only the sub cube really reduces the dimension.

## Examples

```
Subcube = "LOOKUP( LEVEL( Product, 1, 2 ), Fact:Amount ) | Fact:Amount"
```

This expression builds a subcube with only Products containing data (the fact "Amount") and that fact. If the query uses a header at the Y-axis with an iteration like "LEVEL( Product, 1 )" with drilldown, it now will only show products with data, even when the header is opened for drilldown. Because the fact "Amount" is also included in the expression, there could be a X-header with the iteration "Fact:Amount", showing the fact as header.

## See also

Filter

## Suppress Cols

### Type

Boolean

### Since

1.1

### Description

If the "Suppress Cols" expression returns true, all empty columns will be deleted from the query-result and only columns containing at least one value will be visible. The suppress-feature only pays attentions to the table-cells, not to the headers (which still may contain text when the column is deleted).

Note that event with "Suppress Cols" enabled, the whole query will be executed and generated in memory before the columns will be deleted. To avoid performance-problems, you shouldn't use the "Suppress Cols" or "Suppress Rows" property to reduce sparse filled results. In this case, use the LOOKUP-function for better performance.

## Examples

```
Suppress Cols = "true"
```

## See also

Suppress Rows

# Suppress Rows

## Type

Boolean

## Since

1.1

## Description

If the "Suppress Rows" expression returns true, all empty rows will be deleted from the query-result and only rows containing at least one value will be visible. The suppress-feature only pays attentions to the table-cells, not to the headers (which still may contain text when the column is deleted).

Note that event with "Suppress Rows" enabled, the whole query will be executed and generated in memory before the columns will be deleted. To avoid performance-problems, you shouldn't use the "Suppress-Cols" or "Suppress-Rows" property to reduce sparse filled results. In this case, use the LOOKUP-function for better performance.

## Examples

```
Suppress-Rows = "true"
```

## See also

Suppress Cols

# Header

---

## Align

### Type

String ('left', 'center' or 'right')

### Since

1.2

### Description

This property sets the text-alignment for the table-headers generated by this header-element. This property will not influence the alignment of the cells belonging to the header, use the Cell-Align property instead. The expression must return one of the following strings:

- **left:** left alignment
- **center:** center alignment
- **right:** right alignment

### Examples

```
Align = "' left' "
```

```
Align = "' center' "
```

```
Align = "' right' "
```

### See also

Cell Align

## Assertion

### Type

Double

### Since

1.2

### Description

With the "Assertion" property, you can implement a traffic-light analysis inside your queries. The assertion-property holds a Doubleexpression which result sets the assertion for each individual

cell inside a query (the assertion-expression will be evaluated for each individual cell and might result to different assertions for each cell). Depending on the result a different color is applied to the cell-background:

- **value <= -1**: The assertion generates red cells
- **-1 < value < 0**: The assertion generates yellow cells
- **value = 0**: The assertion generates un-asserted cells
- **0 < value**: The assertion generates green cells
- **value = NULL**: The assertion generates un-asserted cells

The assertion-property is not only used for the visualization of queries but also for some automation-task. E.g. you may want only to send emails with query-result when some exception occurs in you query. Then you can set some assertions in your query and use them in the automation.

## Examples

```
Assertion = "-1"
```

Generates red assertions

```
Assertion = "-0.5"
```

Generates yellow assertions

```
Assertion = "1"
```

Generates green assertions

```
Assertion = "0"
```

Generates no assertion

```
Assertion = "NULL"
```

Generates no assertion

## Background

### Type

String

### Since

1.2

### Description

The "Background" property sets the background-color for the table-headers generated by this header-element. The expression must result to a string containing the color-name or hex-code (with a leading #).

This property will not influence the background-color of the cells belonging to the header, use the "Cell Background" property instead to set their background-color.

## Examples

```
Background = "' red' "
```

Sets the background-color to red

```
Background = "' #FFFF00' "
```

Sets the background-color to yellow

## See also

Cell Background, Foreground

## Bottom Color

### Type

String

### Since

1.2

### Description

This property sets the color of the lower-border for all table-headers generated by this header-element. The expression must result to a string containing the color-name or hex-code (with a leading #). If this property stays unused or the expression results to NULL, no lower-border will be displayed. This property will not influence the border of the cells belonging to the header, use the "Cell-Bottom-Color" property instead to set their border-color.

## Examples

```
Bottom-Color = "' black' "
```

## See also

Cell-Bottom-Color, Left-Color, Right-Color, Top-Color

## Bottom Padding

### Type

String

### Since

2.1

## Description

This property sets the lower padding (in pixels) for all table-headers generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used. This property will not influence the padding of the cells belonging to the header, use the "Cell-Bottom-Padding" property instead to set their border-color.

## Examples

```
Bottom-Padding = "10"
```

## See also

Cell-Bottom-Padding, Left-Padding, Right-Padding, Top-Padding

## Cell Align

### Type

String

### Since

1.2

## Description

This property sets the text-alignment for the table-cells generated by this header-element. The expression must return one of the following strings:

- **left**: left alignment
- **center**: center alignment
- **right**: right alignment

## Examples

```
Cell Align = "' left' "
```

```
Cell Align = "' center' "
```

```
Cell Align = "' right' "
```

## See also

Align, Cell Vertical Align

## Cell Background

### Type

String

### Since

1.2

### Description

The "Background" property sets the background-color for the tablecells generated by this header-element. The expression must result to a string containing the color-name or hex-code (with a leading #).

### Examples

```
Cell Background = "' red' "
```

```
Cell Background = "' #FFFF00' "
```

### See also

Background, Cell Color

## Cell Bottom Color

### Type

String

### Since

1.2

### Description

This property sets the color of the lower-border for all table-cells generated by this header-element. The expression must have a color format as result.

If this property stays unused or the expression results to NULL, no lower border will be displayed.

### Examples

```
Cell Bottom Color = "' black' "
```

### See also

Bottom Color, Cell Left Color, Cell Right Color , Cell Top Color

## Cell Bottom Padding

### Type

Integer

### Since

2.1

### Description

This property sets the lower padding (in pixels) for all table-cells generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result.

If this property is unused or the expression results to NULL, the default padding will be used.

### Examples

```
Cell Bottom Padding = "10"
```

### See also

Bottom Padding, Cell Left Padding, Cell Right Padding, Cell Top Padding

## Cell Color

### Type

String

### Since

1.2

### Description

The "Cell Color" property sets the text-color for the table-cells generated by this header-element. The expression must have a color format as result.

### Examples

```
Cell-Color = "' red' "
```

```
Cell-Color = "' #FFFF00' "
```

### See also

Cell Background, Foreground

## Cell Font

### Type

String

### Since

1.2

### Description

This property sets the font for the table-cells generated by this header-element.

### Examples

```
Cell Font = "' Arial' "
```

### See also

Cell Font Size, Cell Font Weight, Font

## Cell Font Size

### Type

Integer

### Since

1.2

### Description

This property sets the font-size (in pixels) for the table-cells generated by this header-element.

### Examples

```
Cell Font Size = "10"
```

### See also

Cell Font, Cell Font Weight, Font-Size

## Cell Font Weight

### Type

String ('bold' or 'lighter')

## Since

1.2

## Description

This property sets the font-weight for the table-cells generated by this header-element. The expression must return one of the following strings:

- **lighter**: normal
- **bold**: bold

## Examples

```
Cell Font Weight = "' lighter' "
```

```
Cell Font Weight = "' bold' "
```

## See also

Cell Font, Cell Font Size, Font Weight

## Cell Left Color

### Type

String

### Since

1.2

### Description

This property sets the color of the left-border for all table-cells generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no left-border will be displayed.

### Examples

```
Cell Left Color = "' black' "
```

### See also

Cell Bottom Color, Cell Right Color, Cell Top Color, Left Color

## Cell Left Padding

### Type

Integer

### Since

2.1

### Description

This property sets the left padding (in pixels) for all table-cells generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used.

### Examples

```
Cell-Left-Padding = "10"
```

```
Cell-Left-Padding = "DRILLLEVEL() * 10"
```

### See also

Cell Bottom Padding, Cell Right Padding, Cell Top Padding, Left Padding

## Cell Link

### Type

String

### Since

1.1

### Description

This property is used to generate the link for the table-cells generated by this header-element. The generated link must be a valid URL (relative or absolute URLs are both allowed). You also may add some parameters (delimited by ? and &) to the URL.

The link will be automatically extended by all keys in the current filter (which is influenced from both headers at the top and left). If you don't want the system to append all keys from the current filter to the URL you can use the "Cell-Link-Keys" property to define exactly which keys to add.

### Examples

```
Cell Link = "'otherquery.html'"
```

Links to "otherquery", displayed in HTML-format

```
Cell Link = "' otherquery.pdf' "
```

Links to "otherquery", displayed in PDF-format

```
Cell Link = "' otherquery.html?limit=10' "
```

Links to "otherquery", displayed in HTML-format, and set the parameter "limit" to 10

### See also

Cell Link Icon, Cell Link Keys, Cell Link Name, Cell Link Target, Link

## Cell Link Icon

### Type

String

### Since

1.2

### Description

With this property you can specify a link-icon to be displayed inside the table-cells generated by this header-element. If you generate a link-icon, the link will be linked to the target instead of the headertext. The result of this property must be a valid URL for an icon (in JPEG, GIF or PNG format).

### Examples

```
Cell Link Icon = "' /iolap/icons/icon_chart.gif' "
```

### See also

Cell Link, Cell Link Keys, Cell Link Name, Cell Link Target, Link Icon

## Cell Link Keys

### Type

Key

### Since

1.2

### Description

This property controls the generation of cell-links. If no Link-Keys are defined, the link-generator will append all selections of all dimension to the link, so the target-query will exactly have the

same (input) filter as this cell has. For example, a cell showing the fact "Amount" for "Product:A" and "Time:2004" will append this filter to its generated links.

If you don't want the generator exactly to transfer the filter of a cell and control the list of parameter yourself, you can use the "Cell-Link-Keys" property to generate a list of keys, which will be appended as parameters to the generated link. The property expects a Key-Expression and each keys of the expression-result will become one parameter appended to the link. The expression can refer to the current selection, but you also can leave out keys or use more complex expression.

## Examples

```
Cell-Link-Keys = "Product | NEXT( Time )"
```

Link with the current Product, but for the following year / month / day

## See also

Cell Link, Cell Link Icon, Cell Link Name, Cell Link Target, Link Keys

## Cell Link Name

### Type

String

### Since

1.2

### Description

This property sets a name for the links of the cells. The link-name will be displayed as a popup-window whenever the user holds the mouse above the linked cell-text or link-icon (if existing).

## Examples

```
Cell Link Name = "' Show details for ' + {Product}"
```

## See also

Cell Link, Cell Link Icon, Cell Link Keys, Cell Link Target, Link Name

## Cell Link Target

### Type

String

## Since

2.1

## Description

By default, the target of a cell-link appears in the same window as the source report. By setting this property, a new window (with the name defined in this property) will be opened, containing the targetreport or -page.

## Examples

```
Cell Link Target = "'window1'"
```

## See also

Cell Link, Cell Link Icon, Cell Link Keys, Cell Link Name, Link Target

## Cell Right Color

### Type

String

### Since

1.2

### Description

This property sets the color of the right-border for all table-cells generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no right-border will be displayed.

### Examples

```
Cell-Right-Color = "'black'"
```

### See also

Cell Bottom Color, Cell Left Color, Cell Top Color, Right Color

## Cell Right Padding

### Type

Integer

### Since

2.1

## Description

This property sets the right padding (in pixels) for all table-cells generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used.

## Examples

```
Cell-Right-Padding = "10"
```

## See also

Cell Bottom Padding, Cell Left Padding, Cell Top Padding, Right Padding

## Cell Top Color

### Type

String

### Since

1.2

## Description

This property sets the color of the upper-border for all table-cells generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no upper-border will be displayed.

## Examples

```
Cell-Top-Color = "' black' "
```

## See also

Cell Bottom Color, Cell Left Color, Cell Right Color, Top Color

## Cell Top Padding

### Type

Integer

### Since

2.1

## Description

This property sets the upper padding (in pixels) for all table-cells generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used.

## Examples

```
Cell-Top-Padding = "10"
```

## See also

Cell Bottom Padding, Cell Left Padding, Cell Right Padding, Top Padding

## Cell Vertical Align

### Type

String

### Since

2.1

## Description

This property determines the vertical alignment for the table-cells generated by this header-element. The expression must return one of the following strings:

- **top:** top-alignment
- **middle:** middle-alignment
- **bottom:** bottom-alignment

## Examples

```
Cell Vertical Align = "' top' "
```

```
Cell Vertical Align = "' middle' "
```

```
Cell Vertical Align = "' bottom' "
```

## See also

Cell Align, Vertical Align

## Drilldown

### Type

Boolean

### Since

1.0

### Description

This property enables drilldown for this header (if the Boolean-Expression in the property return "true").

By default, the drilldown will show all children of the current key. You also can influence the shown keys with the "Drilldown-Iteration" property.

There are also two different-styles of drilldown (vertical and encapsulated). Use the Drilldown-Encapsulate property to control the style of the drilldown.

### Examples

```
Drilldown = "true"
```

Enables drilldown for a header:

Sales		
	Quantity	Amount
Desktop	258.260	524.724.708
▼ Notebook	409.104	1.080.361.524
Acer TravelMate 233XV Notebook	51.436	136.610.138
Acer TravelMate 281XV Notebook		
Acer TravelMate TM233XVI Notebook PC		
Acer TravelMate TM233LCI Notebook PC		
Acer TravelMate TM883LCI Notebook		
Apple iBook Notebook 141 M9969LL/A	50.802	167.316.766
Apple PowerBook Notebook 121 M9892LL/A		
Apple PowerBook Notebook 121 Z876		
Apple Powerbook Notebook 152 M8859LL/A		

### See also

Drilldown Encapsulate, Drilldown Iteration, Drilldown Prefetch

## Drilldown Encapsulate

### Type

Boolean

### Since

1.2

### Description

Whenever you enable drilldown for a header with the Drilldown property, the user may unfold or fold the header's elements to view the children of a keys with its data. When unfolding a key, the

children are by default display under (or right of, if the header is located at the X-axis) their parents.

You can use an alternative display-style for headers when setting the "Drilldown-Encapsulate" property to "true". In this case the children-header becomes an encapsulated header of the parent-header. Encapsulating headers for drilldown gives the user a better overview (because the hierarchy of the keys is displayed in a better way) but uses more space. Also, the parent itself has no own data-row (or column) in the encapsulated display-style.

### Examples

Drilldown-Encapsulate = "false"

Drilldown-Encapsulate = "true"

Sales		
	Quantity	Amount
Desktop	258 280	524 724 708
▼ Notebook	51 436	136 610 138
Acer TravelMate 233XV Notebook		
Acer TravelMate 281XV Notebook		
Acer TravelMate TM233XVI Notebook PC		
Acer TravelMate TM234LCI Notebook PC		
Acer TravelMate TM88DL-CI Notebook		
Apple Book Notebook 141 M909LL/A	50 802	187 316 766
Apple PowerBook Notebook 121 M9892LL/A		
Apple PowerBook Notebook 121 Z07B		
Apple Powerbook Notebook 152 M8859LL/A		

### See also

Drilldown, Drilldown Iteration, Drilldown Prefetch

## Drilldown Iteration

### Type

Key

### Since

1.2

### Description

The Drilldown-Iteration allows to control, which keys are shown when the user drills down inside a header. By default a (drilldown-enabled) header shows the children of its key when opened. For the most time this would be exactly what the user want, but sometimes a more complex behaviour is wanted. E.g. a drilldown could open keys from other dimensions (like all Customers of a product) or reduce the number of children to the keys having values.

The Drilldown-Iteration is a simple Key-Expression. Each key of the expression-result will become one new header in the drilldown. Note that the same expression is used for the new generated headers as drilldown-iteration.

### Examples

Drilldown-Iteration = "Product.Customers"

## See also

Drilldown, Drilldown Encapsulate, Drilldown Prefetch

## Drilldown Prefetch

### Type

Boolean

### Since

1.2

### Description

All drilldown-enabled headers will have an drilldown-icon in front of their text only if there is anything to drilldown. To find out, if there is anything to drilldown, the engine will count the number of children for each row (or execute the expression in the Drilldown-Iteration property).

Especially when using the Drilldown-Iteration, this can be a timeconsuming task, e.g. when the Drilldown-Iteration is performing some database-query. In this case you can disable the prefetch-count with this property (set it to false). Then all headers will show a drilldown-icon, nevertheless if they will contain sub headers or not when opened.

### Examples

```
Drilldown Prefetch = "false"
```

## See also

Drilldown, Drilldown Encapsulate, Drilldown Iteration

## Filter

### Type

Key

### Since

1.2

### Description

This optional property applies the result of its key-expression to the filter (selection) of this header. This is done before any other property of the header is evaluated, so the filter will affect all text-, value and styling-settings of the header and its cells.

The property expects a Key-Expression and the result of the Key- Expression will be applied to the filter. Applying means, for all dimensions contained in the result the dimension in the filter will be changed to the keys of this dimension. All other dimensions will stay unaffected.

## Examples

```
Filter = "Product:A"
```

Apply "Product:A" to the header-filter

## See also

Group By, Iteration

## Font

### Type

String

### Since

1.2

### Description

This property determines the font for the table-headers generated by this header-element. This property will not influence the font of the cells belonging to the header, use the Cell-Font property instead to set their font.

### Examples

```
Font = "'Arial'"
```

### See also

Cell Font, Font Size, Font Weight

## Font Size

### Type

Integer

### Since

1.2

### Description

This property sets the font-size (in pixels) for the table-headers generated by this header-element. This property will not influence the font-size of the cells belonging to the header, use the "Cell-Font-Size" property instead to set their font-size.

## Examples

```
Font Size = "10"
```

## See also

Cell Font Size, Font, Font Weight

## Font Weight

### Type

String ('bold' or 'lighter')

### Since

1.2

### Description

This property sets the font-weight for the table-headers generated by this header-element. The expression must return one of the following strings:

- **lighter**: normal
- **bold**: bold

This property will not influence the font-weight of the cells belonging to the header, use the Cell-Font-Weight property instead to set their font-weight.

### Examples

```
Font-Weight = "' lighter' "
```

```
Font-Weight = "' bold' "
```

### See also

Cell Font Weight, Font, Font Size

## Foreground

### Type

String

### Since

1.2

## Description

The "Foreground" property sets the text-color for the table-headers generated by this header-element. The expression must have a colorformat as result. This property will not influence the text-color of the cells belonging to the header, use the "Cell-Foreground" property instead to set their text-color.

## Examples

```
Foreground = "' red' "
```

```
Foreground = "' #FFFF00' "
```

## See also

Background, Cell Foreground

## Format

### Type

String

### Since

1.1

## Description

The "Format" property allows to set the number- or date-format for the cells generated by this header. Depending on the cell's data-type, one of both formats is expected. This property expects a String-Expression.

If no format is defined inside the header, the system will use the facts default format for the cells (if possible). If no fact-format is existing or the system can't figure out which fact-format to use (e.g. because a Formula was used to generate the cell's content), the default-formatting for the cell will be used.

*Note that you also can display number as date or time by setting a date-/time-format with this properties. In this case the number is interpreted as milliseconds and you may need to transform the number before.*

## Examples

```
Format = "' #, ###, ##0.00' "
```

Use number-format for the cells

```
Format = "' HH: mm: ss' "
```

Use date/time-format for the cells

## See also

Formula

## Formula

### Type

Value

### Since

1.1

### Description

There are two different ways of defining which values a table will display inside the cells: Using facts as headers-iterations or using formulas.

- Using facts in headers is the common way: By using a single fact as the iteration of a header in the X- or Y-axis, this will set the filter to this fact for the whole row or column of a table. This is normally done by dragging a fact onto a header with the visual query-editor of the instantOLAP Workbench. For each cell having no formula the system will check if there is a fact in the filter and use this fact as the standard-formula for the cell.
- Defining formulas is only needed when more a complex calculation inside a cell is needed (e.g for aggregating many values or if no fact contains the needed value) or if you don't want the fact to appear in the header.

To define a formula for a header insert a Value-Expression into the "Formula" property. The Value-Expression is executed for each single cell generated by the header. The expression can refer to all facts and dimensions.

### Examples

```
Formula = "Amount() * 2"
```

Calculates and displays the double amount

## See also

Format

## Group By

### Type

Value

### Since

2.2.6

## Description

The "Group By" property allows to group the keys returned by the "Iteration" of the header by a free definable value-expression. All keys with the same result for this expression will be grouped to a single header and this header will have all grouped keys in its filter.

By using an aggregated expression (like SUM(...)) in the formula of the header you can display aggregated facts or other expressions for a grouped row or column.

If headers are grouped by an expression, there will be (in difference to non- grouped heades) a default-text for the header because more than one keys could be in a group and displaying this keys generally would make no sense. Therefore the result of the group expression (which is logically the same for all keys in this header) will be displayed as the default header text.

## Examples

```
Group By = "Product.CustomerName"
```

Groups the headers by the attribute "CustomerName" of the Product dimension.

## See also

Filter, Iteration

## Height

### Type

Integer

### Since

1.2

## Description

The "Height" property sets the height of the table-headers and -cells (in pixels) generated by this header-element. If no height is set, it will be evaluated automatically to the best-fitting value when the result is displayed inside the browser.

## Examples

```
Height = "50"
```

## See also

Width

## Input

### Type

Boolean

### Since

1.2

### Description

Use this property to enable table-cells for input. It expects a Boolean-Expression which will be executed for each generated cell of this header. Each cell the expression returns "true" for (and the user has sufficient rights to edit the facts) will become edit able.

When using cell to input data, you should not use formulas for that cells, only iterations. Whenever you use a formula, the system wouldn't be able to backtrace the input-value to a single fact (which is possible if the fact was defined by one of the cell-headers iterations).

Note that only making table-cells edit able doesn't guarantee the values will be stored in any database. You also must enable to corresponding cube to store values back to their datasource. See the "Enable Store" property of SQL-cubes for more information.

### Examples

```
Edit = "true"
```

Makes the whole row/column editable

```
Edit = "X() = 5"
```

Only makes column 5 editable

## Iteration

### Type

Key

### Since

1.0

### Description

This optional property allows to iterate (repeat) a header by a number of keys. Iterating headers is used for displaying a larger number of rows or cells (e.g. each product) without hard coding the single headers. Whenever the dimensions of a model change, the queries using iterated header instead of hard coded ones change automatically, too.

The iteration must be defined in a key-expression which returns a number of keys (dimension-elements). Each returned key will generate exactly one copy of the header. The filter of the headercopy will be generated by copying the global filter of the containing block and

applying the key to it. Because of the different filters, each header will show different values (e.g. a different product) and each header-property referring the iterated dimension (e.g the "Text" property) will result to different values.

If the iteration of a key returns NULL, the header disappears. If no iteration is set, the header will generate exactly one row or column without changing any filter.

## Examples

```
Iteration = "LEVEL( Product, 1 )"
```

Show one header of each product

```
Iteration = "Fact: Amount"
```

Show a single header displaying the fact "Amount"

## See also

Filter, Group By

## Left Color

### Type

String

### Since

1.2

### Description

This property sets the color of the left-border for all table-headers generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no left-border will be displayed. This property will not influence the border of the cells belonging to the header, use the "Cell-Left-Color" property instead to set their border-color.

### Examples

```
Left Color = "' black' "
```

### See also

Bottom Color, Cell Left Color, Right Color, Top Color

## Left Padding

### Type

Integer

## Since

2.1

## Description

This property sets the left padding (in pixels) for all table-headers generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used. This property will not influence the padding of the cells belonging to the header, use the Cell-Bottom-Padding property instead to set their border-color.

## Examples

```
Left-Padding = "10"
```

## See also

Bottom Padding, Cell Left Padding, Right Padding, Top Padding

## Link

### Type

String

### Since

1.1

### Description

The result of this property defines the link for this header-element. The generated link must be a valid URL (relative or absolute URLs are both allowed). You also may add some parameters (delimited by ? and &) to the URL.

The link will be automatically extended by all keys in the current filter. If you don't want the system to append all keys from the current filter to the URL you can use the Link-Keys property to define exactly which keys to add.

This property doesn't generate links for the table-cells generated by this header. To add links to cells you must use the Cell-Link property.

### Examples

```
Link = "' otherquery.html' "
```

Links to "otherquery", displayed in HTML-format

```
Link = "' otherquery.pdf' "
```

Links to "otherquery", displayed in PDF-format

```
Link = "'otherquery.html?limit=10' "
```

Links to "otherquery", displayed in HTML-format, and set the parameter "limit" to 10

### See also

Cell Link, Link Keys, Link Name, Link Target

## Link Icon

### Type

String

### Since

1.2

### Description

With this property you can specify a link-icon to be displayed inside the table-header generated by this header-element. If you generate a link-icon, the link will be linked to the target instead of the headertext. The result of this property must be a valid URL for an icon (in JPEG, GIF or PNG format).

This property doesn't generate link-icons for the table-cells generated by this header. To add icons to cells you must use the Cell-Link-Icon property.

### Examples

```
Link-Icon = "'/iolap/icons/icon_chart.gif' "
```

### See also

Cell Link Icon, Link, Link Keys, Link Name, Link Target

## Link Keys

### Type

Key

### Since

1.2

### Description

This property controls the generation of header-links. If no Link-Keys are defined, the link-generator will append all selections of all dimension to the link, so the target-query will exactly have the same (input) filter as this cell has. For example, a header showing the "Product:A" at "Time:2004" will append this filter to its generated links.

If you don't want the generator exactly to transfer the filter of a cell and control the list of parameters yourself, you can use the Link-Keys properties to generate a list of keys, which will be appended as parameters to the generated link. The property expects a Key-Expression and each key of the expression-result will become one parameter appended to the link. The expression can refer to the current selection, but you also can leave out keys or use more complex expression.

## Examples

```
Link Keys = "Product | NEXT( Time )"
```

Link with the current Product, but for the following year / month / day

## See also

Cell Link Keys, Link, Link Icon, Link Name, Link Target

## Link Name

### Type

String

### Since

1.2

### Description

This property sets a name for the links of the header. The link-name will be displayed as a popup-window whenever the user holds the mouse above the linked text or icon (if existing).

## Examples

```
Link Name = "'Show details for ' + {Product}'"
```

## See also

Cell Link Name, Link, Link Icon, Link Keys, Link Target

## Link Target

### Type

String

### Since

2.1

## Description

By default, the target of a header-link appears in the same window as the source report. By setting this property, a new window (with the name defined in this property) will be opened, containing the targetreport or -page.

## Examples

```
Link Target = "' window1' "
```

## See also

Cell Link Target, Link, Link Icon, Link Keys, Link Name

## Right Color

### Type

String

### Since

1.2

## Description

This property sets the color of the right-border for all table-headers generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no right-border will be displayed. This property will not influence the border of the cells belonging to the header, use the Cell-Right-Color property instead to set their border-color.

## Examples

```
Right-Color = "' black' "
```

## Right Padding

### Type

Integer

### Since

2.1

## Description

This property sets the right padding (in pixels) for all table-headers generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used. This property will not influence the padding of

the cells belonging to the header, use the Cell-Bottom-Padding property instead to set their border-color.

## Examples

```
Right-Padding = "10"
```

## See also

Bottom Padding, Cell Right Padding, Left Padding, Top Padding

## Rotate

### Type

Boolean

### Since

1.2

### Description

If the expression in this property results to "true", all header-texts generated by this header will be rotated by 90 degrees.

### Examples

```
Rotate = "true"
```

## Sort

### Type

Value

### Since

2.1

### Description

The "Sort" property allows to sort the table-headers generated by this header by a freely definable expression. The expression will be evaluated for each key from the iteration and the headers will be sorted by their expression-result. If two or more headers result to the same value, the original order of these headers will be kept.

By default, the headers are sorted ascending (from the smallest to the biggest sort-value). Use the "Sort Descending" property to sort them descending.

## Examples

```
Sort = "Amount()"
```

## See also

Sort Descending

## Sort Descending

### Type

Boolean

### Since

2.1

### Description

By default, headers sorted with the Sort property are sorted ascending (from the smallest to the biggest sort-value). Set this property to "true" (to an expression resulting to true) sort the headers descending.

### Examples

```
Sort-Descending = "true"
```

Sorts the generated headers descending

```
Sort-Descending = "false"
```

Sorts the generated headers ascending

### See also

Sort

## Text

### Type

String

### Since

1.1

### Description

The String-Expression in this property defines the text which will be displayed for each text-header generated by this header-element. If no text-expression is defined, the header-cell

will display the default text-attribute of the iterated keys (or their ID if they don't own any default-text). If no "Iteration" and no "Text" expression was defined, the headers will stay empty.

## Examples

```
Text = "TOSTRING( Product )"
```

```
Text = "LIMIT( TOSTRING( Product ), 50 )"
```

```
Text = "Product.ShortText"
```

## See also

Iteration, Title

## Title

### Type

String

### Since

1.2

### Description

This property sets the title of a header. The title of a header is shown in the upper left corner (in the same row or column as the header). If no title is defined for a header, the corner stays empty (or display its own text).

## Examples

```
Title = "' Product' "
```

## See also

Text

## Top Color

### Type

String

### Since

1.2

## Description

This property sets the color of the upper-border for all table-headers generated by this header-element. The expression must have a color format as result. If this property stays unused or the expression results to NULL, no upper-border will be displayed. This property will not influence the border of the cells belonging to the header, use the Cell-Top-Color property instead to set their border-color.

## Examples

```
Top Color = '' black' "
```

## See also

Bottom Color, Cell Top Color, Left Color, Right Color

## Top Padding

### Type

Integer

### Since

2.1

## Description

This property sets the upper padding (in pixels) for all table-headers generated by this header-element. The padding is the inner space between the cell-text and the cell-border. The expression must have an integer-number as result. If this property is unused or the expression results to NULL, the default padding will be used. This property will not influence the padding of the cells belonging to the header, use the "Cell-Bottom-Padding" property instead to set their border-color.

## Examples

```
Top Padding = "10"
```

## See also

Bottom Padding, Cell Top-Padding, Left Padding, Right Padding

## Vertical Align

### Type

String

### Since

2.1

## Description

This property sets the vertical text-alignment for the table-headers generated by this header-element. This property will not influence the alignment of the cells belonging to the header, use the Cell-Align property instead. The expression must return one of the following strings:

- **top**: top-alignment
- **middle**: middle-alignment
- **bottom**: bottom-alignment

## Examples

```
Vertical Align = "'top' "
```

```
Vertical Align = "'middle' "
```

```
Vertical Align = "'bottom' "
```

## See also

Align, Cell Vertical Align

## Visible

### Type

Boolean

### Since

1.2

## Description

This property determines whether the table-headers and -cells generated by this header-element will be visible. If the expression results to false, the cell will become invisible. If all cells of a row or column inside a table are invisible, the whole row or column will vanish.

This property is used both for table-headers and -cells. You can use the X and Y function to set different visibilities for them.

## Examples

```
Visible = "false"
```

Hide whole row or column

```
Visible = "X() > 1"
```

Hide first columns

## Width

### Type

Integer

### Since

1.2

### Description

The "Width" property sets the width of the table-headers and -cells (in pixels) generated by this header-element. If no width is set, it will be evaluated automatically to the best-fitting value when the result is displayed inside the browser.

### Examples

```
Width = "200"
```

### See also

Height

## Z-Order

### Type

Integer

### Since

2.0

### Description

Many properties of a header influence the cell in the headers row or column. Each cell may be influenced by two different headers: The header in the X-axis at the same column and the header in the Y-axis at the same row. If both header set the same cell-property (e.g. "Cell Background" or "Formula") to different values, the header with the higher Z-Order will gain control over the cell and the other header's settings will be ignored.

The default "Z-Order" for a header is "0". Both positive and negative values are allowed.

### Examples

```
Z-Order = "1"
```

Higher order than default

# Comment

---

## Author

### Type

String

### Since

1.1

### Description

This optional property contains the name of the comment's author. Note that this property expects a simple text and no expression!

### Examples

```
Author = "admin"
```

### See also

Date

## Copy To Result

### Type

Boolean

### Since

1.1

### Description

This optional property contains the creation-date of the comment. There is no specific format for the date, you can use any format you want.

### Examples

```
Copy To Result = "true"
```

```
Copy To Result = "false"
```

**See also**

Export

**Date****Type**

String

**Since**

1.1

**Description**

This optional property contains the creation-date of the comment. There is no specific format for the date, you can use any format you want.

**Examples**

```
Date = "01.01.2004"
```

**See also**

Author

**Export****Type**

Boolean

**Since**

1.2

**Description**

If this property contains "true", the comment will be copied to the result whenever the user exports a comment (to PDF or Excel). Otherwise, this comment will disappear.

**Examples**

```
Export = "true"
```

```
Export = "false"
```

**See also**

Copy To Result, Visible

## Formula

### Type

String

### Since

1.2

### Description

There are two different ways to store the content of a comment: As a static text within the "Text" property or as a expression in this property. If you use his property, you can calculate the comment-text with a String-expression. If this property is used, the property "Text" will be ignored.

### Examples

```
Formula = "' Hello ' + $USER"
```

### See also

Text

## Padding

### Type

Integer

### Since

2.2.2

### Description

The "Padding" property defines the left padding of the comment. You can use this property the increase the distance to the left border of the comment.

In combination with the "Text Indent" property you can also shift the comment from the second line.

### Examples

```
Padding = "50"
```

### See also

Text Indent

## Text

### Type

String

### Since

1.1

### Description

This property holds the constant text of the comment. If you want your comment to be generated by an expression, leave this property empty and use the "Formula" property instead.

### Examples

```
Text = "This is an comment"
```

### See also

Formula

## Text Indent

### Type

Integer

### Since

2.2.2

### Description

The "Text Indent" property defines the text indent for the first line of the comment. This can be a negative or positive value. Whenever you use negative values you should also change the "Padding" - otherwise the text will leave its containing frame on the display.

### Examples

```
Text Indent = "-50"
```

### See also

Padding

## Visible

### Type

Boolean

### Since

2.2.4

### Description

If the expression stored in this optional property result to "false", the comment will become invisible inside the result.

### Examples

```
Visible = "true"
```

```
Visible = "$SHOW_COMMENTS = 'true' "
```

### See also

Export



# CHAPTER 2:

## Chart properties

### Contents of this chapter:

Line-Chart properties .....	120
Bar-Chart properties .....	163
Pie-Chart properties .....	207

# Line-Chart properties

---

## 3D Depth

**Type**

Integer

**Since**

1.2

**Description**

Sets the depth of the 3D effect in pixels.

**Examples**

```
3D Depth = "10"
```

**See also**

3D Mode On

## 3D Mode On

**Type**

Boolean

**Since**

1.2

**Description**

Turns the 3D-Mode on or off.

**Examples**

```
3D Mode On = "true"
```

```
3D Mode On = "false"
```

**See also**

3D Depth

## Area\_0

### Type

String

### Since

2.2.4

### Description

Fills the area between two lines with the specified color. This property expects a string as value which contains the index of the two series and the color separated by commas.

### Examples

```
Area_0 = "'0,1,yellow' "
```

## Auto Label Spacing On

### Type

Boolean

### Since

1.2

### Description

By default all the sample labels are painted, even if there is not enough space for them all (they will overlap). If this parameter is set only labels there are room for will be painted.

### Examples

```
Auto Label Spacing On = "true"
```

```
Auto Label Spacing On = "false"
```

## Chart Background

### Type

String

### Since

1.2

## Description

This is the color of the chart grid background. The expression must have a color format as result.

## Examples

```
Chart Background = "' blue' "
```

```
Chart Background = "' #FFFF00' "
```

## Chart Foreground

### Type

String

### Since

1.2

## Description

This is the color of the chart grid outline. The expression must have a color format as result.

## Examples

```
Chart Foreground = "' black' "
```

```
Chart Foreground = "' #FFFF00' "
```

## Chart Title

### Type

String

### Since

1.2

## Description

The result of this property becomes the title of the chart.

## Examples

```
Chart Title = "' Turnover' "
```

## Connected Lines On

### Type

Boolean

### Since

1.2

### Description

By default, a line with undefined values will have gaps in the line. By using this parameter you can bridge these gaps by painting a line to the next defined value in the series. Undefined values does not work with stacked lines.

### Examples

```
Connected Lines On = "true"
```

```
ConnectedLinesOn = "false"
```

## Default Grid Lines Color

### Type

String

### Since

1.2

### Description

Sets the color of default grid lines.

### Examples

```
Default Grid Lines Color = "' black' "
```

```
Default Grid Lines Color = "' #FFFF00' "
```

## Default Grid Lines On

### Type

Boolean

**Since**

1.2

**Description**

Turns on the vertical grid lines.

**Examples**

```
Default Grid Lines On = "true"
```

```
Default Grid Lines On = "false"
```

**Floating Label Font****Type**

String

**Since**

1.2

**Description**

Sets the font used for the floating value and sample labels.

**Examples**

```
Floating Label Font = "' Arial' "
```

**Floating On Legend Off****Type**

Boolean

**Since**

1.2

**Description**

By default floating labels are displayed for all the samples in the selected series when the mouse moves over a series label in the legend. This parameter turns this behavior off.

**Examples**

```
Floating On Legend Off = "true"
```

```
FloatingOnLegendOff = "false"
```

## Font

### Type

String

### Since

1.2

### Description

Sets the default font for the chart labels.

### Examples

```
Font = "' Arial' "
```

## Foreground

### Type

String

### Since

1.2

### Description

This is the color of the title, legend labels, value labels, sample labels, and the range labels. The expression must have a color format as result.

### Examples

```
Foreground = "' blue' "
```

```
Foreground = "' #ffff00' "
```

## Graph Insets

### Type

String

### Since

1.2

## Description

Use this parameter to add space between the chart grid and the chart component edges. The parameter value consists of 4 parameters with the following sequence: top, left, bottom, right. A value of -1 uses the default inset. The expression must result to a string containing all these 4 values separated by commas.

## Examples

```
Graph Insets = "'-1, 50, -1, -1'"
```

## Grid Adjustment On

### Type

Boolean

### Since

1.2

### Description

You can adjust the chart grid at runtime by selecting a grid edge and dragging the mouse button. Double-click on the grid edge to set it to the default position.

### Examples

```
Grid Adjustment On = "true"
```

```
Grid Adjustment On = "false"
```

## Grid Image

### Type

String

### Since

2.1

### Description

You can set an image to be used as a background for the grid.

### Examples

```
Grid Image = "'/iolap/images/chartbg.gif'"
```

```
Grid Image = "'http://myserver/images/chartbg.gif'"
```

## Grid Line Colors

### Type

String

### Since

1.2

### Description

Sets the color of individual vertical grid lines.

### Examples

```
Grid Lines Colors = "' black' "
```

```
Grid Lines Colors = "' #FFFF00' "
```

## Grid Lines

### Type

String

### Since

1.2

### Description

Sets the vertical grid lines. Add one value per grid line. The value is relative to the sample axis range (default 0 to 100) and the width of the chart grid. All values must be passed in a single string, containing the values separated by comma.

### Examples

```
Grid Lines = "' 10, 20, 30, 40, 50, 60, 70, 80, 90' "
```

## Grid Lines Color

### Type

String

### Since

1.2

## Description

Sets the color of all except default grid lines. The expression must have a color format as result.

## Examples

```
Grid Lines Color = "' black' "
```

```
Grid Lines Color = "' #FFFF00' "
```

## Label\_0

### Type

String

### Since

2.1

## Description

This parameter can be used to set a label to any point of the chart. First parameter is label text. Second and third parameters are X and Y coordinates of the label. If X and Y are higher than 0 and lower than 1, the label position is calculated relatively to the chart bounds. Otherwise, the absolute coordinates of the chart are used. There are optional 4th and 5th parameters which are index and series of the sample, pointed by the label.

## Examples

```
Label = "' orange sales,100,100' "
```

```
Label = "' apple sales,0.4,0.5' "
```

```
Label = "' banana sales,200,200,4,0' "
```

## See also

Label Url\_0, Label URL Target\_0

## Label Url\_0

### Type

String

### Since

2.1

## Description

This parameter can be used to assign an URL to an anywhere label. The URL address will be opened when user clicks the label.

## Examples

```
Label URL_0 = "' details.html' "
```

## See also

Label\_0, Label URL Target\_0

## Label URL Target\_0

### Type

String

### Since

2.1

## Description

This parameter controls where a HTML page will be opened when clicked the label.

## Examples

```
Label URL Target_0 = "' _blank' "
```

## See also

Label\_0, LabelURLTarget\_0

## Legend Colors

### Type

String

### Since

1.2

## Description

Sets the colors for the default legend boxes. If this is not set, the colors are taken from the sampleColors parameter. The expression must contain a list of colors in the color format, separated by commas.

## Examples

```
Legend Colors = "' red, green, blue' "
```

## Legend Columns

### Type

Integer

### Since

2.1

### Description

Sets the number of columns that should be used to display legend labels.

### Examples

```
Legend Columns = "2"
```

## Legend Font

### Type

String

### Since

1.2

### Description

Sets the font for the labels in the legend.

### Examples

```
Legend Font = "' Arial' "
```

## Legend Image

### Type

String

### Since

1.2

## Description

Sets an image to be used in front of the legend label instead of the default legend box.

## Examples

```
Legend Image = "/iolap/images/legend.gif"
```

## Legend Labels

### Type

String

### Since

2.2

## Description

This property allows to replace the labels in the legend with you own custom labels. All labels have to be provided in a single string and must be separated with commas.

## Examples

```
Legend Labels = "A, B"
```

## Legend On

### Type

Boolean

### Since

1.2

## Description

Turns on the legend or off.

## Examples

```
Legend On = "true"
```

```
Legend On = "false"
```

## Legend Position

### Type

String ('right', 'left', 'top' or 'bottom')

### Since

1.2

### Description

This property defines the position of the legend inside the chart.

### Examples

```
Legend Position = "right"
```

```
Legend Position = "left"
```

```
Legend Position = "top"
```

```
Legend Position = "bottom"
```

## Legend Reverse On

### Type

Boolean

### Since

2.1

### Description

Set on or off the inverted legend. Default legend entry order is from top to bottom and from left to right.

### Examples

```
Legend Reverse On = "true"
```

```
Legend Reverse On = "false"
```

## Line Stroke

### Type

String

## Since

2.1

## Description

Used to display dashed and dotted lines. Sets an array of values to use as repeating dash lengths and clear sections. Use only one value if all the dashes and clear sections have the same length. For instance, `lineStroke = 3` produces a repeating dash of 3 pixel length followed by space of same length, while `lineStroke = 3|6` produces dash of 3 pixel length followed by 6 empty pixels. The length of the stroke will cycle through the array if more drawing space is available. You can also set different strokes for different series separating arrays with commas.

## Examples

```
Line Stroke = "' 3' "
```

```
Line Stroke = "' 3| 6' "
```

```
Line Stroke = "' 3| 6, 2| 4' "
```

## Line Width

### Type

String

### Since

1.2

### Description

Sets the width of each series line. You set the width of each series by using a comma separated list of values. If you have more than one series, you can set the width of all the series by using 1 value. Default width is 2.

### Examples

```
Line Width = "' 1' "
```

```
Line Width = "' 1, 2' "
```

## Locale

### Type

String

### Since

2.2.1

## Description

Sets the locale of the chart. The locale affects the display of value and range labels, with the correct grouping and decimal signs.

## Examples

```
Locale = "' en, GB' "
```

## Lower Range

### Type

Number

### Since

1.2

## Description

Sets the lower value of the range axis (y-axis). If this is not set, the lower range will be automatically set to the smallest sample (or 0 if no samples are negative).

## Examples

```
Lower Range = "40"
```

## Max Value Line Count

### Type

Integer

### Since

1.2

## Description

Sets the maximum number of value lines to be displayed in the chart. You can use this to control the step between the grid labels to certain extent.

## Examples

```
Max Value Line Count = "5"
```

## Range

### Type

Number

### Since

1.2

### Description

Sets the upper value of the range axis (y-axis). If this is not set, the upper range will be automatically set to the largest sample.

### Examples

```
Range = "100"
```

## Range 2

### Type

Number

### Since

2.6.1

### Description

Sets the upper value of the second range in an overlay chart. For a detailed description of this property, refer to the "Range" property.

### Examples

```
Range = "100"
```

### See other

Range

## Range Adjusted

### Type

Integer

## Since

1.2

## Description

Controls which range(s) are adjusted with which adjuster. By default adjuster 1 controls range 1 and adjuster 2 controls range 2.

## Examples

```
Range Adjusted = "1"
```

```
Range Adjusted = "2"
```

## Range 2 Adjusted

### Type

Integer

### Since

2.6.1

### Description

Controls which range(s) are adjusted with which adjuster in overlay charts. For a detailed description of this property, refer to the "Range adjusted" property.

### See also

Range Adjusted

## Range Adjuster On

### Type

Boolean

### Since

1.2

### Description

Turns on the range adjusters. The range adjuster allows the user to adjust the upper and lower ranges at runtime. Both range adjusters default to the right side of the chart. Use the rangeAdjusterPosition parameter to control the position of the adjusters.

## Examples

```
Range Adjuster On = "true"
```

```
Range Adjuster On = "false"
```

## Range 2 Adjuster On

### Type

Boolean

### Since

2.6.1

### Description

Turns on the range adjusters for overlay charts. For a detailed description of this property refer to the "RangeAdjusterOn" property.

### See also

Range Adjuster On

## Range Adjuster Position

### Type

Integer

### Since

1.2

### Description

Sets the position of the range adjusters. 0 means the left side, 1 the right side. The default is the right side.

### Examples

```
Range Adjuster Position = "1"
```

## Range 2 Adjuster Position

### Type

Integer

## Since

2.6.1

## Description

Sets the position of the range adjusters in overlay charts. For a detailed description of this property, refer to the "RangeAdjusterPosition" property.

## See also

Range Adjuster Position

## Range Axis Label

### Type

String

### Since

1.2

### Description

Adds a label to the range axis. The label is horizontal by default, but the angle can be controlled using the RangeAxisLabelAngle property.

### Examples

```
Range Axis Label = "' Amount' "
```

## Range 2 Axis Label

### Type

String

### Since

2.6.1

### Description

Adds a label to the range axis of the overlay chart. For a detailed description of this property, refer to the "RangeAxisLabel" property.

### See also

Range Axis Label

## Range Axis Label Angle

### Type

Integer

### Since

1.2

### Description

Sets the clockwise angle of the range axis labels.

### Examples

```
Range Axis Label Angle = "270"
```

## Range 2 Axis Label Angle

### Type

Integer

### Since

2.6.1

### Description

Sets the clockwise angle of the range axis labels for the overlay chart.

### See also

```
Range Axis Label Angle = "270"
```

## Range Axis Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the rangeAxisLabels.

## Examples

```
Range Axis Label Font = "'Arial'"
```

## Range 2 Axis Label Font

### Type

String

### Since

2.6.1

### Description

Sets the font used for the rangeAxisLabels of the overlay chart.

### See also

Range Axis Label Font

## Range Color

### Type

String

### Since

1.2

### Description

Sets the color of the range value labels and tick marks. The expression must have a color format as result.

### Examples

```
Range Color = "'green' "
```

```
Range Color = "'#FFFF00'
```

## Range 2 Color

### Type

String

## Since

2.6.1

## Description

Sets the color of the range value labels and tick marks in the overlay chart. The expression must have a color format as result.

## See also

Range Color

## Range Decimal Count

### Type

Integer

### Since

1.2

### Description

Sets the number of fixed decimals to use for the range labels.

### Examples

```
Range Decimal Count = "10"
```

## Range 2 Decimal Count

### Type

Integer

### Since

2.6.1

### Description

Sets the number of fixed decimals to use for the range labels in the overlay chart.

### See also

Range Decimal Count

## Range Label Font

### Type

String

### Since

1.2

### Description

Sets the font for the range labels.

### Examples

```
Range Label Font = "'Arial'"
```

## Range 2 Label Font

### Type

String

### Since

2.6.1

### Description

Sets the font for the range labels in overlay charts.

### See also

Range Label Font

## Range Label Postfix

### Type

String

### Since

1.2

### Description

Adds a postfix after all value labels.

## Examples

```
Range Label Postfix = "'ms' "
```

## Range 2 Label Postfix

### Type

String

### Since

2.6.1

### Description

Adds a postfix after all value labels. This property applies to the range of the overlay chart.

### See also

Range Label Postfix

## Range Label Prefix

### Type

String

### Since

1.2

### Description

Adds a prefix before the range labels.

### Examples

```
Range Label Prefix = "'$' "
```

## Range 2 Label Prefix

### Type

String

### Since

2.6.1

## Description

Adds a prefix before the range labels. This property applies to the range of the overlay chart.

## See also

Range Label Postfix

## Range Labels Off

### Type

Boolean

### Since

1.2

### Description

Turns off the range labels for the range.

### Examples

```
Range Labels Off = "true"
```

## Range 2 Labels Off

### Type

Boolean

### Since

2.6.1

### Description

Turns off the range labels for the range of the overlay chart.

### See also

Range Labels Off

## Range On

### Type

Boolean

**Since**

1.2

**Description**

Turns on the range.

**Examples**

```
Range On = "true"
```

**Range 2 On****Type**

Boolean

**Since**

2.6.1

**Description**

Turns on the range for the overlay chart.

**See also**

Range On

**Range Position****Type**

Integer

**Since**

1.2

**Description**

Sets the position of the range. 0 is left, 1 is right. The default position for the first range is to the left.

**Examples**

```
Range Position = "0"
```

## Range 2 Position

### Type

Integer

### Since

2.6.1

### Description

Sets the position of the overlay ranges. 0 is left, 1 is right. The default position for the overlay range is, like for the first range, right, so you should always set this property to a different value than "RangePosition" when working with overlay charts and different ranges.

### Examples

```
Range 2 Position = "2"
```

### See also

Range Position

## Range Step

### Type

Number

### Since

1.2

### Description

If the range is not set directly it will default to the largest value of the chart or 0. If RangeStep is set, the range will be set to the next value divisible by the step. This parameter works for the lower range as well.

If the maximum value of the chart is 325, and the step is set to 100, the range will automatically be set to 400.

### Examples

```
Range Step = "100"
```

## Range 2 Step

### Type

Number

### Since

2.6.1

### Description

Set the range step for the overlay chart. For a detailed description of this property refer to the property "RangeStep".

### See also

Range Step

## Sample Axis Label

### Type

String

### Since

1.2

### Description

Adds a label below the sample (X) axis.

### Examples

```
Sample Axis Label = "'Date' "
```

## Range Axis Label Angle

### Type

Integer

### Since

1.2

### Description

Sets the clockwise angle of the range axis labels.

## Examples

```
Range Axis Label Angle = "270"
```

## Sample Axis Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the SampleAxisLabel.

### Examples

```
Sample Axis Label Font = "' Arial' "
```

## Sample Colors

### Type

String

### Since

1.2

### Description

Sets the colors of the lines in the chart. The property expects a list of Color-formats in a string, separated by commas.

### Examples

```
Sample Colors = "' red, green, blue' "
```

## Sample Decimal Count

### Type

Integer

### Since

1.2

## Description

Sets the number of fixed decimals to use for the value labels.

## Examples

```
Sample Decimal Count = "2"
```

## Sample Highlight Image

### Type

String

### Since

2.1

## Description

Use this parameter to set a sample highlight image for all sample points.

## Examples

```
Sample Highlight Image = "/iolap/images/dot.gif"
```

## Sample Highlight On

### Type

String

### Since

1.2

## Description

Each data series can have the sample points highlighted by either a circle, a square, or a diamond shape. Use this parameter to turn sample highlighting on or off for the data series. The default highlight type is circle and highlighting is turned off by default.

This property expects a string containing all values (true or false) in a comma-separated list. Each value is for a single series.

## Examples

```
Sample Highlight On = "true, false"
```

## Sample Highlight Size

### Type

String

### Since

1.2

### Description

Use this parameter to set the size of the sample highlighting for each data series. The default size is 6.

This property expects a string containing all values (integers) in a comma-separated list. Each value is for a single series.

### Examples

```
Sample Highlight Size = "2, 4, 2"
```

## Sample Highlight Style

### Type

String

### Since

1.2

### Description

Use this parameter to set the sample highlight style for each data series. The possible styles are circle, circle\_opaque, circle\_filled, square, square\_opaque, square\_filled, diamond, diamond\_opaque, or diamond\_filled.

This property expects a string containing all values in a comma-separated list. Each value is for a single series.

### Examples

```
Sample Highlight Style = "diamond, circle"
```

## Sample Label Angle

### Type

Integer

## Since

1.2

## Description

Sets the clockwise angle of the sample labels.

## Examples

```
Sample Label Angle = "270"
```

## Sample Label Colors

### Type

String

### Since

1.2

### Description

Sets the colors of the sample labels. This will affect the x-axis labels and the legend labels if they are displayed there. The expression must contain a list of colors in the color format, separated by commas.

### Examples

```
Sample Label Colors = "' red, green, blue' "
```

## Sample Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the sample labels.

### Examples

```
Sample Label Font = "' Arial' "
```

## Sample Labels

### Type

String

### Since

2.2

### Description

This property allows to display custom labels in the sample axis. All labels must be provided in a single string and comma separated.

### Examples

```
Sample Labels = "' A, B, C' "
```

## Sample Labels On

### Type

Boolean

### Since

1.2

### Description

Turns on the sample labels.

### Examples

```
Sample Labels On = "true"
```

```
Sample Labels On = "false"
```

## Sample Label Style

### Type

String('below', 'floating', 'outside' or 'below\_and\_floating')

### Since

1.2

## Description

Controls how the sample labels are displayed. The sample labels can either be displayed below the chart grid, around outside the sample point on the chart grid, floating above a sample point when the mouse moves over it, or both below and floating.

## Examples

```
Sample Label Style = "' below' "
```

```
Sample Label Style = "' floating' "
```

```
Sample Label Style = "' below_and_floating' "
```

## Sample Scroller On

### Type

Boolean

### Since

1.2

### Description

The user can scroll and zoom into the bars at runtime by turning on the sample scroller.

### Examples

```
Sample Scroller On = "true"
```

```
Sample Scroller On = "false"
```

## Series Label Colors

### Type

String

### Since

1.2

### Description

Sets the colors of the sample labels.

### Examples

```
Series Label Colors = "' red, green, blue' "
```

## Series Labels On

### Type

Boolean

### Since

1.2

### Description

Turns on floating series labels. The series labels will be displayed below a sample point in a line when the mouse moves over it.

### Examples

```
Series Labels On = "true"
```

```
Series Labels On = "false"
```

## Series Label Style

### Type

String('inside' or 'outside')

### Since

1.2

### Description

Controls how the series labels are displayed. The series labels can either be displayed around outside the sample point on the chart grid or floating above a sample point when the mouse moves over it.

### Examples

```
Sample Label Style = "'inside' "
```

```
Sample Label Style = "'outside' "
```

## Series Line Off

### Type

Value

## Since

1.2

## Description

Turns off the series lines for all the series, or only the specified data series. This can be used with the sample highlights to make a plotter chart or a combined line and plotter chart. However all the plots are distributed evenly across the x-axis.

## Examples

```
Series Line Off = "false"
```

Turns all lines off

```
Series Line Off = "'1,3' "
```

Turns off the lines for Series 1 and 3

## Single Click URL On

### Type

Boolean

### Since

1.2

### Description

Turn on this parameter to use single click instead of double clicks when doing drilldown charts.

### Examples

```
Single Click URL On = "true"
```

```
Single Click URL On = "false"
```

## Stacked On

### Type

Boolean

### Since

1.2

## Description

Paints filled and stacked lines (area chart).

## Examples

```
Stacked On = "true"
```

```
Stacked On = "false"
```

## Target Labels Position

### Type

String ('left' or 'right')

### Since

2.1

## Description

Sets the position of the target value labels. Can be "left" or "right". By default the target value labels are displayed at the same side as the first range.

## Examples

```
Target Labels Position = "' left' "
```

```
Target Labels Position = "' right' "
```

## Target Value Line\_0

### Type

String

### Since

1.2

## Description

You can set grid lines with a specified label and color at any value position in the chart. The first parameter is the label, the second is the value, the third is the color. There is an optional fourth parameter to control whether only the label or the value should be displayed as opposed to both as the default behavior is. All parameters are expected in one single string and comma-separated.

## Examples

```
Target Value Line_0 = "break even, 150, green"
```

## Thousands Delimiter

### Type

String

### Since

1.2

### Description

Use this parameter to control the thousands delimiter in numerical labels.

### Examples

```
Thousands Delimiter = "','"
```

## Title Font

### Type

String

### Since

1.2

### Description

Sets the font used for the chart title.

### Examples

```
Title Font = "' Arial'"
```

## Url Target

### Type

String

### Since

2.1

## Description

This parameter controls where links will be opened:

- **\_self**: Open new page in same window or frame.
- **\_blank**: Open in a new blank window.
- **name**: Open in the frame or window with the specified name.

## Examples

```
Url Target = ''_self''
```

```
Url Target = ''window1''
```

## Value Label Angle

### Type

Integer

### Since

2.1

### Description

Sets the clockwise angle of the static value labels.

### Examples

```
Value Label Angle = "270"
```

## Value Label Font

### Type

Integer

### Since

1.2

### Description

Sets the font for the static value labels.

### Examples

```
Value Label Font = ''Arial''
```

## Value Label Postfix

### Type

String

### Since

1.2

### Description

Adds a postfix after all value labels.

### Examples

```
Value Label Postfix = "' ms' "
```

## Value Label Prefix

### Type

String

### Since

1.2

### Description

Adds a prefix before all value labels.

### Examples

```
Value Label Prefix = "' $' "
```

## Value Labels On

### Type

Boolean

### Since

1.2

### Description

Turns on the value labels for the sample points in a line.

## Examples

```
Value Labels On = "true"
```

```
Value Labels On = "false"
```

## Value Label Style

### Type

String ('inside', 'outside' or 'floating')

### Since

1.2

### Description

Sets the display style of the value labels. The value labels can be painted directly above or below the sample point, directly on top of the sample point, or floating above the sample point when the mouse hovers over it.

### Examples

```
Value Label Style = "' inside' "
```

```
Value Label Style = "' outside' "
```

```
Value Label Style = "' floating' "
```

## Value Lines Color

### Type

String

### Since

1.2

### Description

Sets the color of the value grid lines. The expression must have a color format as result.

### Examples

```
Value Lines Color = "' red' "
```

```
Value Lines Color = "' #FFFF00' "
```

## Value Lines On

### Type

Boolean

### Since

1.2

### Description

Turns on the horizontal value grid lines in the chart background.

### Examples

```
Visible Samples = "10,25"
```

## Visible Samples

### Type

String

### Since

1.2

### Description

Displays a subset of the samples in the chart. The first argument is the index of the starting sample, the second argument is the number of samples to be displayed.

If you turn on the sample scroller it will reflect the subset currently displayed. Both arguments are expected in one string and comma-separated.

### Examples

```
Visible Samples = "10,25"
```

## Zoom On

### Type

Boolean

### Since

2.1

## Description

If ZoomOn is set to true, the user can use mouse box to zoom into the samples of the chart.

## Examples

```
ZoomOn = "true"
```

```
ZoomOn = "false"
```

# Bar-Chart properties

---

## 3D Depth

**Type**

Integer

**Since**

1.2

**Description**

Sets the depth of the 3D effect in pixels.

**Examples**

```
3D Depth = "10"
```

**See also**

3D Mode On

## 3D Mode On

**Type**

Boolean

**Since**

1.2

**Description**

Turns the 3D-Mode on or off.

**Examples**

```
3D Mode On = "true"
```

```
3D Mode On = "false"
```

**See also**

3D Depth

## Auto Label Spacing On

### Type

Boolean

### Since

1.2

### Description

By default all the sample labels are painted, even if there is not enough space for them all (they will overlap). If this parameter is set only labels there are room for will be painted.

### Examples

```
Auto Label Spacing On = "true"
```

```
Auto Label Spacing On = "false"
```

## Bar Alignment

### Type

String ('vertical' or 'horizontal')

### Since

1.2

### Description

Sets vertical (default) or horizontal bars.

### Examples

```
Bar Aligment = "'vertical'"
```

```
Bar Aligment = "'horizontal'"
```

## Bar Label Angle

### Type

Integer

### Since

1.2

## Description

Sets the clockwise angle of the bar labels.

## Examples

```
Bar Label Angle = "270"
```

## Bar Label Colors

### Type

String

### Since

2.1

## Description

Sets the colors of the bar labels. This will affect only the bar labels below the grid.

## Examples

```
Bar Label Colors = "' red, #cc00cc, blue' "
```

## Bar Label Font

### Type

String

### Since

1.2

## Description

Sets the font used for the static bar labels.

## Examples

```
Bar Label Font = "' Arial' "
```

## Bar Labels

### Type

String

## Since

2.2

## Description

This property allows to replace the bar-labels with your own custom labels. All labels have to be provided in a single string, separated by commas.

## Examples

```
Bar Labels = "' A, B, C' "
```

## Bar Labels On

### Type

Boolean

### Since

1.2

### Description

Turns on the bar labels.

### Examples

```
Bar Labels On = "true"
```

```
Bar Labels On = "false"
```

## Bar Label Style

### Type

String ('below', 'floating' or 'below\_and\_floating')

### Since

1.2

### Description

Sets the display style of the bar labels.

### Examples

```
Bar Label Style = "' below' "
```

```
Bar Label Style = "'floating' "
```

```
BarLabelStyle = "'below_and_floating'"
```

## Bar Outline Color

### Type

String

### Since

1.2

### Description

The color of the bar outline. The expression must have a color format as result.

### Examples

```
Bar Outline Color = "'red' "
```

```
Bar Outline Color = "'#FFFF00' "
```

## Bar Outline Off

### Type

Boolean

### Since

1.2

### Description

Turns off the outline around the bars.

### Examples

```
Bar Outline Off = "true"
```

```
Bar Outline Off = "false"
```

## Bar Type

### Type

String ('stacked' or 'side')

## Since

1.2

## Description

Sets the bar type when multiple series are used. Either stacked bars or side-by-side bars (default).

## Examples

```
Bar Type = "' stacked' "
```

```
Bar Type = "' side' "
```

## Bar Width

### Type

Number

### Since

1.2

### Description

Sets the relative width of each bar. If the width is set to 1.0, there will be no space between the bars.

### Examples

```
Bar Width = "1.0"
```

```
Bar Width = "0.5"
```

## Chart Background

### Type

String

### Since

1.2

### Description

This is the color of the chart grid background. The expression must have a color format as result.

## Examples

```
Chart Background = "' blue' "
```

```
Chart Background = "' #FFFF00' "
```

## Chart Foreground

### Type

String

### Since

1.2

### Description

This is the color of the chart grid outline. The expression must have a color format as result.

### Examples

```
Chart Foreground = "' black' "
```

```
Chart Foreground = "' #FFFF00' "
```

## Chart Title

### Type

String

### Since

1.2

### Description

The result of this property becomes the title of the chart.

### Examples

```
Chart Title = "' Turnover' "
```

## Default Grid Lines Color

### Type

String

**Since**

1.2

**Description**

Sets the color of default grid lines.

**Examples**

```
Default Grid Lines Color = "' black' "
```

```
Default Grid Lines Color = "' #FFFF00' "
```

**Default Grid Lines On****Type**

Boolean

**Since**

1.2

**Description**

Turns on the vertical grid lines.

**Examples**

```
Default Grid Lines On = "true"
```

```
Default Grid Lines On = "false"
```

**Floating Label Font****Type**

String

**Since**

1.2

**Description**

Sets the font used for the floating value and sample labels.

## Examples

```
Floating Label Font = "' Arial' "
```

## Floating On Legend Off

### Type

Boolean

### Since

1.2

### Description

By default floating labels are displayed for all the samples in the selected series when the mouse moves over a series label in the legend. This parameter turns this behavior off.

## Examples

```
Floating On Legend Off = "true"
```

```
FloatingOnLegendOff = "false"
```

## Font

### Type

String

### Since

1.2

### Description

Sets the default font for the chart labels.

## Examples

```
Font = "' Arial' "
```

## Foreground

### Type

String

## Since

1.2

## Description

This is the color of the title, legend labels, value labels, sample labels, and the range labels. The expression must have a color format as result.

## Examples

```
Foreground = "' blue' "
```

```
Foreground = "' #ffff00' "
```

## Graph Insets

### Type

String

### Since

1.2

### Description

Use this parameter to add space between the chart grid and the chart component edges. The parameter value consists of 4 parameters with the following sequence: top, left, bottom, right. A value of -1 uses the default inset. The expression must result to a string containing all these 4 values separated by commas.

### Examples

```
Graph Insets = "' -1, 50, -1, -1' "
```

## Grid Adjustment On

### Type

Boolean

### Since

1.2

### Description

You can adjust the chart grid at runtime by selecting a grid edge and dragging the mouse button. Double-click on the grid edge to set it to the default position.

## Examples

```
Grid Adjustment On = "true"
```

```
Grid Adjustment On = "false"
```

## Grid Image

### Type

String

### Since

2.1

### Description

You can set an image to be used as a background for the grid.

### Examples

```
Grid Image = "/iolap/images/chartbg.gif"
```

```
Grid Image = "http://myserver/images/chartbg.gif"
```

## Grid Line Colors

### Type

String

### Since

1.2

### Description

Sets the color of individual vertical grid lines.

### Examples

```
Grid Lines Colors = "black"
```

```
Grid Lines Colors = "#FFFF00"
```

## Grid Lines

### Type

String

### Since

1.2

### Description

Sets the vertical grid lines. Add one value per grid line. The value is relative to the sample axis range (default 0 to 100) and the width of the chart grid. All values must be passed in a single string, containing the values separated by comma.

### Examples

```
Grid Lines = "' 10, 20, 30, 40, 50, 60, 70, 80, 90' "
```

## Grid Lines Color

### Type

String

### Since

1.2

### Description

Sets the color of all except default grid lines. The expression must have a color format as result.

### Examples

```
Grid Lines Color = "' black' "
```

```
Grid Lines Color = "' #FFFF00' "
```

## Label\_0

### Type

String

### Since

2.1

## Description

This parameter can be used to set a label to any point of the chart. First parameter is label text. Second and third parameters are X and Y coordinates of the label. If X and Y are higher than 0 and lower than 1, the label position is calculated relatively to the chart bounds. Otherwise, the absolute coordinates of the chart are used. There are optional 4th and 5th parameters which are index and series of the sample, pointed by the label.

## Examples

```
Label = "' orange sales,100,100' "
```

```
Label = "' apple sales,0.4,0.5' "
```

```
Label = "' banana sales,200,200,4,0' "
```

## See also

Label Url\_0, Label URL Target\_0

## Label Url\_0

### Type

String

### Since

2.1

### Description

This parameter can be used to assign an URL to an anywhere label. The URL address will be opened when user clicks the label.

### Examples

```
Label URL_0 = "' details.html' "
```

### See also

Label\_0, Label URL Target\_0

## Label URL Target\_0

### Type

String

**Since**

2.1

**Description**

This parameter controls where a HTML page will be opened when clicked the label.

**Examples**

```
Label URL Target_0 = '' _blank' "
```

**See also**

Label\_0, LabelURLTarget\_0

**Legend Colors****Type**

String

**Since**

1.2

**Description**

Sets the colors for the default legend boxes. If this is not set, the colors are taken from the `sampleColors` parameter. The expression must contain a list of colors in the color format, separated by commas.

**Examples**

```
Legend Colors = '' red, green, blue' "
```

**Legend Columns****Type**

Integer

**Since**

2.1

**Description**

Sets the number of columns that should be used to display legend labels.

## Examples

```
Legend Columns = "2"
```

## Legend Font

### Type

String

### Since

1.2

### Description

Sets the font for the labels in the legend.

### Examples

```
Legend Font = "' Arial' "
```

## Legend Image

### Type

String

### Since

1.2

### Description

Sets an image to be used in front of the legend label instead of the default legend box.

### Examples

```
Legend Image = "' /iolap/images/legend.gif' "
```

## Legend Labels

### Type

String

### Since

2.2

## Description

This property allows to replace the labels in the legend with you own custom labels. All labels have to be provided in a single string and must be separated with commas.

## Examples

```
Legend Labels = "' A, B' "
```

## Legend On

### Type

Boolean

### Since

1.2

## Description

Turns on the legend or off.

## Examples

```
Legend On = "true"
```

```
Legend On = "false"
```

## Legend Position

### Type

String ('right', 'left', 'top' or 'bottom')

### Since

1.2

## Description

This property defines the position of the legend inside the chart.

## Examples

```
Legend Position = "' right' "
```

```
Legend Position = "' left' "
```

```
Legend Position = "' top' "
```

```
Legend Position = "' bottom' "
```

## Legend Reverse On

### Type

Boolean

### Since

2.1

### Description

Set on or off the inverted legend. Default legend entry order is from top to bottom and from left to right.

### Examples

```
Legend Reverse On = "true"
```

```
Legend Reverse On = "false"
```

## Locale

### Type

String

### Since

2.2.1

### Description

Sets the locale of the chart. The locale affects the display of value and range labels, with the correct grouping and decimal signs.

### Examples

```
Locale = "' en, GB' "
```

## Lower Range

### Type

Number

## Since

1.2

## Description

Sets the lower value of the range axis (y-axis). If this is not set, the lower range will be automatically set to the smallest sample (or 0 if no samples are negative).

## Examples

```
Lower Range = "40"
```

## Max Value Line Count

### Type

Integer

### Since

1.2

### Description

Sets the maximum number of value lines to be displayed in the chart. You can use this to control the step between the grid labels to certain extent.

### Examples

```
Max Value Line Count = "5"
```

## Multi Color On

### Type

Boolean

### Since

1.2

### Description

Gives each bar a separate color. Use the sampleColors parameter to set the individual bar colors.

### Examples

```
Multi Color On = "true"
```

```
Multi Color On = "false"
```

## Range

### Type

Number

### Since

1.2

### Description

Sets the upper value of the range axis (y-axis). If this is not set, the upper range will be automatically set to the largest sample.

### Examples

```
Range = "100"
```

## Range 2

### Type

Number

### Since

2.6.1

### Description

Sets the upper value of the second range in an overlay chart. For a detailed description of this property, refer to the "Range" property.

### Examples

```
Range = "100"
```

### See other

Range

## Range Adjusted

### Type

Integer

## Since

1.2

## Description

Controls which range(s) are adjusted with which adjuster. By default adjuster 1 controls range 1 and adjuster 2 controls range 2.

## Examples

```
Range Adjusted = "1"
```

```
Range Adjusted = "2"
```

## Range 2 Adjusted

### Type

Integer

### Since

2.6.1

### Description

Controls which range(s) are adjusted with which adjuster in overlay charts. For a detailed description of this property, refer to the "Range adjusted" property.

### See also

Range Adjusted

## Range Adjuster On

### Type

Boolean

### Since

1.2

### Description

Turns on the range adjusters. The range adjuster allows the user to adjust the upper and lower ranges at runtime. Both range adjusters default to the right side of the chart. Use the rangeAdjusterPosition parameter to control the position of the adjusters.

## Examples

```
Range Adjuster On = "true"
```

```
Range Adjuster On = "false"
```

## Range 2 Adjuster On

### Type

Boolean

### Since

2.6.1

### Description

Turns on the range adjusters for overlay charts. For a detailed description of this property refer to the "RangeAdjusterOn" property.

### See also

Range Adjuster On

## Range Adjuster Position

### Type

Integer

### Since

1.2

### Description

Sets the position of the range adjusters. 0 means the left side, 1 the right side. The default is the right side.

### Examples

```
Range Adjuster Position = "1"
```

## Range 2 Adjuster Position

### Type

Integer

## Since

2.6.1

## Description

Sets the position of the range adjusters in overlay charts. For a detailed description of this property, refer to the "RangeAdjusterPosition" property.

## See also

Range Adjuster Position

## Range Axis Label

### Type

String

### Since

1.2

### Description

Adds a label to the range axis. The label is horizontal by default, but the angle can be controlled using the RangeAxisLabelAngle property.

### Examples

```
Range Axis Label = "' Amount' "
```

## Range 2 Axis Label

### Type

String

### Since

2.6.1

### Description

Adds a label to the range axis of the overlay chart. For a detailed description of this property, refer to the "RangeAxisLabel" property.

### See also

Range Axis Label

## Range Axis Label Angle

### Type

Integer

### Since

1.2

### Description

Sets the clockwise angle of the range axis labels.

### Examples

```
Range Axis Label Angle = "270"
```

## Range 2 Axis Label Angle

### Type

Integer

### Since

2.6.1

### Description

Sets the clockwise angle of the range axis labels for the overlay chart.

### See also

```
Range Axis Label Angle = "270"
```

## Range Axis Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the rangeAxisLabels.

## Examples

```
Range Axis Label Font = "' Arial' "
```

## Range 2 Axis Label Font

### Type

String

### Since

2.6.1

### Description

Sets the font used for the rangeAxisLabels of the overlay chart.

### See also

Range Axis Label Font

## Range Color

### Type

String

### Since

1.2

### Description

Sets the color of the range value labels and tick marks. The expression must have a color format as result.

### Examples

```
Range Color = "' green' "
```

```
Range Color = "' #FFFF00' "
```

## Range 2 Color

### Type

String

## Since

2.6.1

## Description

Sets the color of the range value labels and tick marks in the overlay chart. The expression must have a color format as result.

## See also

Range Color

## Range Decimal Count

### Type

Integer

### Since

1.2

### Description

Sets the number of fixed decimals to use for the range labels.

### Examples

```
Range Decimal Count = "10"
```

## Range 2 Decimal Count

### Type

Integer

### Since

2.6.1

### Description

Sets the number of fixed decimals to use for the range labels in the overlay chart.

### See also

Range Decimal Count

## Range Label Font

### Type

String

### Since

1.2

### Description

Sets the font for the range labels.

### Examples

```
Range Label Font = "'Arial'"
```

## Range 2 Label Font

### Type

String

### Since

2.6.1

### Description

Sets the font for the range labels in overlay charts.

### See also

Range Label Font

## Range Label Postfix

### Type

String

### Since

1.2

### Description

Adds a postfix after all value labels.

## Examples

```
Range Label Postfix = "'ms' "
```

## Range 2 Label Postfix

### Type

String

### Since

2.6.1

### Description

Adds a postfix after all value labels. This property applies to the range of the overlay chart.

### See also

Range Label Postfix

## Range Label Prefix

### Type

String

### Since

1.2

### Description

Adds a prefix before the range labels.

### Examples

```
Range Label Prefix = "'$' "
```

## Range 2 Label Prefix

### Type

String

### Since

2.6.1

## Description

Adds a prefix before the range labels. This property applies to the range of the overlay chart.

## See also

Range Label Postfix

## Range Labels Off

### Type

Boolean

### Since

1.2

### Description

Turns off the range labels for the range.

### Examples

```
Range Labels Off = "true"
```

## Range 2 Labels Off

### Type

Boolean

### Since

2.6.1

### Description

Turns off the range labels for the range of the overlay chart.

### See also

Range Labels Off

## Range On

### Type

Boolean

**Since**

1.2

**Description**

Turns on the range.

**Examples**

```
Range On = "true"
```

**Range 2 On****Type**

Boolean

**Since**

2.6.1

**Description**

Turns on the range for the overlay chart.

**See also**

Range On

**Range Position****Type**

Integer

**Since**

1.2

**Description**

Sets the position of the range. 0 is left, 1 is right. The default position for the first range is to the left.

**Examples**

```
Range Position = "0"
```

## Range 2 Position

### Type

Integer

### Since

2.6.1

### Description

Sets the position of the overlay ranges. 0 is left, 1 is right. The default position for the overlay range is, like for the first range, right, so you should always set this property to a different value than "RangePosition" when working with overlay charts and different ranges.

### Examples

```
Range 2 Position = "2"
```

### See also

Range Position

## Range Step

### Type

Number

### Since

1.2

### Description

If the range is not set directly it will default to the largest value of the chart or 0. If RangeStep is set, the range will be set to the next value divisible by the step. This parameter works for the lower range as well.

If the maximum value of the chart is 325, and the step is set to 100, the range will automatically be set to 400.

### Examples

```
Range Step = "100"
```

## Range 2 Step

### Type

Number

### Since

2.6.1

### Description

Set the range step for the overlay chart. For a detailed description of this property refer to the property "RangeStep".

### See also

Range Step

## Sample Axis Label

### Type

String

### Since

1.2

### Description

Adds a label below the sample (X) axis.

### Examples

```
Sample Axis Label = "'Date' "
```

## Range Axis Label Angle

### Type

Integer

### Since

1.2

### Description

Sets the clockwise angle of the range axis labels.

## Examples

```
Range Axis Label Angle = "270"
```

## Sample Axis Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the SampleAxisLabel.

### Examples

```
Sample Axis Label Font = "'Arial'"
```

## Sample Axis Range

### Type

Number

### Since

1.2

### Description

Sets the range of the sample axis range. This range is used to calculate where the vertical grid lines set in the gridLines parameter will be displayed.

### Examples

```
Sample Axis Range = "1000"
```

## Sample Colors

### Type

String

### Since

1.2

## Description

Sets the colors of the bars in the chart. By default the chart is set to single color mode, and all the bars will have the first color. If multiColorOn is set to true, each bar will have a different color. The property expects a list of Color-formats in a string, separated by commas.

## Examples

```
Sample Colors = "' red, green, blue' "
```

## Sample Decimal Count

### Type

Integer

### Since

2.2.2

### Description

Sets the number of fixed decimals to use for the value labels.

### Examples

```
Sample Decimal Count = "2"
```

## Sample Label Angle

### Type

Integer

### Since

1.2

### Description

Sets the clockwise angle of the sample labels.

### Examples

```
Sample Label Angle = "270"
```

## Sample Label Colors

### Type

String

### Since

1.2

### Description

Sets the colors of the sample labels. This will affect the bar labels and the legend labels if they are displayed there. The expression must contain a list of colors in the color format, separated by commas.

### Examples

```
Sample Label Colors = "' red, green, blue' "
```

## Sample Label Font

### Type

String

### Since

1.2

### Description

Sets the font used for the sample labels.

### Examples

```
Sample Label Font = "' Arial' "
```

## Sample Labels

### Type

String

### Since

2.2

## Description

This property allows to display custom labels in the sample axis. All labels must be provided in a single string and comma separated.

## Examples

```
Sample Labels = "' A, B, C' "
```

## Sample Label Selection Color

### Type

String

### Since

1.2

### Description

Sets the color used for the sample labels when a sample is selected. The expression must have a color format as result.

### Examples

```
Sample Label Selection Color = "' red' "
```

```
Sample Label Selection Color = "' #FFFF00' "
```

## Sample Labels On

### Type

Boolean

### Since

1.2

### Description

Displays the sample labels inside or outside each bar.

### Examples

```
Sample Labels On = "true"
```

```
Sample Labels On = "false"
```

## Sample Label Style

### Type

String ('inside' or 'outside')

### Since

1.2

### Description

Displays the sample labels inside or outside each bar. Default style is 'outside'.

### Examples

```
Sample Label Style = "'inside' "
```

```
Sample Label Style = "'outside' "
```

## Sample Scroller On

### Type

Boolean

### Since

1.2

### Description

The user can scroll and zoom into the bars at runtime by turning on the sample scroller.

### Examples

```
Sample Scroller On = "true"
```

```
Sample Scroller On = "false"
```

## Series Label Colors

### Type

String

### Since

1.2

## Description

Sets the colors of the series labels when used in the legend. The property expects a list of Color-formats in a string, separated by commas.

## Examples

```
Series Label Colors = "' red, green, blue' "
```

## Series Label Font

### Type

String

### Since

2.1

## Description

Sets the font used for the series labels.

## Examples

```
Series Label Font = "' Arial' "
```

```
Series Label Font = "' Dialog, plain, 12' "
```

## Series Labels On

### Type

Boolean

### Since

1.2

## Description

Displays the series labels inside or outside each bar. The default style is 'outside', and can be set with the seriesLabelStyle parameter.

## Examples

```
Series Labels On = "true"
```

```
Series Labels On = "false"
```

## Series Label Style

### Type

String ('inside' or 'outside')

### Since

1.2

### Description

Displays the seriesLabels inside or outside each bar. The default style is 'outside'. Use seriesLabelsOn to turn on these labels. In a stacked chart, the series labels can only be used inside.

### Examples

```
Series Label Style = "'inside' "
```

```
SeriesLabelStyle = "'outside' "
```

## Single Click URL On

### Type

Boolean

### Since

1.2

### Description

Turn on this parameter to use single click instead of double clicks when doing drilldown charts.

### Examples

```
Single Click URL On = "true"
```

```
Single Click URL On = "false"
```

## Target Labels Position

### Type

String

## Since

2.1

## Description

Sets the position of the target value labels. Can be "left" or "right". By default the target value labels are displayed at the same side as the first range.

## Examples

```
Target Labels Position = "' left' "
```

```
TargetLabelsPosition = "' right' "
```

## Target Value Line\_0

### Type

String

### Since

1.2

### Description

You can set grid lines with a specified label and color at any value position in the chart. The first parameter is the label, the second is the value, the third is the color. There is an optional fourth parameter to control whether only the label or the value should be displayed as opposed to both as the default behavior is. All parameters are expected in one single string and comma-separated.

### Examples

```
Target Value Line_0 = "break even, 150, green"
```

## Thousands Delimiter

### Type

String

### Since

1.2

### Description

Use this parameter to control the thousands delimiter in numerical labels.

## Examples

```
Thousands Delimeter = "','"
```

## Title Font

### Type

String

### Since

1.2

### Description

Sets the font used for the chart title.

### Examples

```
Title Font = "'Arial'"
```

## Url Target

### Type

String

### Since

2.1

### Description

This parameter controls where links will be opened:

- **\_self**: Open new page in same window or frame.
- **\_blank**: Open in a new blank window.
- **name**: Open in the frame or window with the specified name.

### Examples

```
Url Target = "'_self'"
```

```
Url Target = "'window1'"
```

## Value Label Angle

### Type

Integer

### Since

2.1

### Description

Sets the clockwise angle of the static value labels.

### Examples

```
Value Label Angle = "270"
```

## Value Label Font

### Type

Integer

### Since

1.2

### Description

Sets the font for the static value labels.

### Examples

```
Value Label Font = "'Arial'"
```

## Value Label Postfix

### Type

String

### Since

1.2

### Description

Adds a postfix after all value labels.

## Examples

```
Value Label Postfix = "' ms' "
```

## Value Label Prefix

### Type

String

### Since

1.2

### Description

Adds a prefix before all value labels.

### Examples

```
Value Label Prefix = "' $' "
```

## Value Labels On

### Type

Boolean

### Since

1.2

### Description

Turns on the value labels for the bars. You can display the value labels outside the bars, inside the bars, or as tool tip labels. Control the style with the `valueLabelStyle` property.

### Examples

```
Value Labels On = "true"
```

```
ValueLabelsOn = "false"
```

## Value Label Style

### Type

String('inside', 'outside' or 'floating')

## Since

1.2

## Description

Sets the style of the value labels. The value labels can be inside each bar, outside each bar, or floating above each bar as the mouse moves over it. The default is 'outside'.

## Examples

```
Value Label Style = "'inside' "
```

```
ValueLabelStyle = "'outside' "
```

```
ValueLabelStyle = "'floating' "
```

## Value Lines Color

### Type

String

### Since

1.2

### Description

Sets the color of the value grid lines. The expression must have a color format as result.

### Examples

```
Value Lines Color = "' red' "
```

```
Value Lines Color = "' #FFFF00' "
```

## Value Lines On

### Type

Boolean

### Since

1.2

### Description

Turns on the horizontal value grid lines in the chart background.

## Examples

```
Value Lines On = "true"
```

## Visible Samples

### Type

String

### Since

1.2

### Description

Displays a subset of the samples in the chart. The first argument is the index of the starting sample, the second argument is the number of samples to be displayed.

If you turn on the sample scroller it will reflect the subset currently displayed. Both arguments are expected in one string and comma-separated.

### Examples

```
Visible Samples = "10,25"
```

## Zoom On

### Type

Boolean

### Since

2.1

### Description

If ZoomOn is set to true, the user can use mouse box to zoom into the samples of the chart.

### Examples

```
ZoomOn = "true"
```

```
ZoomOn = "false"
```

# Pie-Chart properties

---

## 3D Mode On

### Type

Boolean

### Since

1.2

### Description

Turns the 3D-Mode on or off.

### Examples

```
3D Mode On = "true"
```

```
3D Mode On = "false"
```

### See also

3D Depth

## Angle

### Type

Double

### Since

1.2

### Description

Sets the angle of the 3D pie. Possible values are 10-90. The default angle is 30.

### Examples

```
Angle = "10"
```

## Chart Title

### Type

String

### Since

1.2

### Description

The result of this property becomes the title of the chart.

### Examples

```
Chart Title = "' Turnover' "
```

## Depth

### Type

Double

### Since

2.1

### Description

Sets the depth of the 3D pie. The depth is relative to the pie width and possible values are 0.0 - 1.0. The default depth is 0.4.

### Examples

```
Depth = "0.1"
```

```
Depth = "0.6"
```

## Detached Distance

### Type

Double

### Since

1.2

## Description

Controls the distance the slices are detached from the pie. The value is a factor of the pie radius, so if the parameter value is set to 0.5, the slice is detached halfway out of the pie.

## Examples

```
Detached Distance = "0.75"
```

## Detached Slices

### Type

String

### Since

1.2

## Description

Detaches slices in the pie. The slices are specified as a comma separated list of slice numbers. Detached slices in a specific pie by specifying the pie number in the parameter.

## Examples

```
Detached Slices = "'1,3,5'"
```

## Floating Label Font

### Type

String

### Since

1.2

## Description

Sets the font used for the floating value and sample labels.

## Examples

```
Floating Label Font = "'Arial'"
```

## Floating On Legend Off

### Type

Boolean

### Since

1.2

### Description

By default floating labels are displayed for all the samples in the selected series when the mouse moves over a series label in the legend. This parameter turns this behavior off.

### Examples

```
Floating On Legend Off = "true"
```

```
FloatingOnLegendOff = "false"
```

## Font

### Type

String

### Since

1.2

### Description

Sets the default font for the chart labels.

### Examples

```
Font = "'Arial'"
```

## Foreground

### Type

String

### Since

1.2

## Description

This is the color of the title, legend labels, value labels, sample labels, and the range labels. The expression must have a color format as result.

## Examples

```
Foreground = "' blue' "
```

```
Foreground = "' #ffff00' "
```

## Graph Insets

### Type

String

### Since

1.2

## Description

Use this parameter to add space between the chart grid and the chart component edges. The parameter value consists of 4 parameters with the following sequence: top, left, bottom, right. A value of -1 uses the default inset. The expression must result to a string containing all these 4 values separated by commas.

## Examples

```
Graph Insets = "' -1, 50, -1, -1' "
```

## Inside Label Color

### Type

String

### Since

1.2

## Description

Sets the color used for the inside labels for all pies. The expression must have a color format as result.

## Examples

```
Inside Label Color = "' green' "
```

## Inside Label Font

### Type

String

### Since

1.2

### Description

Sets the font used with the sample/value/percent labels when inside each pie slice.

### Examples

```
Inside Label Font = "' Arial' "
```

## Label\_0

### Type

String

### Since

2.1

### Description

This parameter can be used to set a label to any point of the chart. First parameter is label text. Second and third parameters are X and Y coordinates of the label. If X and Y are higher than 0 and lower than 1, the label position is calculated relatively to the chart bounds. Otherwise, the absolute coordinates of the chart are used. There are optional 4th and 5th parameters which are index and series of the sample, pointed by the label.

### Examples

```
Label = "' orange sales,100,100' "
```

```
Label = "' apple sales,0.4,0.5' "
```

```
Label = "' banana sales,200,200,4,0' "
```

### See also

Label Url\_0, Label URL Target\_0

## Label Url\_0

### Type

String

### Since

2.1

### Description

This parameter can be used to assign an URL to an anywhere label. The URL address will be opened when user clicks the label.

### Examples

```
Label URL_0 = "' details.html' "
```

### See also

Label\_0, Label URL Target\_0

## Label URL Target\_0

### Type

String

### Since

2.1

### Description

This parameter controls where a HTML page will be opened when clicked the label.

### Examples

```
Label URL Target_0 = "' _blank' "
```

### See also

Label\_0, LabelURLTarget\_0

## Legend Colors

### Type

String

## Since

1.2

## Description

Sets the colors for the default legend boxes. If this is not set, the colors are taken from the `sampleColors` parameter. The expression must contain a list of colors in the color format, separated by commas.

## Examples

```
Legend Colors = "' red, green, blue' "
```

## Legend Columns

### Type

Integer

### Since

2.1

### Description

Sets the number of columns that should be used to display legend labels.

### Examples

```
Legend Columns = "2"
```

## Legend Font

### Type

String

### Since

1.2

### Description

Sets the font for the labels in the legend.

### Examples

```
Legend Font = "' Arial' "
```

## Legend Image

### Type

String

### Since

1.2

### Description

Sets an image to be used in front of the legend label instead of the default legend box.

### Examples

```
Legend Image = "/iolap/images/legend.gif"
```

## Legend Labels

### Type

String

### Since

2.2

### Description

This property allows to replace the labels in the legend with you own custom labels. All labels have to be provided in a single string and must be separated with commas.

### Examples

```
Legend Labels = "A, B"
```

## Legend On

### Type

Boolean

### Since

1.2

### Description

Turns on the legend or off.

## Examples

```
Legend On = "true"
```

```
Legend On = "false"
```

## Legend Position

### Type

String ('right', 'left', 'top' or 'bottom')

### Since

1.2

### Description

This property defines the position of the legend inside the chart.

### Examples

```
Legend Position = "' right' "
```

```
Legend Position = "' left' "
```

```
Legend Position = "' top' "
```

```
Legend Position = "' bottom' "
```

## Legend Reverse On

### Type

Boolean

### Since

2.1

### Description

Set on or off the inverted legend. Default legend entry order is from top to bottom and from left to right.

### Examples

```
Legend Reverse On = "true"
```

```
Legend Reverse On = "false"
```

## Locale

### Type

String

### Since

2.2.1

### Description

Sets the locale of the chart. The locale affects the display of value and range labels, with the correct grouping and decimal signs.

### Examples

```
Locale = "' en, GB' "
```

## Outside Label Color

### Type

String

### Since

2.1

### Description

Sets the color used for the outside labels for all pies.

### Examples

```
Outside Label Color = "' red' "
```

```
Outside Label Color = "' #FFFF00' "
```

## Outside Label Font

### Type

String

### Since

2.1

## Description

Sets the font used with the sample/value/percent labels when outside each pie slice.

## Examples

```
Outside Label Font = "' Arial' "
```

```
Outside Label Font = "' Verdana, bold, 12' "
```

## Percent Decimal Count

### Type

Integer

### Since

1.2

### Description

Sets the number of fixed decimals to use for the percent labels.

### Examples

```
Percent Decimal Count = "2"
```

## Percent Labels On

### Type

Boolean

### Since

1.2

### Description

By turning the percent labels on, the percentage for a pie segment will appear floating on top of the segment when the mouse pointer moves across the segment or the sample label in the legend. The percent labels can also be displayed statically by using the `percentLabelStyle` property.

### Examples

```
Percent Labels On = "true"
```

```
Percent Labels On = "false"
```

## Percent Label Style

### Type

String ('floating' or 'inside')

### Since

1.2

### Description

The percent labels can be displayed floating over the pie slices or statically inside the slices.

### Examples

```
Percent Label Style = "'floating' "
```

```
Percent Label Style = "'inside' "
```

## Pie Label Font

### Type

String

### Since

1.2

### Description

Sets the font to use for the pie labels. The pie labels are displayed below each pie if multiple pies are used and "Pie Labels On" is true.

### Examples

```
Pie Label Font = "'Arial' "
```

## Pie Labels On

### Type

Boolean

### Since

1.2

## Description

The pie labels are displayed below each pie if multiple series/pies are used. The labels displayed are the sample labels.

## Examples

```
Pie Labels On = "true"
```

```
Pie Labels On = "false"
```

## Pie Rotation On

### Type

Boolean

### Since

2.1

### Description

Controls if the pie can be rotated by user. If the parameter is on (the expression returns true), the user can rotate the pie by dragging mouse pointer over it.

### Examples

```
Pie Rotation On = "true"
```

## Pointing Label Color

### Type

String

### Since

2.1

### Description

Sets the color used for the pointing labels for all pies.

### Examples

```
Pointing Label Color = "' red' "
```

```
Pointing Label Color = "' #FFFF00' "
```

## Pointing Label Font

### Type

String

### Since

2.1

### Description

Sets the font used with the sample/value/percent labels when pointing to each pie slice.

### Examples

```
Point Label Font = "' Arial' "
```

```
Point Label Font = "' Verdana, bold, 12' "
```

## Sample Colors

### Type

String

### Since

1.2

### Description

Sets the colors of the pie slices in the chart. The expression must contain a list of colors in the color format, separated by commas.

### Examples

```
Sample Colors = "' red, green, blue' "
```

## Sample Decimal Count

### Type

Integer

### Since

1.2

## Description

Sets the number of fixed decimals to use for the value labels.

## Examples

```
Sample Decimal Count = "2"
```

## Sample Label Colors

### Type

String

### Since

1.2

## Description

Sets the colors of the sample labels displayed in the legend. The expression must contain a list of colors in the color format, separated by commas.

## Examples

```
Sample Label Colors = "' red, green, blue' "
```

## Sample Labels

### Type

String

### Since

2.2

## Description

This property allows to display custom labels in the sample axis. All labels must be provided in a single string and comma separated.

## Examples

```
Sample Labels = "' A, B, C' "
```

## Sample Labels On

### Type

Boolean

### Since

1.2

### Description

By turning on the sample labels, the sample label for a pie segment will appear floating on top of the segment when the mouse pointer moves across the segment or the sample label in the legend. The sample labels can also be displayed statically by using the `sampleLabelStyle` parameter.

### Examples

```
Sample Labels On = "true"
```

```
Sample Labels On = "false"
```

## Sample Label Style

### Type

String ('inside' or 'floating')

### Since

1.2

### Description

The sample labels can be displayed floating over the pie slices or statically the slices.

### Examples

```
Sample Label Style = "'inside'"
```

```
Sample Label Style = "'floating' "
```

## Selection Style

### Type

String ('detached', 'triangle' or 'circle')

## Since

1.2

## Description

Sets the effect painted when the user selects a pie slice or a series in the legend. The effects available is a triangle or circle marker within the pie slice, or detaching the slice as it is selected.

## Examples

```
Selection Style = "' detached' "
```

```
SelectionMode = "' triangle' "
```

```
SelectionMode = "' circle' "
```

## Series Label Colors

### Type

Boolean

### Since

1.2

### Description

Sets the colors of the series labels in the legend. The expression must contain a list of colors in the color format, separated by commas.

### Examples

```
Series Label Colors = "' red, green, blue' "
```

## Series Labels On

### Type

String

### Since

1.2

### Description

By turning on the series labels, the series labels for a pie segment will appear floating on top of the segment when the mouse pointer moves across the segment or the label in the legend. The

series labels can also be displayed statically inside each slice by using the `seriesLabelStyle` parameter.

### Examples

```
Series Labels On = "true"
```

```
Series Labels On = "false"
```

## Series Label Style

### Type

String ('floating', 'inside', 'outside' or 'pointing')

### Since

1.2

### Description

The series labels can be displayed floating over the pie slices, statically inside the slices, outside the slices or pointing to the slices.

### Examples

```
Series Label Style = "'floating' "
```

```
Series Label Style = "'inside' "
```

```
Series Label Style = "'outside' "
```

```
Series Label Style = "'pointing' "
```

## Single Click URL On

### Type

Boolean

### Since

1.2

### Description

Turn on this parameter to use single click instead of double clicks when doing drilldown charts.

### Examples

```
Single Click URL On = "true"
```

```
Single Click URL On = "false"
```

## Slice Seperator Color

### Type

String

### Since

1.2

### Description

Controls the slice separator color. If this is not set, the color used is the same as the slice color, only a little darker. The expression must have a color format as result.

### Examples

```
Slice Seperator Color = "'black' "
```

## Slice Seperator On

### Type

Boolean

### Since

1.2

### Description

Displays a line between the slices in a pie. You can set the color of the lines using the "Slice Seperator Color" parameter.

### Examples

```
Slice Seperator On = "true"
```

```
Slice Seperator On = "false"
```

## Thousands Delimiter

### Type

String

**Since**

1.2

**Description**

Use this parameter to control the thousands delimiter in numerical labels.

**Examples**

```
Thousands Delimeter = "'.'" 
```

**Title Font****Type**

String

**Since**

1.2

**Description**

Sets the font used for the chart title.

**Examples**

```
Title Font = "' Arial' "
```

**Url Target****Type**

String

**Since**

2.1

**Description**

This parameter controls where links will be opened:

- **\_self**: Open new page in same window or frame.
- **\_blank**: Open in a new blank window.
- **name**: Open in the frame or window with the specified name.

## Examples

```
Url Target = ""_self"
```

```
Url Target = ""window1"
```

## ValueLabelPostfix

### Type

String

### Since

1.2

### Description

Adds a postfix after all value labels.

### Examples

```
Value Label Postfix = ""ms"
```

## Value Label Prefix

### Type

String

### Since

1.2

### Description

Adds a prefix before all value labels.

### Examples

```
Value Label Prefix = ""$"
```

## Value Labels On

### Type

Boolean

## Since

1.2

## Description

By turning the value labels on, the value for a pie segment will appear floating on top of the segment when the mouse pointer moves across the segment or the sample label in the legend. The value labels can also be displayed statically by using the "Value Label Style" property.

## Examples

```
Value Labels On = "true"
```

```
Value Labels On = "false"
```

## Value Label Style

### Type

String ('inside' or 'floating')

### Since

1.2

### Description

The value labels can be displayed floating over the pie slices or statically inside the slices.

### Examples

```
Value Label Style = "'inside'"
```

```
Value Label Style = "'floating'"
```



# CHAPTER 3:

## Configuration properties

### Contents of this chapter:

Configuration .....	232
Database .....	238
Table .....	259
Column .....	261
Alias .....	262
Link .....	264
Link-Expression .....	267
Table-Expression .....	269
Dimension .....	272
Grant .....	277
Key .....	279
Key-Attribute .....	283
SQL-Keyloader .....	288
SQL-Attribute .....	301
Time-Keyloader .....	308
Time-Attribute .....	316
Number-Keyloader .....	321
SQL-Cube .....	324
SQL-Dimension .....	333
SQL-Fact .....	342
Store .....	344
Store Dimension .....	349
Store Fact .....	350
Formula .....	351
File-Cache .....	354
Include .....	357

# Configuration

---

## Allow ambiguous cube results for equal coordinates

### Type

Constant Boolean

### Since

2.6.0

### Description

Whenever a fact is mapped without an aggregation and used outside a list report, the SQL cube may generate wrong SQL statements and results, because the missing aggregation will result into too many result rows and multiple result rows will contain different values for the same coordinate.

Usually, this will only generate a warning "Blocked existing value" in the Log, but when this property is set to "true" the model will convert this warning into a fatal error and stop the report execution.

Till now, the default value for this property is "false" and misconfigured cubes only generate warnings. In future releases of instantOLAP we may change the default value to "true" in order to avoid unknown errors for the users.

### Examples

```
Allow ambiguous cube results for equal coordinates = "false"
```

## Build Offline Stores immediatley

### Type

Constant Boolean

### Since

2.2.2

### Description

Setting this property to "true" will let the system rebuild all Stores of the model whenever a they are offline (and the model is online). This will avoid any Store beeing offline unless it contains series errors.

If the property is set to false, empty Stores will stay offline until their next refresh period is reached (defined by the cron-pattern in the stores).

## Examples

```
Build Offline Stores = "true"
```

Build all stores immediately.

```
Build Offline Stores = "false"
```

## Enable Adhoc Queries

### Type

Constant Boolean

### Since

2.5

### Description

When this property is set to "true", Powerusers will be able to create pivot tables on the model defined by this configuration in the web frontend. Initially this property is set to "false".

### Examples

```
Enable Adhoc Queries = "false"
```

```
Enable Adhoc Queries = "true"
```

## Load Timeout

### Type

Constant Integer

### Since

2.2

### Description

Defines the timeout for loading this configuration in seconds. The default-value is 1200 (20 minutes).

### Examples

```
Load Timeout = "2400"
```

Changes the timeout to 40 minutes

## Lock-On-Error Time

### Type

Constant Integer

### Since

2.5

### Description

Whenever the system is unable to load a model, the model will be locked for a given time and the system will not try to load it again within this timespan or until the configuration has been changed by an administrator. This prevents the system to try to load a malicious model very often.

The value in this property determines the lock-time in seconds. The default value for this property is 300 (5 minutes).

### Examples

```
Lock-On-Error Time = "900"
```

## Max Cell Count

### Type

Constant Integer

### Since

2.5

### Description

This property defines the maximum size (in number of cells) for tables of queries using this model. The default value for this property is 250000.

### Examples

```
Max Cell Count = "500000"
```

## Max Result Size

### Type

Constant Integer

**Since**

2.5

**Description**

This property defines the maximum size (in number of rows or column) for tables of queries using this model. The default value for this property is 5000.

**Examples**

```
Max Result Size = "10000"
```

**Max Selector Size****Type**

Constant Integer

**Since**

2.5

**Description**

This property defines the maximum number of options any selector in queries using this model may have. The default value for this property is "10000".

**Examples**

```
Max Selector Size = "20000"
```

**Preprocessor****Type**

Constant String

**Since**

2.6.0

**Description**

This property allows to import a Java class which will be executed every time before a model is loaded or synchronized. This is useful whenever an external datasource has to be informed before instantOLAP loads its database from it.

## Query Timeout

### Type

Constant Integer

### Since

2.5

### Description

This property defines the maximum number of seconds any query using this model may perform until it is interrupted by the system and times out. The default value for this property is "120" (2 minutes).

### Examples

```
Query Timeout = "180"
```

## Shutdown-On-Idle Time

### Type

Constant Integer

### Since

2.5

### Description

In order to save system resources, every model will be shut down by the system automatically whenever it has not been used for defined amount of time. With this property you can change this time (in seconds), the default value for this property is 86400 (24 hours).

### Examples

```
Shutdown-On-Idle Time = "172800"
```

## Start model automatically

### Type

Constant Boolean

### Since

2.2.6

## Description

If this property is set to "true", the model defined by this configuration will be automatically started whenever the server is restarted or the model is offline.

## Examples

```
Autostart = "true"
```

```
Autostart = "false"
```

# Database

---

## Catalog

### Type

Constant String

### Since

1.2

### Description

The catalog-properties defines, which catalog of the database will be connected and used by this database-element. If no catalog is defined, the default-catalog for the connecting user will be used. Not all databases support catalogs (means they only have one catalog). If the connected database doesn't support catalogs, this property will be ignored.

### Examples

```
Catalog = "Northwind"
```

## Charset

### Type

Constant String

### Since

2.0.3

### Description

This property defines the message-char set of the database. This char set will be used for decoding messages and exception retrieved from the database.

In general, there is no need to change this property. But there are some database-systems (e.g. SAS on a mainframe system) sending their messages in a different char set than expected. In this case you can use this property to translate the messages into a readable format.

### Examples

```
Charset = "ISO-8859-1"
```

## Column-end bracket

### Type

Constant String

### Since

2.0

### Description

This property defines the escape-character which is put after of the column-name by the SQL-generator. Together with the Column-Start bracket property you can define the escape-characters for column names of the used database. This is necessary when using column-names with white spaces or special chars.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

```
Column-end bracket = "]"
```

## Column-start bracket

### Type

Constant String

### Since

2.0

### Description

This property defines the escape-character which is put in front of the column-name by the SQL-generator. Together with the Column-End bracket property you can define the escape-characters for column names of the used database. This is necessary when using column-names with white spaces or special chars.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

```
Column-start bracket = "["
```

## Concat-Operator

### Type

Constant String

### Since

2.1

### Description

This property sets the concat-operator generator by the SQL-Generator for this database. The standard ANSI SQL/92 operator is ||, but some databases (e.g. MS Access) don't accept this and have other operators.

### Examples

```
Concat-Operator = "+"
```

## Datasource

### Type

Constant String

### Since

1.2

### Description

The "Datasource" property allows Administrators to connect a database via an already defined J2EE-datasource of the Java application-server: If the administrator application-server defined the datasource-connection, there is no need to define URL, JDBC Driver, User and Password. The only thing you need is the JDNI-Name of the datasource and put it into this property. Then instantOLAP will connect that datasource and use its properties and connection-pool etc.

### Examples

```
Data source = "java:comp/env/jdbc/salesDS"
```

## Drop IN with NULL

### Type

Constant Boolean

### Since

2.0

## Description

instantOLAP generates SQL-statements with IN-lists for filtering values out of dimensions. If a dimension-key is mapped to value NULL, this value will also appear inside the IN-list. A few databases (e.g. Microsoft's Access) don't support NULL inside IN-list and return an error. In this case, this property must be set to "true".

For the most common database-systems this property is already pre-defined and there is no need to change this property.

## Examples

Drop IN with NULL = "false":

```
SELECT Amount, ProductID FROM SALES WHERE ProductID IN ( NULL, '1', '2' )
```

Drop IN with NULL = "true":

```
SELECT Amount, ProductID FROM SALES WHERE ProductID IN ( '1', '2' )
```

## Escape Character

### Type

Constant String

### Since

2.0

### Description

This property defines the character used as escape character inside generated SQL-statements. If no value is given for this property, special character won't be escaped.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

Escape character = "":

```
Select ... WHERE ProductName IN ( 'HÄhnereier' )
```

Escape character = "\\":

```
Select ... WHERE ProductName IN ( 'H\\Ähnereier' )
```

## Force Connection

### Type

Constant Boolean

### Since

2.2.6

### Description

Since version 2.2.6 a model will be started even if not all databases defined in its configuration are reachable. This allows to work with persistent dimensions and offline stores even if the used databases are offline.

However it is possible to forbid the model to startup if a special database is not reachable. This is done by setting the property "Force Connection" of the database definition to "true".

### Examples

```
Force Connection = "true"
```

Do not start the model if the database is not connectable.

```
Force Connection = "false"
```

Default setting.

## Group By Index

### Type

Constant Boolean

### Since

2.1

### Description

Some databases only allow GROUP BY clauses in the statements with the index of the selected column instead of its expression or alias-name. Setting this property to "true" will force the SQL-Generator to use the expression-numbers for GROUP BY.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

```
Group By Index = "false":
```

Creates the SQL-statement

```
SELECT SUM( SALES.AMOUNT ), SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1,
4 ) FROM SALES GROUP BY CUSTOMER_ID, SUBSTR( SALES.DATE, 1, 4 )
```

Group By Index = "true":

Creates the SQL-statement

```
SELECT SUM( SALES.AMOUNT ), SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1,
4 ) FROM SALES GROUP BY 2, 3
```

## See also

Group By Subselect

## Group By Subselect

### Type

Constant Boolean

### Since

2.2.1

### Description

If an aggregation is generated which has to group the summarized columns by complex expressions, some database will refuse to execute this statement because they do not allow to group by complex expressions, only by simple columns.

In this case you can use this property to use a subselect which first creates a temporary table containing all result of the complex expressions and then aggregates by them. This works because in this case the complex expressions become simple columns in the temporary table.

*Note that you also must set the property "Use aliases" to "true" in order to use this feature. The aliases will become the column names of the temporary table.*

### Examples

Group By Subselect = "false":

```
SELECT SUM( SALES.AMOUNT ), SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1,
4 ) FROM SALES GROUP BY CUSTOMER_ID, SUBSTR( SALES.DATE, 1, 4 )
```

Group By Subselect = "true":

```
SELECT SUM( A1 ), A2, A3 FROM ( SELECT SUM( SALES.AMOUNT ) A1,
SALES.CUSTOMER_ID A2, SUBSTR( SALES.DATE, 1, 4 ) A3 FROM SALES ) TMP
GROUP BY A2, A3
```

**See also**

Group By Index

**Initial Query****Type**

Constant String

**Since**

2.6.0

**Description**

This property may contain a SQL query (without result set) which will be executed every time a new connection is established. You can use this query e.g. to set some database or connection properties.

The statement will only be executed for new connection. Whenever a report or cube uses a cached connection, this property will be ignored.

**Examples**

```
JDBC Driver = "com.sun.jdbc.JdbcOdbcDriver"
```

**See also**

Validation Query

**JDBC-Driver****Type**

Constant String

**Since**

1.0

**Description**

This property defines the JDBC-Driver used for a database-connection (the qualified class-path of the driver).

Each database-system uses a different driver for Java-Application, which has to be installed into your application-server before you can use it. To find out, which driver is needed for your database, you should read the database-manual or look at the instantOLAP-website.

For the most common databases, the driver will be suggested automatically by the workbench, but you still must install the drivers into your application (the most drivers are copyrighted and

mustn't be delivered with instantOLAP). The only driver you don't have to install before you can use it is the JDBC-ODBC bridge, which is a part of the Java installation.

If you plan to connect a database via a application-server datasource, you won't need this property.

## Examples

```
JDBC Driver = "com.sun.jdbc.JdbcOdbcDriver"
```

## Limit Syntax

### Type

Constant String

### Since

2.1

### Description

This property defines the syntax generated by the SQL-Generator to perform top-10 queries to the database. Some (but not all) database have a special syntax to return only the first n records for a select statement.

The syntax must be defined as a string with two placeholders \$1 and \$2. The placeholder \$1 will be replaced by the origin SELECT-statement (without the keyword SELECT), the placeholder \$2 by the limit.

If you define no Limit Syntax, instantOLAP will generate simple SELECT-statements and only read the first n records from the result. This can be slower or may slow down the database.

For the most common databases, the limit is already pre-defined.

## Examples

MySql Limit-Syntax:

```
Limit Syntax = "SELECT $1 LIMIT $2"
```

Oracle Limit-Syntax:

```
Limit Syntax = "SELECT * FROM ( SELECT $1 ) WHERE ROWNUM <= $2"
```

## Load Links

### Type

Constant String

## Since

2.1

## Description

If this property is "true", instantOLAP will try to load all foreign key definitions from the database-definition and automatically generate instantOLAP links for them. Not all database-vendors support foreign keys.

## Examples

```
Load Links = "true"
```

```
Load Links = "false"
```

## Load Table-Sizes

### Type

Constant Boolean

### Since

2.1

### Description

instantOLAP will query the size of each table (in rows) when this property is set to true and use it for query-optimization. The SQL-Generator will sort the tables in the FROM-clause of the generated statements from large to small (some databases like Oracle perform much better then).

### Examples

```
Load Table-Sizes = "true"
```

```
Load Table-Sizes = "false"
```

## Max connection age

### Type

Constant Integer

### Since

1.2

### Description

When you connect the database with the instantOLAP-internal connection pool (means when you don't use the Data source property), this property defines the maximum age of connections

inside the connection-pool in seconds. Every connection being older than the value defined here will be reconnected automatically. This is helpful to avoid timed out connections, e.g. when the database-server was restarted and the database-driver isn't able to reconnect automatically on its own.

## Examples

```
Max connection age = "300"
```

Reconnect every connection older than 5 minutes

## Max connection count

### Type

Constant Integer

### Since

1.2

### Description

When you connect the database with the instantOLAP-internal connection-pool (means when you don't use the Data source property), this property defines the maximum number of connections hold inside the connection-pool. All users of instantOLAP share this connection to the database, so you should open enough connections when you have a large number of users. But it's possible to work with more users on a database than connections are opened to it.

## Examples

```
Max connection count = "5"
```

Open at maximum five connections to the database

## Max Table-Count

### Type

Constant Integer

### Since

2.2

### Description

Defines the maximum number of table-definitions to be loaded from the database. If more than this number of tables exist, the Workbench will raise an error when displaying the tables in the explorer and you'll either have to increase this property-value or define the table-names manually using the property Table Names.

## Examples

```
Max table-count = "1000"
```

## See also

Table Names

## Max IN-Count

### Type

Constant Integer

### Since

1.2

### Description

The SQL-generator of instantOLAP generates IN-lists to filter dimension-keys. The lists can become quite long and there is a maximum length of IN-lists for each individual database. Generally, IN-lists can have 1000 or more entries, but some databases only allow smaller lists. The default-value for this value is 500, by changing this property you can allow to generate longer or shorter lists. If the number of needed elements in a lists exceeds this value, the SQL-statement will be split into two or more statements by the SQL-generator.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

## Examples

```
Max In-Count = "100"
```

## Name

### Type

Constant String

### Since

1.0

### Description

This property defines the logical name of the database-connection. Every following element in the configuration (e.g. Key-Loaders or SQL-Cubes) refer to this database by the name defined in this property. This property is mandatory.

## Examples

```
Name = "myDB"
```

## Order By Index

### Type

Constant Boolean

### Since

2.2

### Description

Some databases only allow ORDER BY clauses in the statements with the index of the selected column instead of its expression or alias-name. Setting this property to "true" will force the SQL-Generator to use the expression-numbers for ORDER BY.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

## Examples

```
Order By Index = "false":
```

```
SELECT SUM( SALES.AMOUNT ), SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1,
4 ) FROM SALES GROUP BY CUSTOMER_ID, SUBSTR( SALES.DATE, 1, 4 ) ORDER
BY CUSTOMER_ID, SUBSTR( SALES.DATE, 1, 4 )
```

```
Order By Index = "true":
```

```
SELECT SUM( SALES.AMOUNT ), SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1,
4 ) FROM SALES GROUP BY SALES.CUSTOMER_ID, SUBSTR( SALES.DATE, 1, 4 )
ORDER BY 2, 3
```

## Password

### Type

Constant String

### Since

1.0

### Description

This property defines the password when you connect to the database with the instantOLAP-internal connection-pool (means when you don't use the Data source property). If the user has no password you can leave this property empty.

If you use a Data source-connection managed by the application-server, you won't need this property.

### Examples

```
Password = "tiger"
```

### See also

User

## Read Only

### Type

Constant Boolean

### Since

2.2

### Description

If this property is set to "true", the connection to the database will be opened as read-only connection and the system will not be able to insert any data into the database-tables. The default value for this property is "false".

### Examples

```
Read Only = "true"
```

```
Read Only = "false"
```

## Schema

### Type

Constant String

### Since

1.1

### Description

The schema-properties defines, which schema of the database will be connected and used by this database-element. If no schema is defined, the default-schema for the connecting user will be used. Not all databases support schemes. If the connected database doesn't support schemes, this property will be ignored.

All expressions in the configuration using this database-definitions will automatically access tables from the schema defines with this property. However, you still will be able to connect

tables from other schemes by putting the schema-name in front of the table-name, followed by a dot '.'.

## Examples

```
Schema = "Northwind":
```

The expression "mytable.mycolumn accesses" the table "mytable" in the current schema "Northwind"

The expression "otherschema.othertable.othercolumn" accesses the table "othertable" in the schema "otherschema"

## Schema Names

### Type

Constant String

### Since

2.2.4

### Description

If no default-schema is defined with the property "Schemna", the Workbench will display all schemes with their tables when opening a database in the database-explorer.

Instead of showing all schemes, you can also define a (comma-separated) list of schema-name-patterns with this property (you can use the wildcards '\*' and '?' in the names). Then the Workbench will only show this schemes. The value "\*" will show all schemes.

### Examples

```
Schema Names = "SALES, USERDB"
```

```
Schema Names = "*" 
```

```
Schema Names = "S*"
```

### See also

Table Names, Schema

## Single IN Operator

### Type

Constant String

## Since

2.1

## Description

instantOLAP uses the SQL IN operator to filter the SELECT-statements. Each IN operator expects a list of values as second parameter, e.g. CUSTOMER\_ID IN ( 1, 2, 7 ). If there is only value to compare with, you can force the generator to use another operator than IN, e.g. '=' or 'LIKE'.

If this property is not set, the generator will always generate IN operators. This property is pre-defined for the most common databases.

## Examples

Single IN Operator = NULL:

Create a SQL-statement like:

```
SELECT ... WHERE ... CUSTOMER_ID IN ( 1 ) ...
```

Single IN Operator = "=":

Create a SQL-statement like:

```
SELECT ... WHERE ... CUSTOMER_ID = 1 ...
```

## Table-end bracket

### Type

Constant String

### Since

2.0

### Description

This property defines the escape-character which is put after of the table-name by the SQL-generator. Together with the Table-start bracket property you can define the escape-characters for table-names of the used database. This is necessary when using table-names with white spaces or special chars.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

Table-end bracket = "]"

## Table-start bracket

### Type

Constant String

### Since

2.0

### Description

This property defines the escape-character which is put before the table-name by the SQL-generator. Together with the Table-end bracket property you can define the escape-characters for table-names of the used database. This is necessary when using table-names with white spaces or special chars.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

### Examples

```
Table-start bracket = "["
```

## Table Names

### Type

Constant String

### Since

2.1

### Description

By default, the Workbench will display all tables with their columns when opening a database in the database-explorer. Depending on the number of tables in your database, this can take very long or raise an exception, when the maximum allowed number of tables was exceeded.

Instead of showing all tables, you can also define a (comma-separated) list of table-name-patterns with this property (you can use the wildcards "\*" and "?" in the names). Then the Workbench will only show this tables. The value "\*" will show all tables.

### Examples

```
Table Names = "SALES, CUSTOMER, PRODUCT"
```

```
Table Names = "*"
```

```
Table Names = "F_*"
```

## See also

Max Table-Count, Table Types

## Table Types

### Type

Constant String

### Since

2.0

### Description

This property defines the table-types being displayed in the database-explorer of the Workbench. You have to provide the table-types as a comma-separated list. The standard-types for databases are "TABLE,VIEW". Some databases have more types of tables, e.g. "SNAPSHOT". If you want to display the Workbench these additional types, you must declare them with this property.

### Examples

```
Table-Types = "TABLE, VIEW, SNAPSHOT"
```

## See also

Table Names

## Timeout

### Type

Constant Integer

### Since

1.2

### Description

This property defines the timeout (in seconds) for every query inside instantOLAP when this database is used. When this property is left empty, no timeout will be set for the queries. Not all database-drivers support timeouts, so this property might have no effect for your queries.

*Note: If query is being executed, the actual timeout being set for a single SQL-query is the minimum of this value and the seconds being left for the whole query-execution.*

### Examples

```
Timeout = "90"
```

Sets the timeout to 1.5 minutes

## URL

### Type

Constant String

### Since

1.0

### Description

When connecting a database with the instantOLAP connection-pool (means when you don't use the Datasource property), this property defines the URL for a database-connection. The URL sets the used driver, target database-server and database and optional arguments.

The format of a connection-URL is "jdbc:<driver-name>:<connection-options>". Each database has a different URL-syntax and expects different options, so you should refer to the documentation of the database or JDBC-driver. Note that a connection-URL is only valid when the corresponding is set with the JDBC-Driver property.

In the Workbench there is a suggestion-list with standard-URLs for the most common databases which will be offered when creating a new database. You must replace the parts in brackets with your settings.

If you plan to connect a database via a application-server datasource, you won't need this property.

### Examples

```
URL = "jdbc:mysql://myserver/SALES"
```

Creates a connection to a database "SALES" on a MYSQL-database-server

## Use Aliases

### Type

Constant Boolean

### Since

1.2

### Description

The SQL-Generator of instantOLAP support two different ways of generating SQL-statements: With or without aliases. Aliases are logical names being assigned for column-expression, under which these expressions will be used in the WHERE, SORT and GROUP BY-clause. Some databases don't allow complex expressions in these clauses, then you will need aliases. On the

other hand, some other databases don't allow aliases and will throw an exception like "unknown column ..." - for these database you'll have to disable aliases.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

## Examples

Use aliases = "true":

```
SELECT SUM( amount ), SUBSTR( date, 1, 4 ) Al FROM sales GROUP BY Al
```

Use aliases = "false":

```
SELECT SUM( amount ), SUBSTR( date, 1, 4 ) FROM sales GROUP BY SUBSTR(
date, 1, 4 )
```

## Use Joins

### Type

Constant Boolean

### Since

2.2.1

### Description

The SQL-Generator of instantOLAP supports two different ways of generating JOINS between tables in the SQL-statements: By adding simple conditions to the WHERE clause of the statement or by adding "JOIN" clauses to the statements as being defined in the SQL-92 standard and later. By setting this property to "true" you will force the SQL-generator to use this "JOIN" clauses.

JOIN clauses not only can result into a better performance of the database (because it may be easier for the database to use existing indices on the tables), it also allows to create outer joins or conditional joins between tables (which is not possible with the ANSI-SQL generator).

So whenever you define Links in the database as outer joins you must switch on this property in order to have any effect on the SQL-generator.

For the most common database-systems this property is already pre-defined and there is no need to change this property.

## Examples

Use Joins = "false":

```
SELECT .. FROM SALES, PRODUCTS WHERE SALES.PRODUCT_ID = PRODUCTS.ID
```

Use Joins = "true":

```
SELECT .. FROM SALES LEFT INNER JOIN PRODUCTS ( SALES.PRODUCT_ID =  
PRODUCTS.ID )
```

## User

### Type

Constant String

### Since

1.0

### Description

This property defines the user if you connect the database with the instantOLAP-internal connection-pool (means when you don't use the "Datasource" property). If you use a Data source-connection managed by the application-server, you won't need this property.

### Examples

```
User = "scott"
```

### See also

Password

## Use Schema

### Type

Constant Boolean

### Since

2.2.6

### Description

Use this property to configure instantOLAP to use schemas (or not) for this database. If no schemas are used, instantOLAP will not try to determine the standard schema for the user or for tables and will generate SQL statements without any schema information.

This property should be set to "true" if the connected database systems supports schema or to "false" if it does not. For the most common database vendors, this property is preconfigured and has not to be set manually.

### Examples

```
Use Schema = "true"
```

```
Use Schema = "false"
```

## Validation Query

### Type

Constant String

### Since

2.2

### Description

This property can hold any SQL-statement which will be executed every time when a connection is taken from the connection pool. If the statement fails (e.g. because the database-server was restarted meanwhile and the connection is no longer valid), the connection will be rebuild.

By default, this property is empty and no validation will be performed. This property is only used when defining a connection manually using the JDBC-Driver and URL properties. If a datasource defined by the webservice (using the "Datasource" property) is used, it has to define its own validation.

### Examples

```
Validation Query = "SELECT 1"
```

```
Validation Query = "SELECT COUNT(*) FROM PRODUCTS"
```

### See also

Initial Query

# Table

---

## Name

### Type

Constant String

### Since

1.0

### Description

In normal case, all tables of a database are read automatically out of the database repository. But in very few cases you might be forced to define a table on your own, e.g. because the database-driver isn't able to list the tables, if the table-columns won't have the types you need (columns with unknown types disappear in instantOLAP) or if you have no access to the repository of the database.

For this case, this property keeps the name of the table. This name must be equal to the original name of the table inside the database. When a table is defined this way, it can be used as usual in the later SQL-expressions.

### Examples

```
Name = "Sales"
```

## Schema

### Type

Constant String

### Since

2.2.2

### Description

For manually defined tables you can specify the name of the schema they are located in with this property. When setting this property to another value than empty, the SQL-generator will output this schema-name before the table-name every time the table is used.

### Examples

```
Schema = ""
```

E.g. generates the SQL-statement "SELECT ... FROM SALES ..."

Schema = "ERP"

E.g. generates the SQL-statement "SELECT ... FROM ERP.SALES ..."

# Column

---

## Name

### Type

Constant String

### Since

1.0

### Description

For self-defined tables you must also define all of its columns and their type. This property defines the name of column of a table. The name of a column must be equal to its name inside the database, otherwise a "Column xyz unknown" error will be thrown by the database when executing queries later.

### Examples

```
Name = "Amount"
```

## Type

### Type

Constant String ('string','integer','double' or 'date')

### Since

1.0

### Description

For self-defined tables you must also define all of its column and their type. This property defines the type of column and must have one of the values 'string', 'integer', 'double' or 'date'. In instantOLAP there are only a few number of types for columns, less than the most databases have. All columns are converted (if possible) from the database-type to the instantOLAP type when reading the result from the query.

### Examples

```
Type = "integer"
```

# Alias

---

## Name

### Type

Constant String

### Since

1.2

### Description

This property defines the name of an alias defined for a database. The new alias defined with this element can be used like a regular table - the SQL-generator will use the table defined by the Table property (filtered by the Where-Expression defined in the Where property) for this alias.

### Examples

```
Name = "ActiveProducts"
```

## SQL-Where

### Type

SQL Expression

### Since

1.2

### Description

Whenever you define an alias inside your database-definition, you can apply a where-filter to this table. This means, whenever the SQL-generator uses this alias, this where-expression will automatically be added to the SQL-expression. With this feature, you can reduce the alias to a subset of the original table-data.

*Note: The Where-expression must always refer to the name of the new alias instead of the original table-name! E.g. if you create a new alias "ActiveProducts" for the original table "Products", the where-expression must use "ActiveProducts" as the table-name. Otherwise the SQL-generator would throw an exception or, even worse, filter the wrong table.*

### Examples

```
SQL-Where = "ActiveProducts.active = 1"
```

## Table

### Type

Constant String

### Since

1.2

### Description

With this property you can define the original table to which this alias refers.

### Examples

```
Table = "Products"
```

# Link

---

## Direction

### Type

Constant String ('left', 'right', 'both', 'outer left', 'outer right' or 'outer both')

### Since

2.0

### Description

With this property you can define the direction of a link. The direction of a link determines, from which table the link will be visible and usable for the SQL-generator.

E.g. if you have three tables A, B and C and there will be two links:

- A --> B (direction: right)
- B <-- C (direction: left)

the SQL-generator won't be able to generate a statement with the fact in table A and dimensions in table B and C, because there is no visible path from A (the generator always starts at the table containing the facts) to C (the second link has the wrong direction). You'll have to change the direction of the second link to "left" or "both" in this case.

Be careful with the link-directions, otherwise the SQL-generator could mess up and generate statements resulting in too large values, because more than one fact-table could be included in one statement. Try only to use "left" or "right" for your links.

The direction of a link will be also visible in the ERM-chart for a database.

Since version 2.2 instantOLAP is also able to generate JOINS in SQL expressions using the SQL-92 syntax. This syntax allows to create outer joins between tables (an outer join will still return values even some of the records of one tables do not have corresponding records in the second table). Therefore you can also use the new directions "outer left", "outer right" and "outer both" when the new syntax is activated for your database.

### Examples

```
Direction = "left"
```

```
Direction = "right"
```

```
Direction = "both"
```

```
Direction = "outer left"
```

```
Direction = "outer right"
```

```
Direction = "outer both"
```

## SQL-Where

### Type

Constant Boolean

### Since

2.6.0

### Description

By default, links are only included into a generated SQL statement when they are necessary to connect two of the needed tables. If more than one combination of links is available to connect all needed tables, the generator will use the combination with less tables and links.

By setting this property to "true" you can force the generator to use this link whenever there is more than one alternative.

### Examples

```
Mandatory = "TRUE"
```

## Name

### Type

Constant String

### Since

2.0

### Description

This optional property defines the name of a link, under which it will be displayed in the database-view of the workbench and in the ERM-diagrams (also inside the Workbench).

### Examples

```
Name = "sales_customer"
```

## SQL-Where

### Type

SQL Expression

## Since

2.2.1

## Description

With this property you can define a where condition for this link. Then this link will only be valid if the condition matches - this is known as conditional JOINS since SQL-92.

In order to use where conditions for JOINS they must be enabled for your database.

## Examples

```
SQL-Where = "SALES.CUSTOMERTYPE = 1"
```

Could create a statement like:

```
SELECT ... FROM SALES INNER JOIN CUSTOMER ON ( SALES.CUSTOMER_ID =  
CUSTOMER.ID AND SALES.CUSTOMERTYPE = 1 )
```

# Link-Expression

---

## Operator

### Type

Constant String ('<', '<=', '=', '>=', '>', '<>', 'IN' or 'LIKE')

### Since

2.0.1

### Description

This property defines the logical operator for this Link-Expression. The operator can be one of '<', '<=', '=', '>=', '>', 'IN' or '<>'. This operator defines the operator being used by the SQL-Generator of instantOLAP.

### Examples

```
Operator = "="
```

```
Operator = "<="
```

```
Operator = ">="
```

```
Operator = "<>"
```

## Source Expression

### Type

SQL Expression

### Since

2.0.1

### Description

With this property you can define the source-expression of the link used by the SQL-generator. The source-expression can be a simple table/column expression or a more complex expression, e.g. a function call in SQL.

*Note: If you define more than one Link-Expression for one Link, all source-expression must use the same table as base. Otherwise an error will be raised by the system.*

## Examples

```
Source Expression = "sales.productID"
```

```
Source Expression = "@SUBSTR( sales.date, 1, 4 )"
```

## Target Expression

### Type

SQL Expression

### Since

2.0.1

### Description

With this property you can define the target-expression of the link used by the SQL-generator. The target-expression can be a simple table/column expression or a more complex expression, e.g. a function call in SQL.

*Note: If you define more than one Link-Expression for one Link, all target-expression must use the same table as base. Otherwise an error will be raised by the system.*

## Examples

```
Target Expression = "sales.productID"
```

```
Target Expression = "@SUBSTR( sales.date, 1, 4 )"
```

# Table-Expression

---

## Name

### Type

Constant String

### Since

2.2

### Description

This mandatory property defines the name of the table-expression. The table-expression will become visible in the target Table as a "virtual" column with this name.

### Examples

```
Name = "Year"
```

```
Name = "MONTH"
```

## SQL-Expression

### Type

SQL Expression

### Since

2.2

### Description

The property "SQL-Expression" defines the SQL-expression which is used for the "virtual" column defined by this table-expression.

Every time this "virtual" column is used, the SQL-generator will replace it by this expression. If the expression uses other tables than the one this Table-Expression belongs to (defined by the Table attribute), the generator will add these other tables and all necessary links. This means, you can also use Table-Expressions to "import" columns from other tables to a table.

### Examples

```
SQL-Expression = "@TO_CHAR( Customer.Date, 'yyyy' )"
```

## Table

### Type

Constant String

### Since

2.2

### Description

Each Table-Expression must be assigned to a table and will appear as a "virtual" column of this table whenever the administrator views the table in the database-explorer. Therefore this property must contain the name of an existing table of the database.

### Examples

```
Table = "Customer"
```

## Type

### Type

Constant String ('string','integer','double' or 'date')

### Since

2.2

### Description

This property defines the (return-) type of this Table-Expression. You always should set the type because when using functions the system will not be able to figure out the type automatically.

### Examples

```
Type = "integer"
```

## Type

### Type

Constant String (comma separated table names)

### Since

2.6.0

## Description

If the expression defined for this table expression is escaped (means, it uses the phrase @"..." to escape its content), the SQL generator can no longer decide which tables are used by this expression and cannot add them to the FROM clause of the generated statement whenever this table expression is used.

This may cause a runtime error whenever this expression is used. To avoid this you can give a hint to the SQL generator by listing all used tables, separated by commas, in this property.

## Examples

```
Used Tables = "PRODUCTS, CATEGORIES"
```

# Dimension

---

## Cron Pattern

### Type

Constant String (Cron pattern)

### Since

1.1

### Description

The Cron pattern of a dimension determines if and how often the keys of a dimension will be synchronized by the system. The Cron pattern is a time-scheme (a cron pattern string) defining exactly when this synchronize will happen.

Whenever a synchronizing of a dimension is triggered, all Key-Loaders defined for this dimension will be asked to query their datasource for new changes. When a loader detects a change, all keys of the dimension will be discarded and reloaded again, now with the new version of the database-content.

Synchronizing a dimension can cost a lot of time and should be used wisely. Don't synchronize your dimensions too often and try to represent the datasource-change frequency with this property. E.g a period-dimension doesn't have to be updated more than once a day, because there wouldn't be more than one day per day to add to the calendar.

### Examples

```
Cron pattern = "*":
```

Synchronize every minute

```
Cron pattern = "* * 0 0":
```

Synchronize every day at 00:00

## Default Text-Attribute

### Type

Constant String

### Since

2.1

## Description

Usually the ID of a key will be displayed inside queries whenever a key is used in headers and no other text is defined. With this property you can change this behaviour and display an attribute of this dimension instead.

If this attribute is not defined for each key of the dimension, the keys without the attribute will display no text inside the queries.

## Examples

```
Default Text-Attribute = "Text"
```

## Description

### Type

Constant String

### Since

2.2

## Description

This property can hold a brief description of the dimension. The description will then be displayed inside the query-editor, adhoc-tool etc.

## Examples

```
Description = "All products"
```

## Level-Names

### Type

Constant String

### Since

2.2

## Description

The names of the levels of a dimension can be set using two different methods: Either by the loaders, which set each single level-name when loading a level into a dimension or by defining all level-names in this property.

If you use this property, you must list all level names (beginning with the level 0 of the root key) and use commas to separate them.

## Examples

```
Level-Names = "ALL, GROUP, PRODUCT"
```

## Load Children

### Type

Constant Boolean

### Since

2.2.2

### Description

This property allows to switch of the automatic loading of children for dynamic dimensions (only). This can be usefull for very large dynamic dimension, especially if there are many keys directly placed under the root-key of the dimension.

### Examples

```
Load Children = "true"
```

```
Load Children = "false"
```

## Max Size

### Type

Constant Integer

### Since

2.2

### Description

This property defines the maximum number of keys a dimension can contain. The default value for this property is 100000, increase the maximum size if you want to store more than this number of keys in your dimension.

### Examples

```
Max Size = "250000"
```

## Name

### Type

Constant String

## Since

1.0

## Description

This property defines the name of a dimension under which it will be used later in a configuration or query.

The name of a dimension always should be in the single-form, e.g. use "Product" instead of "Products" or "Customer" instead of "Customers". This is because in the queries, the dimension-name is the default-text display ahead selectors or above table-columns.

## Examples

```
Name = "Product"
```

## Storage

### Type

Constant String ('persistent' or 'dynamic')

### Since

2.0

### Description

Whenever a dimension is initially created or synchronized later, the dimension-data (not the facts!) is transferred out of the dimension into the instantOLAP-system. There are two option how to store the dimension-data: All at once on the hard disc (with the "persistent") or dynamically (whenever a new key is needed it will be loaded from the databases and cached locally). storage to the desired value with this property.

- The persistent storage is practically not limited in the number of keys (only the hard disc-size limits the number of keys, but dozens of millions of keys should be no problem) but initially slower than the dynamic storage.
- The dynamically loaded dimensions will load their keys (usually from SQL-Keyloaders) in the moment they are needed the first time and then cached. Because of this real-time behaviour, the loader-structure of dynamic dimension must be less complex then usually (e.g. there must be only one nested keyloader per loader or automatic evaluation of unique attributes will not work).

Dynamic loading only makes sense if the dimensions will own a deep hierarchy with not too many childs per key, because the keys are always loaded in "packages" of childs per key.

### Examples

```
Storage = "persistent"
```

```
Storage = "dynamic"
```

## Visible

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

Defines if a dimension is visible for users inside the query-editor or adhoc-tool. If this property is set to "true" (the default value), users can see and use the dimension inside the query editors. Otherwise it will be invisible but does still exist.

### Examples

```
Visible = "true"
```

```
Visible = "false"
```

# Grant

---

## Action

### Type

Constant String ('read' or 'write')

### Since

2.1

### Description

When designing access rules for a dimension, each rule can either limit the read- or the write-access to a dimension. This property defines which kind of access this rule influences.

### Examples

```
Action = "read"
```

```
Action = "write"
```

## See also

Expression

## Expression

### Type

Boolean

### Since

2.1

### Description

Whenever the system evaluates the access-rights of a user for a single key of a dimension, it will execute the expressions of all Grants of this dimension (if the "Action" of the grant is the right one) and let the user access the key if at least one expression returned "true".

Because of this, the expression of a Grant must return a boolean value. In this expression you can access the desired key of the dimension by simply using the dimension name (because the key will be the only filtered of this dimension when the expression is executed).

## Examples

```
Expression = "HASROLE( 'iolapAdmin' )"
```

```
Expression = "USER() = 'joe' AND Region = 'North' "
```

```
Expression = "HASROLE( PRODUCT ).departmentName"
```

## See also

Action

# Key

---

## Format

### Type

Constant String (Number format)

### Since

1.2

### Description

For facts (keys defined inside your fact-dimension), this property defines the number- or date-format. Whenever the fact is used in a query (as a header-iteration), this format will be applied to the table-cells of the displayed result.

Fact-formats won't be used when you define the cell-content with a formula. In this case you must define the display-format inside the query manually.

This format only works for facts (for keys being defined inside the fact-dimension). If you use this property for a key of a regular dimension, the system will throw an error.

### Examples

```
Format = "###,###,##0.00"
```

```
Format = "0%"
```

## Description

### Type

Constant String

### Since

2.6.0

### Description

This property allows to place a description for the key or fact. The description is displayed in several locations, e.g. when using Excel as frontend.

## ID

### Type

Constant String

### Since

1.0

### Description

Each key inside a dimension must have a unique ID which is used to refer a key later inside the configuration and the queries. With this property you can define the ID for a manually created key in your dimension.

The ID is also used to display keys as a header inside queries. If you want to display an attribute instead of the key-ID instead you can change this behaviour with the Default Text-Attribute property of the dimension.

If the ID of the key is already used by another key inside the same dimension, the system will throw an error.

### Examples

```
ID = "Product A"
```

```
ID = "Amount"
```

### See also

Default Text-Attribute

## Level Name

### Type

Constant String

### Since

2.2

### Description

Each level of a dimension has a name, which is unique inside the dimension. When defining keys manually, this property defines the name of the level where this key will be added.

If no name is defined, the system will generate a level-name automatically or use the name defined in other Keyloaders or the dimension itself. Note that different names for the same level are forbidden, so the system may report an error if the name in this property differs from other names in other loaders for the same level.

## Examples

```
Level Name = "Customer"
```

## Type

### Type

Constant String ('integer', 'double', 'string', 'boolean' or 'date')

### Since

2.2

### Description

This property sets the type of a fact (a key defines inside your fact-dimension). The type of a fact is important inside expression and also determines the return-type of formulas returning this fact.

There are several possible types ('integer', 'double', 'string', 'boolean' and 'date'). The standard-type for facts is 'double'.

Note that this property is only valid for facts and cannot be used in regular dimensions.

## Examples

```
Type = "string"
```

```
Type = "integer"
```

```
Type = "double"
```

```
Type = "date"
```

```
Type = "boolean"
```

## Unit

### Type

Constant String

### Since

1.2

### Description

This property defines the unit for a fact (a static key defined inside the fact-dimension). The unit is an optional and freely definable string which will be displayed inside the reports (in the headers) whenever a fact is used.

If you define a unit for an other key than a fact, the system will throw an error.

### Examples

```
Unit = "EUR"
```

## Unit

### Type

Constant String

### Since

2.6.0

### Description

This property defines if the key or fact is visible, by default all keys are visible. If a key or fact is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

### Examples

```
Visible = "FALSE"
```

# Key-Attribute

---

## Description

### Type

Constant String

### Since

2.6.0

### Description

This property allows to place a description for the attribute.

## Ignore Missing Targets

### Type

Constant Boolean

### Since

2.1

### Description

When generating this Attribute as a link, the System will search for keys in the Target Attribute with an ID matching the result of the SQL Expression. If no target-key was found, the system will stop loading the model and show an error.

You can change this behaviour and ignore missing link-target by setting this property to "true".

### Examples

```
Ignore Missing Targets = "true"
```

## Name

### Type

Constant String

### Since

2.0

### Description

Every attribute of a key must have a name. Define the name of an attribute with this property.

### Examples

Name = "Color"

## Relink-Attribute

### Type

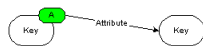
Constant String

### Since

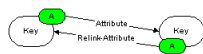
2.0

### Description

Keys from one dimension can be linked to keys from other dimensions as attributes (see the Dimension property). Whenever a key is linked to another one you can use the attribute inside your expression like you would use other attributes, but they will have keys as result. E.g the expression "Customer.Product" would result to the Product(s) of the current customer (when the Customers have been loaded with a linked attribute named "Product").



By default, links are only uni-directional. The first (source) key (which attribute-loader defined the link with its Dimension property) would point to the second (target) key but not vice versa. If you also want the target key to point backwards to source key, you can define the name of the backward-linking attribute with this property. E.g. if the Relink-Attribute for the sample above would be "Customers", you now could use the expression "Product.Customers".



### Examples

Relink-Attribute = "Customers"

## Target Attribute

### Type

Constant String

### Since

2.2.6

## Description

This property allows the more advanced linking between keys of two different dimensions.

The "Type" property of an attribute allows the simple linking between two keys by searching the target-key with its id. But sometimes the source-Keyloader can't link to the target-keys with the id, because it is not possible to load the target-id from its database or tables (or it might be too time-consuming because a couple of database-join would become necessary then).

In this case you might try to connect two keys by searching the target-key with one of its attributes which is known by the source-key. E.g. if the table you load the source-keys "Product" from has a column "MANUFACTURER\_ID", you can link the products to their manufacturers by searching them with their ManufacturerID. Of course, the manufacturers must have a so called attribute, otherwise the system would raise an error.

## Examples

```
Target-Attribute = "CustomerID"
```

Search the target-keys by its Customer-ID

## Type

### Type

Constant String

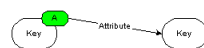
### Since

2.2

## Description

Attributes can either have a basic type (string, integer, double, boolean, date) or point to a key from another dimension (then called a "link"). You can define the type of an attribute with this, it will automatically be converted when loading the attribute. By default, the type of an attribute is "string".

If you want the attribute to point to another dimension, you must specify the target-dimension with this property by using the target dimension-name as type-name. The loader will then search for the key in the target-dimension with the ID loaded from the CSV-column (defined with the property Column) and generate a link to this key.



If no key with the ID was found, the system will raise an error and stop loading the model.

## Examples

```
Type = "String"
```

```
Type = "Integer"
```

```
Type = "Customer"
```

This attribute will become a link to the dimension "Customer"

## Unique

### Type

Constant String ('true', 'false' or 'auto')

### Since

2.2

### Description

An attribute of a dimension can be declared as "unique". Only unique attributes can be used in cubes to map dimension via this attribute (e.g. a cube could map the dimension "Product" with the unique attribute "ProductID" to a column "PRODUCT\_ID" of the fact-table).

If an attribute is unique, only one key per dimension may have a specific value stored in this attribute - e.g. it would be forbidden to have two different keys of a dimension "Product" with the same "ProductID" if it was declared as unique attribute.

There are two different ways to define that an attribute is unique:

- Set the property "Unique" to true: The attribute is expected to be unique. If the same attribute-value is loaded for two different keys, the loader will raise an error and stop loading the model.
- Set the property "Unique" to auto: The loader will automatically find out, if the attribute is unique. This is a bit dangerous, because a unique attribute could become non-unique later and some cube-mapping couldn't work anymore.

If you set the property to false, the attribute won't be unique. Note that unique attributes use a bit more system-resources, so you should avoid to have unique attributes unless you want to use them for mapping cubes.

### Examples

```
Unique = "true"
```

```
Unique = "false"
```

```
Unique = "auto"
```

## Value

### Type

Constant String

### Since

1.2

## Description

This property defines the value of the attribute. Because the key is constant, its attribute-value is constant, too and is a simple string. Depending on the attribute type (simple string or dimension-link), this property must hold the string-value or the ID of the target-key in the target-dimension.

## Examples

```
Value = "ProductA"
```

## Unit

## Type

Constant String

## Since

2.6.0

## Description

This property defines if the attribute is visible, by default attributes are visible. If an attribute is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

## Examples

```
Visible = "FALSE"
```

# SQL-Keyloader

---

## Database

### Type

Constant String

### Since

1.0

### Description

The Database-property defines, from which database the Keyloader will load and generate the keys. This property must refer to a database prior defined (with the logical name of the database-definition).

### Examples

```
Database = "SalesDB"
```

## Distinct

### Type

Constant Boolean

### Since

1.2

### Description

This property defines if the generated SQL-statement will use the DISTINCT clause to generate a distinct list of keys. The DISTINCT clause in SQL eliminates all result rows which appear a second or more time. Some databases don't support the DISTINCT clause at all and some others don't allow DISTINCT in combination with the selection of BLOB-columns or others. In this case you should disable this property and load the data without.

Even without this option, instantOLAP will only load distinct entries but eliminate the non-distinct entries manually. This will cause a slower loading of the keys because the database-result can become much larger. Therefore you should try to avoid to disable this option.

### Examples

```
Distinct = "true"
```

```
Distinct = "false"
```

## Format

### Type

Constant String

### Since

1.2

### Description

With this property you can convert the results of the SQL-Expression into strings. Depending on the result-type of the expression, this property must contain a Number-Format or Date-Format and will then convert the info a string using this format. If you don't specify any format, these types will be converted into string using the standard-format.

### Examples

```
Format = "#, ###, #0"
```

```
Format = "0.00"
```

```
Format = "dd. MM. yyyy"
```

```
Format = "dd. MM. yyyy HH: mm: ss"
```

## Ignore Hierarchy Violation

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

The SQL-Keyloader usually performs checks when building the dimension-hierarchy: If any key is loaded more than once, its parent-key must always be the same (because a key must always have only one parent). If the loader returns different parent-keys for the same key, an error message saying "Hierarchy violation: ..." will be raised.

With this property you can switch of this check. However, this does not mean the key is placed under two or more different parent at the same time. Only the first parent being assigned to a key will be used, any later assignment to a different parent will be ignored without raising an error.

*Setting this property to "true" is dangerous, because the suppressed error messages always signal a damaged hierarchy structure and it can cause broken aggregation and other effect.*

*Instead of using this property you should consider to redesign your model or to rename the loaded keys e.g. by adding the parent-ids to their ids.*

## Examples

```
Ignore Hierarchy Violation = "true"
```

```
Ignore Hierarchy Violation = "false"
```

## Level Name

### Type

Constant String

### Since

2.2

### Description

Each level of a dimension has a name, which is unique inside the dimension. When loading keys from a database with a SQL-Keyloader, this property defines the name of the level the keys are loaded into.

If no name is defined, the system will generate a level-name automatically or use the name defined in other Key-loaders or the dimension itself. Note that different names for the same level are forbidden, so the system may report an error if the name in this property differs from other names in other loaders for the same level.

## Examples

```
Level Name = "Customer"
```

## Mode

### Type

Constant String ('insert\_and\_extend', 'insert' or 'extend')

### Since

2.0

### Description

instantOLAP support three different modes for Keyloaders: insert\_and\_extend, insert and extend. This mode defines the behavior of instantOLAP for keys loaded from the Keyloader when the key already exists in the dimension:

- **insert:** Only non-existing keys may be loaded and inserted with this Keyloader. If a key already exists, the Keyloader will raise an exception and loading of the complete model will be stopped.

- **insert\_and\_extend:** The loader will ignore the fact that a key already exists and add the new attributes loaded with this execution to the existing key. You can use this mode to create new keys and/or extend existing keys with new attributes.
- **extend:** The loader will only add the new attributes loaded by this loader to existing keys. There will be no new keys being generated by this loader. You can use this mode to add some attributes to keys loaded prior by other loaders (e.g. to add a ID or text to keys loaded from another database).

## Examples

```
Mode = "insert_and_extend"
```

```
Mode = "insert"
```

```
Mode = "extend"
```

## Null-Value

### Type

Constant String

### Since

2.0

### Description

This optional property determines which string will be used as ID for NULL-values loaded out of the database. You can replace NULL-values with valid an ID using this property. If no Null-Value was defined, loaded keys with a NULL-ID will be dropped.

### Examples

```
Null-Value = "Unknown customer"
```

## Order-By-ID

### Type

Constant Boolean

### Since

2.2

### Description

Usually instantOLAP orders the loaded keys by their IDs if no other order is set. It then simply adds an ORDER BY clause to the statements which is exactly equal to the Sql-Expression of the loader.

If you don't want the system to add *any* order clause to the statement, you can set this property to "true".

## Examples

```
OrderById = "true"
```

```
OrderById = "false"
```

## Parent Attribute

### Type

Constant String

### Since

1.2

### Description

Whenever you manually define the SQL Parent-Expression for a Keyloader, it will sort the new loaded key into the hierarchy by search parent key with the ID defined by the Parent SQL-Expression. In some case you don't know the parent's ID but only one of it attributes (maybe the technical id, its product-number etc.). In this case you can define with which attribute you want to search the parent key (of course, the Parent SQL-Expression must then contain the corresponding value).

## Examples

See the following example-tables CATEGORY and PRODUCT:

CATEGORY_ID	NAME
1	BEER
1	WINE

PRODUCT_ID	CATEGORY_ID	NAME
1	1	LIGHT BEER
2	1	STRONG BEER
3	2	WHITE WINE
4	2	RED WINE

Now imagine two SQL-Keyloaders, the first one loading the categories:

- SQL-Expression = "CATEGORY.NAME"

- SQL-Attribute with name "CategoryId" and SQL-Expression "CATEGORY.CATEGORY\_ID"

And the second loader for the products:

- SQL-Expression = "PRODUCT.NAME"
- Parent SQL-Expression = "PRODUCT.CATEGORY\_ID"
- Parent Search-Attribute = "CategoryId"

Now the second loader would sort its keys correctly into the hierarchy without knowing the ID (name) of their parents, because it uses the CategoryId (which is available in the PRODUCT-table) to search the parent keys.

## See also

Parent Trim, Parent SQL-Expression

## Parent Format

### Type

Constant String

### Since

1.2

### Description

With this property you can convert the results of the SQL-Parent-Expression into strings. Depending on the result-type of the expression, this property must contain a Number-Format or Date-Format and will then convert the info a string using this format. If you don't specify any format, these types will be converted into string using the standard-format.

### Examples

```
Parent Format = "0"
```

## Parent Null-Value

### Type

Constant String

### Since

2.2

### Description

This optional property determines which string will be used as ID for NULL-values when searching for a parent key. You can replace NULL-values from the Parent-Expression with an

valid ID using this property. If no Parent Null-Value was defined, loaded keys with a parent NULL-ID will be dropped.

## Examples

```
Parent Null-Value = "Unknown group"
```

## See also

Parent SQL-Expression

## Parent SQL-Expression

### Type

SQL Expression

### Since

1.2

### Description

The optional property "SQL Parent-Expression" defines the SQL Expression which generates the ID for the parent keys. The expression must be a valid SQL-fragment for the database used by this loader. You may use simple table/column expression or more complex expressions with concatenations and formulas.

Setting the Parent-Expression will not create the parent keys - it only defines the IDs of the keys under which the loaded keys will be sorted. If no parent key with this ID exists, a key will not be loaded anyway.

If no Parent-Expression was defined, the Keyloader will use the ID-Expression of the surrounding Loader as its Parent-Expression (for example when you dragged a Keyloader onto another one). Embedded Keyloaders are the common case and defining your own Parent-Expression with this property is quite advanced.

If you want to search the parent with an attribute instead of its ID you can define this with the property Parent Search-Attribute.

## Examples

```
SQL-Parent-Expression = "STAFF.COMPANY"
```

Search for keys with the id stored the COMPANY-column as parent

## Parent Trim

### Type

Constant Boolean

## Since

1.2

## Description

With the Trim parent-property you can force the loader to trim the parent-IDs generated by the SQL-query after they have been read out of the query-result. If the parent ID have unnecessary white spaces at their end, the loader would be unable to find the parent-keys and to sort the new keys into the hierarchy. Then you should enable this option.

## Examples

```
Trim parent = "true"
```

```
Trim parent = "false"
```

## Prefix

### Type

Constant String

### Since

2.5

### Description

The prefix defined in this property will be added to all SQL-result before searching for a key with an ID or attribute matching the result. This property can be used instead of concatenating prefixes to the SQL-expression in order to make IDs unique.

### Example

```
Prefix = 'Product_'
```

## Recursive

### Type

Constant Boolean

### Since

1.2

### Description

This is done with an iteration which does not stop before no new key has been loaded. Because one iteration could load the parent key for one of the future iterations, this will create a whole

tree of keys within the dimension. Whenever you use the Recursive-Mode, you must define an SQL-Parent Expression.

## Examples

See the following example-table PRODUCT:

PRODUCT_ID	PARENT_ID	NAME
5	2	WHITE WINE
4	2	RED WINE
3	1	BEER
2	1	WINE
1	NULL	ALL PRODUCTS

With this table, you could set up a SQL-Keyloader with the following properties and attributes:

- SQL-Expression = "PRODUCT.NAME"
- SQL-Parent-Expression = "PRODUCT.PARENT\_ID"
- Parent-Attribute = "ProductID" (search the parent with its ID)
- SQL-Attribute named "ProductID" with the SQL-Expression "PRODUCT.PRODUCT\_ID"
- Recursive = "true"

Now the loader would load all keys within 3 iterations:

First iteration:

- ALL PRODUCTS loaded (because it has no parent and will be placed at the loaders position)

Second iteration:

- WINE loaded (because the parent ALL PRODUCTS now exists)
- BEER loaded (because the parent ALL PRODUCTS now exists)

Third iteration:

- WHITE WINE loaded (because the parent WINE now exists)
- RED WINE loaded (because the parent WINE now exists)

Fourth iteration:

- Nothing loaded, the loader stops

## See also

Parent Search-Attribute, Parent SQL-Expression

## SQL-Check Expression

### Type

SQL Expression

### Since

1.2

### Description

Every time a dimension is synchronized, it will ask all its loaders if any change did happen in their data since the last load. For a SQL-Keyloader, you can control the change-condition with this property. The dimension will trigger a new load when the SQL-Expression defined in this property returns a different value than before. If no SQL-Check is defined, the dimension will reload its keys every time.

### Examples

```
SQL-Check = "MAX( Products.ROWID )"
```

Check the latest ROWID for any changes (works with Oracle-Databases)

```
SQL-Check = "COUNT( Products.ID )"
```

Have some products been added or deleted?

## SQL-Expression

### Type

SQL Expression

### Since

1.0

### Description

The property "SQL-Expression" defines the SQL Expression which generates the ID for the new keys. The expression must be a valid SQL-fragment for the database used by this loader. You may use simple table/column expression or more complex expressions with concatenations and formulas.

Every key inside a dimension must have a unique ID. Because of this, the SQL-expression should generate unique texts.

## Examples

```
SQL-Expression = "Customer.Name"
```

```
SQL-Expression = "Customer.CustomerID || Customer.Name"
```

```
SQL-Expression = "@SUBSTR( Customer.CusomerName, 1, 10 )"
```

## SQL-Order

### Type

SQL Expression

### Since

1.1

### Description

To define the order of the loaded keys inside a dimension you can use this optional property. The property expects a SQL-Expression, by which the result set of the database-query will be ordered.

### Examples

```
Order = "Cusomer.ID"
```

## SQL-Order Descending

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

Use this property to reverse the order of the loaded key when using the SQL-Order property. The default value 'false' for this property means that the keys are ordered ascending. When setting the property to 'true', the order will be descending.

### Examples

```
SQL-Order Descending = "true"
```

```
SQL-Order Descending = "false"
```

## See also

SQL-Order

## SQL-Where

### Type

SQL Expression

### Since

1.1

### Description

Use this property to filter the result-set of the SQL-query performed by this SQL-Keyloader. The WHERE-condition defined by this optional property will be appended to the generated SQL-statement every time the loader is executed. The property expects a SQL-expression with a boolean result-type (means only use true, false or comparators in the WHERE-property).

### Examples

```
SQL-Where = "Products.State = '1'"
```

Only load products with state "1"

## Trim

### Type

Constant Boolean

### Since

1.2

### Description

With the Trim-property you can force the loader to trim the IDs generated by the SQL-query after they have been read out of the query-result. Use this property to avoid unnecessary white spaces after the IDs if the if database produces them.

### Examples

```
Trim = "true"
```

```
Trim = "false"
```

## Unit

### Type

Constant Boolean

### Since

2.6.0

### Description

This property defines if the keys loaded by this loader are visible. Invisible keys will not appear in the Query Editor, Analyzer or other, external tools.

### Examples

```
Visible = "FALSE"
```

# SQL-Attribute

---

## Description

### Type

Constant String

### Since

2.6.0

### Description

This property allows to generate a description for the attribute.

## Format

### Type

Constant String

### Since

2.0

### Description

With this property you can convert the results of the SQL Expression into strings. Depending on the result-type of the expression, this property must contain a Number-Format or Date-Format and will then convert the info a string using this format before storing it as attribute or use it to search a linked key.

If you don't specify any format, the result of the SQL-Expression will be converted using the standard-format.

### Examples

```
Format = "#,###,##0"
```

```
Format = "0.00"
```

```
Format = "dd. MM. yyyy"
```

```
Format = "dd. MM. yyyy HH: mm: ss"
```

## Ignore Missing Targets

### Type

Constant Boolean

### Since

2.1

### Description

When generating this Attribute as a link, the System will search for keys in the Target Attribute with an ID matching the result of the SQL Expression. If no target-key was found, the system will stop loading the model and show an error.

You can change this behavior and ignore missing link-target by setting this property to "true".

### Examples

```
Ignore Missing Targets = "true"
```

## Load Distinct

### Type

Constant Boolean

### Since

2.6.0

### Description

By default, if the result of the SQL statement generating the keys contains multiple equal values for the same attribute, the loader only adds a single attribute value to the keys and skips all following, equals values.

By setting this property to "true", the loader will also add equal values and the key may contain multiple and equal values for the same attribute.

### Examples

```
Load Distinct = "true"
```

```
Load Distinct = "false"
```

## Name

### Type

Constant String

### Since

2.0

### Description

Every attribute of a key must have a name. Define the name of an attribute with this property.

### Examples

```
Name = "Color"
```

## Null-Value

### Type

Constant String

### Since

1.0

### Description

This optional property determines which values will be used for attributes whenever the SQL Expression for the attribute returns NULL. If you don't set this property, all attributes with NULL values will be dropped.

### Examples

```
Null-Value = "Unnamed"
```

## Relink-Attribute

### Type

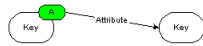
Constant String

### Since

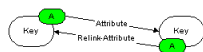
2.0

## Description

Keys from one dimension can be linked to keys from other dimensions as attributes (see the Dimension property). Whenever a key is linked to another one you can use the attribute inside your expression like you would use other attributes, but they will have keys as result. E.g the expression "Customer.Product" would result to the Product(s) of the current customer (when the Customers have been loaded with a linked attribute named "Product").



By default, links are only uni-directional. The first (source) key (which attribute-loader defined the link with its Dimension property) would point to the second (target) key but not vice versa. If you also want the target key to point backwards to source key, you can define the name of the backward-linking attribute with this property. E.g. if the Relink-Attribute for the sample above would be "Customers", you now could use the expression "Product.Customers".



## Examples

Relink-Attribute = "Customers"

## SQL-Expression

### Type

SQL Expression

### Since

1.2

### Description

The mandatory property SQL-Expression defines the SQL Expression which generates the value for the attribute. The expression must be a valid SQL-fragment for the database used by this loader. You may use simple table/column expression or more complex expressions with concatenations and formulas.

### Examples

SQL-Expression = "CUSTOMER. NAME"

Load the customer name as attribute value

## Target Attribute

### Type

Constant String

## Since

2.0

## Description

This property allows the more advanced linking between keys of two different dimensions.

The "Type" property of an Attribute-Loader allows the simple linking between two keys by searching the target-key with its id. But sometimes the source-Keyloader can't link to the target-keys with the id, because it is not possibility to load the target-id from its database or tables (or it might be too time-consuming because a couple of database-join would become necessary then).

In this case you might try to connect two keys by searching the target-key with one of its attributes which is known by the source-key. E.g. if the table you load the source-keys "Product" from has a column "MANUFACTURER\_ID", you can link the products to their manufacturers by searching them with their ManufacturerID. Of course, the manufacturers must have a so called attribute, otherwise the system would raise an error.

## Examples

```
Target-Attribute = "CustomerID"
```

Search the target-keys by its Customer-ID

## Trim

### Type

Constant Boolean

### Since

1.2

### Description

With the Trim-property you can force the attribute-loader to trim the IDs generated by the SQL-query after they have been read out of the query-result. Use this property to avoid unnecessary white spaces after the attributes if the if database produces them.

### Examples

```
Trim = "true"
```

```
Trim = "false"
```

## Type

### Type

Constant String

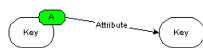
## Since

2.2

## Description

Attributes can either have a basic type (string, integer, double, boolean, date) or point to a key from another dimension (then called a "link"). You can define the type of an attribute with this, it will automatically converted when loading the attribute. By default, the type of an attribute is "string".

If you want the attribute to point to another dimension, you must specify the target-dimension with this property by using the target dimension-name as type-name. The loader will then search for a key in the target-dimension with the ID generated with the property SQL-Expression and generate a link to this key.



If no key with the ID was found, the system will raise an error and stop loading the model. This behaviour can be changed with the property Ignore Missing Targets.

## Examples

```
Type = "String"
```

```
Type = "Integer"
```

```
Type = "Customer"
```

This attribute will become a link to the dimension "Customer"

## See also

Ignore Missing Targets, Relink-Attribute

## Unique

### Type

Constant String ('true', 'false' or 'auto')

### Since

2.0

### Description

An attribute of a dimension can be a unique attribute. Only unique attributes can be used in cubes to map dimension via this attribute (e.g. a cube could map the dimension "Product" with the unique attribute "ProductID" to a column "PRODUCT\_ID" of the fact-table). If an attribute is unique, only one key per dimension may have a specific value stored in this attribute - e.g. it would be forbidden to have two different keys with the same "ProductID".

There are two different ways to define that an attribute is unique:

- Set the property "Unique" to true: The attribute is expected to be unique. If the same attribute-value is loaded for two different keys, the loader will raise an error and stop loading the model.
- Set the property "Unique" to auto: The loader will automatically find out, if the attribute is unique. This is a bit dangerous, because a unique attribute could become non-unique later and some cube-mapping couldn't work anymore.

If you set the property to false, the attribute won't be unique. Note that unique attributes use a bit more system-resources, so you should avoid to have unique attributes unless you want to use them for mapping cubes.

## Examples

```
Unique = "true"
```

```
Unique = "false"
```

```
Unique = "auto"
```

## Unit

### Type

Constant Boolean

### Since

2.6.0

### Description

This property defines if the generated keys are visible, by default all keys are visible. If a key is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

### Examples

```
Visible = "FALSE"
```

# Time-Keyloader

---

## Exclude Pattern

### Type

Date format

### Since

1.2

### Description

The Time-Keyloader allows the exclusion of keys when generating them (exclusion means you don't want to generate keys for special dates or days, e.g. weekends). The exclusion needs two different properties: The Exclude-Pattern (this property) and the Exclude Values. This property defines an additional Date format, which will generate a string for each key. If this string is contained in the Exclude-Values, the key won't be added to the dimension. The Exclude-values must contain a comma-separated list of strings.

### Examples

```
Exclude-Pattern = "eee"
```

```
Exclude-Values = "Sat, Sun"
```

This example avoids adding weekend-days to the dimension, because the pattern "eee" (weekday) results to "Sat" or "Sun" for weekend-days, which are included in the Exclude-values.

### See also

[Exclude Values](#)

## Exclude Values

### Type

Constant String

### Since

1.2

### Description

The Time-Keyloader allows the exclusion of keys when generating them (exclusion means you don't want to generate keys for special dates or days, e.g. weekends). This property is used for

the exclusion. See the description of the Exclude-Pattern property for a detailed description and examples.

### See also

Exclude Pattern

## Language

### Type

Constant String

### Since

2.2.6

### Description

This property determines the language for the generated time keys (eg. for the names of the weekdays). By default, the server used the local server language for the generation, but you may override the language of a load by setting this property to a ISO language name.

### Examples

```
Language = "en"
```

Generates English time patterns.

```
Language = "de"
```

Generates German time patterns.

## Level Name

### Type

Constant String

### Since

2.2

### Description

Each level of a dimension has a name, which is unique inside the dimension. When generating keys with a Time-keyloader , this property defines the name of the level the keys are loaded into.

If no name is defined, the system will generate a level-name automatically or use the name defined in other Keyloaders or the dimension itself. Note that different names for the same level

are forbidden, so the system may report an error if the name in this property differs from other names in other loaders for the same level.

## Examples

```
Level Name = "Year"
```

## Locale Format

### Type

Date format

### Since

1.2

### Description

Some properties of the Time-Keyloader expects date- and time-values (the "Start" and "Stop" property). This properties will expect dates in the local-format of the server (e.g. an US/English server will only accept dates in the American format, a German only in the German and so on). Use this property to define the date-format you want to use in all other properties and make the Keyloader independent of the server-language.

### Examples

```
Locale Format = "dd.MM.yyyy"
```

Now accepts Start and End in the "dd.MM.yyyy" format

```
Locale Format = "yyyy/dd/MM"
```

Now accepts Start and End in the "yyyy/MM/dd" format

## Mode

### Type

Constant String ('insert\_and\_extend', 'insert' or 'extend')

### Since

2.0

### Description

instantOLAP supports three different modes for Keyloaders: insert\_and\_extend, insert and extend. This mode defines the behavior of instantOLAP for keys loaded from the Keyloader when the key already exists in the dimension:

- **insert:** Only non-existing keys may be loaded and inserted with this Keyloader. If a key already exists, the Keyloader will raise an exception and loading of the complete model will be stopped.
- **insert\_and\_extend:** The loader will ignore the fact that a key already exists and add the new attributes loaded with this execution to the existing key. You can use this mode to create new keys and/or extend existing keys with new attributes.
- **extend:** The loader will only add the new attributes loaded by this loader to existing keys. There will be no new keys being generated by this loader. You can use this mode to add some attributes to keys loaded prior by other loaders (e.g. to add a ID or text to keys loaded from another database).

## Examples

```
Mode = "insert_and_extend"
```

```
Mode = "insert"
```

```
Mode = "extend"
```

## Parent

### Type

Date format

### Since

1.0

### Description

Whenever a key is created by the Time-Keyloader it will be sorted into the dimension's hierarchy. To sort it in, the parent of a key must be defined with the Parent-option, which holds a Time-Pattern. For every generated key, the parent-id will also be generated with this pattern and the parent-key will be searched. If the parent-key was found, the new key will be attached to it (become a new child of it) - if the parent key wasn't found, the loader will drop the new key. Note that loader does not generate the parents, it only searches for them.

In most cases, you won't need this property because a Key-Loader can determine its own parent-pattern when it is embedded into another Time-Keyloader (e.g. by using drag&drop inside the workbench).

If no Parent-pattern is defined and the Keyloader is not embedded into another one, all keys are placed directly under the root-key of your dimension.

### Examples

```
Parent = "yyyy"
```

Search a parent with the year as ID

## See also

Start

## Pattern

### Type

Date format

### Since

1.0

### Description

A Time-Keyloader generates keys for a period of time. Each key must have a unique ID (a string) which only appears once in a dimension. The "Pattern" property defines how the Time-Keyloader will generate the IDs of the keys. It must contain a Time-Pattern, which converts the date into a string (the ID).

The pattern also defines the step-width of a Time-Keyloader. E.g. if the pattern is "yyyy" (a four-digit year identifier), the Time-Keyloader will generate keys in year-steps. If the pattern was "MMM/yy" (month and year), the loader will generate keys in month-steps and so on.

### Examples

```
Pattern = "yyyy"
```

Create keys for years (... , 2004, 2005, ...)

```
Pattern = "MMM/yyy"
```

Create keys for months (... , Jan/2004, Feb/2004, ...)

```
Pattern = "dd. MM. yyyy"
```

Create keys for days (... , 01.01.2004, 02.01.2004, ...)

### See also

Stopshift, Start

## Start

### Type

Date format

### Since

1.0

## Description

This property defines the timestamp from which the Time-Keyloader will start generating its keys. The property expects a Time-Pattern with constants and variable elements (which will be replaced by the corresponding values of the current date) in the locale date-format. If you don't want to use the locale format of the server, you can override the format with the Locale Format pattern of this Keyloader.

If no "Start" property is defined, the Keyloader will use the "Start" property from the parent Keyloader (if this Keyloader is encapsulate in another one). If this Keyloader is not encapsulated and no Start is defined, the Keyloader will use the current timestamp as start.

## Examples

```
Start = "01.01.2004"
```

Time-Keyloader starts generating from 01.01.2004

```
Start = "01.01.yyyy"
```

Time-Keyloader starts generating from 01.01 in the current year

```
Start = "dd.MM.yyyy"
```

Time-Keyloader starts generating from today

## See also

Stopshift, Locale Format, Startshift

## Startshift

### Type

Constant Integer (Number of seconds)

### Since

1.2

### Description

The "Start" property only allows to set the start-time to a constant year, month or day or to the current year, month or day. It is not possible to create a Time-Keyloader which e.g. starts yesterday, because this is a relative setting. This property allows to shift the start-time forward or backwards by setting the number of seconds (positive or negative) to shift.

### Examples

```
Start = "dd.MM.yyyy"
```

```
Startshift = "-86400"
```

This setting lets the loader start yesterday, because the original start (today) defined by the Start-property is shifted backwards by  $24 * 60 * 60$  (= 86400) seconds.

### See also

Stopshift, Start

## Stop

### Type

Date format

### Since

1.0

### Description

This property defines the timestamp until which the Time-Keyloader will start generating its Keys. The property expects a Date format with constants and variable elements (which will be replaced by the corresponding values of the current date) in the locale date-format. If you don't want to use the locale format of the server, you can override the format with the Locale Format pattern of this Keyloader.

If no "Stop" property is defined, the Keyloader will use the "Stop" property from the parent Keyloader (if this Keyloader is encapsulate in another one). If this Keyloader is not encapsulated and no End is defined, the Keyloader will use the current timestamp as end.

### Examples

```
Stop = "31.12.2004"
```

Time-Keyloader generates till 31.12.2004

```
Stop = "31.12.yyyy"
```

Time-Keyloader generates till 31.12 of the current year

```
Stop = "dd.MM.yyyy"
```

Time-Keyloader generates Keys till today

```
Stop = ""
```

Time-Keyloader generates Keys till today

## Stopshift

### Type

Constant Integer (Number of seconds)

## Since

1.2

## Description

The "Stop" property only allows to set the stop-time to a constant year, month or day or to the current year, month or day. It is not possible to create a Time-Keyloader which e.g. stop tomorrow, because this is a relative setting. This property allows to shift the stop-time forward or backwards by setting the number of seconds (positive or negative) to shift.

## Examples

```
Stop = "dd. MM. yyyy"
```

```
Endshift = "86400"
```

This setting lets the loader stop tomorrow, because the original end (today) defined by the "Stop" property is shifted forward by  $24 * 60 * 60$  (= 86400) seconds.

## See also

Start, Stop

## Unit

## Type

Constant Boolean

## Since

2.6.0

## Description

This property defines if the generated keys are visible, by default all keys are visible. If a key is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

## Examples

```
Visible = "FALSE"
```

# Time-Attribute

---

## Description

### Type

Constant String

### Since

2.6.0

### Description

This property allows to generate a description for the attribute.

## Ignore Missing Targets

### Type

Constant Boolean

### Since

2.6.0

### Description

When generating this Attribute as a link, the System will search for keys in the Target Attribute with an ID matching the generated attribute pattern. If no target-key was found, the system will stop loading the model and show an error.

You can change this behavior and ignore missing link-target by setting this property to "true".

### Examples

```
Ignore Missing Targets = "true"
```

## Name

### Type

Constant String

### Since

2.0

## Description

Every attribute of a key must have a name. Define the name of an attribute with this property.

## Examples

```
Name = "Weekday"
```

## Pattern

### Type

Date format

### Since

1.0

## Description

The mandatory property Pattern defines the Pattern (Date format) which generates the value for the attribute by converting the key's date into a string.

## Examples

```
Pattern = "eee"
```

This attribute will contain the weekday

## Relink-Attribute

### Type

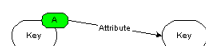
Constant String

### Since

2.0

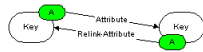
## Description

Keys from one dimension can be linked to keys from other dimensions as attributes (see the Dimension property). Whenever a key is linked to another one you can use the attribute inside your expression like you would use other attributes, but they will have keys as result. E.g the expression "Time.Weekday" would result to the Weekday(s) of the current customer (when the Weekdays have been loaded with a linked attribute named "Weekday").



By default, links are only uni-directional. The first (source) key (which attribute-loader defined the link with its Dimension property) would point to the second (target) key but not vice versa. If you also want the target key to point backwards to source key, you can define the name of the

backward-linking attribute with this property. E.g. if the Relink-Attribute for the sample above would be "Days", you now could use the expression "Weekdays.Days".



## Examples

```
Relink-Attribute = "Days"
```

## See also

Name, Type

## Target Attribute

### Type

Constant String

### Since

2.6.0

### Description

This property allows the more advanced linking between keys of two different dimensions.

The "Type" property of an Attribute-Loader allows the simple linking between two keys by searching the target-key with its id. But sometimes the source-Keyloader can't link to the target-keys with the id, because it is not possibility to load the target-id from its database or tables (or it might be too time-consuming because a couple of database-join would become necessary then).

In this case you might try to connect two keys by searching the target-key with one of its attributes which is known by the source-key. E.g. if the table you load the source-keys "Product" from has a column "MANUFACTURER\_ID", you can link the products to their manufacturers by searching them with their ManufacturerID. Of course, the manufacturers must have a so called attribute, otherwise the system would raise an error.

## Examples

```
Target-Attribute = "CustomerID"
```

Search the target-keys by its Customer-ID

## Type

### Type

Constant String

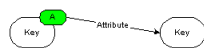
## Since

2.2

## Description

Attributes can either have a basic type (string, integer, double, boolean, date) or point to a key from another dimension (then called a "link"). You can define the type of an attribute with this, it will automatically converted when loading the attribute. By default, the type of an attribute is "string".

If you want the attribute to point to another dimension, you must specify the target-dimension with this property by using the target dimension-name as type-name. The loader will then search for the key in the target-dimension with the ID generated by Pattern attribute and generate a link to this key.



If no key with the ID was found, the system will raise an error and stop loading the model.

## Examples

```
Type = "String"
```

```
Type = "Integer"
```

```
Type = "ProductionTime"
```

This attribute will become a link to the dimension "ProductionTime"

## See also

Pattern, Relink-Attribute

## Unique

### Type

Constant String ('true', 'false' or 'auto')

### Since

2.0

### Description

An attribute of a dimension can be a unique attribute. Only unique attributes can be used in cubes to map dimension via this attribute (e.g. a cube could map the dimension "Product" with the unique attribute "ProductID" to a column "PRODUCT\_ID" of the fact-table). If an attribute is unique, only one key per dimension may have a specific value stored in this attribute - e.g. it would be forbidden to have two different keys with the same "ProductID".

There are two different ways to define that an attribute is unique:

- Set the property "Unique" to true: The attribute is expected to be unique. If the same attribute-value is loaded for two different keys, the loader will raise an error and stop loading the model.
- Set the property "Unique" to auto: The loader will automatically find out, if the attribute is unique. This is a bit dangerous, because a unique attribute could become non-unique later and some cube-mapping couldn't work anymore.

If you set the property to false, the attribute won't be unique. Note that unique attributes use a bit more system-resources, so you should avoid to have unique attributes unless you want to use them for mapping cubes.

## Examples

```
Unique = "true"
```

```
Unique = "false"
```

```
Unique = "auto"
```

## Unit

### Type

Constant Boolean

### Since

2.6.0

### Description

This property defines if the generated attributes are visible, by default all attributes are visible. If an attribute is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

### Examples

```
Visible = "FALSE"
```

# Number-Keyloader

---

## Format

### Type

Constant String

### Since

2.2.6

### Description

This property defines the number format for the generated key-IDs.

### Examples

```
Format = "0"
```

Generates "1", "2", "3", ..

```
Format = "000"
```

Generates "001", "002", "003", ..

## Level Name

### Type

Constant String

### Since

2.2

### Description

Each level of a dimension has a name, which is unique inside the dimension. When loading keys with a Number-Keyloader, this property defines the name of the level the keys are loaded into.

If no name is defined, the system will generate a level-name automatically or use the name defined in other Keyloaders or the dimension itself. Note that different names for the same level are forbidden, so the system may report an error if the name in this property differs from other names in other loaders for the same level.

## Examples

```
Level Name = "LINE"
```

## Max

### Type

Constant Integer

### Since

1.2

### Description

This property defines the end of the range for the generated numbers (key). The last generated key will have this number.

### Examples

```
Max = "1000"
```

### See also

Min

## Min

### Type

Constant Integer

### Since

1.2

### Description

This property defines the start of the range for the generated numbers (key). The first generated key will have this number.

### Examples

```
Min = "1"
```

### See also

Max

## Unit

### Type

Constant String

### Since

2.6.0

### Description

This property defines if the generated keys are visible, by default all keys are visible. If a key is invisible, it will not appear in the Query Editor, Analyzer or other, external tools.

### Examples

```
Visible = "FALSE"
```

# SQL-Cube

---

## Active

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

If this property is set to "false", the cube will be deactivated and no longer used. The default setting is "true".

### Examples

```
Active = "true"
```

```
Active = "false"
```

## Database

### Type

Constant String

### Since

1.0

### Description

The mandatory "Database" property defines from which database a cube will load its data. The Database-property must contain the name of a prior defined database.

### Examples

```
Database = "SalesDB"
```

## Distinct

### Type

Constant String

**Since**

2.6.0

**Description**

This property allows to define the description for this cube.

**Distinct****Type**

Constant Boolean

**Since**

2.2.2

**Description**

By setting this property to "true" the SQL-Cube will generate all its SQL-statements with the "DISTINCT" clause and the results will only contain unique result-lines.

This property is useful whenever you query non-aggregated facts out of a cube in combination with the line-dimension (by using the ROWNUM() function in your queries) and don't want double results in your report. For normal pivot-tables this property has no effect but may cost some execution-time when the database executes the SQL-statement.

**Examples**

```
Distinct = "false"
```

E.g. generates the SQL-statement "SELECT SALES.TIME, SALES.AMOUNT FROM..."

```
Distinct = "true"
```

E.g. generates the SQL-statement "SELECT DISTINCT SALES.TIME, SALES.AMOUNT FROM..."

**Enable Load****Type**

Constant Boolean

**Since**

1.2

## Description

SQL-Cubes have both abilities to load and store data out of or into databases. Both features can be enabled or disabled for each cube. The "Enable Load" property is used for enabling or disabling the loading of data.

The default value for this property is "true". Each cube with load enabled will load the bound facts out of its datasource if the fact- and dimension-mapping matches the searched coordinate. If the property is set to "false" this cube won't return any data.

## Examples

```
Enable Load = "true"
```

```
Enable Load = "false"
```

## See also

Enable Store

## Enable Store

### Type

Constant Boolean

### Since

1.2

## Description

SQL-Cubes have both abilities to load and store data out of or into databases. Both features can be enabled or disabled for each cube. The "Enable Store" property is used for enabling or disabling the storage of data.

The default value for this property is "false". If the property is set to "true", this cube will write back data into its database if the facts and all dimension of the (to store) coordinate match correctly.

Storing data will only work when all dimension of the coordinate are mapped with simple SQL-expression (pure table/column constructs). Otherwise, the cube won't be able to calculate the needed value for the column(s) of the bound tables. E.g. a binding like "PRODUCT.ID" could be used (because the system will store the product-id directly into it), but a SQL-expression like "SUBSTR( SALES.DATE, 1, 4 )" couldn't be converted into a valid column-value.

## Examples

```
Enable Store = "true"
```

```
Enable Store = "false"
```

## See also

Enable Load

## Match

### Type

Boolean

### Since

2.6.0

### Description

If a cube contains stores (materialized views of the cube content), the stores will replace the cube itself and it becomes invisible. This is usually only useful if the store(s) contain the same data as the cube and span the same number of dimensions as the original cube, otherwise the model will behave different and data may be missing when the stores are used instead of the cube.

If you want to materialize only parts of the original cube and keep the cube in order to provide all data not contained in its stores, you must set this property to "true". Then, the model will contain all stores of this cube **and** the cube itself, which is then placed **behind** its stores.

Whenever a reports needs data, the model will try to load it from one of the cube stores and, if no store contain the data, query the cube itself.

### Examples

```
Keep Adhoc Cube With Stores = "TRUE"
```

## Match Expression

### Type

Boolean

### Since

1.2

### Description

Each cube may have a match-condition which exactly defines what data should be loaded out of it. This property expects a boolean-expression. This expression will be applied to each coordinate to determine if its data may be found here. See the general description of cubes for further details.

## Examples

```
Match = "HASKEYS( Time:'2004' )"
```

Only load data for the year 2004

```
Match = "HASLEVEL( Time, 2 )"
```

Only load data for the second level of Time

## See also

Offline Match

## Match Mode

### Type

Constant String ('best', 'first', 'exact' or 'incomplete')

### Since

2.2

### Description

With the "Match Mode" you can control if this cube will be used to load a value. Each value belongs to a coordinate which consists out of one fact and a number of keys. The Match Mode defines, how much of the coordinate must be mapped by a cube to be served:

- **First:** If this is the first cube maps the needed fact and at least one other key, it will be used for loading the value.
- **Exact:** If this is the first cube maps the needed fact and all other keys (not more or less), it will be used.
- **Best (default):** Like "First", but if any other cube exists which matches more than this cube, it will be skipped and the other cube will be used instead.
- **Incomplete:** In this mode, the data will be loaded out of this cube if the fact and at least one dimension of the coordinate is mapped. But in difference to "First" or "Best", this System will not stop searching for the source of the value and continue matching the other cubes. This allows to load data from one cube **and** other cubes in the case that the value wasn't found in this cube. This mode replaces the "complete" property in prior versions.

If a cube maps only parts of a dimension (e.g. only one or more levels or only keys with special attributes), it may not match a coordinate and could pass the coordinate on to other cubes.

## Examples

```
Match Mode = "best"
```

```
Match Mode = "first"
```

```
Match Mode = "exact"
```

```
Match Mode = "incomplete"
```

### See also

Complete

## Name

### Type

Constant String

### Since

2.0

### Description

This property defines the name of the cube. The property is optional and only used for the display inside the workbench and debugging-output.

### Examples

```
Name = "SALESCUBE"
```

## SQL-Order

### Type

SQL Expression

### Since

2.0

### Description

With this property you can add an ORDER BY clause to all SQL-statements generated by this SQL-cube.

Adding orders can be especially interesting when loading lists with a mapped Line-Dimension, because then you might want the result to be in a special order. If you work without a Line-Dimension, the order of the result would be without any effect and this property becomes useless.

The default order when sorting the records is ascending. Use the "SQL-Order Descending" property to switch the order to descending if needed.

### Examples

```
SQL-Order = "ORDER. ORDER_NO"
```

With this property-value the generated SQL-statement would be:

```
SELECT ... FROM ORDER, ... WHERE ... ORDER BY ORDER.ORDER_NO
```

### See also

SQL-Order Descending, SQL-Where

## SQL-Order Descending

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

Use this property to reverse the order of the loaded records when using the SQL-Order. The default value 'false' for this property means that the records are ordered ascending. When setting the property to 'true', the order will be descending.

### Examples

```
SQL-Order Descending = "true"
```

```
SQL-Order Descending = "false"
```

### See also

SQL-Order

## Table Order

### Type

Constant String

### Since

2.2

### Description

This property is for optimizing issues only. When generating SQL-statements, the SQL-generator will use this optional property to define the order of the tables in the FROM-clause of the generated SELECT statement.

In some databases, the order of the tables in the statements can have a big effect on the performance because the order influences the used indices.

## Examples

```
Table Order = "Products, Customers, Productgroups"
```

## SQL-Where

### Type

SQL Expression

### Since

1.2

### Description

If you want the SQL-cube to load only a part of the database-table you can add a filter (in form of a SQL WHERE-clause) to all generated SQL-statements of this cube.

In some cases it will be more efficient to use Aliases with WHERE-clauses instead of this property and use the aliases inside this cube. This allows the central management of the table-filter.

## Examples

```
SQL-Where = "INVOICE.STATE = 1"
```

With this property-value the generated SQL-statement would be:

```
SELECT ... FROM ORDER,... WHERE ... AND INVOICE.STATE = 1 ...
```

## See also

SQL-Order

## Unit

### Type

Constant Boolean

### Since

2.6.0

### Description

This property defines if the cube is visible. Invisible cubes will not appear in the Query Editor, Analyzer or other, external tools.

## Examples

```
Visible = "FALSE"
```

# SQL-Dimension

---

## Attribute

### Type

Constant String

### Since

2.0

### Description

Dimensions can be bound in two different ways, with or without attributes.

If you leave this property empty, the dimension defined in the Dimension property will be bound directly (without attribute) to the database (to the SQL-expression defined in the SQL-Expression property). To bind dimensions "directly" means that the cube will search keys with IDs matching the return-value of the SQL-Expression when loading the data. Because all keys have IDs this will bind all keys in a dimension (at all hierarchy levels, including the root key).

But usually you don't want to bind all hierarchy levels with the same SQL-expression to the database and you don't want to bind the root-key of a dimension:

- Different levels of a hierarchy are bound with different SQL-statements (to generate different aggregations) and
- The root level is not bound in the normal case (so the SQL-generator will leave out the dimension for root keys and simple aggregate the values)

Both can be easily done with attributes. Instead of binding the complete hierarchy to a SQL-expression, only keys with a special attribute are bound (the name of the attributes must be defined in this property). Then the SQL-generator only maps keys having this attributes and searches for keys by their attribute-value instead of their ID (which is very useful when using technical ID for keys).

For example a dimension "Product" could have a level 1 containing product-groups with a attribute named "GroupID" and a second level with the products and an attribute name "ProductID". Both attributes could have separate mappings to the database.

If multiple bindings for a dimension exist in a SQL-cube, each with a different attribute, the SQL-generator will create multiple statements and load the data for each key with its own binding.

### Examples

```
Attribute = "ProductID"
```

Only bind to keys with "ProductID" (e.g. to a SQL-expression SALES.PRODUCT\_ID)

**See also**

Dimension, Level

**Dimension****Type**

Constant String

**Since**

1.0

**Description**

The mandatory property `Dimension` sets the name of the mapped dimension for this mapping-entry.

Dimensions can be bound with two different styles, with or without attributes. See the `Attribute` property for a more detailed description. If no attribute is used, the result of the SQL Expression will be mapped directly to this dimension and the values must exactly match the IDs of the dimension-keys.

**Examples**

```
Dimension = "Product"
```

**See also**

Attribute, Level

**Format****Type**

Constant String

**Since**

1.2

**Description**

The `format` property allows to format the result of the SQL-expression after it is read from the database and before the engine searches the corresponding key in the dimension. Depending on the type of the SQL-Expression for this binding, the format must be a `Number-Format` or `Date-Format`.

**Examples**

```
Format = "0"
```

```
Format = "dd. MM. yyyy"
```

## Key

### Type

Constant String

### Since

2.2

### Description

This property allows to map a single key of a dimension instead of the whole dimension or a part of it (all keys with a specific attribute or a specific level of a dimension).

Mapping single keys is very useful in combination with the Where-Condition which can also be defined for a dimension-mapping. This allows to add specific Where-Condition for single keys of the dimension.

### Examples

```
Key = "Best Customers"
```

### See also

Where

## Level

### Type

Constant String

### Since

2.2

### Description

With this property you can reduce the mapping for a dimension to a single level of the dimension. This property must contain an existing level-name.

If this property is left empty, the whole dimension is mapped (unless you use the property Attribute to map only keys with a certain attribute). If this property is set and the property "Attribute" is used to map only keys with a certain attribute, only keys from the defined level containing the attribute are mapped.

### Examples

```
Level = "PRODUCT"
```

**See also**

Attribute, Dimension

**Mode****Type**

Constant String ('complete', 'select' or 'where')

**Since**

2.5

**Description**

The "Mode" property determines if the dimension-mapping is used in the SELECT-Clause, in the WHERE-Clause or (which is the default) in both clauses of the generated SQL statements.

When a mapping is only used for the WHERE-Clause, the generated SQL will be filtered by the current selection of the report but the result will not be mapped to the corresponding keys of the filtered dimension. This can be quite useful, e.g. if you want to use a dimension only for filtering the SQL but don't want to summarize the result manually. You can also use dimension mapped in the 'where' mode to filter lists.

**Null-ID****Type**

Constant String

**Since**

2.0

**Description**

This property defines, which key (the ID or the attribute-value) will be bound when the SQL-expression returns NULL. If no Null-ID is defined, the NULL values will be dropped and stays unbound.

**Examples**

```
Null-ID = "UnknownProduct"
```

**Omit-Percentage****Type**

Constant Double (between 0.0 and 1.0)

## Since

1.2

## Description

The Omit-Percentage property is used by the SQL-generator for optimizing the generated statements. The Omit-Percentage controls if the generator add WHERE clauses for the SQL-statements to filter to searched keys out of the database or perform a manual filtering after reading the result out of the database.

Adding filters to the SQL-statements (in form on IN-clauses with lists) is very fast for loading database for a subset of all possible values but generates very long statements in the case of a large number of keys. In the worst case, the generated statement become too long and has to be split in multiple statements. Dropping the filter generates shorter statements but load more data out of database than needed.

The property controls the percentage of keys from which the generator starts dropping the filter and load the complete dimension-content. Its default-value is 1.00 (100%). E.g. if you load 8 or more keys from a dimension with 10 keys, the generator will create a statement without a filter (for this dimension). If you load less, the generate will append filters.

Note that this property can also be used for cube-optimization because querying tables without filters is faster in the case the table owns no index for the queried column.

## Examples

Imagine a dimension with 10 keys (products). Each product is bound with its attribute "ProductID" to the SQL-expression "SALES.PRODUCT\_ID". The user queries 8 of the 10 products with his query now. If the Omit-Percentage was set to "0.9", this will generate the following SQL query:

```
Omit-Percentage = "0.9":
```

```
SELECT SUM( AMOUNT ), PRODUCT_ID FROM SALES WHERE PRODUCT_ID IN ( 1,
2, 3, 4, 5, 6, 7, 8 )
```

All selected products are listed in the IN filter and the database only returns values for these 8 products.

If the Omit-Percentage has a value less than 0.8, the generator drops the IN-filter and generates the following statement:

```
Omit-Percentage = "0.7"
```

```
SELECT SUM( AMOUNT ), PRODUCT_ID FROM SALES
```

This load data for all products from table. The cube ignores all product-data for the unwanted products. Because the database has no filter to perform and the statement is smaller, this can be much faster than the statement above.

## Operator

### Type

Constant String ( '=' or 'BETWEEN' )

### Since

2.2.3

### Description

With this property you can define the operator which the SQL-generator will use to map the dimension-values with the expression. Usually the SQL-generator uses the operator '=', which is the default value for this property.

You can also use the operator 'BETWEEN' - the generator will then use the lowest and the highest value (alphanumeric order) and create a BETWEEN-clause with them. This can bring better performance, especially when mapping a time-dimension.

### Examples

```
Operator = '='
```

will e.g. generate a SQL-statement like

```
SELECT ... WHERE F_SALES.YEAR IN ( 2004, 2005, 2006 )
```

```
Operator = 'BETWEEN'
```

will e.g. generate a SQL-statement like

```
SELECT ... WHERE F_SALES.YEAR BETWEEN 2004 AND 2006 )
```

### See also

Swap Expressions

## Prefix

### Type

Constant String

### Since

2.5

### Description

When loading keys from a table, the SQL-Keyloader will add the prefix defined in this property to every ID of every generated key. You can use this property to make keys unique inside a dimension, e.g. when you load products and product groups into the same dimension and both

have equals IDs. Then you could add prefixes like 'group\_' and 'product\_' to the loaders. This is the same as adding prefixes to IDs using the concat operator of SQL.

When adding prefixes to keys, you may also need to add them to the dimension mappings inside of SQL-cubes.

### Example

```
Prefix = 'Product_'
```

## SQL-Expression

### Type

SQL Expression

### Since

1.0

### Description

This property holds the SQL-Expression the Dimension (or dimension-Attribute) is bound to. The SQL-generator will use this expression to select the key-values out of the fact table and to group and filter by the keys. Depending on the binding style (by ID or by attribute) the expression must return a value matching the ID or a special attribute of the keys. See the description of the Dimension and Attribute properties for more details.

### Examples

```
SQL-Expression = "SALES.PRODUCT_ID"
```

### See also

Attribute, Dimension

## SQL Where

### Type

Constant String

### Since

2.2

### Description

If any Where-condition is defined in this property, it will be added to the generated SQL-statement whenever this dimension-mapping is used.

Where-condition are very useful in combination with single mapped keys because this allows to map different where-conditions to different keys of a dimension. E.g. you could map a condition

"SALES.AMOUNT >= 1000000" to a key "BigCustomers" and another condition "SALES.AMOUNT < 1000000" to a key "SmallCustomers". Then the user had the possibility to filter his reports by this categories.

## Examples

```
Where = "SALES.AMOUNT >= 1000000"
```

## See also

Key

## Swap Expression

### Type

Constant Boolean

### Since

2.2.3

### Description

Setting this property to "true" will swap the expressions before and after the match operator in the generated SQL-statement. Especially when using the BETWEEN operator this allows to map a dimension to two different columns.

Note that swapping the expression will only allow to map single keys to an expressions. Therefore the cube would not work when the query filters more than one key.

### Examples

E.g. using BETWEEN with an expression like "CONTRACT.STARTDATE AND CONTRACT.ENDDATE" will generate a SQL-expression like

```
SELECT ... WHERE 20060501 BETWEEN CONTRACT.STARTDATE AND  
CONTRACT.ENDDATE
```

Then the select-statement would return all contracts which valid at the selected date 20060501.

### See also

Swap Expressions

## Trim

### Type

Constant Boolean

## **Since**

2.2.6

## **Description**

Setting this property to "true" will trim the values loaded from the mapped expressions before searching the keys in the mapped dimension.

# SQL-Fact

---

## Fact

### Type

Constant String

### Since

2.0

### Description

This mandatory property sets the name of the fact bound inside a cube. Usually you don't have to set this property, because the name of a bound fact will be automatically set when using the Workbench to create a cube, e.g. when dragging a table-column onto the cube-editor.

A cube will load all data for a bound fact (as long as all dimensions of the fact-coordinate are bound, too). See the Match property if you want to reduce the binding of a fact with a condition.

### Examples

```
Fact = "Amount"
```

### See also

Match

## Match

### Type

Boolean

### Since

1.0

### Description

When a fact is bound to a SQL-cube, the cube will load all occurrences of the fact (as long as all dimensions of the loaded fact-coordinate are also bound within this cube). With the Match-property you can reduce the binding of a fact and add some condition to a single fact-binding.

With the match-property it is possible to bind the same fact to different SQL-expression under different conditions - e.g. you could map the same two 12 different columns of a table, each column providing data for one month of the year.

## Examples

```
Match = "Time.Month = 'Jan' "
```

This binding is only valid for the month January

## See also

Fact

## SQL-Expression

### Type

SQL Expression

### Since

1.0

### Description

This property defines the SQL Expression used by the cube to load the (aggregated) fact out of the database. The expression should always use an aggregating SQL-function (SUM, MIN, MAX, AVG, COUNT or a database-specific function), otherwise the SQL-generator wouldn't be able to return aggregated values for higher hierarchy-levels of the dimensions.

### Examples

```
SQL-Expression = "SUM( SALES.AMOUNT ) "
```

## SQL-Where

### Type

SQL Expression

### Since

2.2.6

### Description

This property defines an SQL where expression, which will be appended if (and only if) the fact is queried from the SQL cube.

### Examples

```
SQL-Where = "SALES.TYPE = 1"
```

# Store

---

## Active

### Type

Constant Boolean ('true' or 'false')

### Since

2.6.0

### Description

If this property is set to "false", the store will be deactivated and no longer used. The default setting is "true".

### Examples

```
Active = "true"
```

```
Active = "false"
```

## Algorithm

### Type

Constant Boolean ('true' or 'false')

### Since

2.6.1

### Description

With version 2.5, instantOLAP introduced a very efficient algorithm for loading and aggregating values from a SQL cubes into their materialized views, the "stores". This algorithm aggregates all existing combinations of all mapped dimensions and their keys and therefore stores all possible answers. Because the algorithm eliminates all invalid combinations of dimensions and keys and uses an internal pattern matching algorithm to identify equals aggregations, we call it the "condensed" algorithm.

However, the time this algorithm needs to process a cube grows linear with the number of imported rows from the database and exponentially with the number of included dimensions, because the number of possible answers also grows exponentially with the dimension count.

Therefore, version 2.6.1 introduces a second algorithm for stores, the "indexed" algorithm. This does **not** aggregate the data when loading it, it only creates a copy of the input data and a inverse index for later access. Therefore, it is nearly independent on the number of included dimension and allows a significant larger number of dimensions to be included. But it has much

slower response times than the "condensed" algorithm, which can even be slower than the original datasource (but increasing the "granularity" for "indexed" stores can give a big performance boost, though).

The reason we introduced this new algorithm is its possible combination with condensed stores: When combining a number of condensed stores, which contain a part of the dimensions in different combinations, with a fully dimensional indexed store, you can hold the complete cube in an offline version even if that would be too big for a single store. And the slower indexed store would only be used if the requested combination of dimension is not covered by any condensed cube.

## Examples

```
Active = "condensed"
```

```
Active = "indexed"
```

## See also

Granularity

## Build Timeout

### Type

Constant Integer

### Since

2.6.0

### Description

Whenever a store is build, this property determines the timeout for the loading process. This property expects an integer values which defines the timeout in seconds. The default value is "3600" (one hour).

## Examples

```
Build Timeout = "3600"
```

One hour

```
Build Timeout = "900"
```

15 Minutes

## Cron Pattern

### Type

Constant String (Cron pattern)

## Since

2.6.0

## Description

The Cron pattern of a store determines if and how often it will rebuild the system. The Cron pattern is a time-scheme (a cron pattern string) defining exactly when this will happen.

Whenever a store rebuild is triggered, the system starts loading it and its new version will exist parallel to the old version. At the moment the new store version is completes it will replace the previous.

Note that a store will also be rebuild if the definition of the source database or one of the contained dimensions changes.

## Examples

```
Cron pattern = "* 0 15 30 45"
```

Rebuild every 15 minutes

```
Cron pattern = "* 0 0"
```

Rebuild every day at 00:00

## Granularity

### Type

Constant Integer

### Since

2.6.0

This property contains aggregating / loading hints for the store and allows to influence its load and response time. Depending on the used algorithm, it has a different meaning and effect:

For **condensed cubes** the granularity defines, how many source values a new and aggregated value must have to be aggregated and stored into the store. If less values are necessary to aggregate it, it will not be written onto the disc but be aggregated at runtime whenever a reports accesses the needed value. This reduces the build time and storage size of the store, but also results into slower answer times.

For **indexed cubes** it defines how complex the generated index will be: By default, the store creates one index for each dimension and combines these indices at runtime. By increasing the granularity, the store will create combined indices for two or more dimensions (depending on this property) and has to combine less indices at runtime. However, multidimensional indices need exponential longer to build - but remember to build-time only grows exponentially with the values of this property and not with the number of the dimensions loaded into the store!

## Description

## Examples

```
Granularity = "5" (condensed store)
```

Aggregate subtotals with 4 or less source elements at runtime.

```
Granularity = "3" (indexed store)
```

Build a multidimensional index which combines 3 dimensions to one index key.

## See also

Algorithm

## Match

### Type

Boolean

### Since

2.6.0

### Description

The "Match" property defines - using a boolean expression - which part of the cube will be loaded into the store.

Note that is more efficient to disable single dimension and facts to avoid loading them. The "Match" property should only be used in combination with them in order to reduce the loaded data to a part of a level or dimension, e.g. to load only the last 3 months.

The "Offline Match" will always be logically joined (AND) with the "Match" property of the owner Cube, therefore the store can never hold data which is not part of the Cube itself.

### Examples

```
Match = "HASKEYS( Time:'2004' )"
```

Only store data for the year 2004 offline.

## Name

### Type

Constant String

## Since

2.6.0

## Description

This property defines the name of the store. The name must be unique inside the owner cube or may be left empty (then the system will create a name for the store automatically).

## Examples

```
Name = "Store 1"
```

## Name

## Type

Constant Integer

## Since

2.6.0

## Description

The property "SQL Fetch Size" defines the number of rows the system will fetch from the SQL result when building a store. E.g. a fetch size of "100" means, the server loads 100 rows from the database, processes them, then loads the next 100 rows and so on.

If no fetch size is defined, the system will use the default fetch size of the connection or database driver.

We strongly recommend to set this property, because some database servers load by default the **complete** result set into the memory before it is passed to the client application (in this case instantOLAP). If you work with very large result set (which is not very seldom for stores), this uses a large amount of memory and can cause "Out Of Memory" exceptions in the worst case.

A special case are MySQL server, which **only** accept "-1" or "0" as a valid for this property. "-1" means to load the result set row by row, "0" will load the complete result set into memory.

## Examples

```
SQL Fetch Size = "100"
```

```
SQL Fetch Size = "-1"
```

Recommended for MySQL databases

# Store Dimension

---

## Load Mode

### Type

Constant String ('none', 'rollup', 'query')

### Since

2.6.0

### Description

This property determines how the included dimension will be aggregated inside the store. It accepts three different values: "rollup", "query" and "none" (the default value):

- `_none_`: The dimension will not be part of the store and store will be independent of this dimension.
- `_rollup_`: Only the dimension-mappings for the lowest levels and the special bindings for single keys will be queried from the database. After loading the facts from the lowest levels, they will be rolled up inside the dimension up to the root key.
- `_query_`: The system will create a sql-query for every combination of all mapped dimension levels and keys, execute it and load the result into the store. There is no aggregation or rollup performed by the system. If any level of the dimension is loaded, the root level will automatically also be loaded.

The 'rollup' is usually the fastest and performs much less queries on the database than the 'query' mode. However there are examples where you cannot use this mode. E.g. a fact using COUNT DISTINCT cannot be aggregated using rollup and you have to use the 'query' mode in this case.

### Examples

```
Load Mode = "none"
```

```
Load Mode = "rollup"
```

```
Load Mode = "query"
```

# Store Fact

---

## Aggregation

### Type

Constant String ('auto', 'sum', 'min', 'max', 'avg' or 'none')

### Since

2.6.0

### Description

When loading a fact into a store, the system tries to evaluate its aggregation function from the SQL-expression (this is the 'auto' mode, which is the default value for this property). If the system is not able to evaluate the function (e.g. it may use a database-specific and unknown function), you can set the aggregation manually here.

Especially for "COUNT(DISTINCT ...)" aggregations, you have the possibility to set the aggregation here, because without manual configuration the system would throw an error message (count distinct cannot be aggregated, but sometimes the distinct is only used for database reasons and the values can be summarized with 'sum').

If you set the mode to 'none' the system will not rollup the fact at all and fill all upper levels with NULL values.

### Examples

```
Offline aggregation = "sum"
```

# Formula

---

## Active

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

If this property is set to "false", the formula will be deactivated and no longer used. The default setting is "true".

### Examples

```
Active = "true"
```

```
Active = "false"
```

## Description

### Type

Constant String

### Since

2.6.0

### Description

This property allows to place a description for the formula.

## Expression

### Type

Value

### Since

1.1

## Description

This property defines the expression of a formula. The expression does the real work and calculates the return value of the formula. The expression must be a Value-Expression (an expression returning any type of values, but not keys).

The expression can use any other facts or refer to the current dimension-elements. Formulas can for example be used to perform some aggregation (if the cubes aren't able to) or to calculate missing facts which are not mapped by any cube.

## Examples

```
Expression = "Quantity() / Price()"
```

Calculate the Amount out of two other facts

```
Expression = "SUM( Quantity( CHILDREN( Time ) ) )"
```

Summarize the Quantity of the next time-level

## Fact

### Type

Constant String (fact-name)

### Since

2.0

### Description

Each formulas calculates the values of one special fact. This property holds the name of the fact this formula is calculating.

### Examples

```
Fact = "Amount"
```

This formula calculates the Amount

### See also

Match

## Match

### Type

Boolean

## Since

1.2

## Description

With the Fact-property you can define which fact a formula will calculate. If you only define the fact of a formula, it will calculate this fact for the complete cube (for all elements of all dimensions). But if you want the formula to only calculate a part of the cube (maybe only a special level of one dimension), you can add a match-expression to the formula with this property.

The match-expression limits the usage of this formula to all coordinates for which the expression returns "true" (the match-expression is a Boolean-Expression). The match-expression could for example test the current coordinate for a level of for the occurrence of a special key. Match-expression in formulas are similar to the match-expression in cubes or caches.

## Examples

```
Match = "HASLEVEL( Time, 1 )"
```

Only calculate the level 1 of the Time-dimension

```
Match = "HASKEYS( Product:A )"
```

Only calculate for Product "A"

## See also

Fact

# File-Cache

---

## Active

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

If this property is set to "false", the cache will be deactivated and no longer used. The default setting is "true".

### Examples

```
Active = "true"
```

```
Active = "false"
```

## Cache Line-Numbers

### Type

Constant Boolean ('true' or 'false')

### Since

2.2

### Description

Usually caches would not cache results of lists (which are pivot-tables using the Line-Dimension of instantOLAP to display single rows from a SQL query in a table).

By setting this property to "true" the cache will even store this values. Be aware that the cache can grow very big by caching this kind of data.

### Examples

```
Cache Line-Numbers = "true"
```

```
Cache Line-Numbers = "false"
```

## Cron-Pattern

### Type

Constant String (Cron pattern)

### Since

2.0

### Description

Filecaches are checked frequently for out-of-date entries (which are older than the time-span defined with the MaxAge-Property) and delete them. Because this can be a time-consuming task, the Filecache can be triggered to perform this more or less often by defining a CronPattern with this property.

Using the Cron-Pattern property also gives you the possibility to delete all entries at a given point of time when setting the MaxAge property to zero. For example, a Cron-Pattern `"* 0 0"` combined with a Max-Age of `"0"` will remove all entries at 0am every night.

### Examples

```
Cron = "* 0 0"
```

Check the Filecache every night at 0am for outdated entries

## Match-Expression

### Type

Boolean

### Since

1.2

### Description

If this property is left empty, the cache will store all entries (all facts for all coordinates). If you only want to store a part of the data (e.g. a special fact or all entries older than a month) you can set this property to a Boolean-Expression which controls which data will be accepted for storage. The Boolean-Expression must return TRUE for all accepted coordinates. You may use boolean functions as HASKEYS or HASLEVEL for checking the current coordinate.

### Examples

```
HASKEYS( Fact: Amount )
```

Only store the fact "Amount"

```
EXISTS( NEXT( Time ) )
```

Don't store the current year, month, day etc.

## Max Age

### Type

Constant Integer (seconds)

### Since

1.2

### Description

This property defines the maximum age of entries in seconds. Each entry of this cache will automatically be deleted if they become older than the maximum age defined by this property. Additional to the maximum age you can define, when and how often the out-of-date check is performed. Use the Cron-Pattern property to change the check-frequency.

If the maximum age is undefined, the entries will never expire. If the value is "0" all values will expire immediately (which only makes sense in combination with setting a different check-frequency in the Cron-Pattern than its default value).

### Examples

```
Max Age = "900"
```

Maximum age is 15 minutes

### See also

Cron-Pattern

## Filename

### Type

Constant String

### Since

2.2

### Description

Each Filecache must have a unique name, which has to be defined with this mandatory property.

### Examples

```
Name = "CACHE1"
```

# Include

---

## Model

### Type

Constant String

### Since

2.0

### Description

When you import another model into your model, you have to define the other model-name with this property. Note this property expects a model name, not a configuration-filename (so don't append .config to the model-name).

### Examples

```
Model = "general/DefaultModel"
```



# CHAPTER 4:

# Expressions

## Contents of this chapter:

The Type-system .....	360
Syntax .....	362
Constants .....	370
Functions .....	374

# The Type-system

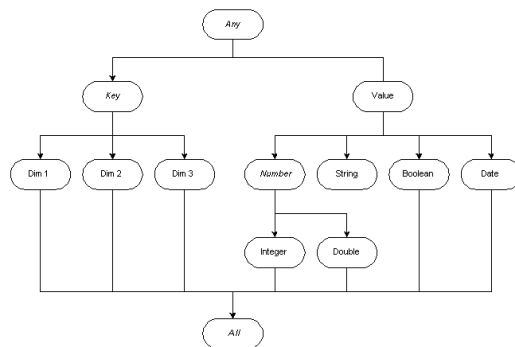
---

From the version 2.0 instantOLAP uses a type-system (known from programming-languages), where all values belong to a specific type. With the new type-system, instantOLAP can perform better syntax-checks and give better error-messages for your expressions. Also, functions can be combined in a much better way now, because the type-system now exactly controls, which function can be used as argument for other functions.

The type-system is organized in a hierarchy, with types, subtypes and super-types. Some types are createable, which means values with exactly this type can be created. Other types are only "help-types", which group other types to a super-type but can never have instances. "Value" is such a help-type.

A super-type of a type is another type which lies above the type in the hierarchy. E.g. "Value" is a super-type of "Integer". A subtype is a type which, lies under a type in the hierarchy. If a functions expects an argument of type X you can pass any value which has exactly this type X or a subtype of X. E.g. a function expecting an argument of the type "Number" would accept both values of the type "Integer" or "Double".

The following graphs shows all types in instantOLAP and their relationships:



Note that dimensions automatically become types in the graph and are always a subtype of "Key". Because of this, each function which expects arguments of the type "Key" will accept keys from every dimensions.

## All

The special type "All" is sub-type of all other types. The only value having "All" as type is NULL, which means NULL can be passed for any argument because NULL fits to every type.

NULL always represents an "undefined" value.

## Any

The type "Any" is the basic type in the type-system - every other type is a subtype of "Any" and every value belongs to the type "Any". If a function expects an argument of the type "Any", you can pass anything you want.

There can be no instance of the type "Any" itself.

## Boolean

Boolean-values are logical values which can have the values "true" or "false". The type Boolean is a subtype of "Value".

## Date

The type "Date" represents a timestamp (including years, months, days, hours, minutes, seconds and milliseconds).

## Double

The type "Double" represents numbers with a fraction. The type "Double" is a subtype of "Number".

## Integer

Integers are numbers without any fractions. The type "Integer" is a subtype of "Number".

## Key

All elements (keys) of a dimension have the type "Key" (more exact: each dimension becomes its own type in the model, which is a subtype of "Key". The elements of the dimension have their dimension as type). Keys are no value so the only super type of "Key" is "Any".

## Number

"Number" is another type which groups all kinds of numbers (Integers and Doubles). No number belongs to this type directly because each number is an instance of a subtype of it.

## String

Strings have the type "String" which is a subtype of "Value".

## Value

The type "Value" is the super types for all kinds of values - Strings, Numbers, Booleans, Dates and so on. No value has really the type "Value", because every value belongs to a subtype of this.

# Syntax

---

## Dimensions and selections

### Syntax

```
<dimension-syntax> := <dimension-name>
```

### Result-Type

Key

### Description

With the dimension-name itself you can access a dimension respective the selection (the filter) of a dimension. In the filter, no, one or more keys are stored for each dimension. With the dimension-name you can access these keys.

The filter is influenced by a number of query-elements, e.g. by

- the URL of a query,
- the selectors,
- the filter of the query, the blocks, the pivot-tables or the headers and
- the iterations of blocks and headers

### Examples

Time

Current selection of the dimension "Time"

## Dimension-Levels

### Syntax

```
<level-expression> := <dimension-name> '::' <level-name>
```

```
<level-expression> := <level-name>
```

### Result-Type

Key

### Description

Similar to the access to dimensions and selections, you can access the filtered keys of a specific level of a dimension. But in difference to the dimension-access, this will not return the selected keys directly but all keys above or underneath the selected keys, depending on the

desired level and on its position (above or underneath the level of the level of the selected keys).

## Examples

If the currently selected key of the dimension "Time" was "Feb/2004", the expression

```
Time::Year
```

would result to Time:'2004' and

```
Time::Day
```

to Time:'01.02.2004' + Time:'02.02.2004' + ... + Time:'29.02.2004'

## See also

Dimensions and selections

# Operators

## Using operators

Some functions are also available as operators which you can use instead of writing down the whole function-name. A few operators have no corresponding function.

Operators have a priority which defines the order in which the engine will evaluate them. The engine will evaluate the operators starting with the highest priority, down to the smallest. If two operators have the same priority, they will be evaluated from the left to the right.

If you want to change the execution-order of your expression, you must use brackets to override the operators priorities.

## List of operators

Op.	Pri	Function	Example
:	6	Key-operator	Time:'2004'
::	6	Level-operator	Time::Month
.	5	Attribute-Operator	Product.color
[]	5	CUBE	[CHILDREN( Product )]
{ }	5	TOSTRING	{Product}
*	4	MUL	10 * 20
/	4	DIV	10 / 20
%	4	MOD	20 % 10
+	3	ADD	10 + 20

-	3	SUB	10 - 20
		3	JOIN
IN	3	IN	Product:ProductA IN Product
<	2	LESS	10 < 20
>	2	GREATER	10 > 20
=	2	EQUAL	10 = 20
<=	2	LESS_OR_EQUAL	10 <= 20
>=	2	GREATER_OR_EQUAL	10 >= 20
<>	2	UNEQUAL	10 <> 20
LIKE	2	LIKE	'Hello world' LIKE '*world'
?	2	MATCH	Product ? Product.color = 'red'
AND	1	AND	Product.color = 'red' and Amount() > 0
OR	1	OR	Product.color = 'red' or Amount() > 0

Product:Product

### See also

Brackets

## Brackets

### Syntax

```
<bracket-expression> := '(' <expression> ')'
```

### Description

With brackets you can change the execution-order of your expressions. Normally expressions are executed in the order of the used operators priorities, beginning with the highest. Operators with equal priorities are executed from the left to right. By encapsulating parts of your expression in brackets, you will force the system to evaluate the content of the brackets first before using their result with the outer operators. You also may encapsulate multiple brackets - then the expressions will be executed from the inner to the outer.

Note that functions also operate like brackets: All arguments of an function-call will be evaluated first (beginning with the first argument), then the function will be executed and after all the result will be used together with the remaining part of you expression. If you encapsulate multiple functions (using functions as arguments for other functions), the inner functions will be executed first.

Brackets may be used with any type of expressions (booleans, keys, numbers etc.).

### Examples

```
( 10 + 20 ) * 30
```

```
= 900
```

```
10 + 20 * 30
```

```
= 610
```

### See also

Function calls

## Accessing attributes

### Syntax

```
<attribute-expression> := <key-expression> '.' <attribute-name>
```

### Return-Type

Equal to the attribute-type

### Description

Each element (key) of a dimension may have no, one or more attributes. Attributes can be simple values (Strings, Numbers, Booleans etc.) or elements (Keys) from other dimensions. When attributes have the type "Key", this connection between two dimensions is called a "Link".

To access an attribute you must use the dot-operators '.' immediately after a key-expression (attributes can only be read from keys). The return-type of the operator is equal to the type of the attribute. E.g. if an attribute has the type "String", the return type of the attribute-expression is also "String". If the type of the attribute is "Key", the return-type is "Key" and you may again use the attribute-operator again to access other attributes of the attribute.

If the previous key-expression results to NULL, the result of the attribute-expression is also NULL.

### Examples

```
Product.Color
```

Color of the current product

```
Product.Vendor.Name
```

Name of the vendor of the current product

## See also

ATTRIBUTENAMES, ATTRIBUTES

## Accessing variables

### Syntax

```
<variable-expression> := '$' <variable-name>
```

### Return-Type

String

### Description

instantOLAP allows the definition and usage of variables inside queries. Variables can be defined in two different ways:

- By adding selectors to your query which let the user choose the value of a variable,
- by passing arguments to the server in the URL

#### ***Variables defined with selectors***

Each selector automatically creates a variable with the name of the selector. A selector can automatically influence the filter of a query, because the user can select one or more elements of a dimension, but they also create a variable with the name of the dimension (e.g. a selector for "Time" creates a variable named "Time"). If a selector does not influence a dimension (because you gave a name to which matches no dimension), only the variable will be created (e.g. a selector with the name "X" will create a variable named "X" but not influence any dimension if there is no dimension with this name).

#### ***Variables passed in the URL***

Whenever a query is called from outside the system (e.g. with a constant link from another website) you can append additional parameters to the queries URL. All parameters will be converted into variables.

#### ***Types and evaluation***

Variables are always strings. If you want variables to be interpreted as different types, for example as a number, you can use the EVAL-function. The EVAL-functions expects a string as arguments and compiles it to a valid expression (or throws an error if the parsing fails) and returns the result of the dynamically created expression. This also lets you use variables in a very flexible way, e.g. you could offer a list of key-expressions inside your query and use them inside headers for the iteration.

#### ***Lists***

Variables can have no, one or more values, depending on the way you defined them. If a variable was defined by a Single-Selector, it will select one value at maximum. If it was defined by a Multiple-Selector, it can have more selections. When the variable was passed as an URL-appendix, each occurrence of the variable in the URL will become one value.

## Predefined variables

- **\$COUNTRY:** The COUNTRY variable contains the country-code of the current user. This is equivalent to the country-code configured in the browser.
- **\$LANGUAGE:** The LANGUAGE variable contains the language-code of the current user. This is equivalent to the language-code configured in the browser.
- **\$QUERYAUTHOR:** The author of the current query.
- **\$QUERYFILENAME:** The filename of the query.
- **\$QUERYFOLDER:** The path of the folder where the query is located.
- **\$QUERYDATE:** The creation-date of the query.
- **\$QUERYNAME:** The name of the query.
- **\$QUERYPATH:** The full path of the query inside the repository.
- **\$USER:** The USER variable contains the account name of the current user.

## Examples

`$COUNTRY`

E.g. returns 'US' (if the current user has configured 'US' as his country in his browser)

`$LANGUAGE`

E.g. return 'de' (if the current user has configured 'de' as his preferred language in his browser)

`$USER`

E.G. returns 'admin' (if the current user is named 'admin')

`EVAL( $X )`

Evaluates the variable X interpreted as an expression

## See also

[EVAL](#)

## Function calls

### Description

Like in programming-languages, instantOLAP supports functions for evaluating values at runtime. Each function has a name, certain expected arguments and a return type. Whenever you use a function inside an expression, the function-call will be interpreted as an expression of the return-type and can be passed to other functions as arguments (if they expect an argument of this type or a super-type of it).

To call a function you'll have to type the function-name first, followed by brackets surrounding the arguments which are separated by commas. The arguments itself can also be expressions, e.g. you can pass constants, functions or dimensions as arguments.

## Examples

```
NEXT( Time )
```

```
FIRST( LEVEL( Time, 1 ) )
```

## Level-Functions

### Syntax

```
<levelfunction-expression> := <level-name> '(' [ <Key> { ',' <Key> } ]  
' )'
```

### Description

For each level of each dimension the system automatically creates a function, which you can use to find the keys of these level underneath or above the currently selected keys, depending of the position of the level and the level of the selected keys. With these functions you can form expressions like "give me all months of the selected year" or "give me all product-groups of the selected products".

Additionally you can pass keys as arguments to change the current selection (the filters) for this functions-calls.

### Examples

```
Month()
```

```
Month( NEXT( Time ) )
```

## Fact-Functions

### Syntax

```
<fact-expression> := <fact-name> '(' [ <Key> { ',' <Key> } ] )'
```

### Description

For each fact, instantOLAP automatically generates a function which you can use to access the values of this fact from the cubes. Like in the CUBE-function you also can pass keys as arguments to this functions, changing the current filter which is used to access the cubes. Read the documentation of the CUBE-function for a more detailed description.

### Examples

```
Amount()
```

```
Amount( NEXT( Time ) )
```

## Comments

### Description

Since version 2.5 you can also add comments to expressions. To start a comment you must place a double-slash (`/ /`) to your expression, the parser will then skip the rest of the line and continue parsing from the beginning of the following line.

# Constants

---

For the most type you can use constants inside expressions:

- NULL as constant for the type All
- TRUE and FALSE as constants for the type Boolean
- Strings as constants for the type String
- Integer numbers as constants for the type Integer
- Double numbers as constants for the type Double
- Dimension-Keys as constants for the type Key

## Boolean constants

### Syntax

```
<Boolean-Constant> := 'TRUE' | 'true' | 'FALSE' | 'false'
```

### Description

TRUE and FALSE (written in capital or small letters) are the only possible constants for the type Boolean.

### Examples

```
IIF( TRUE, 10, 20 )
```

```
= 10
```

### See also

AND, NOT, OR

## Integer numbers

### Syntax

```
<Integer-Constant> = [ '-' ] { <digit> }
```

### Description

Integer-Constant represent frictionless numbers and have the type Integer.

### Examples

```
0
```

-1

100

### See also

Integer

## Double numbers

### Syntax

```
<Double-Constant> = ['-'] { <digit> } '.' { <digit> }
```

### Description

Double-Constant represent numbers with fraction and have the type Double.

### Examples

0.0

100.10

-10.100

### See also

Double

## Strings

### Syntax

```
<String-Constant> := '"' { <character> } '"' | "'" { <character> } "'"
```

### Description

Strings are constants of the type String. Strings inside expressions must be embedded in delimiters (like in programming languages). You may use two different types of delimiters, either the char " or the char '. Delimiters can be embedded (the delimiter ' can be embedded in " and vice versa).

### Examples

'Hello World'

"Jim's car"

## See also

String

## Dimension-Keys

### Syntax

```
<Key-Constant> = <dimension-name> ':' <id> | <dimension-name> ":" <id>
> " " "
```

### Description

Key-Constants point to a single Key of a specific Dimension and have the type Key. Whenever you use Key-Constants in queries, the expression is not influenced by the current filter, because this is a constant Link to a key.

Each Key-constants consists of the dimension-name followed by the char ":" and the ID of the reference key. If the ID contains white spaces or special chars, you must embed it within the delimiters " or '.

### Examples

```
Product: Coffee
```

```
Time: '01.01.2004'
```

## See also

Key

## NULL

### Syntax

```
<NULL-Constant> := ' NULL'
```

### Description

Like in SQL-databases, NULL is the constant representing an undefined value for all types. NULL is the only value having the type All (which is a subtype of all other types), so you might it at any position and for any argument of a function.

Understanding the behavior of NULL and of functions with NULL-arguments is very important for database- and OLAP-systems. E.g. if your reports queries data out of a database, the database will return NULL for the value not being stored inside their tables (for example, when no turnaround was stored for a specific date). The CUBE-function then will also return NULL for this values and all functions working with this value will have to deal with this NULL-value.

The most functions also return NULL if any of their parameters is NULL because there is no possible calculation for a valid return-value. But some functions exists especially for the treatment of NULL-values, the most important functions are ISNULL, EXISTS, and ZERO.

## Examples

```
Amount( Time: Dec/2004 )
```

Results to NULL if there is no amount stored for December 2004

```
DIV( NULL, 5 )
```

Always returns NULL (impossible to calculate)

## See also

EXISTS, ISNULL, ZERO

# Functions

---

## ABC

### Syntax

```
<abc-expression> := 'ABC(' <Key> ',' <Number> ',' <Number> ',' <Number> '> )'
```

### Since

2.1

### Return-type

Key

### Description

The ABC functions helps building an ABC-analysis. With this function you can analyze keys by any fact and find the keys with the (summarized) N% of the fact.

The first argument determines the list of keys you want to group. The second arguments is the formula you want to apply on the keys (usually a simple fact). The third and fourth arguments set the boundaries (maximum as first and minimum as second percentage, expected as values between 0.0 and 1.0) of the group you want to extract from the keys.

If any of the arguments is NULL the function will return NULL.

### Examples

```
ABC( LEVEL( Products, 1 ), Amount(), 1.0, 0.5 )
```

Find the Products with the top 50% amounts in sum

## ABS

### Syntax

```
<abs-expression> := 'ABS(' <Number> ')'
```

### Since

2.0

### Return-type

Number

## Description

Returns the absolute value of a the argument. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. If the argument is NULL, the value NULL is returned.

## Examples

```
ABS( 1.0 )
```

```
= 1.0
```

```
ABS( -1 )
```

```
= 1
```

```
ABS( 0 )
```

```
= 0
```

```
ABS( NULL )
```

```
= NULL
```

## See also

CEIL, FIRST

# ADD

## Syntax

```
<add-expression> := 'ADD(' <Any> [ ',' { <Any> } ] )'
```

## Since

1.0

## Return-type

Depends on the argument's type, see below

## Description

The ADD-function adds (or concatenates) values of any type. The behavior and return-type of this function depends on the argument's type:

- If all arguments are Integers, the return-value is the sum of all values (the return type is Integer)
- If all arguments are Numbers, the return-value is the sum of all values (the return type is Double)
- If all arguments are Keys, the return value is the joined list of all Keys (the return type is Key)

- If all arguments are Strings, the return value is the concatenated String of all arguments (the return type is String)
- Otherwise the return-value is the concatenation of all values converted to String

NULL-values will be ignored (even when arguments contain NULLs, the function will return the sum or concatenation of all other values).

This function is not type-safe because it add anything and does no type-checks. Use the SUM-function to perform a pure number-addition (the SUM-function only accepts and returns values of the type Number).

Instead of the ADD-function you also can use the operators "+".

## Examples

```
ADD( 10, 20 )
```

```
= 30
```

```
ADD( 'Hello', ' ', 'World' )
```

```
= 'Hello World'
```

```
ADD( 'Hello', 10 )
```

```
= 'Hello10'
```

```
ADD( Product:A, Product:B )
```

```
= Product:A | Product:B
```

## See also

DIV, MUL, SUB, SUM

# ALL

## Syntax

```
<all-expression> := 'ALL(' <String> ')'
```

## Since

2.1

## Return-type

Key

## Description

The ALL function returns all keys of a dimension or level. The name of the dimension or level must be passed as a string-expression.

This function is very similar to the LEVEL function but accepts real level-names or dimension-names as parameters.

## Examples

```
ALL( 'Time' )
```

Returns all time keys

```
ALL( 'Week' )
```

Returns all weeks (one specific level of the dimension Time)

## See also

FILTER, LEVEL

# ANCESTORS

## Syntax

```
<ancestors-expression> := ' ANCESTORS(' <Key> ' )'
```

## Since

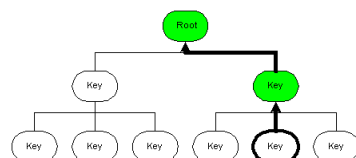
1.2

## Return-type

Key

## Description

This function returns all keys above the argument's keys in their hierarchy (including their parents, their parent's parents and so on, and the dimension's root-key). If the argument is NULL, the return-value is also NULL.



## Examples

```
ANCESTORS( Time: '01.07.2003' )
```

Returns Time:'Jul/2003' + Time:'2003' + Time:'Complete Period'

### See also

CONCAT, FILTER, PARENT, PEDIGREE

## AND

### Syntax

```
<and-expression> := 'AND(' <Boolean> ',' <Boolean> ')'
```

```
<and-expression> := <Boolean> ' AND <Boolean>
```

### Since

1.0

### Return-type

Boolean

### Description

The logical operator AND returns one of the Boolean constants TRUE, FALSE or NULL, depending on the arguments' values:

- If both arguments result to TRUE, the result is also TRUE.
- If one of the arguments results to FALSE, the result is also FALSE.
- If one of the arguments results to NULL, the result is also NULL.

Instead of this function you also can use the operator "AND".

### Examples

```
AND( TRUE, TRUE )
```

```
= TRUE
```

```
TRUE and TRUE
```

```
= TRUE
```

```
AND( FALSE, TRUE )
```

```
= FALSE
```

```
AND( TRUE, FALSE )
```

```
= FALSE
```

```
AND( TRUE, NULL )
```

```
= NULL
```

```
AND( FALSE, NULL )
```

```
= NULL
```

```
AND( NULL, NULL )
```

```
= NULL
```

### See also

NOT, OR

## ATTRIBUTENAMES

### Syntax

```
<attributes-expression> := ' ATTRIBUTES(' <Key> ' )'
```

### Since

2.0

### Return-type

String

### Description

This functions returns the names of all attributes of all keys passed as argument.

### Examples

```
ATTRIBUTENAMES( Time: '01.07.2003' )
```

E.g. returns 'Weekday' + 'DayID' + ...

### See also

BEAUTIFY, DIMENSIONNAME

## ATTRIBUTENV

### Syntax

```
<attributes-expression> := ' ATTRIBUTENV(' <Key> ' )'
```

## Since

2.0

## Return-type

String

## Description

This functions returns a list containing all attribute-names and -values for the given keys. The values are returned as a comma-separated string. If the arguments is NULL, the functions returns NULL.

## Examples

```
ATTRIBUTENV( Product:Product1 )
```

E.g. returns 'ProductID' + 'P1' + 'Customer' + 'C1, C2, C3' + ...

## See also

ATTRIBUTENAMES, ATTRIBUTES

# ATTRIBUTES

## Syntax

```
<attributes-expression> := ' ATTRIBUTES(' <Key> ')'
```

## Since

2.0

## Return-type

Value

## Description

This functions returns the values of all attributes of the keys passed as argument. If the argument is NULL, the function returns NULL.

## Examples

```
ATTRIBUTES( Time:'01.07.2003' )
```

E.g. returns Weekday:'Tue' + Week:'26/2003' + ...

## See also

ATTRIBUTENAMES

## AVAILABLEOFFLINE

### Syntax

```
<availableoffline-expression> := 'AVAILABLEOFFLINE(' [ <Key> { ',' <Key> } ] ')'
```

### Since

2.5

### Return-type

Boolean

### Description

The AVAILABLEOFFLINE function checks, if the fact passed as argument (or which is currently selected in the filter) is available in any offline cube for the currently selected dimension-keys (which also can be overridden by passing keys as arguments).

This function can be used to avoid error messages when using offline data, e.g. you can use it to display more information in a report inside the instantOLAP Player if the computer is connected to the database and to hide special areas of a query whenever it is disconnected.

### Examples

```
AVAILABLEOFFLINE( Amount )
```

Returns TRUE if the "Amount" Fact for the currently selected filter is available in any offline cube.

```
AVAILABLEOFFLINE( Amount, Product:1 )
```

Same check but for a special product.

## AVG

### Syntax

```
<avg-expression> := 'AVG(' <Number>, { ',' <Number> } ')'
```

### Since

1.0

## Return-type

Double

## Description

The function AVG returns the average value of all values of all arguments.

If the argument contains NULL-values, these will be ignored! If you don't want to ignore NULL-values, you can use the ZERO-function to interpret them as zero. If only NULL-values are passed as argument, the function returns NULL.

## Examples

```
AVG( 2, 4 )
```

```
= 3
```

```
AVG( Amount( LEVEL( Time, 3 ) ) )
```

Average amount of all days (level 3 of time-dimension)

```
AVG( ZERO( Amount( LEVEL( Time, 3 ) ) ) )
```

Average amount of all days (level 3 of time-dimension), interpreting NULL as 0

```
AVG( NULL )
```

```
= NULL
```

## See also

ALL, COUNT, MAX, MIN, ZERO

## AVGKEY

### Syntax

```
<avgkey-expression> := ' AVGKEY(' <Key> { ',' <Value> } )'
```

### Since

2.2.6

### Return-type

Key

## Description

The AVGKEY function returns the key of the first argument with the result for the expression passed as the second argument being the nearest to the average value of all results. If any of the arguments is NULL the function returns NULL.

## Examples

```
AVGKEY( PRODUCT, Amount() )
```

Returns the product with the closest amount to the average value over all products (for the current filter).

## See also

MAXKEY, MINKEY

# BEAUTIFY

## Syntax

```
<beautify-expression> := 'BEAUTIFY(' <String> ')'
```

## Since

2.0

## Return-type

String

## Description

The function BEAUTIFY changes to letters' case to make a string more readable. After performing this function, the first letter of each word will be in upper case, the rest of each word will be lower case. If you pass NULL to this function, it will return NULL, too.

## Examples

```
BEAUTIFY( 'hello WORLD' )
```

```
= 'Hello World'
```

## See also

TOLOWER, TOUPPER

## BELONGSTO

### Syntax

```
<belongsto-expression> := ' BELONGSTO(' <Key> ' ) '
```

### Since

2.2

### Return-type

Boolean

### Description

This function returns true, if the selected key (or at least one of the selected keys) of the dimension defined by the arguments is equal, child of or parent of the key passed as the argument. If the argument is NULL, the function returns FALSE.

This function is mainly used in matching expression, e.g. for cubes.

### Examples

```
BELONGSTO( Time:'01/2005' )
```

Returns true for Time:'01.01.2005', Time:'2005' etc.

### See also

ISCHILD OF, ISPARENT OF

## CATCH

### Syntax

```
<catch-expression> := ' CATCH(' <Any>, <Any> ' ) '
```

### Since

2.5

### Return-type

The common supertype of both arguments

### Description

The CATCH function catches and suppresses any error thrown while the evaluation of the first arguments and returns the result of the second argument instead. If no error occurs, the result of the first argument is returned.

The return type of this function is the common supertype of both arguments. E.g. if the first argument has the type "Double" and the second has "String", the return type would be "Value".

## Examples

```
CATCH( Amount(), 'Not available' )
```

Evaluates "Amount()" and returns (if no error was thrown) the result of this expression. If any error occurs (e.g. if the fact Amount is not available offline and no database is connected), the string 'Not available' is returned.

## See also

SWITCH

# CASE

## Syntax

```
<case-expression> := 'CASE(' <Boolean> ',' <Any> { ',' <Boolean> ',' <Any> } ')'
```

## Since

2.2.2

## Return-type

The return-type is the super-type of all values.

## Description

The CASE-function allows the conditional evaluation of one or more arguments. The CASE-function is very similar to the IIF function but accepts more than one condition.

Beginning with the first argument (which must be a boolean condition), the system evaluates the condition. If the condition returns TRUE, the following argument is evaluated and its result is returned as the result of the CASE function. If it returns FALSE, then the following condition (the third argument) is evaluated and so on.

If any of the evaluated conditions returns NULL then the function returns NULL. If no condition return TRUE or NULL the function also returns NULL.

Because there is no default return-value for this CASE function, you may add TRUE as the last condition, followed by the default value. Then this condition will always be returned if no other condition matches.

## Examples

```
CASE( HASLEVEL( Time, 1 ), 'L1', HASLEVEL( Time, 2 ), 'L2', TRUE, 'Other' )
```

Returns 'L1' for years, 'L2' for months and 'Other' for any other level of the current time-selection.

```
CASE( HASLEVEL( Time, 1 ), 'L1', NULL, 'L2' )
```

Returns 'L1' for years and NULL for any other level of the time selection.

### See also

IIF

## CEIL

### Syntax

```
<ceil-expression> := 'CEIL(' <Number> ')'
```

### Since

2.0

### Return-type

Integer

### Description

Returns the smallest (closest to negative infinity) integer that is not less than the argument. Special cases:

- If the argument value is of the type integer, then the result is the same as the argument.
- If the argument is NULL, then the result is also NULL.

### Examples

```
CEIL( 1.5 )
```

```
= 2
```

```
CEIL( -1.5 )
```

```
= -1
```

```
CEIL( 5 )
```

```
= 5
```

```
CEIL( NULL )
```

```
= NULL
```

### See also

FIRST, ROUND

## CHILDREN

### Syntax

```
<children-expression> := 'CHILDREN(' <Key> ')'
```

### Since

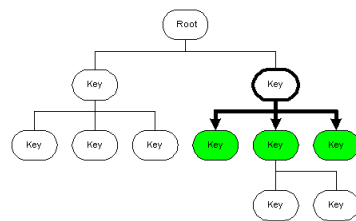
1.0

### Return-type

Key

### Description

The function CHILDREN returns all children of the keys passed as argument. Children of a key are those keys, which are placed directly under a key inside a hierarchy.



If the argument is NULL, this functions returns NULL.

### Examples

```
CHILDREN( Time:'2003' )
```

E.g. returns Time:'Jan/2003' + Time:'Feb/2030' + ... + Time:'Dec/2003'

```
CHILDREN( NULL )
```

= NULL

### See also

ANCESTORS, FIND, LEAFS, NEIGHBOURS, PARENT, PEDIGREE

## CLUSTER

### Syntax

```
<cluster-expression> := 'CLUSTER(' <Key> ')'
```

### Since

2.1.2

## Return-type

Key

## Description

The function CLUSTER replaces keys in the argument by their parents if all siblings of the key also appear in the argument. The result of the CLUSTER-function is clustered again unless it is not cluster-able any more.

This function is useful to reduce the number of loaded keys and facts, e.g. if you want to aggregate a fact for a specified range of time.

If the argument is NULL, the function returns NULL.

## Examples

```
CLUSTER( Time: 'Q1/2003' + Time: 'Q2/2003' + Time: 'Q3/2003' +
Time: 'Q4/2003' + Time: 'Q1/2003' )
```

```
= Time: '2003' + Time: 'Q1/2004'
```

```
CLUSTER( NULL )
```

```
= NULL
```

```
SUM( Amount( CLUSTER( Time ) ) )
```

Efficient aggregation of the fact "Amount" for a time-span

# COALESCE

## Syntax

```
<coalesce-expression> := 'COALESCE( [ <Any> [ ',' <Any> [...] ] ] )'
```

## Since

2.6.0

## Return-type

The common supertype of all arguments

## Description

This function returns the first value from all arguments which is not NULL. If all arguments are NULL, the function will return NULL.

## Examples

```
COALESCE( NULL, NULL, 1, 5)
```

= 1

### See also

ISNULL, ZERO

## COLSPAN

### Syntax

```
<colspan-expression> := 'COLSPAN(' <Integer> ')'
```

### Since

2.2.2

### Return-type

Integer

### Description

This function returns the number of columns spanned by the header with the given y-position (passed as argument). This function is e.g. useful to create subtotals for tables with grouped headers.

### Examples

```
SUM( TONUMBER( MATRIX( X() - 1, Y(), X() - COLSPAN(0), Y() ) ) ) )
```

Returns a subtotal for all columns spanned by the same, grouping header in the X-axis.

### See also

ROWSPAN

## CONCAT

### Syntax

```
<concat-expression> := 'CONCAT(' <String> [ ',' <String> ] )'
```

### Since

2.0

### Return-type

String

## Description

The function CONCAT concatenates all strings from argument one to one string, using the delimiter defined with the optional second argument. If no delimiter is defined, the standard-delimiter "," is used.

If the first argument is NULL, this function returns NULL.

## Examples

```
CONCAT( LEVEL( Customer, 1 ).Name )
```

E.g. returns 'Customer1,Customer2,Customer3'

```
CONCAT( LEVEL( Customer, 1 ).Name, ';' )
```

E.g. returns 'Customer1;Customer2;Customer3'

## See also

ADD

## CONTAINS

### Syntax

```
<contains-expression> := 'CONTAINS(' <Any> ',' <Any> ')'
```

### Since

1.2

### Return-type

Boolean

### Description

The function CONTAINS returns TRUE, if all values of the second argument are contained in the first argument, otherwise it returns FALSE. NULL is treated like any other element.

### Examples

```
CONTAINS( FAMILY( Time:'Complete Period' ), LEVEL( Time, 1 ) )
```

= TRUE

```
CONTAINS( FAMILY( Time:'Complete Period' ), 'Hello World' )
```

= FALSE

```
CONTAINS( FAMILY( Time:'Complete Period' ), NULL )
```

```
= NULL
```

```
CONTAINS( NULL, LEVEL( Time, 1 ) )
```

```
= FALSE
```

### See also

EXISTS, IN, CONTAINSTEXT

## CONTAINSTEXT

### Syntax

```
<containstext-expression> := 'CONTAINSTEXT(' <String> ',' <String> ')'
```

### Since

2.2

### Return-type

Boolean

### Description

Returns TRUE, if the text passed as first argument contains the text defined by the second argument. If any of the arguments is NULL, the function returns NULL. The text-comparison is case-sensitive.

### Examples

```
CONTAINSTEXT( 'Hello world', 'Hello' )
```

```
= TRUE
```

```
CONTAINSTEXT( 'Hello world', 'Foo' )
```

```
= FALSE
```

```
CONTAINSTEXT( 'Hello world', 'World' )
```

```
= FALSE
```

```
CONTAINSTEXT( 'Hello world', NULL )
```

```
= NULL
```

```
CONTAINSTEXT( NULL, 'Hello' )
```

```
= NULL
```

**See also**

CONTAINS

**COUNT****Syntax**

```
<count-expression> := 'COUNT(' <Any> ')'
```

**Since**

1.2

**Return-type**

Integer

**Description**

This function returns the size of the argument. If the argument is NULL, this function returns NULL.

**Examples**

```
COUNT( LEVEL( Fact, 0 ) )
```

```
= 1 (the root-key)
```

```
COUNT( NULL )
```

```
= NULL
```

**See also**

EXISTS

**COUNTRY****Syntax**

```
<country-expression> := 'COUNTRY()'
```

**Since**

2.2.0

**Return-type**

String

## Description

Returns the country-code of the session of the current user.

## Examples

```
COUNTRY( )
```

```
= 'DE'
```

## See also

LANGUAGE, LOCALE

# CUBE

## Syntax

```
<cube-expression> := 'CUBE(' [ <Key> { ',' <Key> } ] ')'
```

```
<cube-expression> := '[' [ <Key> { ',' <Key> } ] ]'
```

```
<cube-expression> := <fact-name> '(' [ <Key> { ',' <Key> } ] ')'
```

## Since

1.0

## Return-type

The type of the returned fact(s).

## Description

With the CUBE-function, you can read data out of one or more cubes. Without any argument, this function will read and return the current selection from the cubes. The current selection is defined by the headers, query- and block-filters, selectors etc.

You could, as an example, define a query with a x-header containing the key Fact:Amount and a y-header containing the key Product:Product1. You also could add a selector with years as option. If the user select Time:2003, the current selection for the cells is Fact:Amount, Product:Product1, Time:2003. If you use the CUBE-function inside the cells, this value will be read out of the cube.

If you want to read values which differ from the current selection, you can pass key-expressions as arguments to the CUBE-function. The results of this key-expression will be applied to the selection before values are read. If you would use the expression CUBE( NEXT( Time ) ) in the above sample, the same value (for Fact:Amount and Product:Product1) would be accessed, but for the next year. Or you could e.g. use CUBE( Fact:Price ) to access another fact (the price in this case).

Note that the CUBE-function can return more than one value. This will happen if the current selection or the arguments contain more than one key for at least one dimension. If you would use CUBE( CHILDREN( Time ) ) in the above sample, it would return 12 values, one per month.

Mathematically spoken, the number of the returned values is  $(N_1) * (N_2) * \dots * (N_d)$ , with  $N_x$  being the number of selected keys for dimension  $x$  and  $d$  being the number of possible dimensions.

The return-type of this function depends on the currently selected fact. If exactly one fact is selected, its type will be the return-type. If more than one fact is selected, the most common super type of their return-types will be returned (which can be ANY in the worst case).

There are two short-forms of this function: Brackets and auto-generated fact-functions. The bracket-form is very close to the original-form - instead of using the function-name CUBE you can enclose all arguments in brackets []. The function CUBE( NEXT( Time ) ) for example is equal to [NEXT( Time )].

The auto-generated fact-functions can be used whenever you exactly know which fact you want to read (which you normally know). In this case, you can use the function which was automatically created and is named like your fact. E.g. if you would define a fact named "Amount" there will be a function called "Amount", too. In this case you could access the fact with Amount() instead of [Fact:Amount]. You also can pass key-expressions to the generated fact-function, a valid example would be Amount( NEXT( Time ) ).

Note that only well-named fact will result in a auto-generated function. If your fact contains special chars of white spaces, the system won't be able to create a function for this.

## Examples

```
CUBE( )
```

Returns the currently selected values

```
[ ]
```

Same as above

```
CUBE( NEXT( Time ) )
```

Returns the currently selected values, but for the next time-element

```
[ NEXT( Time ) ]
```

Same as above

```
[ NEXT( Time ), Fact: Amount ]
```

Returns the value of "Amount" for the next time-element

```
CUBE( NEXT( Time ), Fact: Amount )
```

Same as above

```
Amount( NEXT( Time ) )
```

Same as above

## See also

ZERO

## DATEADD

### Syntax

```
<dateadd-expression> := 'DATEADD(' <Date> ',' <Integer> [ ',' <String> ] )'
```

### Since

2.5

### Return-type

Date

### Description

The DATEADD function adds a number of years, days, hours, minutes, seconds or milliseconds to the date passed as first argument. The last and optional argument defines, what you want to add to the date. You can pass 'year', 'day', 'hour', 'minute', 'second' or 'millisecond' here. The default value for the last argument is 'day'.

If any of the arguments is NULL the function returns NULL.

### Examples

```
DATEADD( Product.Releasedate, 5 )
```

Adds 5 days to the release date of the product.

```
DATEADD( Product.Releasedate, -1, 'year' )
```

Subtracts 1 year from the release date of the product.

## DATEDIFF

### Syntax

```
<datediff-expression> := 'DATEDIFF(' <Date> ',' <Date> )'
```

### Since

2.5

### Return-type

Integer

### Description

The DATEDIFF function returns the difference between both arguments in milliseconds. If any of the argument is NULL, the function returns also NULL.

## Examples

```
DATEDIFF( Project.End, Project.Start ) / ( 24 * 60 * 60 * 1000 )
```

Calculates the duration of a projects and converts it to days (from milliseconds).

## DEBUG

### Syntax

```
<debug-expression> := 'DEBUG(' <Any> [ ',' <String> ] )'
```

### Since

2.2.1

### Return-type

The type of the first argument

### Description

This function is useful to debug calculations in formulas or queries. It will return the value of the first argument but also print the values together with the text defined by the second argument to the debug log.

### Examples

```
DEBUG( PARENT( Time ), 'PARENT OF TIME:' )
```

Returns PARENT( Time ) and also prints the result to the debug log, e.g. "PARENT OF TIME: 2006"

### See also

FILTER, TYPE

## DEFAULTTEXT

### Syntax

```
<defaultttext-expression> := 'DEFAULTTEXT(' <Any> )'
```

### Since

2.2.1

### Return-type

String

## Description

Returns the default display-text of the keys. If the dimension of the keys have a default-text attribute, then the value of this attribute will be returned. If the dimension has not default text-attribute or if the key does not contain any value for this attribute, then the ID of the key will be returned.

The DEFAULTTEXT function exactly returns the same text that pivot-tables would display for keys in headers if no other text-expression was defined.

If the argument is NULL then the function also returns NULL.

## Examples

```
DEFAULTTEXT( Product:1234 )
```

```
= 'Computers'
```

```
DEFAULTTEXT( NULL )
```

```
= NULL
```

## DEVIATION

### Syntax

```
<deviation-expression> := ' DEVIATION( ' <Number> ' , ' <Number> ' ) '
```

### Since

2.2

### Return-type

Number

### Description

Calculates the deviation between the first and the second argument in percent (where 0 means 0%, 1 means 100%, -1 means -100%).

- If any of the arguments is NULL, the function returns 0
- If both values are equal, the function returns 0
- If the first argument is 0 and the second is positive, the function returns 1
- If the first argument is 0 and the second is negative, the function returns -1
- Otherwise the function returns  $(v2 - v1) / v1$  where  $v1$  is the value defined by the first argument and  $v2$  the value defined by the second.

### Examples

```
DEVIATION( 10, 20 )
```

```
= 1
DEVIATION( 20, 10 )

= -0.5
DEVIATION( 0, 20 )

= 1
DEVIATION( 0, -20 )

= -1
DEVIATION( 20, 20 )

= 0
DEVIATION( 0, 0 )

= 0
DEVIATION( NULL, 20 )

= NULL
DEVIATION( 10, NULL )

= NULL
```

### See also

DIV

## DEPTH

### Syntax

```
<depth-expression> := 'DEPTH(' <Key> ')'
```

### Since

2.1

### Return-type

Integer

### Description

This function returns the depth of the dimension parsed as argument. If the argument is NULL, the function returns NULL.

## Examples

```
DEPTH( Time )
```

```
= 4
```

```
DEPTH( NULL )
```

```
= NULL
```

## See also

LEVLLENAMES

# DIMENSIONATTRIBUTENAMES

## Syntax

```
<dimensionattributenames-expression> := ' DIMENSIONATTRIBUTENAMES( ' <Key> ' ) '
```

## Since

2.1

## Return-type

String

## Description

Returns the names of all attributes of all keys of the dimension. If the argument is NULL, the function returns NULL.

## Examples

```
DIMENSIONATTRIBUTENAMES( Product )
```

E.g. returns 'ProductID' + 'Customer' + 'Text' + ...

## See also

ATTRIBUTENAMES

# DIMENSIONNAME

## Syntax

```
<dimensionname-expression> := ' DIMENSIONNAME( ' <Key> ' ) '
```

## Since

2.0

## Return-type

String

## Description

Returns the names of the dimension(s) of the arguments, based on the return type of the argument. If the argument contains more than one return type, this function is returning the corresponding number of dimension-names. If the argument is NULL, this function returns NULL.

Note that this function DOES NOT execute the argument, it only analyses its type and converts it into dimension names.

## Examples

```
DIMENSIONNAME( Product )
```

```
= 'Product'
```

```
DIMENSIONNAME( Product:Product1 )
```

```
= 'Product'
```

```
DIMENSIONNAME( FIRST( Product::1 | Time::1 ) )
```

```
= 'Product' | 'Time'
```

## See also

DIMENSIONNAMES

## DIMENSIONNAMES

### Syntax

```
<dimensionnames-expression> := ' DIMENSIONNAMES() '
```

### Since

2.0

### Return-type

String

## Description

Returns the names of all dimensions.

## Examples

```
DIMENSIONNAMES( )
```

E.g. returns 'Fact' + 'Product' + 'Time' + ...

## See also

DIMENSIONNAME

# DISTINCT

## Syntax

```
<distinct-expression> := 'DISTINCT(' <Any> ')'
```

## Since

2.0

## Return-type

Equal to the argument's type.

## Description

The DISTINCT-function eliminates double values from argument. This means, each value, which occurs more than once in the argument, will appear only once in the result. If the value NULL is contained more than once, it will also be contained only once.

## Examples

```
DISTINCT( Product:Product1 )
```

= Product:Product1

```
DISTINCT( Product:Product1 | Product:Product2 | Product:Product1 )
```

= Product:Product1 + Product:Product2

# DIV

## Syntax

```
<div-expression> := 'DIV(' <Number> ',' <Number> ')'
```

```
<div-expression> := <Number> '/' <Number>
```

## Since

1.0

## Return-type

Double

## Description

The function DIV divides every value from the first argument by the corresponding value of the second argument and returns a number of doubles. Both arguments must have the same number of values, otherwise an error will be raised.

If one of both values is NULL, the result is NULL, too. If the divisor is zero, the function returns the string "ERR".

Instead of this function you also can use the operator "/".

## Examples

```
DIV( 10, 2 )
```

```
= 5
```

```
10 / 2
```

```
= 5
```

```
DIV( 10, NULL )
```

```
= NULL
```

```
DIV( NULL, 5 )
```

```
= NULL
```

```
DIV( 10, 0 )
```

```
= 'ERR'
```

```
DIV( JOIN( 10, 5 ), JOIN( 2, 1 ) )
```

```
= 5 | 5
```

## DLOOKUP

### Syntax

```
<dlookup-expression> := 'DLOOKUP(' <Key> { ',' <Key> } ')'
```

## Since

2.1.2

## Return-type

Key

## Description

This function evaluates, by immediately querying the cubes, for which keys of a dimension or dimension-level (defined by the first argument) a fact (defined by the second argument) is available (not NULL). When querying the cubes, the current filter will be considered. The DLOOKUP-function is very important whenever you build reports showing values for sparse filled cubes.

The DLOOKUP-function is very similar to the function LOOKUP but simpler and faster:

- DLOOKUP only can lookup whole dimensions or dimension-levels. Therefore, only dimension-names or level-names may be passed as first arguments. Also, you can only query for a single dimension with one call.
- DLOOKUP does not consider formulas, so it will only return keys which can be directly loaded out of cubes.

If the ordinary LOOKUP function is able to use this faster function instead, it will forward the operation to this function.

If any of the arguments is NULL, the function returns NULL.

## Examples

```
DLOOKUP( Article, Fact:Amount )
```

Returns all articles with a Amount stored in any cube for the current selection

```
DLOOKUP( Article::Group, Fact:Amount )
```

Returns all article-groups with a Amount stored in any cube for the current selection

```
DLOOKUP( Article::Group + Article::Category, Fact:Amount )
```

Returns all article-groups and -categories with a Amount stored in any cube

```
DLOOKUP( Article::Group, Fact:Amount + PREV( Time ) )
```

Returns all article-groups with a Amount stored in any cube for the previous period

```
DLOOKUP( NULL, NULL )
```

= NULL

**See also**

LOOKUP, SPLIT

**DRILLKEY****Syntax**

```
<drillkey-expression> := ' DRILLKEY()'
```

**Since**

2.2.1

**Return-type**

Key

**Description**

Returns the current drill-key of a header. This is the iteration-key of the header on which the user clicked to open the drilldown and to see the current header.

This function is only valid in pivot-tables.

**Examples**

```
DRILLKEY( )
```

In a normal drilldown using the children hierarchy, e.g in the time-dimension, the drill-key will be e.g. 'Time:2006' for the then opened header containing 'Time:Jan/2006'.

**DRILLLEVEL****Syntax**

```
<drilllevel-expression> := ' DRILLLEVEL()'
```

**Since**

2.1

**Return-type**

Integer

**Description**

Returns the current drill-level of a header. The initial level for a not-drilled header is 0, after a single drill-down 1 and so on. Headers without having the drilldown -feature enabled will always return 0.

This function is only valid in pivot-tables.

## Examples

```
DRILLLEVEL( )
```

Returns 0 if no drilldown happened

Returns 1 if the user drilled once

## DSORT

### Syntax

```
<sort-expression> := 'DSORT(' <Key> ',' <Key> [ ',' <Integer> [ ',' <Boolean> [ ',' <Boolean> ] ] ] )'
```

### Since

2.2

### Return-type

Key

### Description

The function DSORT sorts the complete content or a single or more levels of a dimension (defined by the first argument) by a single fact (defined by the second argument). When querying the cubes, the current filter will be considered and may also be influenced with the second argument.

Like with LOOKUP and DLOOKUP, the DSORT-function is very similar to the function SORT but simpler and faster:

- DSORT only can sort whole dimensions or dimension-levels. Therefore, only dimension-names or level-names may be passed as first arguments. Also, you can only query for a single dimension with one call.
- DSORT does not consider formulas, so it will only return keys which can be directly loaded out of cubes. Therefore you can only pass the fact as key instead as formula.

Like SORT, you can also pass a limit and the order (ascending or descending) as argument three and four. Additionally there is an additional argument which can contain a boolean match-expression which can be used to filter the result after it was queried from the cubes.

If the ordinary SORT function is able to use this faster function instead, it will forward the operation to this function.

If the first or second argument is NULL, the function returns NULL.

### Examples

```
DSORT( Article, Fact:Turnaround )
```

```

DSORT( Article::PRODUCTS, Fact:Turnaround )

DSORT( Article::PRODUCTS, Fact:Turnaround )

DSORT( Article::PRODUCTS, Fact:Turnaround, 10 )

DSORT( Article::PRODUCTS, Fact:Turnaround, 10, true )

DSORT( Article::PRODUCTS, Fact:Turnaround, 10, true, Article.State = 1
)

```

**See also**

DLOOKUP, LOOKUP, SORT

**DURATIONTOSTRING****Syntax**

```
<durationtostring-expression> := ' DURATIONTOSTRING( <Integer> ) '
```

**Since**

2.6.1

**Return-type**

String

**Description**

Formats the passed duration (in milliseconds) into a string. The standard format for durations is "days-HH:MM:SS", future versions of this will allow to define own formats.

If the argument is NULL, the function returns NULL.

**Examples**

```
DURATIONTOSTRING( 60000 )
```

```
= "00:00:01"
```

**See also**

TOSTRING

**ELEMENT\_AT****Syntax**

```
<elementat-expression> := ' ELEMENT_AT(' <Any> ',' <Integer> [ ',' <Integer> ] ) '
```

## Since

2.2.2

## Return-type

The type of the first argument

## Description

This function extracts one or more elements of the first argument by their position. The position of the desired elements is defined by the second argument whereby 0 is the position of the first value. The number of elements to be returned is defined by the optional third argument. If the third argument is left empty, the function will only return one value.

If any of the arguments is NULL then the function also returns NULL.

## Examples

```
ELEMENT_AT( 'A' | 'B' | 'C' , 0 )
```

```
= 'A'
```

```
ELEMENT_AT( 'A' | 'B' | 'C' , 2 )
```

```
= 'C'
```

```
ELEMENT_AT( 'A' | 'B' | 'C' , 1, 2 )
```

```
= 'B' | 'C'
```

```
ELEMENT_AT( 'A' | 'B' | 'C' , NULL )
```

```
= NULL
```

# EMPTY

## Syntax

```
<empty-expression> := 'EMPTY()'
```

## Since

2.1

## Return-type

All

## Description

This function returns an empty list. Because the return-type of this function is "All" you can use it as argument for any other function.

## Examples

```
EMPTY()
```

## EMPTYASNULL

### Syntax

```
<emptyasnull-expression> := 'EMPTYASNULL( <Any> )'
```

### Since

2.6.1

### Return-type

The type of the argument

### Description

If the argument is an empty list, this function returns NULL. Otherwise the function returns the unmodified argument.

### Examples

```
EMPTYASNULL( 10 | 20 )
```

```
= 10 | 20
```

```
EMPTYASNULL( EMPTY() )
```

```
= NULL
```

## ENDSWITH

### Syntax

```
<endswith-expression> := 'ENDSWITH(' <String>, <String> )'
```

### Since

2.1

### Return-type

Boolean

## Description

This function determines if the first argument ends with the string passed as second argument. If any of the argument is NULL, the function returns NULL.

## Examples

```
ENDSWITH( 'Hello World', 'World' )
```

```
= TRUE
```

```
ENDSWITH( 'Hello World', 'Hello' )
```

```
= FALSE
```

```
ENDSWITH( 'Hello World', 'NULL' )
```

```
= NULL
```

# EQUAL

## Syntax

```
<equal-expression> := 'EQUAL(' <Any> ',' <Any> ')'
```

```
<equal-expression> := <Any> '=' <Any>
```

## Since

1.0

## Return-type

Boolean

## Description

This function tests both arguments for equality. They are equal if:

- Both arguments have the same size.
- Each element of argument one is contained in argument two at the same position.

Instead of this function you also can use the operator "=".

## Examples

```
EQUAL( 10, 10 )
```

```
= TRUE
```

```
EQUAL( 10, 20 )
```

```
= FALSE
EQUAL( 10 | 10, 10 )
= TRUE
EQUAL( 10, 'Hello World' )
= FALSE
EQUAL( 10, NULL )
= NULL
EQUAL( NULL, NULL )
= NULL
```

## ERROR

### Syntax

```
<error-expression> := 'ERROR( <String > )'
```

### Since

2.2.0

### Return-type

All

### Description

This function will stop the current query-execution and raises an error with the error-message passed as the argument.

If the argument is NULL and no error-message is defined, no error will be raised.

### Examples

```
IIF( LEVELOF( Time ) = 0, ERROR( 'Invalid time-selection' ), Time
```

## EVAL

### Syntax

```
<eval-expression> := 'EVAL(' <String> ')'
```

### Since

2.0

## Return-type

Any

## Description

The function EVAL allows to parse and execute expression at runtime. The argument must be a string, which must define a valid expression. If the argument is NULL, this function returns NULL.

Because the system cannot ensure the return-type of the function at the compile time, it only returns values of the type 'Any' since the version 2.2.2. You must cast the result to the desired type in order to use the result of the function.

## Examples

```
TOINTEGER( EVAL( '10 * 2' ) )
```

= 5

```
EVAL( $param )
```

Parsing and evaluation of the parameter named 'param'

```
EVAL( NULL )
```

= NULL

# EXISTS

## Syntax

```
<exists-expression> := ' EXISTS(' <Any> ' )'
```

## Since

1.1

## Return-type

Boolean

## Description

This function tests if the argument contains at least one value except NULL.

## Examples

```
EXISTS( NEXT( Time ) )
```

Returns TRUE, if "Time" not currently selects the last element

```
EXISTS( 1 )
```

```
= TRUE
```

```
EXISTS( NULL )
```

```
= FALSE
```

## EXP

### Syntax

```
<exp-expression> := 'EXP(' <Number> ')'
```

### Since

2.1

### Return-type

Number

### Description

This function returns Euler's number  $e$  raised to the power defined by the argument. If the argument is NULL, the function returns NULL.

### Examples

```
EXP( 10 )
```

```
= e^10
```

## FACTROOT

### Syntax

```
<factroot-expression> := 'FACTROOT()'
```

### Since

1.2

### Return-type

Key

### Description

The function "FACTROOT" returns the root-key of the fact-dimension.

## Examples

```
FACTROOT( )
```

E.g. returns Fact:'All Facts'

## See also

NONFACTROOTS

# FAMILY

## Syntax

```
<family-expression> := ' FAMILY(' <Key> ' )'
```

## Since

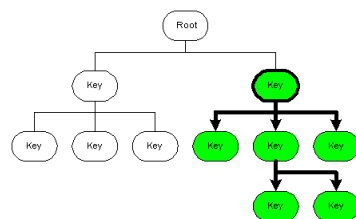
1.2

## Return-type

Key

## Description

The FAMILY-function evaluates the family for each key of the argument and returns it. The members of a key-family are the key itself, its children, the children of the children and so on down to the leaves. If the argument is NULL, the functions returns NULL.



## Examples

```
FAMILY( Time:'2003' )
```

E.g. returns Time:'Jan/2003' + Time:'01.01.2003' + ... + Time:'31.12.2003'

```
FAMILY( NULL )
```

= NULL

## See also

ALL, ANCESTORS, CHILDREN, LEAFS, PARENT

## FILTER

### Syntax

```
<filter-expression> := 'FILTER()'
```

### Since

2.0

### Return-type

String

### Description

The function FILTER returns the current filter as string. The current filter is the collection of all dimension-selections, e.g. in a cell. This is a debug-function, which you can use to test and display the current selection if needed.

### Examples

```
FILTER( )
```

E.g. returns the string '[Time=2003;Product=A]'

### See also

DEBUG, RETURNTYPE, TYPE

## FILTERKEYS

### Syntax

```
<filterkeys-expression> := 'FILTERKEYS()'
```

### Since

2.2.6

### Return-type

Key

### Description

The function FILTERKEYS returns all keys of the current filter as string. The current filter is the collection of all dimension-selections, e.g. in a cell.

## Examples

```
FILTERKEYS( )
```

E.g. returns the keys Time:2003 | Product:A

## See also

FILTER

# FIND

## Syntax

```
<find-expression> := 'FIND( <Key> ',' <String> [ ',' <Boolean> [ ',' <String> ] ] )'
```

## Since

2.0 2.5 (attribute name added)

## Return-type

Key

## Description

The function FIND searches for keys of the dimension defined by the first argument using the pattern defined by argument two.

The third and optional argument determines if you want to use wildcards when searching for keys. When enabling wildcards, you can use the wildcards '\*' and '?' to find more than one key. The default value for this argument is "false".

The last argument defines the name of the attribute you want to search. If no attribute is defined, the function will return all keys which IDs match the pattern. If you define a valid attribute name, the function will return all keys which has values for this attribute which match the pattern. By default the FIND function searches for IDs.

If one of the first three arguments is NULL, the FIND-function returns NULL.

## Examples

```
FIND( Time, 'Jan/2003' )
```

E.g. returns Time:'Jan/2003'

```
FIND( Time, 'J*/2003', true )
```

E.g. returns Time:'Jan/2003', Time:'Jun/2003', Time:'Jul/2003'

```
FIND( Time, NULL )
```

```
= NULL
```

```
FIND( NULL, 'J*/2003', true )
```

```
= NULL
```

```
FIND( Time, '55', false, 'Week' )
```

E.g. returns all Time keys which have a value for the "Week" attribute equals to "55".

### See also

NOW

## FIRST

### Syntax

```
<first-expression> := 'FIRST(' <Any> [ ',' <Integer> ] )'
```

### Since

1.2

### Return-type

The type of the first argument.

### Description

This function returns the first N values of the first argument. If you define no size N (with the second argument), the function only returns the first element of argument. If one of the arguments is NULL, the functions returns NULL.

### Examples

```
FIRST( LEVEL( Time, 2 ) )
```

E.g. returns Time:'Jan/2003'

```
FIRST( LEVEL( Time, 2 ), 3 )
```

E.g. returns Time:'Jan/2003' + Time:'Feb/2003' + Time:'Mrz/2003'

```
FIRST( NULL )
```

```
= NULL
```

### See also

LAST, NEXT, PREV

## FKEY

### Syntax

```
<fkey-expression> := ' FKEY()'
```

### Since

2.5.0

### Return-type

Key

### Description

The FKEY function returns the last key that was added to the filter. In headers, this is the iterated key, in FOEARCH-, MATCH-expression or similar expressions, this is the iterated key.

### Examples

```
FOREACH( PRODUCT | MANUFACTURER, { FKEY() } )
```

### See also

IVALUE

## FLOOR

### Syntax

```
<floor-expression> := ' FLOOR(' <Number> ')'
```

### Since

2.0

### Return-type

Integer

### Description

This function Returns the largest (closest to positive infinity) integer value that is not greater than the argument. If the argument is NULL, it returns NULL.

### Examples

```
FLOOR( 1.5 )
```

= 1

```
FLOOR( -1.5 )
```

```
= -2
```

```
FLOOR( 5 )
```

```
= 5
```

```
FLOOR( NULL )
```

```
= NULL
```

### See also

ABS, CEIL, ROUND

## FOREACH

### Syntax

```
<foreach-expression> := 'FOREACH(' <Any> ',' <Any> ')'
```

### Since

1.2, changed in 2.5.0

### Return-type

The return-type of the second argument.

### Description

The FOREACH-function evaluates the second argument for each value of the first argument and returns all results in a joined list. If one of the arguments is NULL, this function returns NULL.

At the beginning, the first argument is evaluated - which result to a list of keys or values. For keys, the current selection (filter) is manipulated with each key and the second argument will be executed with this filter - otherwise the function will be executed with the parent filter. All results will be joined to a list which return-type of the second parameter.

Since version 2.5.0, the FOREARCH function accepts any value as the first argument. If the first argument does not return keys, you can use the IVALUE() function to access the iterated value.

### Examples

```
FOREACH( LEVEL( Article, 2 ), Article.Color )
```

Returns all colors of all articles at level 2

### See also

MATCH, SORT, IVALUE

# FORECAST

## Syntax

```
<forecast-expression> := 'FORECAST(' <Number> ',' <Key> ',' <Key> [
',' <Integer> [ ',' <Integer> [ ',' <Integer> ] ] ] ')'
```

## Since

2.2

## Return-type

Double

## Description

The FORECAST function performs a forecast for the function defined with argument 1 for a single dimension-element (defined by argument 3) based on the results for the keys defined by the argument 2.

This function offers a number of different forecast-models, from which you can choose one with the 4th argument. This argument expects an integer-number representing one of the following forecast-models:

- 0: Best forecast (default): This option chooses automatically one the following models.
- 1: Simple exponential forecast: A simple exponential smoothing forecast model is a very popular model used to produce a smoothed Time Series. Whereas in simple Moving Average models the past observations are weighted equally, Exponential Smoothing assigns exponentially decreasing weights as the observations get older.

In other words, recent observations are given relatively more weight in forecasting than the older observations. In the case of moving averages, the weights assigned to the observations are the same and are equal to  $1/N$ . In simple exponential smoothing, however, a "smoothing parameter" - or "smoothing constant" - is used to determine the weights assigned to the observations.

This simple exponential smoothing model begins by setting the forecast for the second period equal to the observation of the first period. Note that there are ways of initializing the model. As of the time of writing, these alternatives are not available in this implementation. Future implementations of this model may offer these options.

- 2: Double exponential forecast: Double exponential smoothing - also known as Holt exponential smoothing - is a refinement of the popular simple exponential smoothing model but adds another component which takes into account any trend in the data. Simple exponential smoothing models work best with data where there are no trend or seasonality components to the data. When the data exhibits either an increasing or decreasing trend over time, simple exponential smoothing forecasts tend to lag behind observations. Double exponential smoothing is designed to address this type of data series by taking into account any trend in the data.

Note that double exponential smoothing still does not address seasonality. For better exponentially smoothed forecasts using data where there is expected or known to be seasonal variation in the data, use triple exponential smoothing.

As with simple exponential smoothing, in double exponential smoothing models past observations are given exponentially smaller weights as the observations get older. In other

words, recent observations are given relatively more weight in forecasting than the older observations.

There are two equations associated with Double Exponential Smoothing.

$$f_t = a \cdot Y_t + (1-a) \cdot (f_{t-1} + b \cdot t - 1) \quad b_t = g \cdot (f_t - f_{t-1}) + (1-g) \cdot b_{t-1}$$

where

$Y_t$  is the observed value at time  $t$ .  
 $f_t$  is the forecast at time  $t$ .  
 $b_t$  is the estimated slope at time  $t$ .  
 $a$  - representing alpha - is the first smoothing constant, used to smooth the observations.  
 $g$  - representing gamma - is the second smoothing constant, used to smooth the trend.

- 3: Triple exponential forecast: Triple exponential smoothing - also known as the Winters method - is a refinement of the popular double exponential smoothing model but adds another component which takes into account any seasonality - or periodicity - in the data.

Simple exponential smoothing models work best with data where there are no trend or seasonality components to the data. When the data exhibits either an increasing or decreasing trend over time, simple exponential smoothing forecasts tend to lag behind observations. Double exponential smoothing is designed to address this type of data series by taking into account any trend in the data. However, neither of these exponential smoothing models address any seasonality in the data.

For better exponentially smoothed forecasts of data where there is expected or known to be seasonal variation in the data, use triple exponential smoothing.

As with simple exponential smoothing, in triple exponential smoothing models past observations are given exponentially smaller weights as the observations get older. In other words, recent observations are given relatively more weight in forecasting than the older observations. This is true for all terms involved - namely, the base level  $L_t$ , the trend  $T_t$  as well as the seasonality index  $s_t$ .

There are four equations associated with Triple Exponential Smoothing:

$$\begin{aligned} L_t &= a \cdot (x_t / s_{t-c}) + (1-a) \cdot (L_{t-1} + T_{t-1}) \\ T_t &= b \cdot (L_t - L_{t-1}) + (1-b) \cdot T_{t-1} \\ s_t &= g \cdot (x_t / L_t) + (1-g) \cdot s_{t-c} \\ f_{t,k} &= (L_{t+k} + T_t) \cdot s_{t+k-c} \end{aligned}$$

where

$L_t$  is the estimate of the base value at time  $t$ . That is, the estimate for time  $t$  after eliminating the effects of seasonality and trend.  
 $a$  - representing alpha - is the first smoothing constant, used to smooth  $L_t$ .  
 $x_t$  is the observed value at time  $t$ .  
 $s_t$  is the seasonal index at time  $t$ .  
 $c$  is the number of periods in the seasonal pattern. For example,  $c=4$  for quarterly data, or  $c=12$  for monthly data.  
 $T_t$  is the estimated trend at time  $t$ .  
 $b$  - representing beta - is the second smoothing constant, used to smooth the trend estimates.  
 $g$  - representing gamma - is the third smoothing constant, used to smooth the seasonality estimates.  
 $f_{t,k}$  is the forecast at time the end of period  $t$  for the period  $t+k$ .

There are a variety of different ways to come up with initial values for the triple exponential smoothing model. The approach implemented here uses the first two "years" (or complete cycles) of data to come up with initial values for  $L_t$ ,  $T_t$  and  $s_t$ . Therefore, at least two complete cycles of data are required to initialize the model. For best results, more data is recommended - ideally a minimum of 4 or 5 complete cycles. This gives the model chance

to better adapt to the data, instead of relying on getting - guessing - good estimates for the initial conditions.

- 4: Regression forecast: Implements a single variable linear regression model. The coefficients of the regression - the intercept and the slope - as well as the accuracy indicators are determined from the data set passed to init.

A single variable linear regression model essentially attempts to put a straight line through the data points. For the more mathematically inclined, this line is defined by its gradient or slope, and the point at which it intercepts the x-axis (i.e. where the independent variable has, perhaps only theoretically, a value of zero). Mathematically, assuming the independent variable is x and the dependent variable is y, then this line can be represented as:

$$y = \text{intercept} + \text{slope} * x$$

- 5: Polynomial forecast: Implements a single variable polynomial regression mode. The coefficients of the regression as well as the accuracy indicators are determined from the data set passed to init.

A single variable polynomial regression model essentially attempts to put a polynomial line - a curve if you prefer - through the data points. Mathematically, assuming the independent variable is x and the dependent variable is y, then this line can be represented as:

$$y = a_0 + a_1 * x + a_2 * x^2 + a_3 * x^3 + \dots + a_m * x^m$$

- 6: Multiple linear forecast: Implements a multiple variable linear regression model. The coefficients of the regression, as well as the accuracy indicators are determined from the data set passed to init.

A multiple variable linear regression model essentially attempts to put a hyperplane through the data points. Mathematically, assuming the independent variables are xi and the dependent variable is y, then this hyperplane can be represented as:

$$y = a_0 + a_1 * x_1 + a_2 * x_2 + a_3 * x_3 + \dots$$

where the ai are the coefficients of the regression. The coefficient a0 is also referred to as the intercept. If all xi were zero (theoretically at least), it is the forecast value of the dependentVariable, y.

- 7: Moving average forecast: A moving average forecast model is based on an artificially constructed time series in which the value for a given time period is replaced by the mean of that value and the values for some number of preceding and succeeding time periods. As you may have guessed from the description, this model is best suited to time-series data; i.e. data that changes over time. For example, many charts of individual stocks on the stock market show 20, 50, 100 or 200 day moving averages as a way to show trends.

Since the forecast value for any given period is an average of the previous periods, then the forecast will always appear to "lag" behind either increases or decreases in the observed (dependent) values. For example, if a data series has a noticeable upward trend then a moving average forecast will generally provide an underestimate of the values of the dependent variable.

The moving average method has an advantage over other forecasting models in that it does smooth out peaks and troughs (or valleys) in a set of observations. However, it also has several disadvantages. In particular this model does not produce an actual equation. Therefore, it is not all that useful as a medium-long range forecasting tool. It can only reliably be used to forecast one or two periods into the future.

The moving average model is a special case of the more general weighted moving average. In the simple moving average, all weights are equal.

- 8: Weighted moving average forecast: A weighted moving average forecast model is based on an artificially constructed time series in which the value for a given time period is replaced by the weighted mean of that value and the values for some number of preceding time periods. As you may have guessed from the description, this model is best suited to time-series data; i.e. data that changes over time.

Since the forecast value for any given period is a weighted average of the previous periods, then the forecast will always appear to "lag" behind either increases or decreases in the observed (dependent) values. For example, if a data series has a noticeable upward trend then a weighted moving average forecast will generally provide an underestimate of the values of the dependent variable.

The weighted moving average model, like the moving average model, has an advantage over other forecasting models in that it does smooth out peaks and troughs (or valleys) in a set of observations. However, like the moving average model, it also has several disadvantages. In particular this model does not produce an actual equation. Therefore, it is not all that useful as a medium-long range forecasting tool. It can only reliably be used to forecast a few periods into the future.

9: Naive forecast: A naive forecasting model is a special case of the moving average forecasting model where the number of periods used for smoothing is 1. Therefore, the forecast for a period,  $t$ , is simply the observed value for the previous period,  $t-1$ .

Due to the simplistic nature of the naive forecasting model, it can only be used to forecast up to one period in the future. It is not at all useful as a medium-long range forecasting tool.

This model really is a simplistic model, and is included partly for completeness and partly because of its simplicity. It is unlikely that you'll want to use this model directly. Instead, consider using either the moving average model, or the more general weighted moving average model with a higher (i.e. greater than 1) number of periods, and possibly a different set of weights.

Some models need a periodicity to defined which can be passed with the optional argument 5. The periodicity defines the number of values for a single period - e.g if you want to forecast months in a year, the periodicity should be set to 12.

The optional last arguments allows to define an iteration for the forecast in order to create missing values in the past by forecasting them and then use these values as a base for further iterations.

## Examples

```
FORECAST( Amount(), Time, LEVEL( Time, 2 ) )
```

Best forecast for the fact Amount() for the current Time based on the whole level 2 of the time-period.

```
FORECAST( Amount(), Time, LEVEL( Time, 2 ), 8, 12 )
```

Weighted moving average forecast for the current time, based on the whole level 2 of the time-period with a periodicity of 12 (number of months in a year).

## FPOP

### Syntax

```
FPOP( <Any> )
```

## Since

2.2

## Return-type

Any (equal to the argument-type)

## Description

Many functions influence the current filter, e.g. when iterating the input-set for a match or sort function. Also, the iterations of headers, selector-options change the current selection.

With this function you can access the previous selection from the filter-stack before it was influence by a function or iteration. The expression passed as argument will be simply executed with the previous version of the filter.

Also, you can nest this function to access even earlier versions of the filter.

## Examples

```
FPOP( Product )
```

Returns the selection of the Product-dimension before it was changed by the last function or iteration.

```
FPOP( COUNT( Product ) )
```

Counts the number of originally selected products before the last iteration.

```
FPOP( FPOP( Product ) )
```

Climbs up two levels in the filter-stack.

## See also

FPUSH

## FPUSH

## Syntax

```
FPUSH( <Key>, Any> )
```

## Since

2.2

## Return-type

Any (equal to the argument-type)

## Description

The function FPUSH allows to execute an expression with a different filter than the current.

The expression passed as second argument will be executed after the first argument was applied to the filter. If the first argument is NULL, the filter will remain unchanged. This function does not the filter of the outer scope.

## Examples

```
FPUSH( PREV( Product ), Amount() )
```

```
= Amount( PREV( Product ) )
```

```
FPUSH( PREV( Product ), Amount() / Quantity() )
```

```
= Amount( PREV( Product ) ) / Quantity( PREV( Product ) )
```

```
FPUSH( NULL, Amount() )
```

```
= Amount()
```

## See also

FPOP

## GETDAY

### Syntax

```
<getday-expression> := ' GETDAY( ' <Date> ' ) '
```

### Since

2.5

### Return-type

Integer

### Description

Returns the day of week of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( ' 2008-03-31', ' yyyy-MM-dd' ) )
```

Returns 31 (the day of week of the argument).

## GETHOUR

### Syntax

```
<gethour-expression> := 'GETHOUR(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the hour of day of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 21 (the hour of the argument).

## GETSECOND

### Syntax

```
<getsecond-expression> := 'GETSECOND(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the second of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 14 (the second of the argument).

## GETMINUTE

### Syntax

```
<getminute-expression> := 'GETMINUTE(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the minute of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 29 (the minute of the argument).

## GETMONTH

### Syntax

```
<getmonth-expression> := 'GETMONTH(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the month of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '2008-03-31', 'yyyy-MM-dd' ) )
```

Returns 3 (the month of the argument).

## GETSECOND

### Syntax

```
<getmillisecond-expression> := 'GETMILLISECOND(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the milliseconds of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 521 (the milliseonds of the argument).

## GETYEAR

### Syntax

```
<getyear-expression> := 'GETYEAR(' <Date> ')'
```

### Since

2.5

### Return-type

Integer

### Description

Returns the year of the passed date. If the argument is NULL, the function also returns NULL.

### Examples

```
GETDAY( TODATE( '2008-03-31', 'yyyy-MM-dd' ) )
```

Returns 2008 (the year of the argument).

# GREATER

## Syntax

```
<greater-expression> := ' GREATER( ' <Any> ', ' <Any> ' ) '
```

```
<greater-expression> := <Any> ' > ' <Any>
```

## Since

1.0

## Return-type

Boolean

## Description

This function tests if each value of the first argument is greater than the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function you also can use the operators ">".

## Examples

```
GREATER( 2, 1 )
```

```
= TRUE
```

```
2 > 1
```

```
= TRUE
```

```
GREATER( 2, 2 )
```

```
= FALSE
```

```
2 > 2
```

```
= FALSE
```

```
GREATER( NULL, 2 )
```

```
= NULL
```

```
GREATER( JOIN( 2, 3 ), JOIN( 1, 2 ) )
```

```
= TRUE
```

## See also

EQUAL, GREATER\_OR\_EQUAL, LESS, LESS\_OR\_EQUAL

## GREATER\_OR\_EQUAL

### Syntax

```
<greaterorequal-expression> := 'GREATER_OR_EQUAL(' <Any> ',' <Any> ')'
```

```
<greaterorequal-expression> := <Any> '>=' <Any>
```

### Since

1.0

### Return-type

Boolean

### Description

This function tests if each value of the first argument is greater or equal than the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function you also can use the operator ">="

### Examples

```
GREATER_OR_EQUAL( 2, 1 )
```

```
= TRUE
```

```
2 >= 1
```

```
= TRUE
```

```
GREATER_OR_EQUAL( 2, 2 )
```

```
= TRUE
```

```
2 >= 2
```

```
= TRUE
```

```
GREATER_OR_EQUAL( NULL, 2 )
```

```
= NULL
```

```
GREATER_OR_EQUAL( JOIN( 2, 2 ), JOIN( 1, 2 ) )
```

```
= TRUE
```

### See also

EQUAL, GREATER, LESS, LESS\_OR\_EQUAL

## HASACCESS

### Syntax

```
<hasaccess-expression> := 'HASACCESS(' <Key> ')'
```

### Since

2.2.0

### Return-type

Boolean

### Description

This functions checks if the current user has access to the keys passed as the argument. This function may **only** be used for the definition of the access-rules inside the configuration editor.

If the argument is NULL then the functions returns also NULL.

### Examples

```
HASACCESS( Product.Manufacturer )
```

In an access rule for the Product dimension, this would grant access to a user for all products of the manufacturers he already has access to.

## HASCHILDREN

### Syntax

```
<haschildren-expression> := 'HASCHILDREN(' <Key> ')'
```

### Since

2.2.2

### Return-type

Boolean

### Description

This functions checks if the keys passed as argument have at least one child.

If NULL is passed as argument, the functions returns NULL.

### Examples

```
HASCHILDREN( Time:' Jan/2006' )
```

= TRUE

```
HASCHILDREN( Time:'01.01.2006' )
```

= FALSE

```
HASCHILDREN( NULL )
```

= NULL

### See also

ISLEAF

## HASKEYS

### Syntax

```
<haskeys-expression> := 'HASKEYS(' <Key> { ',' <Key> } ')'
```

### Since

1.1

### Return-type

Boolean

### Description

The function HASKEYS checks if the current selection (filter) contains at least one key from each argument (usually each argument is only one key). This is very useful for building match-expressions, e.g. for cubes or formulas. If any of the arguments is NULL, the function returns NULL.

### Examples

```
HASKEYS( Fact: Amount )
```

Does the current selection contain the fact "Amount"?

```
HASKEYS( Fact: Amount, Time:'Jan/2003' )
```

Does the current selection contain the fact "Amount" and January?

### See also

HASLEVEL, HASPOSITION

## HASLEVEL

### Syntax

```
<haslevel-expression> := 'HASLEVEL(' <Key> { ',' <Integer> } ')'
```

### Since

1.1

### Return-type

Boolean

### Description

Like the HASKEYS-function, this functions checks the keys from the current selection (filter). But this function checks the levels of a dimension (better all keys of the selection belonging to that dimension). The first argument is the dimension-name, all following argument are integers defining the levels. If all selected keys of the dimension match on of the levels, this function returns TRUE, otherwise FALSE. If any argument is NULL, the function returns NULL.

### Examples

```
HASLEVEL( Time, 1, 2 )
```

Is a time-key of level 1 or 2 selected?

### See also

HASKEYS, HASPOSITION

## HASPOSITION

### Syntax

```
<hasposition-expression> := 'HASPOSITION(' <Key> { ',' <Integer> } ')'
```

### Since

2.0

### Return-type

Boolean

### Description

This functions checks if the keys passed as argument have the position defined in the second argument. The position of a key is its position in its parent child-list, beginning with 0.

If any argument is NULL, the function returns NULL.

### Examples

```
HASPOSITION( Time:' Jan/2004', 0 )
```

```
= TRUE
```

```
HASPOSITION( Time:' Jan/2004', 1 )
```

```
= FALSE
```

### See also

POSITIONOF

## HASROLES

### Syntax

```
<hasroles-expression> := ' HASROLES(' <String> { ',' <String> } ')'
```

### Since

2.0

### Return-type

Boolean

### Description

This functions checks if the actual user has all of the roles passed as arguments.

If any of the arguments has more than one value, the user only needs to have one of the roles defines in this argument to pass the check.

If any of the arguments contains NULL the functions returns NULL.

### Examples

```
HASROLES( 'iolapAdmin', 'iolapUser' )
```

Is the current user admin and user?

```
HASROLES( 'iolapAdmin' | 'iolapUser' )
```

Is the current user admin or user?

### See also

HASUSER, USER

## HASUSER

### Syntax

```
<hasuser-expression> := 'HASUSER(' <String> { ',' <String> } ')'
```

### Since

2.0

### Return-type

Boolean

### Description

This functions checks if the actual user has one of the names passed as arguments. If any of the argument is NULL, the function returns NULL.

### Examples

```
HASUSER( 'guest', 'admin' )
```

Is the current user named 'guest' or 'admin'?

### See also

HASROLES, USER

## IIF

### Syntax

```
<iif-expression> := 'IIF(' <Boolean> ',' <Any> ',' <Any> ')'
```

### Since

1.2

### Return-type

The return-type of the second and third argument (the super-type of them).

### Description

The IIF-function allows the conditional evaluation of two arguments: Depending of the first argument's result (which is a boolean expression), the function will evaluate the second or third argument and returns its result or NULL:

- If argument 1 result to TRUE, the second argument will be evaluated and its result will be returned.

- If argument 1 result to FALSE, the third argument will be evaluated and its result will be returned.
- If argument 1 result to NULL, the result is NULL

## Examples

```
IIF( HASLEVEL( Time, 1 ), 'Year', IIF( HASLEVEL( Time, 2 ), 'Month', 'Day' ) )
```

Returns 'Year', 'Month' or 'Day', depending on the level of the selected time.

## See also

CASE, MATCH

# IN

## Syntax

```
<in-expression> := 'IN(' <Any> ',' <Any> ')'
```

```
<in-expression> := <Any> 'IN' <Any>
```

## Since

1.2

## Return-type

Boolean

## Description

This functions checks whether all values from the first argument are contained in the second argument.

Instead of this function you also can use the operator "IN".

## Examples

```
IN( Article.Color, 'Red' | 'Green' )
```

Is the current article red or green?

```
Article.Color IN ( 'Red' | 'Green' )
```

Same as above

## See also

CONTAINS, EXISTS

## INTERSECT

### Syntax

```
<intersect-expression> := 'INTERSECT(' <Any> { ',' <Any> } ')'
```

### Since

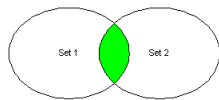
1.2

### Return-type

Supertype of all argument-types.

### Description

The function INTERSECT results in the intersection from all arguments, these are only the values contained in each argument. The return-type is the most common super type of all arguments.



### Examples

```
INTERSECT( Article.Customer, Region.Customer )
```

Returns the customers of the current article AND region

```
INTERSECT( ( 10 | 20 | 30 ), ( 20 | 30 | 40 ) )
```

```
= 20 | 30
```

### See also

JOIN

## ISCHILD OF

### Syntax

```
<ischildof-expression> := 'ISCHILD OF(' <Key> ',' <Key> { ',' <Key> } ')'
```

### Since

1.2

## Return-type

Boolean

## Description

This function checks whether all keys from the first argument are equal to or children of at least one of the keys from the second argument.

A key A is child of a key B, if:

- B is the direct parent of A,
- or the parent of A is a child of B

If more than two arguments are passed, all keys of the first arguments must be a child of at least one key of all following arguments

If one of the arguments is NULL, the function returns NULL. If the first argument is empty and contains no key, the function returns TRUE.

## Examples

```
ISCHILDOF( Time: 'Jan/2003', Time: '2003' )
```

```
= TRUE
```

```
ISCHILDOF( Time: '2003', Time: 'Jan/2003' )
```

```
= FALSE
```

```
ISCHILDOF( Time: '2003', NULL )
```

```
= NULL
```

```
ISCHILDOF( NULL, Time: 'Jan/2003' )
```

```
= NULL
```

## See also

ISPARENTOF

## ISEMPTY

### Syntax

```
<isempty-expression> := 'ISEMPTY(' <Any> ')'
```

### Since

2.5

## Return-type

Boolean

## Description

This function checks if the argument contains no elements. It is equal to "COUNT( <Any> ) = 0".

## Examples

```
ISEMPTY( EMPTY() )
```

```
= TRUE
```

```
ISEMPTY( 10 )
```

```
= FALSE
```

```
ISEMPTY( NULL )
```

```
= FALSE
```

## See also

ISNULL, EXISTS

# ISLEAF

## Syntax

```
<isleaf-expression> := ' ISLEAF( <Key> )'
```

## Since

2.6.1

## Return-type

Boolean

## Description

Return TRUE if the key passed as argument is a leaf and has no children. If more than one keys are passed as argument, the function return the same number of boolean values as result. If the argument is NULL, the function returns NULL.

## Examples

```
ISLEAF( Product:' All products' | NULL )
```

```
= FALSE | NULL
```

**See also**

HASCHILDREN

**ISNULL****Syntax**

```
<isnull-expression> := 'ISNULL(' <Any> ')'
```

**Since**

1.2

**Return-type**

Boolean

**Description**

This function checks if the argument contains any NULL.

**Examples**

```
ISNULL( NULL )
```

```
= TRUE
```

```
ISNULL( 10 )
```

```
= FALSE
```

```
ISNULL( ( 10 | NULL ) )
```

```
= TRUE
```

**See also**

EXISTS

**ISPARENTOF****Syntax**

```
<isparentof-expression> := 'ISPARENTOF(' <Key> ',' <Key> ')'
```

**Since**

1.2

## Return-type

Boolean

## Description

Checks whether all keys from the first argument are parents of each key from the second argument. A key A is father of a key B, if:

- B is directly placed under A,
- A child of A is parent of B

If one of the arguments is NULL, the function returns NULL. If the first argument is empty and contains no key, the function returns TRUE.

## Examples

```
ISPARENTOF( Time:'2003', Time:'Jan/2003' )
```

```
= TRUE
```

```
ISPARENTOF( Time:'Jan/2003', Time:'2003' )
```

```
= FALSE
```

```
ISPARENTOF( Time:'2003', NULL )
```

```
= NULL
```

```
ISPARENTOF( NULL, Time:'2003' )
```

```
= NULL
```

## See also

ISCHILD OF

## ITERATIONKEY

### Syntax

```
<iterationkey-expression> := 'ITERATIONKEY()'
```

### Since

2.2.0

### Return-type

Key

## Description

If this function is used inside a header and if the header was created by an header-iteration, this function returns the single iteration-key for this header.

## Examples

```
ITERATIONKEY( )
```

E.g. return 'Feb/2006' for one of the headers generated by the iteration 'Time::MONTH'.

## IVALUE

### Syntax

```
<ivalue-expression> := ' IVALUE( )'
```

### Since

2.5.0

### Return-type

Value

### Description

This function return the iterated value in the functions MATCH or FOREACH.

### Examples

```
MATCH( PRODUCT.Name, STARTWITH( { IVALUE( ) }, ' P' ) )
```

This example return all product names beginning with 'P'.

### See also

FKEY

## JOIN

### Syntax

```
<join-expression> := ' JOIN( ' <Any> { ',' <Any> } ' )'
```

```
<join-expression> := <Any> '|' <Any>
```

### Since

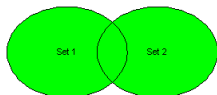
1.2

## Return-type

Depending of both argument's type (their supertype).

## Description

The function JOIN concatenates all results from all arguments to one joined result. The result of the first argument is added to result at first, the from the second and so on. If a argument is NULL, the NULL is simply added to result.



Instead of the function JOIN you also can use the operators "|".

## Examples

```
JOIN( LEVEL( Time, 1 ), LEVEL( Time, 2 ) )
```

is equal to LEVEL( Time, 1, 2 )

```
JOIN( NULL )
```

= NULL

## See also

DISTINCT, INTERSECT

## KEYINDEX

### Syntax

```
<keyindex-expression> := 'KEYINDEX( <Key> )'
```

### Since

2.6.0

### Return-type

Integer

### Description

Returns the internal index of the key(s). By default, the root key of a dimension has always the index 0, all other keys have number greater than zero.

If the argument is NULL, the function returns NULL.

## Examples

```
KEYINDEX( Product:' All products' )  
  
= 0
```

## See also

POSITIONOF

## LANGUAGE

### Syntax

```
<language-expression> := ' LANGUAGE( )'
```

### Since

2.2.0

### Return-type

String

### Description

Returns the language-code of the session of the current user.

### Examples

```
LANGUAGE( )  
  
= 'de'
```

### See also

COUNTRY, LOCALE

## LAST

### Syntax

```
<last-expression> := ' LAST( ' <Any> [ ',' <Integer> ] ' )'
```

### Since

1.2

## Return-type

The type of the first argument.

## Description

This function returns the last N values of the first argument. If you define no size N (with the second argument), the function only returns the last element of argument. If one of the arguments is NULL, the functions returns NULL.

## Examples

```
LAST( LEVEL( Time, 2 ) )
```

```
= Time:'Dez/2003'
```

```
LAST( LEVEL( Time, 2 ), 3 )
```

```
= Time:'Okt/2003' + Time:'Nov/2003' + Time:'Dez/2003'
```

```
LAST( NULL )
```

```
= NULL
```

## See also

FIRST

# LEAFS

## Syntax

```
<leafs-expression> := 'LEAFS(' <Any> ')'
```

## Since

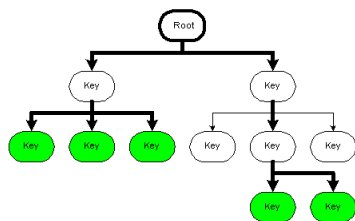
1.1

## Return-type

Key

## Description

Returns all keys below the argument, which do not have own children (they are called "leafs"). The leafs don't need direct children of the argument, but can also be further down in the hierarchies. If the argument is NULL, the function returns NULL. If the argument is only leafs, the function returns an empty result.



## Examples

```
LEAFS( Time:'2003' )
```

E.g. returns Time:'Jan/2003' + ... + Time:'Dez/2003'

```
LEAFS( NULL )
```

```
= NULL
```

## See also

CHILDREN, FAMILY, NONLEAFS

## LEFT

### Syntax

```
<left-expression> := 'LEFT(' <String> ',' <Integer> ')'
```

### Since

2.0

### Return-type

String

### Description

The function LEFT returns the N (defined by the second argument) left chars of the string passed as first argument. If the length of a string is smaller than N, the original string is returned. If any argument is NULL, the function returns NULL.

### Examples

```
LEFT( 'Hello world', 5 )
```

```
= 'Hello'
```

```
LEFT( NULL, 5 )
```

```
= NULL
```

```
LEFT( 'Hello world', NULL )
```

= NULL

### See also

RIGHT, SUBSTR

## LESS

### Syntax

```
<less-expression> := 'LESS(' <Any> ',' <Any> ')'
```

```
<less-expression> := <Any> '<' <Any>
```

### Since

1.0

### Return-type

Boolean

### Description

This function tests if each value of the first argument is less than the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function you also can use the operators "<".

### Examples

```
LESS( 1, 2 )
```

```
= TRUE
```

```
1 < 2
```

```
= TRUE
```

```
LESS( 2, 2 )
```

```
= FALSE
```

```
2 < 2
```

```
= FALSE
```

```
LESS( NULL, 2 )
```

```
= NULL
```

```
LESS( JOIN( 2, 3 ), JOIN( 1, 2 ) )
```

= FALSE

### See also

EQUAL, GREATER, GREATER\_OR\_EQUAL, LESS\_OR\_EQUAL, UNEQUAL

## LESS\_OR\_EQUAL

### Syntax

```
<lessorequal-expression> := 'LESSOREQUAL(' <Any> ',' <Any> ')'
```

```
<lessorequal-expression> := <Any> '<=' <Any>
```

### Since

1.0

### Return-type

Boolean

### Description

This function tests if each value of the first argument is less or equal to the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function you also can use the operator "<=".

### Examples

```
LESS_OR_EQUAL( 1, 2 )
```

= TRUE

```
1 <= 2
```

= TRUE

```
LESS_OR_EQUAL( 2, 2 )
```

= TRUE

```
2 <= 2
```

= TRUE

```
LESS_OR_EQUAL( NULL, 2 )
```

= NULL

```
LESS_OR_EQUAL( JOIN( 2, 3 ), JOIN( 2, 2 ) )
```

= TRUE

### See also

EQUAL, GREATER, GREATER\_OR\_EQUAL, LESS, UNEQUAL

## LEVEL

### Syntax

```
<level-expression> := 'LEVEL(' <Key> { ',' <Integer> } ')'
```

### Since

1.1

### Return-type

Key

### Description

The function returns one or more complete levels from the hierarchies of the dimension indicated with argument 1. The levels are defined by the optional arguments 2 - n. If the first argument is NULL or any level is NULL, the result is NULL. If no level is defined, an empty result is returned.

Note: This function is not affected by the current filter, it will always return complete and unfiltered levels.

### Examples

```
LEVEL( Time, 0 )
```

= Time:'Complete Period'

```
LEVEL( Time, 1, 2 )
```

= Time:'2003' + Time:'Jan/2003' + ... + Time:'Dez/2003'

```
LEVEL( Time )
```

= Empty result

```
LEVEL( NULL )
```

= NULL

```
LEVEL( Time, NULL )
```

= NULL

**See also**

ALL, LEVELOF

**LEVELNAMES****Syntax**

```
<levelnames-expression> := ' LEVELNAMES(' <Key> ')'
```

**Since**

2.1

**Return-type**

String

**Description**

This function returns the names of all hierarchy-levels of the dimension passed as argument. The order of the names is top down. If the argument is NULL, the function returns NULL.

**Examples**

```
LEVELNAMES( Time )
```

E.g. returns 'TimeRoot', 'Year', 'Month', 'Day'

**See also**

DEPTH

**LEVELOF****Syntax**

```
<levelof-expression> := ' LEVELOF(' <Key> ')'
```

**Since**

2.0

**Return-type**

Integer

## Description

Returns the number of the hierarchies-levels the key of the arguments belong to. The most upper level (which only the root-key belongs to) has the level 0, all deeper levels have higher numbers. If the argument is NULL, the function returns NULL.

## Examples

```
LEVELOF( Time: 'Complete Time' )
```

```
= 0
```

```
LEVELOF( Time: '2000' )
```

```
= 1
```

```
LEVELOF( Time: 'Jan/2000' )
```

```
= 2
```

```
LEVELOF( Time: '01.01.2000' )
```

```
= 3
```

```
LEVELOF( NULL )
```

```
= NULL
```

## See also

HASLEVEL, POSITIONOF

## LIKE

### Syntax

```
<like-expression> := 'LIKE(' <String> ',' <String> ')'
```

```
<like-expression> := <String> 'LIKE' <String>
```

### Since

2.5

### Return-type

Boolean

### Description

The LIKE function compares the string passed as first argument with the pattern passed as second argument and returns TRUE if the string matches the pattern or FALSE if not. The pattern can use '?' and '\*' characters as wildcards for a single or number of characters.

Instead of this function you also can use the operator "LIKE".

If any of the arguments is NULL, the function returns NULL.

## Examples

```
LIKE( 'Hello world', '*world' )
```

```
= TRUE
```

```
LIKE( 'Hello world', 'H?llo world' )
```

```
= TRUE
```

```
LIKE( 'Hello world', '*ship' )
```

```
= FALSE
```

```
LIKE( NULL, '*world' )
```

```
= NULL
```

## See also

REGEXP

# LIMIT

## Syntax

```
<limit-expression> := 'LIMIT(' <String> [ ',' <Integer> [ ',' <String> ] ] )'
```

## Since

1.2

## Return-type

String

## Description

This functions limits the size of the strings passed as arguments: If its size is greater than the maximum-size defined by argument 2, a string will be cut to this size and the chars "..." will be appended. Otherwise the original string is returned.

If the second argument is left out, the maximum-size will be the default length of 50. With the third and optional argument you can change the appendix "..." to any other string-pattern.

If any argument is NULL, the function returns NULL.

## Examples

```
LIMIT( 'Hello World', 8 )
```

```
= 'Hello...'
```

```
LIMIT( 'Hello World', 8, '!' )
```

```
= 'Hello!'
```

## See also

BEAUTIFY, SUBSTR, TOLOWER, TOUPPER

## LOCALE

### Syntax

```
<locale-expression> := ' LOCALE( )'
```

### Since

2.2.0

### Return-type

String

### Description

Returns the locale-code of the session of the current user. The locale contains both the country- and language-code.

### Examples

```
LOCALE( )
```

```
= 'DE_de'
```

### See also

COUNTRY, LANGUAGE

## LOOKUP

### Syntax

```
<lookup-expression> := ' LOOKUP( ' <Key> { ',' <Key> } ' )'
```

## Since

2.0

## Return-type

Key

## Description

This function evaluates, by immediately querying the cubes, for which keys of the first argument a fact (defined by the second argument) is available (not NULL). When querying the cubes, the current filter will be considered. The LOOKUP-function is very important whenever you build reports showing values for sparse filled cubes.

If any of the arguments is NULL, the function returns NULL.

## Examples

```
LOOKUP( LEVEL( Artikel, 1 ), Fact:Amount )
```

Returns all articles with a Amount stored in any cube for the current selection

## See also

DLOOKUP, SORT

# LTRIM

## Syntax

```
<ltrim-expression> := 'LTRIM(' <String> ')'
```

## Since

2.2.6

## Return-type

String

## Description

The function TRIM ommits all leading whitespaces from the input string passed as argument. If the argument is NULL the function also returns NULL.

## Examples

```
LTRIM( ' Hello World ' )
```

```
= 'Hello World '
```

## See also

RTRIM, TRIM

# MATCH

## Syntax

```
<match-expression> := 'MATCH(' <Value> ',' <Boolean> ')'
```

```
<match-expression> := <Value> '?' <Boolean>
```

## Since

1.2, changed in 2.5.0

## Return-type

The return type of the first argument

## Description

This function allows filtering a set of value with a boolean match-expression. It only returns the subset of the values defined by the first argument, for which the second argument result to TRUE.

Like the functions SORT and FOREACH, this function creates a copy of the current filter for each key and applies the key to it (if the first argument returns keys). Then the match-expression will be executed. If the result is TRUE, the value will be added to the result.

Since version 2.5.0, the function accepts all values as first argument. If the argument does not return keys, you can use the function IVALUE to access the iterated value.

Instead of this function you can also use the operator "?".

## Examples

```
MATCH( LEVEL( Product, 1 ), Product.ProductType = 'Type A' )
```

Returns all products with type 'Type A'

```
LEVEL( Product, 1 ) ? Product.ProductType = 'Type A'
```

Same as above

## See also

FIND, FOREACH, SORT, IVALUE, FKEY

# MATRIX

## Syntax

```
<matrix-expression> := 'MATRIX(' <Number> [ ',' <Number> [ ',' <Number>
> [ ',' <Number> ] ] } )'
```

## Since

2.0

## Return-type

Value

## Description

The MATRIX-function allows to access the values of cells inside the current sub result. You can use this function e.g. to build SUM-rows or -columns inside your queries.

The function expects at least one argument: The x-position of the cell in the same row, which value you want to access. If you want to access the value of another row, then you can pass the row-number as the second argument.

If only one or two arguments are passed, this function returns exactly one value. With the third and fourth argument you can define a hole area, then more values are returned: The third argument defines the maximum column, the fourth the maximum row to read. E.g. if you read an area of 5 columns and 5 rows, 25 values are returned. Empty cell return NULL.

In connection with this function the functions X, Y, MIN\_X, MIN\_Y, MAX\_X and MAX\_Y are also interesting. These functions can help you to find the current position of a cell or the size of a sub result.

If any of the arguments contains NULL values, the function will return NULL.

## Examples

```
MATRIX( X() + 1 )
```

Returns the value right to the current cell

```
ADD( TO_NUMBER( MATRIX( X(), MIN_Y(), X(), Y - 1() ) ) )
```

Creates a summary row

## See also

MAX\_X, MAX\_Y, MIN\_X, MIN\_Y, X, XHEADER, Y, YHEADER

## MAX

### Syntax

```
<max-expression> := 'MAX(' <Number> { ',' <Number> } ')'
```

### Since

1.0

### Return-type

Double

### Description

The MAX-function returns the greatest value of all values passed as argument. NULL values are not included into the comparison. If all arguments are NULL, the functions returns NULL.

### Examples

```
MAX( Amount( LEVEL( Time, 3 ) )
```

Returns the greatest Amount for all days

### See also

ADD, AVG, MIN, SUM

## MAXKEY

### Syntax

```
<maxkey-expression> := 'MAXKEY(' <Key> { ',' <Value> } ')'
```

### Since

2.2.6

### Return-type

Key

### Description

The MAXKEY function returns the key of the first argument with the greatest result for the expression passed as the second argument. If any of the arguments is NULL the function returns NULL.

## Examples

```
MAXKEY( PRODUCT, Amount() )
```

Returns the product with the greatest amount (for the current filter).

## See also

AVGKEY, MINKEY

## MAX\_X

### Syntax

```
<max_x-expression> := 'MAX_X()'
```

### Since

2.0

### Return-type

Integer

### Description

When executed in a header or cell, this function returns the X-coordinate of the last column in the current sub result. If this functions is used outside a header, an error will be raised.

## Examples

```
MATRIX( MAX_X(), Y() )
```

Returns the value of the last column in the current row

## See also

MATRIX, MAX\_Y, MIN\_X, MIN\_Y, X, XHEADER, Y, YHEADER

## MAX\_Y

### Syntax

```
<max_y-expression> := 'MAX_Y()'
```

### Since

2.0

## Return-type

Integer

## Description

When executed in a header or cell, this function returns the Y-coordinate of the last row in the current sub result. If this functions is used outside a header, an error will be raised.

## Examples

```
MATRIX( X() , MAX_Y() )
```

Returns the value of the last row in the current column

## See also

MATRIX, MAX\_X, MIN\_X, MIN\_Y, X, XHEADER, Y, YHEADER

# MIN

## Syntax

```
<max-expression> := 'MIN(' <Number> { ',' <Number> } )'
```

## Since

1.0

## Return-type

Double

## Description

The MIN-function returns the smallest value of all values passed as argument. NULL values are not included into the comparison. If all arguments are NULL, the functions returns NULL.

## Examples

```
MIN( Amount( LEVEL( Time, 3 ) )
```

Returns the smallest Amount for all days

## See also

ADD, AVG, MAX, SUM

## MINKEY

### Syntax

```
<minkey-expression> := 'MINKEY(' <Key> { ',' <Value> } ')'
```

### Since

2.2.6

### Return-type

Key

### Description

The MINKEY function returns the key of the first argument with the smallest result for the expression passed as the second argument. If any of the arguments is NULL the function returns NULL.

### Examples

```
MINKEY( PRODUCT, Amount() )
```

Returns the product with the smallest amount (for the current filter).

### See also

AVGKEY, MAXKEY

## MIN\_X

### Syntax

```
<min_x-expression> := 'MIN_X()'
```

### Since

2.0

### Return-type

Integer

### Description

When executed in a header or cell, this function returns the X-coordinate of the first non-header column in the current sub result. If this functions is used outside a header, an error will be raised.

## Examples

```
MATRIX( MIN_X(), MIN_Y(), MAX_X(), MIN_Y() )
```

Returns all values of the first row under the header

## See also

MATRIX, MAX\_X, MAX\_Y, MIN\_Y, X, XHEADER, Y, YHEADER

## MIN\_Y

### Syntax

```
<min_y-expression> := 'MIN_Y()'
```

### Since

2.0

### Return-type

Integer

### Description

When executed in a header or cell, this function returns the Y-coordinate of the first non-header row in the current sub result. If this functions is used outside a header, an error will be raised.

## Examples

```
MATRIX( MIN_X(), MIN_Y(), MIN_X(), MAX_Y() )
```

Returns all values of the column right of the header

## See also

MATRIX, MAX\_X, MAX\_Y, MIN\_X, X, XHEADER, Y, YHEADER

## MIX

### Syntax

```
<mix-expression> := 'MIX(' <Any> { ',' <Any> } ')'
```

### Since

2.5

## Return-type

Common supertype of all arguments.

## Description

The MIX function mixes all lists of all arguments by adding all first elements of all arguments, the all seconds elements of all arguments (and so on) to the result.

If any argument has less elements than the other arguments, the function will continue adding the elements of the others arguments after all elements of the smaller arguments are processed.

This function treats NULL like any other value and NULL arguments will be merged to the result list.

## Examples

```
MIX( ( 'A' | 'B' | 'C' ), ( 'D' | 'E' | 'F' ) )
```

```
= 'A' | 'D' | 'B' | 'E' | 'C' | 'F'
```

```
MIX( ( 'A' | 'B' | 'C' ), ( 'D' ) )
```

```
= 'A' | 'D' | 'B' | 'C'
```

```
MIX( ( 'A' | 'B' | 'C' ), NULL )
```

```
= 'A' | NULL | 'B' | 'C'
```

# MOD

## Syntax

```
<mod-expression> := 'MOD(' <Number> ',' <Number> ')'
```

```
<mod-expression> := <Number> '%' <Number>
```

## Since

1.2

## Return-type

Double

## Description

This function computes the remainder of dividing argument one by argument two. If one of both arguments is NULL, the result is NULL.

Instead of this function you also can use the operator "%".

## Examples

```
MOD( 5, 2 )
```

```
= 1
```

```
94 % 10
```

```
= 4
```

```
MOD( NULL, 1 )
```

```
= NULL
```

```
10 % NULL
```

```
= NULL
```

## See also

ADD, DIV, MUL, SUM

# MUL

## Syntax

```
<mul-expression> := 'MUL(' <Number> ',' <Number> ')'
```

```
<mul-expression> := <Number> '*' <Number>
```

## Since

1.0

## Return-type

Double

## Description

This function multiplies all values of all arguments. If any of the values is NULL, the function returns NULL.

Instead of this function you also can use the operator "\*" .

## Examples

```
MUL( 10, 42 )
```

```
= 420
```

```
10 * 42
```

= 420

10 \* 42 \* NULL

= NULL

## See also

ADD, DIV, MOD, SUM

## NEIGHBOURS

### Syntax

```
<neighbours-expression> := ' NEIGHBOURS(' <Key> ' ) '
```

### Since

1.2

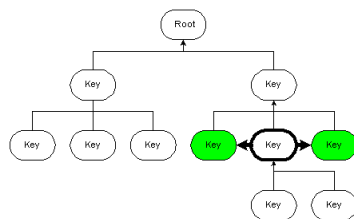
### Return-type

Key

### Description

Determines all "neighbours" of the keys which are passed as argument. The "neighbours" of a key are all other keys of the dimension with the same father, without key itself.

If the argument is NULL, the function returns NULL.



### Examples

```
NEIGHBOURS( Time: ' May/2003' )
```

E.g. returns Time:'Apr/2003' + Time:'Jun/2003'

## See also

CHILDREN, LEVEL, PARENT

## NEXT

### Syntax

```
<next-expression> := 'NEXT(' <Key> [ ',' <Integer> [ ',' <Integer> ] ]
')
```

### Since

1.0

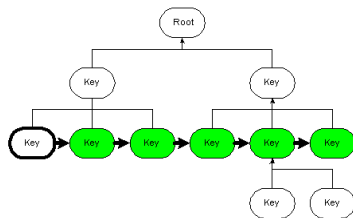
### Return-type

Key

### Description

Evaluates the following keys of the keys from the argument 1. The successor of a key is the next key within the same level of the dimension - the successor must not have compellingly the same father like the key himself.

If no distance (with argument 2) is defined, the directly following key is returned. If a distance N is defined, the key with the distance N from the key is determined. E.g. NEXT (time, 2) corresponds NEXT (NEXT (time)).



With the third argument you can also define the number of keys you want the function to return. The default value is 1.

If the argument is NULL, the result is NULL. If the keys have no successor, because they are last within the level, the result empty.

### Examples

```
NEXT( Time:'Jan/2003' )
```

```
= Time:'Feb/2003'
```

```
NEXT( Time:'Jan/2003', 3 )
```

```
= Time:'Apr/2003'
```

```
NEXT( Time:'Dec/2003' )
```

```
= Empty set
```

```
NEXT( NULL )
```

= NULL

### See also

PREV, SUCC, UPPERNEXT, UPPERPREV

## NONFACTROOTS

### Syntax

```
<nonfactroots-expression> := 'NONFACTROOTS()'
```

### Since

1.2

### Return-type

Key

### Description

Returns the root-keys of all dimensions without the fact-dimension.

### Examples

```
NONFACTROOTS( )
```

E.g. returns Time:'Complete Period' + ... + Product:'All Products'

### See also

FACTROOT

## NONLEAFS

### Syntax

```
<nonleafs-expression> := 'NONLEAFS(' <Key> ')'
```

### Since

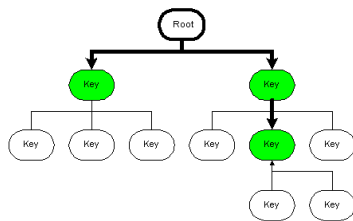
1.2

### Return-type

Key

## Description

The function NONLEAFs determines all keys below the argument which also have own children. The leafs of the hierarchies are not added to the result. If the argument is NULL, the function returns NULL.



## Examples

```
NONLEAFs( Time:'2000' )
```

E.g. returns Time:'Jan/2000', ..., Time:'Dec/2000'

## See also

ANCESTORS, CHILDREN, FAMILY, LEAFs, PARENT

## NOT

### Syntax

```
<not-expression> := 'NOT(' <Boolean> ')'
```

### Since

1.0

### Return-type

Boolean

### Description

This logical function negates the boolean argument:

- If the argument is TRUE, the function returns FALSE
- If the argument is FALSE, the function returns TRUE
- If the argument is NULL, the function returns NULL

### Examples

```
NOT( TRUE )
```

```
= FALSE
```

```
NOT( FALSE )
```

```
= TRUE
```

```
NOT( NULL )
```

```
= NULL
```

### See also

AND, OR

## NOW

### Syntax

```
<now-expression> := 'NOW( ' <Key> ',' <String> ' )'
```

### Since

1.2

### Return-type

Key

### Description

The function NOW finds the key from a time dimension which corresponds to the actual day, month, year etc. Such a function is necessary, because in instantOLAP there is no special time dimension, so you have to find the current time-key in your own time-dimension using this function.

To find the current time key, you must pass the time dimension as the first argument. With the second argument you must define a time-pattern, with which the desired key is searched.

### Examples

```
NOW( Time, 'yyyy' )
```

```
= Time:'2002'
```

```
NOW( Time, 'MMM/yyyy' )
```

```
= Time:'Jan/2002'
```

```
NOW( Time, 'MMM/yy' )
```

```
= NULL
```

# OR

## Syntax

```
<or-expression> := 'OR(' <Boolean> ',' <Boolean> ')'
```

```
<or-expression> := <Boolean> 'OR' <Boolean>
```

## Since

1.0

## Return-type

Boolean

## Description

The logical operator OR returns TRUE, FALSE or NULL depending on the arguments' values:

- If one of the arguments result to TRUE TRUE, the result is also TRUE.
- If both arguments result to FALSE, the result is also FALSE.
- If one of the arguments results to NULL the result is also NULL.

Instead of this function you also can use the operator "or".

## Examples

```
OR( TRUE, TRUE )
```

```
= TRUE
```

```
OR( FALSE, TRUE )
```

```
= TRUE
```

```
OR( TRUE, FALSE )
```

```
= TRUE
```

```
OR( TRUE, NULL )
```

```
= TRUE
```

```
OR( FALSE, NULL )
```

```
= NULL
```

```
OR( NULL, NULL )
```

```
= NULL
```

## See also

AND, NOT

# PARENT

## Syntax

```
<parent-expression> := ' PARENT(' <Key> ')'
```

## Since

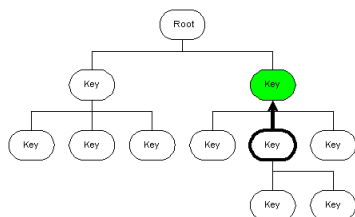
1.0

## Return-type

Key

## Description

The function PARENT determines the parent of a key, this is the key directly above the argument in the hierarchies. All keys except the root-key have a parent. If the argument is NULL, this function returns NULL.



## Examples

```
PARENT( Time:' Jan/2003' )
```

E.g. returns Time:'Q1/2003'

```
PARENT( Time:' Complete Period' )
```

Returns NULL (if Time:'Complete Period' is the root-key)

```
PARENT( NULL )
```

= NULL

## See also

CHILDREN, FAMILY, ISPARENTOF, PEDIGREE

## PEDIGREE

### Syntax

```
<pedigree-expression> := ' PEDIGREE(' <Key> ')'
```

### Since

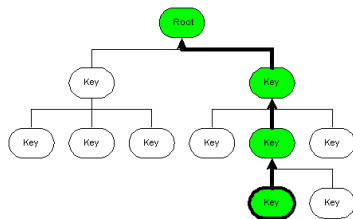
2.1.2

### Return-type

Key

### Description

Returns the keys passed as arguments and all their ancestors. This function returns a distinct and ordered list - even if multiple keys from the argument have the same ancestors they will only appear once in the result.



If the argument is NULL, the function returns NULL.

### Examples

```
NEIGHBOURS( Time:'Jun/2004', Time:'May/2004' )
```

E.g. returns Time:'Complete Period' + Time:'2004' + Time:'May/2004' + Time:'Jun/2004'

### See also

ANCESTORS

## PERCENTILE

### Syntax

```
<percentile-expression> := ' PERCENTILE(' <Number> ',' <Number> ')'
```

### Since

2.2.6

## Return-type

Integer

## Description

The function PERCENTILE calculates the percentile (defined by the second argument as a value between 0 and 100) for the values passed as the first argument.

## Examples

```
PERCENTILE( 10 | 15 | 21 | 18, 90 )
```

# POSITIONOF

## Syntax

```
<positionof-expression> := ' POSITIONOF(' <Key> ' )
```

## Since

2.0

## Return-type

Integer

## Description

The function POSITIONOF determines the position of a key below his parent and returns it. The first key has always the position 0. If the argument contains more than one key, the corresponding number of positions is returned. If the argument is NULL, the function returns NULL.

## Examples

```
POSITIONOF( Time:'01.01.2002' )
```

= 0

```
POSITIONOF( Time:'03.01.2002' )
```

= 2

## See also

LEVELOF

## POW

### Syntax

```
<pow-expression> := 'POW(' <Number> ',' <Number> ')
```

### Since

2.2.6

### Return-type

Number

### Description

Returns the value of the first argument raised to the power of the second argument. If any argument is NULL, the function returns also NULL.

### Examples

```
POW( 2, 3 )
```

```
= 2 * 2 * 2 = 8
```

### See also

SQRT

## PRED

### Syntax

```
<pred-expression> := 'PRED(' <Key> [ ',' <Integer> [ ',' <Integer> [ ',' <Integer> ] ] ] )'
```

### Since

2.2

### Return-type

Key

### Description

Returns the predecessors of the keys passed as first argument. In difference to the function PREV, this function does not return the directly previous key but a child of the previous key of an ancestor. This child will have the same position within its ancestor than the argument.

From which ancestor the previous will be used is defined (with its level-number) with the second argument.

You can use this function to find a corresponding key in a different part of the keys hierarchy, e.g. to find the same month in the previous year.

Like in the PREV function, you can also define the distance of the first key (here with the third argument) and the number of keys (fourth argument) you wish to return.

If any of the argument is NULL, this function returns NULL.

## Examples

```
PRED( Time, 1 )
```

Returns the corresponding day, month etc. of the previous year

```
PRED( Time, 2 )
```

Returns the corresponding day of the previous month

```
PRED( Time, 1, 2 )
```

Returns the corresponding day, month etc. of the previous of the previous year

```
PRED( Time, 1, 1, 3 )
```

Returns the corresponding days, months etc. of the last 3 previous years

```
PRED( NULL, 1 )
```

= NULL

```
PRED( Time, NULL )
```

= NULL

## See also

PREV, SUCC

## PREV

### Syntax

```
<prev-expression> := 'PREV(' <Key> [ ',' <Integer> [ ',' <Integer> ] ]  
' )'
```

### Since

1.0



## RANGE

### Syntax

```
<range-expression> := ' RANGE(' <Key> ',' <Key> ')'
```

### Since

2.2.2

### Return-type

Key

### Description

This function returns the two keys from the arguments and all keys between them. If both keys come from different dimensions, the function will return an empty result. If any of the arguments is NULL, the function will return also NULL.

If both keys come from the same dimension but from different levels inside the dimension, the result will not contain the key with the higher level but its first child (or descendant) within the same level.

If the position of the second argument within the dimension is lower than the position of the first, the function will swap them. The result always starts with the key with the first position.

The function also accepts more than one key for each argument. Then it will return multiple ranges, one for each combination of start- and end-key.

### Examples

```
RANGE( Time:'Jan/2006', Time:'Apr/2006' )
```

```
= Time:'Jan/2006' | Time:'Feb/2006' | Time:'Mar/2006' | Time:'Apr/2006'
```

```
RANGE( Time:'Apr/2006', Time:'2006' )
```

```
= Time:'Jan/2006' | Time:'Feb/2006' | Time:'Mar/2006' | Time:'Apr/2006'
```

```
RANGE( Time:'Jan/2006', NULL )
```

```
= NULL
```

## REGEXP

### Syntax

```
<regexp-expression> := ' REGEXP(' <String> ',' <String> ')'
```

**Since**

2.5

**Return-type**

String

**Description**

The REGEXP function parses the first argument with the regular expression defined by the second arguments and returns all text fragments of the first arguments which match the expression.

If any of the arguments is NULL the function will return NULL.

**Examples**

```
COUNT( REGEXP( 'Hello world', 'l|o' ) )
```

= 5 (the number of occurrences of the characters 'l' and 'o' in the argument)

```
COUNT( REGEXP( 'Hello world', NULL ) )
```

= NULL

**See also**

LIKE

**REGRESSION****Syntax**

```
<regression-expression> := 'REGRESSION(' <Number> ',' <Key> ',' <Key> [ ',' <Integer> ] )'
```

**Since**

2.2

**Return-type**

Double

**Description**

Calculates the regression for the expression defined with the first argument for the data-point(s) defined by the keys passed as argument 3.

For the calculation of the regression you'll also have to define an input-dataset. This can be done with the second argument, which has to be a number of keys for which the expression will be executed. The results of this calculations will then be taken as the input-dataset.

The optional 4th argument allows to shift result up or down to the upper or lower border of the regression (then the regression-line will touch the most upper or lower point of the input-dataset).

## Examples

```
REGRESSION( Amount(), LEVEL( Time, 3 ), Time )
```

Uses all existing values of Amount for the third-time level to calculate the regression for the current Time key.

## See also

FORECAST, VARIANCE

# REPLACE

## Syntax

```
<replace-expression> := 'REPLACE(' <String> ',' <String> ',' <String> ')'
```

## Since

2.0

## Return-type

String

## Description

The replace-function replaces all occurrences of the string defined as argument two in argument one by the string defined in argument three. If any of the arguments is NULL, the function returns NULL.

## Examples

```
REPLACE( 'Hello World', 'World', 'Customer' )
```

```
= 'Hello Customer'
```

## See also

LEFT, RIGHT, STRLEN, SUBSTR

## RETURNTYPE

### Syntax

```
<returntype-expression> := 'RETURNTYPE(' <Any> ')'
```

### Since

2.2.0

### Return-type

String

### Description

This debugging-function returns the name of the return-type passed as the argument. The argument will not be evaluated, only its return-type will be determined.

### Examples

```
RETURNTYPE( 10 )
```

```
= 'Integer'
```

```
RETURNTYPE( NULL )
```

```
= 'All'
```

```
RETURNTYPE( NEXT( Time ) )
```

```
= 'Time'
```

### See also

DEBUG, FILTER, TYPE

## REVERSE

### Syntax

```
<reverse-expression> := 'REVERSE(' <Any> ')'
```

### Since

1.2

### Return-type

The type of the argument.

## Description

This function returns all values from the argument in their reverse order.

## Examples

```
REVERSE( 1 | 2 | 3 )
```

```
= 3 | 2 | 1
```

```
REVERSE( LEVEL( Time, 3 ) )
```

E.g. returns Time:'Dec/2003' + ... + Time:'Jan/2003'

## See also

INTERSECT, JOIN, WITHOUT

# RIGHT

## Syntax

```
<right-expression> := 'RIGHT(' <String> ',' <Integer> ')'
```

## Since

2.0

## Return-type

String

## Description

The function RIGHT returns the N (defined by the second argument) right chars of the string passed as first argument. If the length of a string is smaller than N, the original string is returned. If any argument is NULL, the function returns NULL.

## Examples

```
RIGHT( 'Hello world', 5 )
```

```
= 'world'
```

```
RIGHT( NULL, 5 )
```

```
= NULL
```

```
RIGHT( 'Hello world', NULL )
```

```
= NULL
```

**See also**

LEFT, SUBSTR

**ROUND****Syntax**

```
<round-expression> := ' ROUND( ' <Number> ' ) '
```

**Since**

2.0

**Return-type**

Integer

**Description**

The function ROUND returns the closest integer to the argument. The result is rounded to an integer by adding 1/2, taking the floor of the result, and casting the result to type integer.

If the argument is NULL, the function returns NULL.

**Examples**

```
ROUND( 0.5 )
```

```
= 1
```

```
ROUND( 0.4 )
```

```
= 0
```

```
ROUND( NULL )
```

```
= NULL
```

**See also**

CEIL, FLOOR

**ROWNUM****Syntax**

```
<rownum-expression> := ' ROWNUM( [ <Integer> [ ',' <Integer> ] ] ) '
```

## Since

2.2

## Return-type

Key (members of the LINE\_DIMENSION)

## Description

This function is needed to create detail-reports which show non-aggregated data directly taken from datasources (e.g. SQL-databases).

Because for non-aggregating queries there is nothing you can iterate on the Y-axis of a pivot-table in order to create rows (or on the X-axis in order to create columns), there is a system-dimension named "LINE\_DIMENSION" which contains an endless number of keys representing simple line-numbers. This function helps you to calculate how many line-numbers the query will need in order to display the complete result of the query.

The function has no arguments. It returns members of the LINE\_DIMENSION (beginning with the first key LINE\_DIMENSION:'1') which can then be used in the iteration of a pivot-table.

*When creating a list in the query-editor, this function will be automatically used as the iteration of the Y-axis. Also, the column containing this function will be set invisible to hide the line numbers.*

With both optional arguments you can define the maximum number of rows you want to create (defined by the first argument) and the position of the first line you want to create (defined by the second argument). If any of these arguments is NULL the functions return an empty set.

## Examples

```
ROWNUM( )
```

Returns the number of needed keys of the LINE\_DIMENSION

```
ROWNUM( 100 )
```

Returns the first 100 keys of the LINE\_DIMENSION (or less if there are less rows in the datasource).

```
ROWNUM( 100, 50 )
```

Returns 100 keys of the LINE\_DIMENSION (or less) beginning with key 50.

# ROWSPAN

## Syntax

```
<rowspan-expression> := ' ROWSPAN(' <Integer> ')'
```

## Since

2.2.2

## Return-type

Integer

## Description

This function returns the number of rows spanned by the header with the given x-position (passed as argument). This function is e.g. useful to create subtotals for tables with grouped headers.

## Examples

```
SUM( TONUMBER( MATRIX( X(), Y() - 1, X(), Y() - ROWSPAN(0) ) ) ) )
```

Returns a subtotal for all rows spanned by the same, grouping header in the Y-axis.

## See also

ROWSPAN

# RTRIM

## Syntax

```
<rtrim-expression> := 'LTRIM(' <String> ')'
```

## Since

2.2.6

## Return-type

String

## Description

The function TRIM ommits all trailing whitespaces from the input string passed as argument. If the argument is NULL the function also returns NULL.

## Examples

```
RTRIM( ' Hello World ' )
```

```
= ' Hello World'
```

## See also

LTRIM, TRIM

## SORT

### Syntax

```
<sort-expression> := 'SORT(' <Key> ',' <Value> [ ',' <Integer> [ ',' <Boolean> ] ] )'
```

### Since

2.0

### Return-type

Key

### Description

With the SORT-function a set of keys can be sorted by any criteria. The first argument defines the key-set to be sorted, the second argument is a value-expression which defines the sort-criteria for the keys. Like the MATCH and FOREACH-functions, this function executes the criteria-expression for each key of the key-set (with respect to the current filter). The keys then will be sorted by their criteria-results.

With the (optional) third argument you can define a size-limit for your result. This enables you to pick the greatest or smallest N keys out of the key-set. The default-value for this argument is NULL, which doesn't limit the result-size.

The (optional) fourth argument defines the order of the result: If the argument is TRUE, the keys will be sorted by a descending order. If the argument is FALSE (default), the keys will be sorted in an ascending order.

Like the LOOKUP-function, this function may directly query a cube to perform its operation, e.g. by sending a SQL-query to a database.

### Examples

```
SORT( LEVEL( Article, 3 ), Artikel.Farbe )
```

Returns all articles sorted by their colors

```
SORT( LEVEL( Article, 3 ), Amount(), 10, true )
```

Returns the top-10 articles sorted by their Amount

### See also

DSORT, GREATER, LOOKUP, MATCH

## SPLINE

### Syntax

```
<spline-expression> := 'REGRESSION(' <Number> ',' <Key> ',' <Key> )'
```

## Since

2.2

## Return-type

Double

## Description

This function calculates a spline graph for the expression defined by the first argument. It returns the spline-value for the data-point defined by the second argument.

For the calculation of the spline you'll also have to define an input-dataset. This can be done with the third argument, which has to be a number of keys for which the expression will be executed. The results of this calculations will then be taken as the input-dataset.

## Examples

```
SPLINE( Amount(), TIME, LEVEL( Time, 3 ) )
```

## See also

REGRESSION

# SPLIT

## Syntax

```
<split-expression> := 'SPLIT(' <String> ',' <String> ')'
```

## Since

2.0

## Return-type

String

## Description

The SPLIT-function can be used to split a string into parts. The first argument is the string to split, the second argument defines the delimiter by that the string should be split. The result is a number of strings without the delimiter. If any argument is NULL, the function returns NULL.

## Examples

```
SPLIT( ' Split, this, string', ',' )
```

```
= 'Split' | 'this' | 'string'
```

```
SPLIT( NULL, ',' )
```

= NULL

```
SPLIT( 'Split,this,string', NULL )
```

= NULL

### See also

LEFT, RIGHT, STRLEN, SUBSTR

## SQRT

### Syntax

```
<sqrt-expression> := 'SQRT(' <Number> ')'
```

### Since

2.6.0

### Return-type

Number

### Description

Returns the square root of the numbers passed as argument. If NULL is passed as argument, the functions returns NULL.

### Examples

```
SQRT( 9 )
```

= 3

### See also

POW

## STARTSWITH

### Syntax

```
<startswith-expression> := 'STARTSWITH(' <String>, <String> ')'
```

### Since

2.1

## Return-type

Boolean

## Description

This function determines if the first argument starts with the string passed as second argument. If any of the argument is NULL, the function returns NULL.

## Examples

```
STARTSWITH( 'Hello World', 'World' )
```

```
= FALSE
```

```
STARTSWITH( 'Hello World', 'Hello' )
```

```
= TRUE
```

```
STARTSWITH( 'Hello World', 'NULL' )
```

```
= NULL
```

## See also

ENDSWITH

# STRLEN

## Syntax

```
<strlen-expression> := 'STRLEN( ' <String> ' )'
```

## Since

2.0

## Return-type

Integer

## Description

The function STRLEN returns the length of the string passed as arguments. If the argument is NULL, the function returns NULL.

## Examples

```
STRLEN( 'Hello world' )
```

```
= 11
```

```
STRLEN( NULL )
```

```
= NULL
```

### See also

LEFT, RIGHT, SPLIT, SUBSTR, TEXTPOSITION

## SUB

### Syntax

```
<number-expression> := 'SUB(' <Number> ',' <Number> ')'
```

```
<number-expression> := <Number> '-' <Number>
```

### Since

1.0

### Return-type

Number

### Description

This function evaluates the difference between argument 1 and argument. If one of the arguments is NULL, the function returns NULL.

Instead of this function you also can use the operator "-".

### Examples

```
SUB( 10, 5 )
```

```
=> 5
```

```
-10 - 20
```

```
= -30
```

```
SUB( 10, NULL )
```

```
= NULL
```

### See also

ADD, DIV, MUL

## SUBSTR

### Syntax

```
<substr-expression> := 'SUBSTR(' <String> ',' <Integer> ',' <Integer> ')'
```

### Since

2.0

### Return-type

String

### Description

The function SUBSTR returns a new string that is a substring of the string passed as first argument. The substring begins with the character at index specified by the second argument and extends to the character at the index defined by the third argument - 1. Thus the length of the substring is end Index - begin Index. If any of the arguments is NULL, the function returns NULL.

### Examples

```
SUBSTR( 'Hello World', 0, 5 )
```

```
= 'Hello'
```

```
SUBSTR( NULL, 1, 5 )
```

```
= NULL
```

### See also

LEFT, RIGHT, STRLEN, SUBSTRINGBEHIND

## SUBSTRINGBEHIND

### Syntax

```
<substringbehind-expression> := 'SUBSTRINGBEHIND(' <String> ',' <String> ')'
```

### Since

2.2.3

### Return-type

String

## Description

This function returns the substring of the first argument behind the first occurrence of the second argument. If it does not occur, the function will return NULL.

If any of the arguments is NULL, the function will return NULL.

## Examples

```
SUBSTR( 'Hello World', ' ' )
```

```
= 'World'
```

## See also

SUBSTR

# SUBTOTAL

## Syntax

```
<number-expression> := 'SUBTOTAL(' [ <String> [ ',' <boolean> ] ] )'
```

## Since

2.5

## Return-type

Double

## Description

This function can be used to add totals and subtotals to a pivot table. The function will add all previous values in the same column (if the function is used in a Y-header formula) or in the same row (if the function is used in a X-header formula) and return the summarized value.

The first an optional argument allows to define a variable name for the subtotal. With this variable name, you can use the same subtotal function multiple times in the same row or column, because the function will not summarize values which have been generated with the SUBTOTAL function and the same variable name.

If no variable name is given, the system uses the variable 'DEFAULT' as default name.

The boolean second and also optional argument determines, if the subtotal is absolute (FALSE) or incremental (TRUE). 'Absolute' means, it **only** adds the values since the last row or column the SUBTOTAL was calculated (if the variable name is equals). The 'incremental' mode will add the last SUBTOTAL value plus all new values since them.

The default value for this argument is 'FALSE' (therefore absolute).

## Examples

```
SUBTOTAL( )
```

```
SUBTOTAL( ' CATEGORY' )
```

```
SUBTOTAL( ' CATEGORY' , true)
```

## SUBTOTALS

### Syntax

```
<subtotals-expression> := ' SUBTOTALS( ' <Key> ' )'
```

### Since

2.2

### Return-type

Key

### Description

This function inserts additional keys to the result of the argument in order to add summary lines to an iteration: After each key, the parent key will be displayed if the following key is not child of the same parent. Also, the parent of the parent will be added if it changes and so on.

This function is very similar to the PEDIGREE function but the higher aggregated keys are inserted after the detailed keys.

If the argument is NULL, the function returns NULL.

### Examples

```
SUBTOTALS( LEVEL( TIME, 3 ) )
```

Returns all days with added keys for months, years and the total.

### See also

ANCESTORS, PEDIGREE

## SUCC

### Syntax

```
<succ-expression> := 'SUCC( ' <Key> [ ',' <Integer> [ ',' <Integer> [ ',' <Integer> ] ] ] )'
```

## Since

2.2

## Return-type

Key

## Description

Returns the successors of the keys passed as first argument. In difference to the function NEXT, this function does not return the directly following key but a child of the following key of an ancestor. This child will have the same position within its ancestor than the argument.

From which ancestor the following key will be used is defined (with its level-number) with the second argument.

You can use this function to find a corresponding key in a different part of the keys hierarchy, e.g. to find the same month in the next year.

Like in the NEXT function, you can also define the distance of the first key (here with the third argument) and the number of keys (fourth argument) you wish to return.

If any of the argument is NULL, this function returns NULL.

## Examples

```
SUCC( Time, 1 )
```

Returns the corresponding day, month etc. of the next year

```
SUCC( Time, 2 )
```

Returns the corresponding day of the next month

```
SUCC( Time, 1, 2 )
```

Returns the corresponding day, month etc. of the next after the next year

```
SUCC( Time, 1, 1, 3 )
```

Returns the corresponding days, months etc. of the next 3 following years

```
SUCC( NULL, 1 )
```

= NULL

```
SUCC( Time, NULL )
```

= NULL

## See also

NEXT, PRED

## SUM

### Syntax

```
<sum-expression> := 'SUM(' <Number> [ ',' { <Number>} ] ')'
```

### Since

2.0.3

### Return-type

Double

### Description

This function returns the sum of all values passed as arguments. In difference to the ADD-function, the SUM-function only accepts and returns numerical values as arguments and return-value.

If any of the arguments is NULL, this function returns the sum of all other values. If all arguments are NULL, it return NULL.

### Examples

```
SUM( 10, 20 )
```

```
= 30
```

```
SUM( Amount( CHILDREN( Product ) ) )
```

Returns the sum of all amounts of the sub-products

### See also

ADD

## SWITCH

### Syntax

```
<switch-expression> := 'SWITCH(' <Any> { ',' <Any> ',' <Any> } ')'
```

### Since

2.5.0

### Return-type

The common supertype of all return-value arguments.

## Description

The SWITCH function calculates its first arguments and then compares it to all match values of the following match / return value pairs. Whenever a match-value is equal, the function will return the corresponding return value and stop the evaluation. If no match-value matches, the function will return NULL.

The SWITCH function is similar to the CASE function but calculates only the first value and then compares to other (mostly constant) arguments. Therefore it can be faster and easier to use in some cases.

If the first argument is NULL, the function will return NULL without comparing any values.

## Examples

```
SWITCH( Project.State, 0, 'Not begun', 1, 'In progress', 2, 'Finished'
)
```

Translates the numerical project-state to texts.

## See also

CASE

## SYSDATE

### Syntax

```
<sysdate> := 'SYSDATE()'
```

### Since

2.5

### Return-type

Date

### Description

Returns the current system date.

## TEXTPOSITION

### Syntax

```
<textposition-expression> := 'TEXTPOSITION(' <String>, <String> ')'
```

### Since

2.2

## Return-type

Integer

## Description

If the text defined by the second argument is contained in the text defined by the first argument, this function returns the position of its first char within the other text (beginning with 0 for the first possible position in a text).

If the text is not contained, the function NULL. Also, if any of the arguments is NULL, the function will return NULL.

## Examples

```
TEXTPOSITION( 'Hello world', 'Hello' )
```

```
= 0
```

```
TEXTPOSITION( 'Hello world', 'world' )
```

```
= 6
```

```
TEXTPOSITION( 'Hello world', 'planet' )
```

```
= NULL
```

```
TEXTPOSITION( NULL, 'world' )
```

```
= NULL
```

```
TEXTPOSITION( 'Hello world', NULL )
```

```
= NULL
```

## See also

STRLEN, SUBSTR

## TIMESTAMP

### Syntax

```
<timestamp-expression> := 'TIMESTAMP(' <String> ')'
```

### Since

1.2

### Return-type

String

## Description

This function returns the current date and time as a string. The format is defined by the date-format-pattern passed as second argument. If the format is NULL, the function returns NULL.

## Examples

```
TIMESTAMP( 'yyyy' )
```

```
= '2002'
```

```
TIMESTAMP( 'yyyyMMddHHmm' )
```

```
= '200208011321'
```

## See also

NOW

## TODATE

### Syntax

```
<todate-expression> := 'TODATE(' <Any> [ ',' <String> ] )'
```

### Since

2.2

### Return-type

Date

### Description

This function converts the value passed as argument (if possible) to a date:

- If the argument is a string, the string will be parsed. For the parsing of the string the date-format defined by the second argument will be used. If no format is defined, the function will use the standard date-format depending on the current locale settings.
- If the argument is a date, the date will be date
- If the argument is a number, the number will be used to construct a date using the number as milliseconds.
- For any other type, the function returns NULL

If the argument is NULL, the function returns NULL.

### Examples

```
TODATE( '01.08.2005', 'dd.MM.yyyy' )
```

= 01.08.2005

TODATE( NULL )

= NULL

### See also

TOINTEGER, TONUMBER, TOSTRING

## TOINTEGER

### Syntax

<tointeger-expression> := 'TOINTEGER(' <Any> ')'

### Since

2.2

### Return-type

Integer

### Description

This function converts the value passed as argument (if possible) to an integer value

- If the argument is an integer, the integer will be returned unchanged
- If the argument belongs to any other number-type, it will be returned without its fraction.
- If the argument is a string, it will be parsed. If the strings contains a fraction, this fraction will be ignored.
- For any other type, the function returns NULL

If the argument is NULL, the function returns NULL.

### Examples

TOINTEGER( 42 )

= 42

TOINTEGER( 42.2 )

= 42

TOINTEGER( '42.2' )

= 42

TOINTEGER( NULL )

= NULL

### See also

TODATE, TONUMBER, TOSTRING

## TOKEY

### Syntax

```
<tokey-expression> := 'TOIKEY(' <Any> ')'
```

### Since

2.2.2

### Return-type

Key

### Description

Converts the argument into a list of keys. In fact, this function only filters and returns the keys and NULL values from the first argument and ignores all other values.

### Examples

```
TOKEY( Time:'2006' )
```

```
= Time:'2006'
```

```
TOKEY( Time:'2006' | 100 )
```

```
= Time:'2006'
```

```
TOKEY( NULL )
```

```
= NULL
```

### See also

TODATE, TONUMBER, TOSTRING

## TOLOWER

### Syntax

```
<tolower-expression> := 'TOLOWER(' <String> ')'
```

## Since

1.2

## Return-type

String

## Description

Converts all of the characters in the String to lower case. If the argument is NULL, the function returns NULL.

## Examples

```
TOLOWER( 'Hello World' )
```

```
= 'hello world'
```

## See also

BEAUTIFY, TOUPPER

# TONUMBER

## Syntax

```
<tonumber-expression> := ' TONUMBER( ' <Any> ' ) '
```

## Since

1.2

## Return-type

Number

## Description

This function converts the value passed as argument (if possible) to a number:

- If the argument is a string, the string will be parsed. Depending on the string, the returned value will be a integer- or double-value or NULL (if the string is not convertible)
- If the argument is a number, the number will be returned
- For any other type, the function returns NULL
- If the argument is NULL, the function returns NULL.

## Examples

```
TONUMBER( '10' )
```

```
= 10  
  
TONUMBER( '10.0' )  
  
= 10  
  
TONUMBER( '10.2' )  
  
= 10.2  
  
TONUMBER( 10 )  
  
= 10  
  
TONUMBER( 'Hello World' )  
  
= NULL  
  
TONUMBER( NULL )  
  
= NULL
```

### See also

TODATE, TOINTEGER, TOSTRING

## TOSTRING

### Syntax

```
<tostring-expression> := 'TOSTRING(' <Any> [ ',' <String> ] )'
```

### Since

1.1

### Return-type

String

### Description

The TOSTRING-function converts arguments of any type into a string. According to the type of the argument, different conversions are used:

- Strings are returned without any modification
- Integer-values are converted into a text without post comma places
- Double-values are converted into a text with post comma places
- Boolean-values result to 'TRUE' or 'FALSE'
- Date-values are converted to string using the locale format
- Keys are converted to their ID

- NULL results to NULL

With the (optional) second argument you can refine the conversion: By passing a decimal format as second argument, you can determine exactly the format of converted numbers. Depending on the value-type this must be a data- or number-format.

## Examples

```
TOSTRING( 10 )
```

```
= '10'
```

```
TOSTRING( 10.2 )
```

```
= '10.2'
```

```
TOSTRING( 10.2, '0.00' )
```

```
= '10.20'
```

```
TOSTRING( Time: 'Jan/2003' )
```

```
= 'Jan/2003'
```

```
TOSTRING( TRUE )
```

```
= 'TRUE'
```

```
TOSTRING( NULL )
```

```
= NULL
```

## See also

TODATE, TOINTEGER, TONUMBER

## TOUPPER

### Syntax

```
<toupper-expression> := 'TOUPPER(' <String> ')'
```

### Since

1.2

### Return-type

String

## Description

Converts all of the characters in the String to upper case. If the argument is NULL, the function returns NULL.

## Examples

```
TOUPPER( 'Hello World' )
```

```
= 'HELLO WORLD'
```

```
TOUPPER( NULL )
```

```
= NULL
```

## See also

BEAUTIFY, TOLOWER

## TRIM

### Syntax

```
<trim-expression> := 'TRIM(' <String> ')'
```

### Since

2.2.6

### Return-type

String

### Description

The function TRIM ommits all leading and trailing whitespaces from the input string passed as arguments. If the argument is NULL the function also returns NULL.

### Examples

```
TRIM( ' Hello World ' )
```

```
= 'Hello World'
```

### See also

LTRIM, RTRIM

## TOUPPER

### Syntax

```
<toupper-expression> := 'TUPLE(' [ <Key> { ',' <Key> } ' )'
```

### Since

2.5

### Return-type

Coordinate

### Description

This function creates tuples from all keys passed as arguments and returns them as Coordinates.

### Examples

```
TUPLE( Period:2008, CustomerA: | Customer:'B' )
```

```
= (Period:2008,Customer:A) | (Period:2008,Customer:A)
```

## TYPE

### Syntax

```
<type-expression> := 'TYPE(' <Any> ')'
```

### Since

2.2

### Return-type

String

### Description

This function is for debugging purpose. It returns the name of the type of the value returned by the argument (the name of the Java-class representing the type).

If the argument is NULL, the function returns NULL.

### Examples

```
TYPE( 'Hello world' )
```

```
= 'java.lang.String'
```

```
TYPE( NULL )
```

```
= NULL
```

### See also

DEBUG, FILTER, RETURNTYPE

## UNEQUAL

### Syntax

```
<unequal-expression> := 'UNEQUAL(' <Any> ',' <Any> ')'
```

```
<unequal-expression> := <Any> '<>' <Any>
```

### Since

1.0

### Return-type

Boolean

### Description

This function checks whether the two arguments are unequal. Two values are unequal when they are not equal, so this function returns the negated result of the EQUAL-function. If any argument is NULL, this function returns NULL.

Instead of this function you can also use the operator "<>".

### Examples

```
UNEQUAL( 10, 20 )
```

```
= TRUE
```

```
10 <> 20
```

```
= TRUE
```

```
UNEQUAL( 10, 10 )
```

```
= FALSE
```

```
UNEQUAL( 10, 'Hello World' )
```

```
= TRUE
```

```
UNEQUAL( 10, NULL )
```

```
= NULL
```

**See also**

EVAL, GREATER, GREATER\_OR\_EQUAL, LESS, LESS\_OR\_EQUAL

**UNIT****Syntax**

```
<unit-expression> := 'UNIT(' <Key> ')'
```

**Since**

2.2.1

**Return-type**

String

**Description**

Returns the unit(s) of the fact(s) passed as arguments. If the argument is NULL or if the keys passed as the arguments are no facts or if the facts don't have any unit, then the function returns NULL.

**Examples**

```
UNIT( Fact: Turnaround )
```

```
= 'EUR'
```

```
UNIT( NULL )
```

```
= NULL
```

```
UNIT( Time: 2006 )
```

```
= NULL
```

**UPPERNEXT****Syntax**

```
<uppernext-expression> := 'UPPERNEXT(' <Key> ')'
```

**Since**

1.1

**Return-type**

Key

## Description

The function UPPERNEXT determines, just as the function NEXT, the successor of a key. But in contrast to the function NEXT, for the last child of a parent not the successor of the key is returned, but the successor of the parent.

This function and the UPPERPREV-function are very useful for generating year-to-date aggregates, because you can use higher aggregated levels for your calculations.



If the argument is NULL, this function returns NULL.

## Examples

```
UPPERNEXT( Time: ' Nov/2003' )
```

```
= Time: 'Dec/2003'
```

```
UPPERNEXT( Time: ' Dec/2003' )
```

```
= Time: '2004'
```

```
UPPERNEXT( NULL )
```

```
= NULL
```

## See also

NEXT, PREV, UPPERPREV

## UPPERPREV

### Syntax

```
<upperprev-expression> := ' UPPERPREV( ' <Key> ' )'
```

### Since

1.1

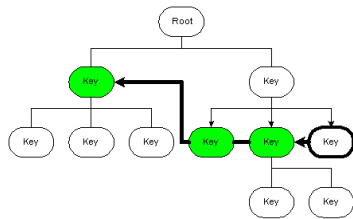
### Return-type

Key

### Description

The function UPPERPREV determines, just as the function PREV, the predecessor of a key at its level. But in contrast to function NEXT, for the first child of a parent not the predecessor of the key is returned, but the predecessor of the parent.

This function and the UPPERNEXT-function are very useful for generating year-to-date aggregates, because you can use higher aggregated levels for your calculations.



If the argument is NULL, this function returns NULL.

## Examples

```
UPPERPREV( Time: 'Feb/2004' )
```

```
= Time: 'Jan/2004'
```

```
UPPERPREV( Time: 'Jan/2004' )
```

```
= Time: '2003'
```

```
UPPERPREV( NULL )
```

```
= NULL
```

## See also

NEXT, PREV, UPPERNEXT

# USER

## Syntax

```
<User-Expression> := 'USER()'
```

## Since

2.0

## Return-type

String

## Description

The function USER returns the name of the current user. Using this function is equivalent to access the variable \$USER.

## Examples

```
USER()
```

E.g. returns 'admin' (means the current user is 'admin')

### See also

HASROLES, HASUSER

## VARIANCE

### Syntax

```
<variance-expression> := 'VARIANCE(' <Number> [ ',' { <Number> } ] )'
```

### Since

2.2

### Return-type

Double

### Description

Calculates the variance for the input dataset defined by all arguments. If all arguments are NULL, the function returns NULL. Otherwise, all NULL values will be ignored.

### Examples

```
VARIANCE( 10, 15, 7, 10 )
```

```
= 11.0
```

### See also

REGRESSION

## WITHOUT

### Syntax

```
<without-expression> := 'WITHOUT(' <Any> ',' <Any> )'
```

### Since

1.2

### Return-type

Common supertype of both arguments.

## Description

The function WITHOUT returns all values from the first argument which are not found in the second argument. If any of the arguments is NULL, the functions returns NULL

## Examples

```
WITHOUT( CHILDREN( Time:'2004' ), Time:'Jan/2004' )
```

Returns all months of the year 2004 except january

## See also

INTERSECT, JOIN

## XML2ASCII

### Syntax

```
<xml2ascii-expression> := 'XML2ASCII(' <String> ')
```

### Since

2.6.0

### Return-type

String

### Description

This function removes all XML or HTML markup elements from the passed string and returns the rest as string. If NULL is passed as argument, the function returns NULL.

### Examples

```
XML2ASCII( "<p style=' color: red;'>Hello <b>world</b></p>")
```

```
= "Hello world"
```

## X

### Syntax

```
<x-expression> := 'X()'
```

### Since

2.0

## Return-type

Integer

## Description

This function returns the current X-position (column-number) of the header or cell when this function is being executed inside pivot-tables. If the X-function is executed outside a pivot-table, an error is raised.

## Examples

```
background="IIF( X() % 2 = 0, 'grey', 'white' )"
```

E.g. you can use the function X() in query-headers to colorize the header-backgrounds.

## See also

MATRIX, MAX\_X, MAX\_Y, MIN\_X, MIN\_Y, XHEADER, Y, YHEADER

# XHEADER

## Syntax

```
<xheader-expression> := ' XHEADER( ' <String> ' )'
```

## Since

2.0

## Return-type

Integer

## Description

With help of the XHEADER function, the position of one or more X-headers inside the current pivot-table can be found. The argument is a search-pattern, which is used to find the headers on the x-axis. The result is a list of integers, representing the headers' positions, or NULL if no header matches.

In the pattern you can use the wildcards "\*" and "?" for specifying the headers name. If the pattern is NULL the functions returns NULL.

The usage of this function outside pivot-tables is invalid.

## Examples

```
XHEADER( ' Sum' )
```

Returns the position of the header(s) with the Text 'Sum'

```
XHEADER( 'Article*' )
```

Returns the positions of all headers starting with 'Article'

### See also

MATRIX, MAX\_X, MAX\_Y, MIN\_X, MIN\_Y, X, Y, YHEADER

## Y

### Syntax

```
<y-expression> := 'Y()'
```

### Since

2.0

### Return-type

Integer

### Description

This function returns the current Y-position (row-number) of the header or cell when this function is being executed. If the Y-function is executed outside a header, an error is raised.

### Examples

```
background="IIF( Y() % 2 = 0, 'grey', 'white' )"
```

E.g. you can use the function Y() in query-headers to colorize the header-backgrounds.

### See also

MATRIX, MAX\_X, MAX\_Y, MIN\_X, MIN\_Y, X, XHEADER, YHEADER

## YHEADER

### Syntax

```
<yheader-expression> := 'YHEADER( ' <String> ' )'
```

### Since

2.0

### Return-type

Integer

## Description

With help of the YHEADER function the position of one or more Y-headers inside the sub result can be determined. The argument is a search-pattern, which is used to find the headers on the y-axis. The result is an integer-set, representing the headers' positions, or NULL if no header matches.

In the pattern you can use the wildcards '\*' and '?' for specifying the headers name. If the pattern is NULL the functions returns NULL.

The usage of this function outside pivot-tables is invalid.

## Examples

```
YHEADER( ' Sum' )
```

Returns the position(s) of the header with the Text 'Sum'

```
YHEADER( ' Article*' )
```

Returns the positions of all headers starting with 'Article'

## See also

MATRIX, MAX\_X, MAX\_Y, MIN\_X, MIN\_Y, X, XHEADER, Y

## YTD

### Syntax

```
<ytd-expression> := ' YTD( ' <Key> ', ' <Key> ' ) '
```

### Since

2.2

### Return-type

Key

### Description

The YTD-function can be used to collect all predecessors of a key which belong to one of it ancestors (are children of). Which ancestor will be used to collect is defined by the second argument. The key itself is always added to the result.

If any of the arguments is NULL, the function returns NULL.

You can use this function to build e.g. the summarized sales from the beginning of the current year till now (therefore YTD means year-to-date function).

## Examples

```
Amout( YTD( Time, Time::YEAR ) )
```

Totals sales from 01.01 till now

## See also

PRED, PREV

# ZERO

## Syntax

```
<zero-expression> := 'ZERO(' <Number> ')'
```

## Since

1.1

## Return-type

Number

## Description

The function ZERO converts all NULL-values from the argument into the integer-number "0". All other values are return without any manipulation. Because most function can't deal with NULL, you can use this function to pre-convert NULLs into a real number.

## Examples

```
ZERO( Amount() )
```

```
AVG( Zero( Amount( LEAFS( Time ) ) ) )
```

## See also

CUBE, EXISTS, ISNULL

# CHAPTER 5:

## Formats

### Contents of this chapter:

Date formats .....	514
Number formats .....	516
Cron patterns .....	517
Colors .....	518

# Date formats

---

## Syntax

Date-Formats are defined by assembling elements from the following table to a pattern which will be used to convert dates into a string. Each element will be replaced by the corresponding value from the date when converting it.

Symbol	25,% Meaning	Presentation	Example
G	era designator	(Text)	AD
y	Year	(Number)	1996
M	month in year	(Number & Text)	July & 07
q	quarter in year	(Number)	2
d	day in month	(Number)	10
h	hour in am/pm (1~12)	(Number)	12
H	hour in day (0~23)	(Number)	0
m	minute in hour	(Number)	30
s	second in minute	(Number)	55
S	millisecond	(Number)	989
E	day in week	(Text)	Tuesday
D	day in year	(Number)	190
F	day of week in month	(Number)	2 (2nd Wed in July)
w	week in year	(Number)	27
W	week in month	(Number)	2
a	am/pm marker	(Text)	PM

k	hour in day (1~24)	(Number)	24
K	hour in am/pm (0~11)	(Number)	0
z	time zone	(Text)	Pacific Time Standard
'	escape for text	(Delimiter)	
''	single quote	(Literal)	'

## Examples

' dd. MM. yyyy'

E.g. returns '01.10.2004'

' dd. MMM yyyy'

E.g. returns '01. October 2004'

' EEE'

E.g. returns 'Monday'

# Number formats

---

## Syntax

Number-Formats are defined by assembling elements from the following table to a pattern which will be used to convert numbers into a string. Each element will be replaced by the corresponding value from the number when converting it.

Symbol	Meaning
0	a digit
#	a digit, zero shows as absent
,	placeholder for decimal separator
E	separates mantissa and exponent for exponential formats
;	separates formats
-	default negative prefix
%	multiply by 100 and show as percentage
?	multiply by 1000 and show as per mille
Â±	currency sign; replaced by currency symbol; if doubled, replaced by international currency symbol
.	If present in a pattern, the monetary decimal separator
X	any other characters can be used in the prefix or suffix
'	used to quote special characters in a prefix or suffix

## Examples

'00#'

E.g. returns '005'

'#.###.##0.00'

E.g. returns '7,532,238.00'

'000%'

E.g. returns 037%

# Cron patterns

---

## Syntax

```
cron-pattern := [ [ [ [ <month-pattern> ' ' ] <day-pattern> ' ' ] <day-of-week-pattern> ' ' ] <hour-pattern> ' ' ] <minute-pattern>
```

```
month-pattern := '*' | { (1..12) { , (1..12) }
```

```
day-pattern := '*' | { (1..31) { , (1..31) }
```

```
day-of-week-pattern := '*' | { (1..7) { , (1..7) }
```

```
hour-pattern := '*' | { (0..23) { , (0..23) }
```

```
minute-pattern := '*' | { (0..59) { , (0..59) }
```

## Description

Cron-Patterns are used in various elements of instantOLAP, e.g. for triggering synchronizing dimensions or the automatic sending of Mails. Cron-Patterns contain a single pattern for each part of the time: One for the month, one for the day, one for the day-of week, one for the hour and one for the minute. Each part can be "\*" (stays for each month, each day and so on) or a comma-separated list of months, days, weekdays, hours or minutes (represented as numbers).

The cron-pattern matches if each part matches the corresponding part of the time (same month, same day etc.) or the corresponding part of the pattern is "\*".

## Examples

```
*
```

Every minute

```
* 0
```

Every hour (00:00, 01:00, ..., 23:00)

```
* 0 0
```

Every day at midnight

```
* 1 0
```

Every day at one a clock

# Colors

---

## Syntax

```
<color> := <predefined-color> | '#' <hex> <hex> <hex>
```

## Description

In instantOLAP, each color information is represented as a String containing a pre-defined color-name or a hexadecimal RGB value. Pre-defined color-names are simple names as 'red', 'green' or 'blue'. For the RGB-representation the String must begin with a '#', followed by a 6-digit hexadecimal value with 2 digits for the red-, green- and blue-value.

## Pre-defined Colors

white, black, red, green, blue, light\_grey, grey, dark\_grey, orange, yellow, cyan

## Examples

```
' #ffffff'
```

white

```
' #000000'
```

black

```
' #ff0000'
```

red

```
' #00ff00'
```

green

```
' #0000ff'
```

blue

```
' red'
```

red

```
' green'
```

green

```
' blue'
```

blue

# CHAPTER 6:

# SQL-Expressions

## Contents of this chapter:

SQL-Expressions in instantOLAP .....	520
Syntax .....	522

# SQL-Expressions in instantOLAP

---

The SQL-Keyloaders and the SQL-Cubes use SQL-Expression for their definition. SQL-Expressions have a SQL-like syntax but only are fragments of a whole SQL-statement. The statements are assembled out of this fragments by the instantOLAP SQL-generator every time a loader or cube queries the database.

## Expression-Types

SQL-Expression also use types but the type-system is less restrictive than the one for expressions. Mainly the system only distinguishes between Strings, Numbers and Boolean-Expressions. Boolean-Expression are expected for all WHERE properties inside the loaders and cubes, all other properties allow any type.

## The SQL-Generator

Every time a loader or cube executes a query, the SQL-statement will be generated before. Basically each time a statement is generated the following steps will be performed:

1. All needed SQL-expressions are collected
2. All needed filters (WHEREs) are collected
3. All used tables are collected from the expressions and filters to generate the FROM clause
4. All links between the used tables are collected to generate the joins in the WHERE clause
5. All non-aggregating expressions are added to the GROUP BY clause

If more than one table is used, the generator will automatically gather all available links between this tables and add them to the statements. If one or more tables can't be linked to the others (because there is no defined or imported link between them and the others), the generator will stop and raise an error. Selecting values from multiple tables without links is not supported!

## Generator-hints in the database-settings

Though instantOLAP generates statements in the ANSI SQL/92 syntax, some SQL features are still database-dependent:

- The concat-operator: Most databases use the ANSI syntax (||) to concatenate strings. But some database (MS Access, MS SQL Server, MySql) use different operator, e.g. the plus-sign or custom-functions.
- The maximum number of values in a WHERE clause using IN: instantOLAP uses IN-clauses to filter expressions for a large number of values (e.g. to filter products by their IDs). This IN-clauses can be become very long and may exceed the maximum length of lists or overall statements. In this case, the SQL-generator will split the statement and execute multiple queries. These lengths are also database-dependent.
- The usage of aliases in WHERE, GROUP BY and ORDER BY clauses: Some database don't allow the usage of complex expressions or functions in the GROUP BY, WHERE or ORDER BY clause and expect aliases instead. Other database don't support aliases in the clauses but allow complex expressions.
- Custom functions and custom aggregations: Most databases have custom functions which are not part of the ANSI SQL/92 syntax but can be very useful.

To cover these features, the database allows to configure most of the database-dependent properties (concat-operator, maximum value count for IN, usage of aliases) in their definition. Also, for the most common database-types these settings are already pre configured and will be used automatically when a pre-configured database is detected.

Only the custom-functions and -aggregation have to be inserted manually into the SQL-expressions by using the SQL Pass-through.

# Syntax

---

## Tables and columns

### Syntax

```
<column-expression> := [ <schema-name> '.' ] <table-name> '.' <column-name>
```

```
<schema-name> := <string> | '"' <string> '"'
```

```
<table-name> := <string> | '"' <string> '"'
```

```
<column-name> := <string> | '"' <string> '"'
```

### Description

This is the easiest way of accessing a column of specific table inside your database. Like in pure SQL, you'll have to pass the table-name with a following '.' and the column name. In difference to SQL you always must provide the name of the table, a stand-alone column-name is invalid.

Optionally you can access tables from other schemata's than the default schema of your database-connection (if your database supports schemata's). In this case, you'll have to provide a preceding schema-name if front of the table-name.

Whenever schema-, table- or column-names contains white spaces or special characters, you can embed its name in delimiters (" or ') to pass it to the instantOLAP SQL-generator. The generator will then automatically embed the name with the database's name-delimiter. Whenever you create SQL-expressions inside the workbench using drag&drop, the names are embedded in delimiters anyway.

### Examples

```
SALES. AMOUNT
```

Accesses the column "AMOUNT" of the table "SALES"

```
ERM. SALES. AMOUNT
```

Accesses the column "AMOUNT" of tables "SALES" in schema "ERM"

```
' SALES' .' AVG AMOUNT'
```

Accesses the column "SALES" in table "AVG AMOUNT" (which has a whitespace in its name)

## Constants

### Syntax

```
<sql-constant> := <string-constant> | <number-constant> | <boolean-constant> | <null-constant>
```

```
<string-constant> := '"' <string> '"'
```

```
<number-constant> := <integer-constant> | <double-constant>
```

```
<integer-constant> := ['-'] { <digit> }
```

```
<double-constant> := ['-'] { <digit> } '.' { <digit> }
```

```
<boolean-constant> := 'TRUE' | 'FALSE'
```

```
<null-constant> := 'NULL'
```

```
<asterix-constant> := '*'
```

### Description

Like in "real" SQL you can use constants in your SQL-expression. There are several types of constants (integer-, double-, boolean- or string-constants and NULL) you can use.

### Examples

```
'Hello World'
```

```
10
```

```
-10
```

```
10.5
```

```
-10.5
```

```
TRUE
```

```
FALSE
```

```
NULL
```

```
*
```

## Operators

### Syntax

```
<operator-expression> := <binary-operator-expression>
```

<binary-operator-expression> := <sql-expression> <operator> <sql-expression>

### Description

instantOLAP supports mostly all common SQL-operators, including "LIKE" and "IS NULL". See the following table for all SQL-operators being supported:

Op.	Meaning	Remark	Example
			String Concatenation
+	Addition	For Numbers and Strings	
-	Subtraction		
*	Multiplication		SALES.QUANTITY * SALES.PRICE
/	Division		SALES.AMOUNT / SALES.PRICE
%	Modulo		
<	Less		SALES.PRICE < 2.00
<=	Less Or Equal		SALES.PRICE <= 2.00
>	Greater		SALES.PRICE > 2.00
>=	Greater or Equal		SALES.PRICE >= 2.00
=	Equal		
LIKE	Pattern Matching	? and % maybe used as placeholder	CUSTOMER.NAME LIKE 'A%'
		IN	Value in List?
AND	Boolean And		
OR	Boolean Or		
NOT	Boolean Not		
IS NULL	Test for NULL		CUSTOMER.NAME IS NULL
IS NOT NULL	Test for not NULL		CUSTOMER.NAME IS NOT NULL

Though the concat-operator is often database-dependent (e.g. MS SQL Server and MS Access except "+" as the concat-operator), all expressions in instantOLAP should use the expression "||" for concatenation. The database-definition in instantOLAP allows to change the operator for

a database and replaces all occurrences of "||" when generating SQL-statements. Also, for the most common database-types, the operator is already pre configured.

## Functions

### Syntax

```
<function-expression> := <aggregation-expression> | <passthrough-expression>
```

### Description

There are two different ways of using functions inside instantOLAP-SQL: Calling aggregation-functions (the only type of functions ANSI-SQL supports) and calling custom (database-specific) function in the SQL pass through syntax.

The first way is the most common and mainly used for binding your facts to the fact-tables. The second way is very generic and maybe used for both, aggregating and other calculations or transformation inside SQL.

See the description of both ways for more details.

### Examples

```
SUM( SALES.AMOUNT )
```

Uses the standard-SQL function to sum all amounts

```
@TOCHAR( SALES.DATE, 'YYYY' )
```

Converts a date to its year

## Aggregation functions

### Syntax

```
<aggregation-expression> := <sum-expression> | <min-expression> | <max-expression> | <avg-expression> | <count-expression> | <count-distinct-expression> || <custom-aggregation-expression>
```

```
<sum-expression> := 'SUM( <sql-expression> )'
```

```
<min-expression> := 'MIN( <sql-expression> )'
```

```
<max-expression> := 'MAX( <sql-expression> )'
```

```
<avg-expression> := 'AVG( <sql-expression> )'
```

```
<count-expression> := 'COUNT( <sql-expression> )'
```

```
<count-distinct-expression> := 'COUNT( DISTINCT <sql-expression> )'
```

```
<custom-aggregation-expression> := '@@' <function-name> '(' <sql-expression> ')'
```

## Description

Aggregation functions are used for mapping facts to a fact-table. In most cases, the facts should be mapped in an aggregated way, so the executed SQL-statement will automatically return the aggregated value for the selected dimension-entries.

Using an aggregation-function will automatically let the instantOLAP SQL-generator append GROUP BY clauses to the generated SQL-statements.

There are several ANSI SQL aggregation-functions: SUM, MIN, MAX, AVG and COUNT or COUNT DISTINCT. Most databases offer more than these functions but their syntax differ from database to database. If you want to use one of these custom-functions, you'll have to use the SQL pass through feature of instantOLAP (which passes any SQL-expression to the database without performing a syntax-check).

## Examples

```
SUM( SALES.AMOUNT )
```

Returns the SUM of all amounts

```
MIN( SALES.AMOUNT )
```

Returns the minimum amount

```
MAX( SALES.AMOUNT )
```

Returns the maximum amount

```
AVG( SALES.AMOUNT )
```

Returns the average amount

```
COUNT( SALES.CUSTOMER_ID )
```

Counts the number of customers

```
COUNT( DISTINCT SALES.CUSTOMER_ID )
```

Counts the (distinct) number of customers

```
COUNT( * )
```

Counts the number of records

```
@@LEAST( SALES.AMOUNT )
```

Uses the database-specific function LEAST (MySQL)

## SQL Passthrough

### Syntax

```
<passthrough-expression> := <function-passthrough-expression> | <full-passthrough-expression>
```

```
<function-passthrough-expression> := [ '@' | '@@' ] <string> '(' { <sql-expression> } ')'
```

```
<full-passthrough-expression> := [ '@' | '@@' ] ''' <string> '''
```

### Description

The instantOLAP SQL-parser only knows the ANSI-SQL 92 functions (like SUM, MIN, MAX etc) and raises an error whenever an unknown function is used in SQL-expression. However, it is possible to use custom or database-specific SQL-expression with this pass-through syntax.

There are two different ways of defining a pass through: Only to define the unknown function-name or escaping the whole expression. The first way means, you only escape the function name but all following arguments of the functions should be checked.

The second way is to pass the whole expression, including its argument, without being checked by instantOLAP. The advantage of the first way is that you still can be sure the rest of your expression is correct and contains no spelling-errors. The second way is more flexible but less safe, because you can pass anything you want.

To escape only a function-name, you must use the passthrough-operator "@" followed by the function-name (e.g. @SUBSTR). To escape a whole expression, a string (encapsulated in delimiters) must immediately follow the operator. The content of this string will then be included in the generated SQL-expression without any change.

There are also two different escape-characters for beginning the passthrough-sequence: The simple "@" escape-character and the aggregating one "@@". Since SQL differs between aggregating and non-aggregating functions, you have to tell the instantOLAP SQL-generator which kind of expression your escaped expression is. This is very important, because the SQL-generator will only append necessary GROUP BY clauses to the SQL-statements when the escaped expression is signed as being aggregating.

### Examples

```
@SUBSTR( SALES.DATE, 1, 4 )
```

The custom-function "SUBSTR" is passed though, but all arguments are still checked

```
@"CONVERT( CHAR(10) SALES.DATE )"
```

Full 'passthrough' of the expression

```
@@' COUNT( DISTINCT SALES.CUSTOMER_ID )'
```

Passes a whole (aggregating) expression without any check

## Brackets

### Syntax

```
<bracket-expression> := '(' <sql-expression> ')'
```

### Description

Brackets in SQL-Expression allow to control the evaluation-order of its sub-expressions. By encapsulating parts of your expression in brackets, you will force the database to evaluate the content of the brackets first before using their result with the outer operators. You also may encapsulate multiple brackets - then the expressions will be executed from the inner to the outer.

### Examples

```
( 10 + 20 ) * 30
```

```
= 900
```

```
10 + 20 * 30
```

```
= 610
```

```
( CUSTOMER.FIRSTNAME IS NULL ) OR ( CUSTOMER.LASTNAME IS NULL )
```