Process functions

The PQL process functions allow you to calculate process specific metrics with simple function calls.

Throughput times, process patterns and flows and process variants are some example on the calculations facilitated by the

Content

Process equals

Available in version: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
process equals [START] first_input [[TO ANY] TO intermediate_input]*
[END] // this is how it should be (maybe)
process [NOT] equals [START] activity [TO activity]* [END] // this is
the actual grammar
```

where each activity is exactly one of

- ANY
- 'activity_name'

Arguments

Inputs

The input arguments are the names of the activities in the eventlog. The activity name has to be wrapped in '. e.g.: **process equals 'My first activity'**

START

START defines the starting activity of a process flow. Used in the process equals statement START allows you to specify to filter on cases that have a certain starting activity. e.g.: **process equals START 'My first activity'**

TO

TO allows you to specify the next following activity in the cases. With this argument it is possible to specify a certain flow from one to another activity in the selected cases. e.g. **process equals 'My first activity' TO 'My second activity'**

TO ANY TO

TO ANY TO expands the possibility to define a process flow that one activity is following the other one at any point in time and not only directly. e.g.: **process equals 'My first activity' TO ANY TO 'My second activity'**

END

END sets the end activities for the inspected cases. END after the last specified activity names flags this activity as the ending activity and will thereby filter for all cases with this ending activity. e.g.: **process equals 'My last activity' END**

Examples

Eventlog

Case	Activity	Time	Number
1	A	01.01.2016	2

1	B	02.01.2016	10
1	A	22.01.2016	8
1	C	24.01.2016	4
2	A	01.02.2016	10
2	B	02.02.2016	9
3	A	01.02.2016	7
3	B	02.02.2016	4
3	C	04.02.2016	6

1) The cases flow through a certain activity as a condition

```
CASE WHEN PROCESS EQUALS 'C' THEN '1' ELSE '0' END
```

This example will return a 1 for all cases that include the activity 'C' and 0 for all cases without the activity 'C'.

Output

Case	Activity	Time	Number	OUTPUT
1	A	01.01.2016	2	1
1	B	02.01.2016	10	1
1	A	22.01.2016	8	1
1	C	24.01.2016	4	1
2	A	01.02.2016	10	0
2	B	02.02.2016	9	0
3	A	01.02.2016	7	1
3	B	02.02.2016	4	1
3	C	04.02.2016	6	1

2) Cases follow a certain path as a condition

Advanced

```
process equals START 'Create Purchase Requisition Item' TO 'Create
Purchase Order' TO ANY TO 'Goods Receipt' END
```

2) Count the activities of cases with a certain starting activity

Advanced

```
count(CASE WHEN process equals 'C' END THEN 1.0 ELSE NULL END)
```

Output

This example counts the number of activities for the cases which are ending with the activity C.

Case	OUTPUT
1	4
2	0
3	3

Calc Throughput

Available in version: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Throughput is used to calculate, for each case, the time between two activities. From which activity the calculated throughput time should start and at which it should end can be configured. We call the starting activity simply "start actvitiy" and the ending activity "end activity".

Syntax:

```
CALC_THROUGHPUT(<START_ACTIVITY> TO <END_ACTIVITY>, <TIMESTAMPS>)
```

```
<START_ACTIVITY> : [ FIRST_OCCURRENCE['activity'] | LAST_OCCURRENCE['activity'] | CASE_START |
ALL_OCCURENCE[''] ]

FIRST_OCCURRENCE['activity'] : Throughput time starts at the first occurrence of the specified activity
type.

LAST_OCCURRENCE['activity'] : Throughput time starts at the last occurrence of the specified activity
type.

CASE_START : Throughput time starts at the first activity of the case.

ALL_OCCURENCE[''] : Has the same meaning as CASE_START. The string parameter is ignored, but has to be
specified.
```

```
<END_ACTIVITY> :  [ FIRST_OCCURRENCE['activity'] | LAST_OCCURENCE['activity'] | CASE_END | ALL_OCCURENCE
[''] ]

FIRST_OCCURRENCE['activity'] : Throughput time ends at the first occurrence of the specified activity
type.

LAST_OCCURRENCE['activity'] : Throughput time ends at the last occurrence of the specified activity
type.

CASE_END : Throughput time ends at the last activity of the case.

ALL_OCCURENCE[''] : Has here the same meaning as CASE_END. The string parameter is ignored, but has to
be specified.
```

If **start and end activity is conflicting**, meaning the end activity is before the start activity the throughput calculation returns null for the conflicting case. Null is also returned if a **case has only one activity.**

```
<TIMESTAMPS> : In theory any integer column, but in praxis only REMAP_TIMESTAMPS
(<ACTIVITY_TIMESTAMP_COLUMN>, <TIME_UNIT>)

<ACTIVITY_TIMESTAMP_COLUMN> : Timestamp column of the activity table.

<TIME_UNIT> : [ <MINUTES> | <HOURS> | <DAYS> ]
```

Examples:

Case Id	Activity	Timestamp
1	A	00:01
1	B	00:02
1	A	00:04

1	B	00:08

Graphical representation:

First A to First B

Result:

"Case Id"	CALC_THROUGHPUT(FIRST_OCCURRENCE['A'] TO FIRST_OCCURRENCE['B'], REMAP_TIMESTAMPS ("TIMESTAMP", MINUTES))
1	1

First B to First A - Conflicting

Result:

"Case Id"	CALC_THROUGHPUT(FIRST_OCCURRENCE['B'] TO FIRST_OCCURRENCE['A'], REMAP_TIMESTAMPS ("TIMESTAMP", MINUTES))
1	null

This combination is **conflicting**. First B is after First A. Therefore null is returned.

Last A to Last B

Result:

"Case Id"	CALC_THROUGHPUT(LAST_OCCURRENCE['B'] TO LAST_OCCURRENCE['A'], REMAP_TIMESTAMPS ("TIMESTAMP", MINUTES))
1	4

First A to Last B

Result:

"Case Id"	CALC_THROUGHPUT(FIRST_OCCURRENCE['A'] TO LAST_OCCURRENCE['B'], REMAP_TIMESTAMPS ("TIMESTAMP", MINUTES))
1	7

Case Start to Case End

Result:

"Case Id"	CALC_THROUGHPUT(CASE_START TO CASE_END, REMAP_TIMESTAMPS("TIMESTAMP", MINUTES))
1	7

All_Occurrences to All_Occurrences

Result:

"Case Id"	CALC_THROUGHPUT(ALL_OCCURRENCES['] TO ALL_OCCURRENCES['], REMAP_TIMESTAMPS("TIMESTAMP", MINUTES))
1	7

Source Target

vailable in version: CELONIS 4.3

Description

The Source Target Operators can be used to create a temporary table, with values from different activities (Rows). On those columns all standard operators can be used. The Source/Target operators regard a case as edges and nodes. Nodes are the activities. Edges connect the activities in the order in which they happened. An edge points from a source activity to a target activity.

Syntax:

```
SOURCE(Column [, (ANY_OCCURRENCE[] | ANY_OCCURRENCE_WITH_SELF[] | FIRST_OCCURRENCE[]) TO (ANY_OCCURRENCE
[] | ANY_OCCURRENCE_WITH_SELF[] | LAST_OCCURRENCE[])] [, Filter_Column] )
Adds the values from Column, which correspond to a source activity, to a temporary table based on the
optional edge configuration and the optional filter column.

TARGET(Column [, (ANY_OCCURRENCE[] | ANY_OCCURRENCE_WITH_SELF[] | FIRST_OCCURRENCE[]) TO (ANY_OCCURRENCE
[] | ANY_OCCURRENCE_WITH_SELF[] | LAST_OCCURRENCE[])] [, Filter_Column] )
Adds the values from Column, which correspond to a target activity, to a temporary table based on the
optional edge configuration and the optional filter column.

(ANY_OCCURRENCE[] | ANY_OCCURRENCE_WITH_SELF[] | FIRST_OCCURRENCE[]) TO (ANY_OCCURRENCE[]
| ANY_OCCURRENCE_WITH_SELF[] | LAST_OCCURRENCE[])
Edge configuration, allows the user to control which edges are drawn between the activities of a case.
The following combinations are supported:
```

- ANY_OCCURENCE[] TO ANY_OCCURENCE[] - Default configuration, edges are drawn between consecutive activities within cases.
- FIRST_OCCURRENCE[] TO ANY_OCCURRENCE[] - From the first activity of each case an edge is drawn to all other activities in this case.
- FIRST_OCCURRENCE[] TO ANY_OCCURRENCE_WITH_SELF[] - From the first activity of each case an edge is drawn to all activities in this case, including the first one.
- ANY_OCCURRENCE[] TO LAST_OCCURRENCE[] - All activities, besides the last one of the case, draw an edge to the last activity of the case.
- FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[] - From the first activity of a case an edge is drawn to the last activity in this case.
- Filter_Column - This columns allows to filter out certain edges. If a value in the filter column is null the corresponding activity in the activity table is not used by SOURCE or TARGET operator.

Example

Activity Table

Case ID	Activity	Timestamp
1	A	08:00
1	B	08:15
1	C	08:45

How Source, Target Operators interpret the case by default:

Example 1:

source(Activity)	source(Timestamp)	target(Activity)	target(Timestamp)

A	08:00	B	08:15
B	08:15	C	08:45

Example 2:

This can be combined with further operators. In the following example is the time between two activities calculated:

source(Activity)	target(Activity)	target(Timestamp) - source(Timestamp)
A	B	00:15
B	C	00:30

Example 3:

Case ID	Activity	Timestamp
1	A	08:00
1	B	08:15
1	C	08:45

Result of:

source(Activity, REMAP_VALUES("TABLE1"."ACTIVITY", ['B', NULL]))	target(Activity, REMAP_VALUES("TABLE1"."ACTIVITY", ['B', NULL]))
A	C

By mapping B to NULL, activity B is skipped and an direct edge between A and C is created.

Example 4:

Calculate the average throughput time between the first occurrence of an activity and the last occurrence of another one:

```
AVG( TARGET( REMAP_TIMESTAMPS("_CEL_P2P_ACTIVITIES"."EVENTTIME",
seconds ) ,FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[]) -SOURCE(
REMAP_TIMESTAMPS("_CEL_P2P_ACTIVITIES"."EVENTTIME", seconds ) ,
FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[]) )
```

If used in the same component or PQL statement, the filter and condition in the SOURCE and TARGET operators have to be the same.

Join behavior

Source/Target generate temporary tables. For each used unique edge and filter column configuration a separate temporary table is created. Source/Target operators with identical Edge and Filter column add columns to the same temporary table. The temporary tables are joined to the case table.

Example:

In this example several calls to the Source/Target operator are done. After each call the tables status is shown.

1. SOURCE(Activity, ANY TO ANY)

   Temporary Edge Table is created. Source column is added to it.
2. TARGET(Activity, ANY TO ANY)
   Target column is added to existing temporary table.

3. SOURCE(Activity, ANY TO ANY, Filter)
   This column is added to a new temporary table, because the configuration is different to the previous ones.

Multiple temporary tables can lead to problems if used within the same query. This happens if Source/Target operators with different Edgify and Filter configurations are used within one query. The generated temporary tables can by default not be joined together and an error is returned. If one really needs to have two different configurations, the values in the temporary tables have be pulled up with PU functions to the case table.

Activation count

Available in version: CELONIS 4.3

Description

The ACTIVATION_COUNT operator returns for every activity in every case, how many times, at a given point in a process this activity has already occurred.

Syntax

ACTIVATION_COUNT(<ACTIVITY_TABLE>)

<ACTIVITY_TABLE>: The activity table, in almost all cases, this will be the "ACTIVITY_EN" column.

Result

For every activity in every case a number *x*, that shows this is the *x*th occurrence of this particular activity in the case.

Example

CASE_KEY	ACTIVITY	ACTIVATION_COUNT
1	A	1
1	B	1
1	C	1
1	A	2
1	B	2
1	A	3
2	A	1
2	B	1

Filtered by CASE_KEY 1 and ACTIVITY A, this leads to:

CASE_KEY	ACTIVITY	ACTIVATION_COUNT
1	A	1
1	A	2
1	A	3

Imagine the unfolding of a process into a sequential graph, as shown below, the algorithm increases the number for a special activity every time he encounters said activity on his way from the beginning to the end of the process:

Example of a process graph with annotated activation counts

Aggregation functions

Aggregate functions execute calculations on a set of values and return one single value. In contrary to classical SQL, in PQL select and group by clauses are not required, as the the Celonis components automatically group the aggregated values.

Examples would be average, sum, variance or count.

- Average
- Count
- MAX
- Median
- Min
- Moving operators
- Pull Up
- Quantile
- Standard deviation
- Sum
- Trimmed mean
- Variance

Average

Available in version: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Calculates the average over an aggregation.

Syntax

```
AVG(<value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- § 4.15.4 (only in regards of grouping(for MOVING_AVG see Average), DISTINCT qualifier is not allowed)
- § 10.9 Syntax Rules

Examples

ID	COUNTRY	VALUE
1	DE	10
1	US	2
1	FR	10
2	DE	3
2	US	40
3	DE	5
3	US	3

Average value per country

Result:

"COUNTRY"	AVG("VALUE")
DE	6
FR	10
US	15

Average value per ID

Result:

"ID"	AVG("VALUE")
1	7
2	21
3	4

Count

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Calculates the count over an aggregation.

Syntax

```
COUNT([DISTINCT] <value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- § 4.15.4 (only in regards of grouping(no Moving COUNT is available), COUNT(*) is not allowed)
- § 10.9 Syntax Rules

Examples

ID	COUNTRY
1	DE
1	US
1	FR
2	DE
2	US
2	DE
2	DE
3	DE
3	US
3	US

Count IDs per country

Result:

"COUNTRY"	COUNT("ID")
DE	5
FR	1

US	4

Count distinct IDs per country

Result:

"COUNTRY"	COUNT(DISTINCT "ID")
DE	3
FR	1
US	3

MAX

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Calculates the maximum over an aggregation.

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- § 4.15.4 (only in regards of grouping(for MOVING_MAX see MAX), DISTINCT qualifier is not allowed)
- § 10.9 Syntax Rules

Examples

COUNTRY	VALUE
DE	10
US	2
FR	10
DE	3
US	40
DE	4
DE	22
DE	5
US	3
US	43

Maximum value per country

Result:

"COUNTRY"	MAX("VALUE")
DE	22
FR	10
US	43

Median

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Calculates the median over an aggregation.

The median is chosen using a partial sorting algorithm that calculates the element at position `median_index`.

`median_index` is calculated by `floor(group_size / 2)`.

Syntax

```
MEDIAN(<value expression>)
```

Examples

ID	COUNTRY	VALUE
1	DE	10
1	US	2
1	FR	10
2	DE	3
2	US	40
2	DE	4
2	DE	22
3	DE	5
3	US	3
3	US	43

Median values per country

Result:

"COUNTRY"	MEDIAN("VALUE")
DE	5
FR	10
US	40

Median values per ID

Result:

"ID"	MEDIAN("VALUE")
1	10
2	22
3	5

Min

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
MIN(<value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- § 4.15.4 ✅ (only in regards of grouping(for MOVING_MIN see Min), DISTINCT qualifier is not allowed)
- § 10.9 Syntax Rules ✅

Examples

COUNTRY	VALUE
DE	10
US	2
FR	10
DE	3
US	40
DE	4
DE	22
DE	5
US	3
US	43

Minimum value per country

Result:

"COUNTRY"	MIN("VALUE")
DE	3
FR	10
US	2

Moving operators

Available in Version: CELONIS 4.3

Description

Moving operators calculate a value across a set of neighboring table rows. A range of neighboring rows, used for the calculation of a moving operator result is called window.

Syntax

There are four moving operators available:

PQL

```
MOVING_AVG(INPUT_COLUMN, START, END) // Average
MOVING_MAX(INPUT_COLUMN, START, END) // Maximum
MOVING_MIN(INPUT_COLUMN, START, END) // Minimum
MOVING_MEDIAN(INPUT_COLUMN, START, END) // Median
MOVING_SUM(INPUT_COLUMN, START, END) // Sum
```

INPUT_COLUMN:[INT,DOUBLE] - Column on which moving operator is executed

`START:INT - Start of the window, relative to the current row.`

`END:INT - End of the window, relative to the current row.`

`START and END are both including.`

Please note: The maximum window size [START, END] is limited to 1000.

Compliance with SQL:2003 (only for this operator & general non compliance applies)

`MOVING_<X>(<value expression>, <start value>, <end value>)` should be (is?) compliant to (Rules for <X> (e.g. **Moving operators**) apply):

SQL:2003

```
<X>(<value expression>) OVER (ROWS BETWEEN <start window bound> AND
<end window bound>)
```

With **`<start window bound>`** =

- **`<absolute start value>`** **`PRECEDING`**, if **`<start value>`** < 0

With **`<end window bound>`** =

- **`<absolute end value>`** **`PRECEDING`**, if **`<end value>`** < 0

With **`<absolute start value>`** = absolute value of **`<start value>`**

With **`<absolute end value>`** = absolute value of **`<end value>`**

- `CURRENT ROW`, if `<start value>` = 0
- `<absolute start value> FOLLOWING`, if `<start value>` > 0

- `CURRENT ROW`, if `<end value>` = 0
- `<absolute end value> FOLLOWING`, if `<end value>` > 0

Examples

Month	Income
1	100
1	300
2	400
3	300
4	500

Ordered Result

Result average:

"MONTH"	MOVING_AVG(AVG("INCOME"), -1, 0)
1	200
2	300
3	350
4	400

`Result maximum:`

"MONTH"	MOVING_MAX(AVG("INCOME"), -1, 0)
1	200
2	400
3	400
4	500

`Result minimum:`

"MONTH"	MOVING_MIN(AVG("INCOME"), -1, 0)
1	200
2	200
3	300
4	300

`Result median:`

MONTH	MOVING_MEDIAN(AVG(INCOME), -1, 0)
1	200
2	400
3	400

4	500

Result:

"MONTH"	MOVING_SUM("INCOME", -1, 0)
1	100
1	400
2	700
3	700
4	800

Aggregated Result

Result:

"MONTH"	AVG("INCOME")	MOVING_AVG(AVG("INCOME"), -1, 0)
1	200	200
2	400	300
3	300	350
4	500	400

Reversed Order

Result:

"MONTH"	AVG("INCOME")	MOVING_AVG(AVG("INCOME"), -1, 0)
4	500	500
3	300	400
2	400	350
1	200	300

MOVING_AVG respects ordering of table.

Corner case

`Result:`

"MONTH"	AVG("INCOME")	MOVING_AVG(AVG("INCOME"), -1, 0)
1	200	-
2	400	200
3	300	400
4	500	300

Pull Up

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Pull up functions allow you to set the basis on which a KPI is aggregated. You can define the dimension to which a KPI is pulled.

Railroad Diagram

Syntax

PQL

```
PU_<X>(<table>, <value expression> [, <filter expression>]) //
PU_COUNT, PU_SUM, PU_AVG, PU_MIN, PU_MAX, PU_MEDIAN, PU_COUNT_DISTINCT
```

PQL

```
PU_<X>(<table>, <value expression> [, <filter expression>] [, <order by
expression>]) // PU_FIRST, PU_LAST
```

PQL

```
PU_QUANTILE(<table>, <value expression>, <quantile> [, <filter
expression>])
```

Quantile

Available in: CELONIS 4.3

Description

Calculates the according quantile over an aggregation.

The quantile is calculated by using a partial sorting algorithm which calculates the element at the requested quantile position `quantile_index`.

The quantile_index is calculated by `floor(quantile * group_size)`, where the `quantile` is [0, 1].

> The quantile function always returns an already existing element out of the requested column, no averaging performed on items.

Properties

QUANTILE(..., x)	Corresponds to
`x = 0.5`	MEDIAN
`x = 0.0`	MIN
`x = 1.0`	MAX

Syntax

Code Block

```
QUANTILE(<value expression>, <quantile>)
```

Standard deviation

Available in: CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
STDEV(<value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

STDEV is compliant to standard STDDEV_SAMP in the following way:

- § 4.15.4 (There is no Moving VAR available)
- § 10.9 Syntax rules

Examples

ID	COUNTRY	VALUE
1	DE	10
1	US	2
1	FR	10
2	DE	3
2	US	40
2	DE	4
2	DE	22
3	DE	5
3	US	3
3	US	43

STDEV over average of values per country

Result:

"COUNTRY"	STDEV("VALUE" average)
DE	7.8549
FR	0

US	22.5536

STDEV over average of values per ID

Result:

"ID"	STDEV("VALUE" average)
1	4.6188
FR	17.5000
US	22.5389

Sum

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
SUM(<value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- § 4.15.4 (only in regards of grouping(no Moving SUM available), DISTINCT qualifier is not allowed)
- § 10.9 Syntax Rules

Arguments

Inputs

The input arguments are the names of the activities in the eventlog. The activity name has to be wrapped in '. e.g.: **process equals 'My first activity'**

START

START defines the starting activity of a process flow. Used in the process equals statement START allows you to specify to filter on cases that have a certain starting activity. e.g.: **process equals START 'My first activity'**

TO

TO allows you to specify the next following activity in the cases. With this argument it is possible to specify a certain flow from one to another activity in the selected cases. e.g. **process equals 'My first activity' TO 'My second activity'**

TO ANY TO

TO ANY TO expands the possibility to define a process flow that one activity is following the other one at any point in time and not only directly. e.g.: **process equals 'My first activity' TO ANY TO 'My second activity'**

END

END sets the end activities for the inspected cases. END after the last specified activity names flags this activity as the ending activity and will thereby filter for all cases with this ending activity. e.g.: **process equals 'My last activity' END**

Examples

COUNTRY	VALUE
DE	10

US	2
FR	10
DE	3
US	40
DE	4
DE	22
DE	5
US	3
US	43

Sum of values per country

Result:

"COUNTRY"	MEDIAN("VALUE")
DE	44
FR	10
US	88

Trimmed mean

Available in: CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Calculates the trimmed mean over an aggregation.

Lower and upper cutoff get rounded to the next smaller whole row number for each group. (e.g. Trimmed mean over a group with 42 rows and a lower & upper cutoff of 10 will result in the cut of the upper & lower 4 rows.)

If the of upper & lower rows cut is greater or equal to the count of the current group, trimmed mean will return 0 for this group.

A float column has a float column as result.

An integer column has an integer column as result. (Note: Values get rounded via the "Round to nearest, round half away from zero" rule)

Syntax

```
TRIMMED_MEAN(<value expression> [, <lower cutoff> [, <upper cutoff>]])
```

- `<lower cutoff>`: Integer between 0 and 100 (other values result in undefined behaviour)
- `<upper cutoff>`: Integer between 0 and 100 (other values result in undefined behaviour)

Compliance with SQL:2003 (only for this operator & general non compliance applies)

- `TRIMMED_MEAN` is not part of the standard.
- `TRIMMED_MEAN(<value expression>, 0, 0)` is compliant to Trimmed mean in the SQL:2003 standard.
- `TRIMMED_MEAN(<value expression>, 0, 0)` not necessarily has the same result as `AVG(<value expression>)`.

Examples

COUNTRY	VALUE
DE	10
US	2
FR	10
DE	3
US	40
DE	4
DE	22
DE	5

US	3
US	43

Average vs. trimmed mean evenly cut off vs. trimmed mean unevenly cut off

Result:

AVG("VALUE")	TRIMMED_MEAN("VALUE",5)	TRIMMED_MEAN("VALUE",2,30)
14	16	5

Average vs. trimmed mean evenly cut off vs. trimmed mean unevenly cut off

Result:

COUNTRY	AVG("VALUE")	TRIMMED_MEAN("VALUE",5)	TRIMMED_MEAN("VALUE",2,30)
DE	8	9	6
FR	10	10	10
US	22	22	15

Variance

Available in: CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
VAR(<value expression>)
```

Compliance with SQL:2003 (only for this operator & general non compliance applies)

VAR is compliant to standard VAR_SAMP in the following way:

- § 4.15.4 (There is no Moving VAR available)
- § 10.9 Syntax rules

Examples

ID	COUNTRY	VALUE
1	DE	10
1	US	2
1	FR	10
2	DE	3
2	US	40
2	DE	4
2	DE	22
3	DE	5
3	US	3
3	US	43

Variance of average values per country

Result:

"COUNTRY"	VAR("VALUE" average)
DE	61.7000
FR	0.0000

US	508.6667

Variance of average values per ID

Result:

"ID"	VAR("VALUE" average)
DE	21.3333
FR	306.2500
US	508.0000

Math functions

- ABS
- CEIL
- FLOOR
- LOG
- POWER
- QNORM
- ROUND
- SQRT

ABS

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

A mathematical function that returns the absolute (positive) value of the specified numeric expression. (ABS changes negative values to positive values. ABS has no effect on zero or positive values.)

Syntax

```
ABS(<column>)
```

Examples

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	-124
2	A1	-24935
3	A2	345
4	A1	2983539
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1

ABS("Cases"."Value")
124
24935
345
2983539
234

CEIL

Available in: CELONIS 4.2 , CELONIS 4.3 , CELONIS 4.3 , CELONIS 4.3

Description

Returns the smallest integer greater than, or equal to, the specified numeric expression.

Syntax

```
CEIL(<column>)
```

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	12.4
2	A1	24935
3	A2	34.5
4	A1	29835.39
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1

CEIL("Cases"."Value")
13
24935
35
29836
234

FLOOR

Available in: CELONIS 4.2 , CELONIS 4.3 , CELONIS 4.3 , CELONIS 4.3

Description

Returns the greatest integer smaller than, or equal to, the specified numeric expression.

Syntax

```
FLOOR(<column>)
```

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	12.4
2	A1	24935
3	A2	34.5
4	A1	29835.39
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1

FLOOR("Cases"."Value")
12
24935
34
29835
234

LOG

Available in: CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the natural logarithm of the specified float expression. By default, LOG() returns the natural logarithm, but you can specifiy the base with an optional parameter.

Syntax

```
LOG(<column> [, base ])
```

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	124
2	A1	24935
3	A2	345
4	A1	2983539
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1

LOG(124)
4.820281565605037

Example 2

LOG("Cases"."Value")
4.820281565605037
10.124027717980223

5.84354441703136
14.908620737754262
5.455321115357702

Example 3

LOG("Cases"."Value", 2)
6.954196310386876
14.605884582552477
8.430452551665532
21.508593204852435
7.870364719583405

POWER

Available in: CELONIS 4.2.1 , CELONIS 4.3 , CELONIS 4.3

Description

Returns the value of the specified expression to the specified power.

Syntax

```
POWER(input , y)
```

Arguments

input

Is an expression of type float or of a type that can be implicitly converted to float.

y

Is the power to which to raise *float_expression*. *y* can be a numeric data type.

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	124
2	A1	24935
3	A2	345
4	A1	2983539
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1

POWER(2,2)
4

Example 2

POWER(4,0.5)
2

Example 3

"Types"."Types"	LOG("Cases"."Value", 2)
A1	565816875264
A2	119025

QNORM

Available in: CELONIS 4.2.1 , CELONIS 4.3

Description

Approximates the inverse of the normal (Gaussian) cumulative distribution function with an error of less than 4.5e-4.

Syntax

```
QNORM(<input>)
```

ARGUMENTS

Input has to be a numeric value, bigger than 0.0 and smaller than 1.0.

ROUND

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns a numeric value to the next whole number.

Syntax

```
ROUND ( numeric_expression )
```

Example

Example 1:

ROUND(2.34)
2

Example 2:

Round(2.56)
3

SQRT

Available in: CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the square root of the specified float value.

Syntax

```
sqrt ( float_expression )
```

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	124
2	A1	24935
3	A2	345
4	A1	2983539
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1:

SQRT(4)
2

Example 2:

SQRT(AVG("cases"."Value"))
776

Example 3:

"Types"."Types"	SQRT(AVG("cases"."Value"))
A1	867
A2	19

Date Operations

The date functions in PQL allow calculations and manipulation of date columns. The following functions are available:

- (Time Unit) Between Functions
- Add (Time Unit) Functions
- Basic (Time Unit) Functions
- Date Between
- REMAP_TIMESTAMPS
- Round (Time Unit) Functions

(Time Unit) Between Functions

Available in: CELONIS 4, CELONIS 4.2, CELONIS 4.2.1, CELONIS 4.3

(Time Unit) Between Functions

PQL
`[TIME_UNIT]_BETWEEN(INPUT_COLUMN 1, INPUT_COLUMN 2 )`

INPUT_COLUMN 1(DATE), INPUT_COLUMN 2(DATE)

[TIME_UNIT] : (YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, MILLIS)

Time unit between functions result in the time difference between two input columns and represents it in the specified time unit.

Result will be in DOUBLE form.

Example

Table1.Column1	Table1.Column2	YEARS_BETWEEN ("Table1"."Column1" , "Table1"."Column2")
1999-12-31 23:59:59.000	1999-12-31 23:59:59.000	0.0
2000-06-01 00:00:00.000	1999-12-31 23:59:59.000	-0.416
1999-12-31 23:59:59.000	2000-12-31 00:00:00.000	1.0
1999-12-31 23:59:59.000	2001-12-31 00:00:00.000	2.0
2000-12-31 00:00:00.000	2001-12-31 00:00:00.000	1.0

Add (Time Unit) Functions

Add (Time Unit) Functions

PQL
`ADD_[TIME_UNIT](INPUT_COLUMN 1, INPUT_COLUMN 2 )`

INPUT_COLUMN 1(DATE), INPUT_COLUMN 2(INT)

[TIME_UNIT] : (YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, MILLIS)

Add time unit functions adds the integer in the second column into the date value on the first column such that the integer represents the specified time unit.

Result will be in DATE form.

Example

Table1.Column1	Table1.Column2	ADD_MINUTES ("Table1"."Column1", "Table1"."Column2")
1999-12-31 23:59:59.000	0	1999-12-31 23:59:59.000
1999-12-31 23:59:59.000	120	2000-01-01 01:59:59.000
2000-02-28 17:01:01.000	48*60	2000-03-01 17:01:01.000

:

Please note: Year 2000 is a leap year which has 366 days.

Basic (Time Unit) Functions

Available in: CELONIS 4, CELONIS 4.2, CELONIS 4.2.1, CELONIS 4.3

Basic (Time Unit) Functions

PQL
`TIME_UNIT(INPUT_COLUMN)`

INPUT_COLUMN (DATE)

[TIME_UNIT] : (YEAR, QUARTER, MONTH, DAY, HOURS, MINUTES, SECONDS, MILLIS, DAY_OF_WEEK, CALENDAR_WEEK)

Time unit functions results in the time unit data expressed in the intervals such that: Year(-), Quarter(1-4), Month(1-12), Day(1-31), Hours(0-23), Minutes(0-59), Seconds(0-59), Millis(0-999), Calendar Week(1-52), Day of Week(0-6).

Please note: For Calendar Week function, weeks starts with Monday.

Please note: For Day of Week function, '0' represents Sunday and goes on like that.

Result will be in integer form.

Example

Table1.Column1	QUARTER ("Table1"."Column1")
2016-03-31	1
2016-06-30	2
2016-12-31	4

Date Between

Available in: CELONIS 4, CELONIS 4.2, CELONIS 4.2.1, CELONIS 4.3

Date Between Functions

PQL

```
DATE_BETWEEN(INPUT_COLUMN 1, INPUT_COLUMN 2, INPUT_COLUMN 3
)
```

INPUT_COLUMN 1(DATE), INPUT_COLUMN 2(DATE), INPUT_COLUMN 3(DATE)

Date between functions result in the either one or zero that represents weather the date value in **first column is in between two date values on other columns** .

Result will be in INTEGER form.

Example

Table1.Column1	Table1.Column2	Table1.Column3	DATE_BETWEEN ("Table1"."Column1" , "Table1"."Column2" , "Table1"."Column3")
2000-01-01 00:00:00.000	1999-12-31 23:59:59.999	2005-05-09 23:59:59.000	1
2005-05-09 23:59:59.000	2000-01-01 00:00:00.000	2005-05-09 23:59:59.100	1
2005-05-09 23:59:59.100	1999-12-31 23:59:59.999	2000-01-01 00:00:00.000	0
2005-05-09 23:59:59.000	2005-05-09 23:59:59.000	1999-12-31 23:59:59.999	0

REMAP_TIMESTAMPS

Available in: **CELONIS 4.2**, **CELONIS 4.2.1**, **CELONIS 4.3**

Description

Remap timestamp operator is a time unit counter in a specified intervals. User can create weekly time blocks and count the number of predetermined time units between date inputs.

Syntax

PQL

```
REMAP_TIMESTAMPS(INPUT_COLUMN,
TIME_UNIT)
//Basic


REMAP_TIMESTAMPS(INPUT_COLUMN, TIME_UNIT, DAY HH:MM-HH:
MM)
//Specified Calendar


REMAP_TIMESTAMPS(TO_TIMESTAMP('01.01.1970 23:30.30', 'DD.MM.YYYY HH:mm.
SS'), TIME_UNIT, DAY (START)HOUR:MINUTE-(END)HOUR:
MINUTE)                                    //with TO_TIMESTAMP
```

Inputs

INPUT_COLUMN[DATE] : Date column on which REMAP_TIMESTAMPS operator is executed.

TO_TIMESTAMP('01.01.1970 23:30.30', 'DD.MM.YYYY HH:mm.SS') : This query expression sets the ending time for the time interval that operator is executed.

TIME_UNIT[DAYS,HOURS,MINUTES,SECONDS] : Sets the time units for the operator so that result will be in the form of this time units.

DAY HH:MM-HH:MM(INT:INT-INT:INT) : Sets the calendar specification for the operator so that the time will be counted only if it is in this specific time interval.

Former HH:MM must be the start hour and minute, and the latter must be the end hour and minute.

Please note: Query may have more than one calendar specification.

Please note: When query has more than one calendar specification for the same day, only the last one is considered and others are ignored.

Result: Int-Array with values representing the count of selected time units.

How It Works

In the basic case, operator counts the number of specified time units from 01.01.1970 to the date values on the input column.

When we set calendar specifications, operator only measures the specified time interval on a week and accumulates it between 01.01.1970 to the date values on the input column.

Then counts the specified time units in the result. Thus we have an integer array which show the occurrence of specified time unit.

When we have TO_TIMESTAMP('01.01.1970 23:30.30', 'DD.MM.YYYY HH:mm.SS') expression instead of the input column, operator simply considers the end time of operator as the time expression in to_timestamp.

Starting time is the same as before(01.01.1970).

Examples

Results for different queries

1)

Query: REMAP_TIMESTAMPS(TO_TIMESTAMP('02.01.1970', 'DD.MM.YYYY'), DAYS)

Result of basic case: 1

2)

Query: REMAP_TIMESTAMPS(TO_TIMESTAMP('02.01.1970', 'DD.MM.YYYY'), HOURS)

Result of basic case: 24

3)

Table1.Column1
null
03.01.1970

Query: REMAP_TIMESTAMPS (Table1.Column1, MINUTES, FRIDAY 11:00-12:30 TUESDAY 14:55-15:00)

Result (Calendar Specified)
null
90

Please note: 01.01.1970 is Thursday and from 01.01.1970 to 03.01.1970 we come up with only Friday.

4)

Query: REMAP_TIMESTAMPS(TO_TIMESTAMP('29.12.1969 06:30.30', 'DD.MM.YYYY HH:mm.SS'), HOURS, MONDAY 8:00-13:00 WEDNESDAY 8:00-13:00)

Result of Calendar Specified with TO_TIMESTAMP: -10

Please note: 01.01.1970 is Thursday and from 29.12.1969 to 01.01.1970 we come up with both Monday and Wednesday.

Furthermore, since the end time for operator is before than the starting time(01.01.1970), counted time unit is negative.

5)

Query: REMAP_TIMESTAMPS(TO_TIMESTAMP('29.12.1969 06:30.30', 'DD.MM.YYYY HH:mm.SS'), HOURS, MONDAY 8:00-13:00 WEDNESDAY 8:00-13:00 MONDAY 11:00-15:00)

Result of Calendar Specified with TO_TIMESTAMP: -9

Please note: Operator only considered the second calendar specification for Monday.

6)

Query: REMAP_TIMESTAMPS(TO_TIMESTAMP('29.12.1969 06:30.30', 'DD.MM.YYYY HH:mm.SS'), HOURS, MONDAY 8:00-13:00 WEDNESDAY 8:00-13:00 MONDAY 15:00-11:00)

Result of Calendar Specified with TO_TIMESTAMP: -1

Please note: Operator only considered the second calendar specification for Monday and since starting time is after than the ending time, these calendar specification is considered as negative.

Round (Time Unit) Functions

Available in: CELONIS 4, CELONIS 4.2, CELONIS 4.2.1, CELONIS 4.3

Round (Time Unit) Functions

PQL
`ROUND_[TIME_UNIT](INPUT_COLUMN)`

INPUT_COLUMN (DATE)

[TIME_UNIT] : (YEAR, MONTH, DAY, WEEK, HOUR, MINUTE, SECOND)

Round time unit functions rounds down the subordinates of chosen time unit into the ground state.

Ground states are such that, Month(01), Day(01), Hour(00), Minute(00), Second(00), Millisecond(000).

For **ROUND_WEEK** function the progress is slightly different. This function also rounds down the Hour, Minute, Second and Millisecond values, but the Day value will not be in its ground state. Instead, the day will be chosen as the Monday of the week of that date.

Result will be in DATE form.

Examples

Table1.Column1	ROUND_WEEK ("Table1"."Column1")
2016-01-01 23:59:59.000	2015-12-28 00:00:00.000
1994-12-31 23:59:59.000	1994-12-26 00:00:00.000
2017-04-11 00:00:00.000	2017-04-10 00:00:00.000

Table1.Column1	ROUND_MONTH ("Table1"."Column1")
2006-05-08 14:58:49.700	2006-05-01 00:00:00.000
2006-12-31 23:58:59.999	2006-12-01 00:00:00.000

Logical statements

- AND, OR, NOT
- CASE WHEN
- IN

AND, OR, NOT

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

The `AND` and the `OR` operator combines the results of two logical statements. The result of these two operators is defined by the **three-valued logic (3VL)**. This applies also for the `NOT` operator which inverts the result of a logical statement.

Syntax

```
<logical expression> AND <logical expression>
<logical expression> OR <logical expression>
NOT <logical expression>
```

Examples

The **three-valued logic (3VL)** is defined as follows:

p	*q*	*p OR q*	*p AND q*	*p = q*
True	True	True	True	True
True	False	True	False	False
True	Null	True	Null	Null
False	True	True	False	False
False	False	False	False	True
False	Null	Null	False	Null
Null	True	True	Null	Null
Null	False	Null	False	Null
Null	Null	Null	Null	Null

p	*NOT p*
True	False
False	True
Null	Null

(see https://en.wikipedia.org/wiki/Null_(SQL))

Differences for older versions

For versions lower than 4.3 the AND and OR statement will always return NULL if any of the input values is NULL

CASE WHEN

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
CASE
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

Example

Eventlog

Case ID	Type	Eventtime	User
1	a	01.01.17 08:00	John
1	b	02.02.17 14:00	Bill
1	c	05.03.17 20:00	Jessica
2	a	08.03.17 02:00	John
2	c	10.05.17 08:00	Bill
3	a	11.06.18 14:00	Jessica
3	b	13.07.18 20:00	Jessica
3	c	15.07.18 02:00	Sebastian
3	c	16.09.17 08:00	John
3	c	18.09.17 14:00	Bill
4	a	18.09.17 20:00	Jessica
4	b	22.12.17 02:00	Bill
4	c	23.12.17 07:00	Jessica
5	c	24.12.17 08:40	Sebastian
5	b	25.12.17 12:00	John

Case ID — Case ID

Cases

Case ID	Type	Value
1	A1	124
2	A1	24935
3	A2	345
4	A1	2983539
5	A1	234

Types — Types

Types

Types	Details1	Details2
A1	1234	1111
A2	9999	2222

Example 1: One condition without else

"Cases"."Case ID"	CASE WHEN "Cases"."Case ID" = 'A1' THEN 'First type' ELSE NULL END
1	First type
2	First type
3	-
4	First type
5	First type

Example 2: Two conditions

"Eventlog"."Case ID"	CASE WHEN "Cases"."Case ID" = 'A1' THEN 'First type' WHEN "Cases"."Case ID" = 'A2' THEN 'Second type' ELSE NULL END
1	First type
2	First type
3	Second type
4	First type
5	First type

Example 3: Case when applied for ratio calculation

"Cases"."Type"	AVG(CASE WHEN "Cases"."Case ID" = 'A1' THEN 1.0 ELSE 0.0 END)
a	0.75
b	0.75
c	0.5714285714285714

IN

Available in: CELONIS 4 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Syntax

```
<value expression> IN (<value expression>, ...)
```

Result

The result of the `IN` operator is True if the result of the left statement is equal to the result of one of the statements in the list. If the list contains NULL and the result of the left statement is NULL then the result of the `IN` operator is True. Otherwise the result of the operator is False.

String functions

- LEFT
- LEN
- LOWER
- LTRIM
- REVERSE
- RIGHT
- RTRIM
- STR_TO_INT
- SUBSTRING
- UPPER

LEFT

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the left part of a character string with the specified number of characters.

Syntax

```
LEFT ( character_expression , integer_expression )
```

Example

Example 1:

LEFT('ABCD', 2)
AB

LEN

Available in: CELONIS 4.2 , CELONIS 4.3 , CELONIS 4.3 , CELONIS 4.3

Description

Returns the number of characters of the specified string expression, including trailing blanks.

Syntax

```
LEN( string_expression )
```

Examples

LEN('abc')
3

LOWER

Available in: CELONIS 4.2 , CELONIS 4.3 , CELONIS 4.3 , CELONIS 4.3

Description

Returns a character expression after converting uppercase character data to lowercase.

Syntax

```
LOWER ( character_expression )
```

Example

LOWER('Hallo')
hallo

LTRIM

Available in: CELONIS 4.2 , CELONIS 4.3 , CELONIS 4.3 , CELONIS 4.3

Description

Returns a character expression after it removes leading blanks.

Syntax

```
LTRIM ( character_expression )
```

Example

LTRIM(' This is a test')
This is a test

REVERSE

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the reverse order of a string value.

Syntax

```
REVERSE ( string_expression )
```

Example

REVERSE('abc')
cba

RIGHT

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the right part of a character string with the specified number of characters.

Syntax

```
RIGHT ( character_expression , integer_expression )
```

Example

RIGHT('abcdefg', 3)
efg

RTRIM

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns a character string after truncating all trailing spaces.

Syntax

```
RTRIM ( character_expression )
```

Example

RTRIM(' This is a test ')
This is a test

STR_TO_INT

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the input of type string as integer value.

Syntax

```
STR_TO_ING ( string_expression )
```

SUBSTRING

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns the substring of the string in column.

Syntax

```
SUBSTRING ( expression ,start , length )
```

Examples

Example 1

SUBSTRING('Hello World', 0, 5)
Hello

Example 2

SUBSTRING('Hello World', 6, 5)
World

UPPER

Available in: CELONIS 4.0 , CELONIS 4.2 , CELONIS 4.2.1 , CELONIS 4.3

Description

Returns a character expression with lowercase character data converted to uppercase.

Syntax

```
UPPER ( character_expression )
```

Example

UPPER('Hello')
HELLO

Advanced statistics

- Decission tree
- kmeans
- Linear regression
- Zscore

Decission tree

Available in: CELONIS 4.3

Description

Like other machine learning functions (such as linear regression or k means), the Decision Tree consists of two steps.

The first step consists of training the decision tree model. The will learn the classification rules of the decision tree (see https://en.wikipedia.org/wiki/Decision_tree) based on the provided training data. In this step, the user can specify which columns the model should use, and on which subset of the data the system should be trained on. Additionally, an extra max_depth parameter can be provided in order to avoid overfitting to the training data.

In a second step, once the model is trained, the user can use it to classify new rows based on the model input columns.

Syntax:

DECISION_TREE(TRAIN_[FILTERED_]DT([EXCLUDED(Dim_1, ...),] INPUT(Column_A, ...), OUTPUT(COLUMN_B) [, max_depth], PREDICT(Column_A, ...))

```
TRAIN_DT - Constructs the decision tree. Authorization objects are still respected.
TRAIN_FILTERED_KM - Constructs the decision tree, filtering the training data with current filters.

EXCLUDED - These are dimension columns (columns that are not aggregations) that should be used as
grouping columns for the INPUT columns of the model training. These columns will be ignored by the
training algorithm and won't be part of the trained model.

INPUT - One or more columns, which is used to train the model.

OUTPUT - One column containing the labels associated to each row that the model should train upon.

max_depth (default=3) - this specifies the maximal depth allowed for the learned tree, this can be used
to prevent overfitting. If max_depth=0, no limit will be set on the depth of the tree.

PREDICT - One or more columns, should be semantically the same as the ones provided as the model INPUT
parameter, contains the class assignments based on the trained decision tree rules.
```

All columns in TRAIN_DT have to be joinable.

The columns in PREDICT do not have to be joinable with the columns in TRAIN_DT.

The input of the model training is regarded as an **independent sub query**. This means if an aggregation is used, it is independent of the dimensions defined in the rest of the query. This also means that the columns within TRAIN_DT have to be joinable, but not with the columns used in the rest of the query.

Filter behavior:

Standard Decision Tree

If rows of a column are filtered, it does **not** affect the decision tree, as long as the decision tree model is not trained on aggregation results. This means independent of filters and selections, the underlying model stays the same. If you want to restrict the input data of a model you can use a CASE WHEN statement and map the values you want to be ignored to null.

If a model is trained on results of an aggregation it still changes with the filtering because the result of the aggregation is affected by the filtering.

Filtered Decision Tree

If a filter or selection changes, the model is retrained and the resulting function adopts to the new of view of data. This has a serious performance impact.

Examples:

Here is the training data used for the following tests

Temperature	Mood
-15	Very Bad
10	Bad
24	Good
22	Good
35	Bad
45	Very Bad

Simple Decision Tree

Result:

"Temperature"	`DECISION_TREE(TRAIN_DT(INPUT(Temperature), OUTPUT(Mood)), PREDICT(Temperature))`
-15	Very Bad
10	Bad
24	Good
22	Good
35	Bad
45	Very Bad

Decision Tree with underfitting

Here, we set the max_depth to 1 meaning that no decision will be taken and every row will be assigned to the same class (the most present class in order to minimize error).

Result:

"Temperature"	`DECISION_TREE(TRAIN_DT(INPUT(Temperature), OUTPUT(Mood), 0.1), PREDICT(Temperature))`
-15	Bad
10	Bad
24	Bad
22	Bad
35	Bad
45	Bad

kmeans

Available in: CELONIS 4.3

Description

like Linear Regression, KMeans consists actually of two steps. First training a model, meaning finding the centroids of the clusters. Second cluster the values by the model. Our KMeans implementation is very flexible and allows the user to specify on which data the algorithm should train, and which data it should cluster. This if often more then most users need. Therefore there is also a simple KMeans operator, which just expects the number of clusters.

Syntax:

KMEANS(K)

Simple Kmeans. It creates K clusters based on all other inputs in the query.

```
KMEANS(TRAIN_[Filtered_]KM([EXCLUDED(Dim_Column_A, ...),] INPUT(Column_B, ...), K), CLUSTER(Column_B, ...))
```

```
TRAIN_KM - Finds K centroids in the cluster. Authorization objects are still respected.
TRAIN_FILTERED_KM - Finds K centroids in the filtered data.
```

```
EXCLUDED - Here you can optionally define dimensions which influence the aggregations used for model training, but are not itself included in the training.
```

```
INPUT - One or more columns, which is used to train the model.
```

```
K - Number of clusters.
```

```
CLUSTER - One or more columns, which are clustered based on the trained centroids.
```

All columns in TRAIN_KM have to be joinable.

The columns in CLUSTER do not have to be joinable with the columns in TRAIN_KM.

The input of the model training is regarded as an **independent sub query**. This means if an aggregation is used, it is independent of the dimensions defined in the rest of the query. This also means that the columns within TRAIN_KM have to be joinable, but not with the columns used in the rest of the query.

Filter behavior:

Standard KMeans

If rows of a column are filtered, it does **not** affect the linear model, as long as the kmeans model is not trained on aggregation results. This means independent of filters and selections, the underlying model stays the same. If you want to restrict the input data of a model you can use a CASE WHEN statement and map the values you want to be ignored to null.

If a model is trained on results of an aggregation it still changes with the filtering because the result of the aggregation is affected by the filtering.

Filtered KMeans

If a filter or selection changes, the model is retrained and the resulting function adopts to the new of view of data. This has a serious performance impact.

Result:

KMeans finds clusters based on the K-means++ algorithm. It assigns to every cluster a number. K-means is not a stable algorithm, so you can get different results when executing the algorithm multiple times. Therefore you must **not** rely that clusters stay the same.

Example 1:

Month	Income

1	100
1	150
2	450
2	500
3	800
3	900

Simple KMeans

Result:

Month	Predicted Income	KMEANS(3)
1	250	1
2	950	2
3	1700	0

Advanced KMeans

Result:

"Month"	"Predicted Income"	KMEANS(TRAIN_KM(INPUT(Month, SUM(Income)), 3), CLUSTER(Month, SUM (Income)))
1	250	1
2	950	2
3	1700	0

Example 2:

Month	Income
1	100
1	50
2	500
3	800
4	600

KMeans without EXCLUDE

Result:

Month	KMEANS(TRAIN_KM(INPUT(Month, SUM(Income)), 3), CLUSTER(Month, sum(Income)))
1	1
2	2
3	2
4	0

Here month 2 and 3 are clustered together, even though the income of month 4 (600) would closer to the income of month 2 (500). This is because we also cluster by the months itself. If we want to cluster only by the income and still use month as dimension, we need to use the EXCLUDED tag as shown in the next example.

KMeans with EXCLUDE

Result:

Month	KMEANS(TRAIN_KM(EXCLUDED(Month), INPUT(SUM(Income)), 3), CLUSTER(sum(Income)))
1	1
2	0
3	2
4	0

Now month 2 and 4 are clustered together. The timeline has now no influence on the clustering but is still used for the aggregation.

Linear regression

Available in: CELONIS 4.3

Description

Linear Regression consists actually of two steps. First training a model. Second predict values by the model.

Syntax:

```
LINEAR_REGRESSION(TRAIN_[Filtered_]LM(INPUT(Column_A, ...), OUTPUT(Column_B)), PREDICT(Column_A, ...));

TRAIN_LN - Trains a Linear Regression model.
TRAIN_FILTERED_LM - The data on which the linear model is trained is filtered.

INPUT - One or more columns, which is used to train the model so that it describes the Output.

OUTPUT - One column, through which the model lays a predictor function.

PREDICT - One or more columns, on which for each row a predicted value is returned by applying the
trained model.
```

All columns in TRAIN_LM have to be joinable.

The columns in PREDICT do not have to be joinable with the columns in TRAIN_LM.

The input of the model training is regarded as an **independent sub query**. This means if an aggregation is used, it is independent of the dimensions defined in the rest of the query. This also means that the columns within TRAIN_LN have to be joinable, but not with the columns used in the rest of the query.

Filter behavior:

Standard Linear Regerssion

If rows of a column are filtered, it does **not** affect the linear model, as long as the linear model is not trained on aggregation results. This means independent of filters and selections, the underlying model stays the same. If you want to restrict the input data of a model you can use a CASE WHEN statement and map the values you want to be ignored to null.

If a model is trained on results of an aggregation it still changes with the filtering because the result of the aggregation is affected by the filtering.

Filtered Linear Regression

If a filter or selection changes, the model is retrained and the resulting function adopts to the new of view of data. This has a serious performance impact.

The examples "Simple linear regression with filtering" and "Simple filtered linear regression with filtering" are showing the difference.

Example:

Month	Income
1	100
1	300
2	400
3	300
4	500

Simple linear regression

Result:

Month	LINEAR_REGRESSION(TRAIN_LM(INPUT(Month), OUTPUT(Income)), PREDICT(Month))
1	217.647
1	217.647
2	302.941
3	388.235
4	473.529

Simple linear regression with constant

Result:

LINEAR_REGRESSION(TRAIN_LM(INPUT(Month), OUTPUT(Income)), PREDICT(1))
217.647

Multiple linear regression

Result:

Month	LINEAR_REGRESSION(TRAIN_LM(INPUT(Month, POWER(Month,2)), OUTPUT(Income)), PREDICT(Month, POWER (Month,2)))
1	215.384
1	215.384
2	307.692
3	392.307
4	469.230

Linear regression with model training on aggregation

Result:

Month	LINEAR_REGRESSION(TRAIN_LM(INPUT(Month), OUTPUT(AVG(Income))), PREDICT(Month))
1	230
1	230
2	310
3	390
4	470

Simple linear regression with filtering

Filter Temporary Month > 1

Result:

Month	TH, LINEAR_REGRESSION(TRAIN_LM(INPUT(Month), OUTPUT(Income)), PREDICT(Month))
2	302.941
3	388.235
4	473.529

Linear model doesn't change. Linear Regression still returns the same values as in the Simple Linear Regression example, without filtering.

Simple filtered linear regression with filtering

Filter Temporary Month > 1

Result:

Month	LINEAR_REGRESSION(TRAIN_FILTERED_LM(INPUT(Month), OUTPUT(Income)), PREDICT(Month))
2	350
3	400
4	450

Linear model is recalculated and adopts to filter changes. Therefore the predicted values are changing.

Zscore

Available in: CELONIS 4.3

Description

Assuming normally distributed data, the z-score or standard score of an entry is `(input - avg)/sigma`, where `avg` is the average and `sigma` the sample standard deviation. It may be used to detect outliers, e.g. by filtering for a z-score with an absolute value bigger than 1 (~32% -quantile), bigger than 2 (~5%-quantile) or bigger than 3 (~.3%-quantile).

Syntax

```
ZSCORE(<column>)
```

where `<column>` must be of type `cel_int_t` or `cel_float_t` and must be no constant, i.e. it has a table as its `owner`.

Examples

Z-scores are independent of the order of the input.

DATA_1	DATA_2	...
1	7	
1	4	
4	7	
7	1	
7	1	

Example 1:

ZSCORE("TABLE"."DATA_1")	ZSCORE("TABLE"."DATA_2")
-1.0	1.0
-1.0	0.0
0.0	1.0
1.0	-1.0
1.0	-1.0

and permutations of the data will produce according permutations of the output above.

Z-scores are not affected by filters (at the moment)

```
ZSCORE(HOURS("_CEL_P2P_ACTIVITIES"."EVENTTIME"))
```

computes the global z-scores, which is not always as intended. Assume that you have filtered for some (e.g. un-automated) activities using the UI, which seem to be normally distributed, for which you would like the z-scores of. In that case you have to filter the column again using PQL, e.g. via `CASE WHEN` which sets some values to `NULL`.