USERGUIDE
EMM

METRON-ENERGY MANAGEMENT MODULE

(JOOL)

POWERED BY METRON

Laurent HANET

Version 1.03.10 - 23/5/2023

Table of contents

1. Basic knowledge of object-oriented language

EMM uses an "object-oriented" programming syntax. This means that each element worked on in EMM will be an "object" to which it will be assigned its own type, properties and functions.

An object can be a string, a date, or even a water meter for example.

Each type of object is associated with a series of functions that can be applied to it. For example, the . avg function, which performs an arithmetic average, is relevant if applied to a series of numbers. It is much less relevant if you try to apply it to a string of characters.

In practice, EMM will therefore use objects, and transform them into other objects depending on the function used.

Example:

We could start with a Meter, retrieve its name, and then count the number of letters in the Meter's name.

The Name recovery function will therefore transform the Meter object (of type "Meter") into another "Channel name" object (of type "character string"). To do this, EMM will retrieve the value of its "Name", which is a character string, from the Meter's properties.

Next, the Character Count function will count the number of characters in the string provided by the previous step and will therefore provide an object of type "Number", which will be the number of characters in the Meter name.

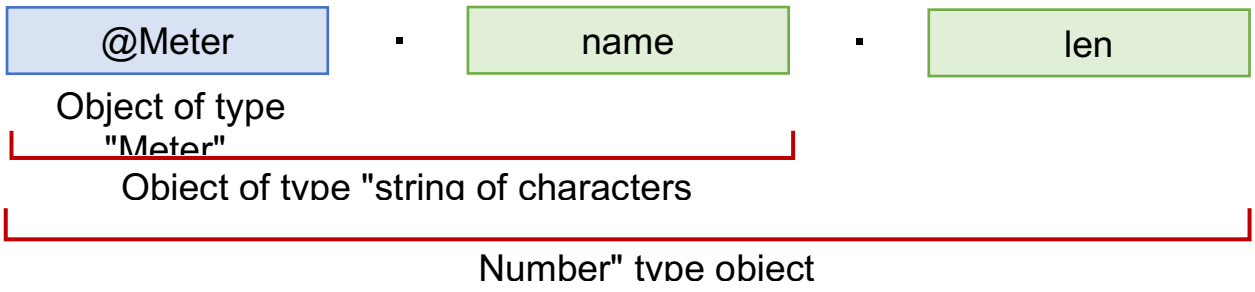
We therefore have a series of cascading operations, all starting from one object and producing a new object.

The EMM syntax will list the different functions one after the other, separated by a dot, in the direction of reading.

.len being the operator that counts the length of a text (from LENgth), the above example will become

@Meter. name. len

This syntax will work as follows:



@Meter = the start object, the Meter, an object of type "Meter

@Meter. name: the . name function applies to an object of type Meter and returns an object of type "string", in this case the name of the Meter in question.

@Meter. name. len: the . len function starts with a string and returns a number, corresponding to the number of characters in the string.

It is important, when writing EMM codes, to always keep in mind what the active object is, to ensure that the following function can be applied to it, and to follow the reasoning step by step until the desired value is achieved.

Some functions are simple and direct and will be applied to their object, without needing additional information, like the functions listed above (.name, .len ...)

Other functions may or must be given one or more additional parameters to work, such as the .round function which rounds a decimal number. This function must receive a parameter indicating the number of digits after the decimal point to keep in the rounding. This parameter will then be passed to the function as an argument. It will be given the value of the parameter to be used between brackets when it is called.

item. **round**(2)

→ this expression will return the value of the number stored in the variable "item", rounded to 2 decimal places.

Many functions will therefore need arguments to work. Some will need several arguments, which will be supplied in brackets, all separated by semicolons, in the order set by the function definition.

selection.data(from;to)

Some parameters may also be optional and not strictly required for the proper functioning of the formula.

Note that the parameters can be of any format (text, numeric, date...), depending on what is required by the function.

2. Basic features

a. Text object

○ General

Text objects are strings of characters, which can be used by the system for various applications, such as identifying, sorting, or characterising Meters.

To specify a value in text format in the system, it will be enclosed in inverted commas " ".

If a text is to contain inverted commas (which should not be interpreted as a text closure), they must be doubled in the text.

Thus, the text hello "world"! should be encoded as follows: "hello ""world""!"

It is possible to concatenate strings via the **+** symbol

"hello" **+** "world" →hello world

Texts can be explicit or contained in a variable. For example, item.name is a text containing the name of the selected item, in text format.

○ Functions associated with texts

Several functions can be applied to objects in text format, whether they are explicit, and noted between " ", or are contained in a text variable.

.len

From a string of characters, this function will return a numerical value indicating the length of this string (number of characters, including spaces)

Examples:

("test"). **len** →4

item.name. **len** →The length of the active item name.

.contains

Receives as argument a string of characters and compares it to the chosen text. The function will return TRUE or FALSE (Boolean) depending on whether or not the string is included in the text.

Note that the comparison is not case sensitive.

Examples:

("this is a test"). **contains**("test") →TRUE

("this is a test"). **contains**("TEST") →TRUE

("this is a test"). **contains**("red") →FALSE

item.name. **contains**("red") →will return TRUE or FALSE depending on whether the word "red" is contained in the active item name

.startswith /.endswith

Receives as argument a string of characters and compares it to the chosen text. The function will return TRUE if the text begins (.startswith) or ends (.endswith) with the string and it will return FALSE (Boolean) otherwise.

Once again, the comparison is case-insensitive.

Examples:

("this is a test"). **startswith**("is") →FALSE

("this is a test"). **endswith**("is") →TRUE

("this is a test"). **startswith**("this") →TRUE

("this is a test"). **startswith**("THIS") →TRUE

item.name. **startswith**("red") →Will return TRUE or FALSE depending on whether the active item name starts with "red" or not

.left /.right

From the chosen text, create a string by copying from the left (.left) or from the right (.right) the given number of characters of the chosen text.

These functions can for example be useful on structured references that would contain a site reference concatenated with a Meter reference, in order to separate the references.

Examples:

("this is a test"). **left** (6) →this is

("this is a test"). **left** (10) →this is u

("this is a test"). **right** (6) →n test

("this is a test"). **right** (15) →eci est un test

Note: The space is always counted as a character like any other.

.replace

This function is used to change a piece of text and replace it with another text.

Note: This function **is** case sensitive when searching for text to replace.

Syntax :

("message to be modified"). **replace**("text to be found/removed"; "text to be inserted")

Examples:

("message to be modified"). **replace**("modify"; "transform") →message to be transformed

("message to be MODIFIED"). **replace**("modify"; "transform") →message to be MODIFIED

("this is a test"). **replace**("e"; "X") →cXci Xst a tXst

b. Digital object

○ General

In EMM databases, numerical values are stored with a dot as a decimal separator, and without a thousands separator, regardless of the user's cultural preferences.

A distinction must be made between the numerical objects as used in the program, which are stored and used with a dot and without thousands separators, and the final numerical objects, which are presented to the user in his grids or graphs, and which are displayed according to his cultural preferences. Using a decimal numerical value in a formula in the syntax will therefore be done by using the dot (.) as a decimal separator.

Numerical values in EMM are encoded in "double" format (standard IT format, where numbers are encoded on 64 bits).

EMM supports basic mathematical operations (+ - * /), whether they are used with explicit numerical values or via variables containing numerical values.

○ Predefined values

Some values are frequently used and are therefore pre-encoded in EMM.

pi = 3.14159265358979

null = the empty value (often used in formulas to test if a data item is empty)

rand = returns a random value between 0 and 1

rand(10) = returns a random value between 0 and 10. The argument passed to the function tells it the maximum value allowed for the random number to be generated.

○ Functions associated with numerical values

The following functions will apply to objects of numeric type, whether they are explicitly specified, or are contained in a variable of numeric type such as a consumption value.

.round

This function is applied to a numerical value and returns the same value, rounded to a number of decimal places specified by the argument provided.

The argument must be an integer, and will therefore indicate the number of digits kept after the decimal point.

Examples:

pi. **round**(4) →3 ,1416

item.value. **round**(4) →Return the value of the item rounded to the 4th decimal place.

.floor /.ceiling

These functions receive a numerical value, and provide the value rounded down (floor) or up (ceiling)

Examples:

(4.8439). **ceiling** →5

(-2.156). **ceiling** →-2

(4.8439). **floor** →4

pi. **floor** →3

item.value. **floor→Return** the value of the item, rounded down

Note: the brackets around the numerical values are not essential, but they can clarify the reading, and they avoid confusing the decimal point with the point separating the successive functions of the object-oriented language.

Remainder of integer division (modulo)

Modulo division simply uses the '%' symbol instead of '/'.

Examples:

5 % 3→2

.abs

Receives a numerical value and returns its absolute value.

Examples:

(-315.4563). **abs→315** ,4563

(42.15) . **abs** →42 ,15

item.value. **abs** →Return the absolute value of the item' s value.

.sqrt

Receives a numeric value and returns its positive square root.

Examples:

16.sqrt →4

(16.5). sqrt →4 ,06201820231798

item.value. sqrt →Return the positive square root of the item's value

.power

Raises a numerical value to the power passed as an argument. The exponent does not have to be an integer or even positive.

Examples:

3.power(2) →32 = 9

2.power(-1) →2-1 = 0,5

4.power(2.1) →42,1 = 18,3791736799526

item.value. power(2) →return the square of the item' s value

.exp /.log /.log10

These functions will respectively return the exponential, the natural logarithm (ln) or the logarithm to the base 10 of the numerical value to which they are applied.

Examples:

3.exp →e3 = 20,0855369231877

12.log →ln (12) = 2.484906649788

1000.log10 →log10(1000) = 3

item.value. exp will return the value of the item

Advanced Note: These functions can be applied to numerical values but also to series of values, data profiles or collections of numbers (see definitions below). In this case, they will provide a new series of values containing the exp, log or log10 of the values of the initial series.

Digital collection (advanced example) :

(100; 10; 100000). log10→2, 1, 5

Data profile (advanced example) :

item.data. **exp→** Returns the list of exponentials of each value in the profile

.sin /.cos /.tan

These functions will respectively return the sine, cosine or tangent of the numerical value to which they are applied.

Examples:

(2). **sin** →sin (2) = 0.909297426825682

(pi/3). **cos** →cos (pi/3) = 0.5

item.value. **tan** →return the tangent of the item value

Advanced Note: These functions can be applied to numerical values but also to series of values, data profiles or collections of numbers. In this case, they will provide a new series of values containing the sine, cos or tan of the values in the original series.

Digital collection (advanced example) :

(pi/3; 0; 25). **cos** →0.5, 1, 0.9912028118634736

Data profile (advanced example) :

item.data. **sin→Returns** the list of sines of each value in the profile

.asin /.acos /.atan

These functions will respectively return the arcsine, arccosine or arctangent of the numerical value to which they apply.

Values passed to asin and acos must be between -1 and 1 or they will be incalculable.

Examples:

(0.5). **asin** →pi/6 = 0.5235987755982988

1.**acos** →0

item.value. **atan** →return the arc whose item value is the tangent

Note: These functions can be applied to numerical values but also to series of values, data profiles or collections of numericals (see definitions below). In this case, they will provide a new series of values containing the asin, acos or atan of the values of the initial series.

Digital collection (advanced example) :

(0.5; 0; 1). **acos** →1 .0471975511965979, 1.5707963267948966, 0

Data profile (advanced example) :

item.data. **asin→Returns** the list of arcsinus of each profile value

.format

This function allows you to give a certain form to numbers for their final display in a table or in a report for example. It is applied to a number and returns a string of characters (text) in the requested format. The code indicating the format is passed as an argument to the function.

Caution: as the result of this function is a "text" object, it cannot be recovered (except in exceptional cases) to perform further calculations.

The format code passed as an argument to this function will be a text, reflecting the structure to be used to display the number. This format uses certain well-defined characters to symbolise the components of the final number.

- the dot represents the decimal separator
- commas represent thousands separators
- The "#" is used to indicate a number, **IF there is** one to be displayed.
- The "0" is used to force the display of a number, and will display a zero if there is no value to display.
- The other characters will be copied as they are in the final result.

The symbols actually used as decimal and thousand separators in the displayed result will be those defined by the user settings.

Examples:

(1.12345). **format**("#.#") →1 ,1

(1.12345). **format**("#.###") → 1,123

(1.12345). **format**("###.###") → 1,123

(0.12345). **format**("#.###") → ,123

(0.12345). **format**("0.###") → 0,123

(0.1). **format**("0.###") → 0,1

(0.1). **format**("0.000") → 0,100

(0.1). **format**("0.##0") → 0,100

(123456). **format**("#") → 123456

(123456789). **format**("###,###,###") → 123 456 789

(123). **format**("# kWh") →123 kWh

Remarks :

The # before the decimal point does not limit the number of digits displayed (see second last example)

The # after the decimal point defines the number of digits displayed after the decimal point (see first examples)

c. Subject date

○ General

In the EMM system, a date is a type of object containing a year, month, day, hour, minutes, seconds and milliseconds.

These objects will be used very frequently in EMM, particularly in the creation of data profiles, composed of "Date-Value" pairs.

In the EMM database, dates are stored in GMT, and are then displayed to the user once they have been transposed into their time zone.

It is possible to encode a date directly in a box, using the following format:

"2015-03-01 15:12:05.020" →first March 2015, 15:12 and 05 seconds, 20 thousandths.

However, the format being rather strict, and the date being automatically considered as in GMT, this makes this use rather off-putting. It is preferable to use the dateeval function (described a little further on) which will allow you to dynamically create a date from its components, and directly in local time.

Displaying a date object as text in a spreadsheet will return the data stored in hard copy in the database, and its format is not the clearest. It is therefore preferable to use the "Date" format to display dates.

○ Predefined dates

There are date values in the system which are created automatically on request.

from	Start date of the active time context
to	End date of the active time context
now	Current date (year, month, day, hour, minute, second, millisecond)
today	Today's date, midnight this morning (= now.synchro(1; "day"))

As a reminder, the time context is the period of time that has been selected in the time selector at the top of the EMM window in use. In the particular case of an automatic task, running without EMM open, and therefore without a selected context, the active context will be defined as the period between

- from = now minus the time step configured for the task
- to = now, date of execution = now.

When you are in the From or To box of an editor (spreadsheet, HTML report, etc.), the from and to keywords will refer to the start and end dates of the time context displayed in the selector. These From or To boxes can be used to modify the active context via a formula, locally, for the needs of the current tool. For example, the dates can be transposed one week backwards, so that in a dashboard based on the current week, the consumption of the past week can be displayed in one of its widgets.

Once inside the definition of the tool (table,HTML report...), the from and to keywords will then refer to the beginning and end of the local context, as a result of any modifications made in the From and To boxes.

Note: As we will see later, a temporal context different from the active context can be imposed when calling an array of data, or via the use of an execute or when calling an array. These contexts will then be substituted for the context of the selector. All the rest of the previous reasoning remains similar.

Note: Similarly, some functions can be given optional parameters that include a from/to context. These functions will then limit their results to what is within the time period defined by this temporary context.

- Functions associated with dates

Date objects can be used or modified by functions, for example to synchronise data sets or to find another date from the start date.

dateeval

Construct a date object from its numerical components (year, month...). This function allows you to compose a date without having to worry about the details of the format. It is strongly recommended to use this function when it is necessary to encode a date manually in hardcopy.

Note that this formula does not apply to an already existing object. It constructs a date on the basis of the parameters provided to it.

Syntax :

dateeval(year; month; day; hour; minute; second)

The time periods are encoded in numerical values (no need for "), and apart from the first mandatory parameter, all the others are optional. If not all the values are given as parameters (for example, if we stop at hours), the date constituted will be that of the beginning of the associated period.

Examples:

dateeval(2015; 12; 20; 6) →December 20, 2015 at 06:**00**:**00**

dateeval(2015; 12; 20) →On 20 December 2015 at **00**:**00**:**00**

dateeval(2015; 12) →On **1** December 2015 at **00**:**00**:**00**

dateeval(2015) →On **1 January** 2015 at **00**:**00**:**00**

.add

Adds a defined duration to a given date. The function receives 2 parameters: the number of time intervals, and the type of time interval, in order to tell it the duration to add, and provides a value in date format.

Syntax :

. **add**(nbr_intervals; "type_interval")

- nbr_intervals: indicates the number of intervals to add (numerical value)
- "type_interval": indicates the type of interval. Text format (i.e. between " "), choose from year, month, day, hour, minute or second

Note: Instead of creating another function to subtract a duration, the .add function accepts negative durations and will then perform a duration subtraction.

Examples:

now. **add**(1; "day") →tomorrow at the same time (+1 day)

now. **add**(1; "month") →next month, same day, same time (+1 month)

from. **add**(-1; "year") →One year before the start date of the current context

to. **add**(2; "month") →Two months after the end date of the current context

dateeval(2015; 2; 10). **add**(2; "day")

→ will return the date 12 February 2015, 00:00:00 (10 February + 2 days)

.sync

Changes a date back to the beginning of a given period. For example, it can be used to obtain the start date of the month or the start date of the year containing the date selected in the context.

Syntax :

. **sync**(interval_size ; "interval_type")

- interval_size: indicates the size of the considered interval (numerical value)
- "type_interval": indicates the type of interval. Text format (i.e. between " "), choose from year, month, day, hour, minute or second

Examples:

now. **synchro**(1; "day") →This morning, 00:00:00

now. **synchro**(1; "month") →Midnight , the morning of the first day of the current month

now. **synchro**(3; "month") →Midnight , the morning of the first day of the current quarter

now. **synchro**(1; "year") →Night , the morning of the first of the year of this year

to. **sync**(1; "month") →Midnight , the morning of the 1st day of the end month of the context

dateeval(2015; 2; 10). **synchro**(1; "month") →The first of February 2015, 00:00:00

.year / .month /.day /.hour /.minute /.second

Returns the **number** (numeric value) of the year, month, day, hour, minute or second of the given date (no additional parameters)

Examples:

now. **hour** →The current time

to. **year** →The year of the end of the context

from. **month** →The number of the month in which the context begins

dateeval(2015; 12; 20). **month** → 12

.weekday / .day365

Similar to the .day function, the .weekday function will return the number of the weekday associated with the given date: 1 for Monday, 2 for Tuesday ... and 0 for Sunday.

The .day365 function will return the serial number of the given day in the year (over 365/366 days)

Warning: In the case of leap years, the .day365 function will count the 29th of February. The 1st of March will therefore not have the same value whether it is a leap year or not. This detail is important if we try to compare the relative progress in the year between two different years for example.

.format

This function starts from a date and generates a text, translating the given date according to the format passed as argument.

The time/date format passed as a parameter is of text type (so between " ") and follows the formats of c#.

Code	Period	Explanation	Example
G	Era	The era	A.D.
yy	Year	The last two figures of the year	15
yyyyy	Year	The 4 figures of the year	2015
M	Month	The month	1 - 12
MM	Month	The month in 2 digits	01 - 12
MMM	Month	The month in short text (localized by user)	Oct
MMMM	Month	The full name of the month (localized by the user)	October
d	Day	Day of the month	1 - 31
dd	Day	Day of the month with 2 digits	01 - 31
ddd	Day	Short name of the day (located by user)	on.
dddd	Day	Full name of the day (localized by user)	Monday
h	Time	Time in 12-hour format	1 - 12
hh	Time	Time in 12-hour format on 2 digits	01 - 12

H	Time	Time in 24-hour format	0 - 23
HH	Time	Time in 24-hour format on 2 digits	00 - 23
m	Minute	The minute	0 - 59
mm	Minute	The 2-digit minute	00 - 59

(full table -> https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-date-and-time-format-strings)

Examples:

now. **format**("dd/MM/yyyy hh:mm:ss") →16/10/2015 12:01:45

now. **format**("dd/MM/yy hh:mm") →16/10/15 12:01

now. **format**("dd-MMM-yy") →16-Oct-15

now. **format**("ddd dd-MMM-yyyy") → Fri . 16-Oct-2015

datediff

This function receives two dates and a type of time step (text) as arguments and constructs a numerical value counting the number of time steps of that type entirely between the two dates.

Note that this formula does not apply to an already existing object. It is constructed solely on the basis of its parameters.

Syntax :

dateiff(date1; date2; "day")

→Will return a numeric = the number of integer days between date1 and date2.

Dates can be entered in hard text format (not recommended), or using dateeval, or based on a formula: from, to, now, from.synchro(1; "day") ...

Example:

datediff(now.synchro(1; "month"); now; "day")

→Return the number of whole days since the beginning of the current month.

daysinyear / daysinmonth

Functions receiving a date as an argument, they will return the number of days in the year/month containing the given date.

Note that this formula does not apply to an already existing object. It is constructed solely on the basis of its parameters.

Examples:

daysinyear(now)

→Will return a numeric indicating the number of days in the year containing the date "now". Basically, this will return 365 or 366 depending on whether or not it is a leap year.

daysinyear(dateeval(2015; 12; 20; 6))

→Renverra 365. Indeed, the year 2015, containing the date 20/12/2015 6:00:00, is not a leap year and therefore contains 365 days.

daysinmonth(dateeval(2015; 12; 20; 6)) →31

daysinmonth(dateeval(2017; 2)) →28

daysinmonth(column("FROM"))

→ Will return the number of days contained in the month that contains the date present in the "FROM" column. If the column contains anything other than a date, the function will return an error.

d. Logical object (Booleans)

○ General

The values true and false represent the logical values "true" and "false" and can be used directly in the system. These values are often the result of logical tests (with if, contains, startswith... for example), and are rarely written explicitly in a formula.

○ Logic functions and connectors

All functions that generate, use or combine one or more logical values are listed here.

> < = >= <= <>

Logical operations can be used directly in the syntax. They compare their arguments and return a logical value that can be used later in a logical function such as an if for example.

The following operations are available:

>	bigger
<	smaller
=	equal
<=	smaller or equal
>=	greater than or equal to
<>	different

Examples:

2 < 1 →false

2 > 1 →true

item.value = 1 → Returns true if the item is 1 and false otherwise

.not

Very basic, this function allows you to reverse the value of a Boolean. Thus, a true value will become false, and a false value will become true

(2 < 1). not →true

(2 > 1). not →false

(item.value = 1). **not→Returns** false if item is 1 and true otherwise

.isnull / .isnotnull

These functions receive a value and check if it exists or if it is empty, which corresponds to the computer value null.

Remember: Null ≠ 0. Null means that the variable does not exist, while 0 indicates that the variable exists, and that its value is zero. We are dealing here with the same nuance as between a consumption whose value has not been received (null) and a consumption actually at zero over a given period (0).

Examples:

(null). **isnull** → true

(0). **isnull** → false (0 is not "empty", it is the numerical value zero)

(item.value). **isnotnull**

→ Returns true if the item value exists and false otherwise

and / or

The keywords and and or are recognised by the system to combine tests or conditions. They correspond to logical "and" and "or" and will produce a logical value from two logical starting values or conditions.

Examples:

true **and** true →true

true **and** false →false

(2 < 1 **or** 4 > 3) →true

Advanced example:

selection.where(item.name.startswith("A") **or** item. name.startswith("B"))

→ Will return selected entities whose name begins with A or B

if

If you want to conditionally retrieve one value or another based on an expression, you can use an "if" condition.

The conditional expression must provide a true or false value, but these values can be the result of a more complex formula.

Syntax :

if(condition; yes_value; no_value)

Examples:

if (true; 1; 0) →1

if (2 < 1; 1; 0) →2 < 1 = false, so the if condition will return 0

Advanced examples:

if (item.value > 0; "ok"; "zero")

→The conditional expression checks whether the value of the current item is greater than zero. If so, the if condition will return the text "ok" and if not, it will return the text "zero".

if (@bxl.data.isnull ; @default_profile.data ; @bxl.data)

→The conditional expression checks whether the data in the bxl Meter is empty. In this case, a default profile is taken, and if not (if the Meter data was indeed received for example), the actual data is taken.

if (@bxl.alarm("//MAX_ACTIVE") ; "Alarm Max ON" ; "Alarm Max OFF")

→The condition will look for the value of the alarm parameter "//MAX_ACTIVE", which is a Boolean (the property has been defined this way). This Boolean value is then used directly as the result of the condition and the function will return "Alarm Max ON" or "Alarm Max OFF" depending on whether the alarm is active or not.

Under this condition, the following two expressions will have the same effect:

if (@bxl.alarm("//MAX_ACTIVE") = true ; "Alarm Max ON" ; "Alarm Max OFF")

if (@bxl.alarm("//MAX_ACTIVE") ; "Alarm Max ON" ; "Alarm Max OFF")

Advanced remark: If the Boolean value comes from a property, the use of the .not function must be well understood. Indeed, a property can have as value, not only true and false, but also null, in the case where the property is not defined.

In this case, it should be remembered that true.not ≠ false

Indeed, true.not simply indicates that the value is not true. It can therefore be false **or** null.

3. Entities / Meters / Channels and Properties

a. Entity objects and Meters

○ Accessing an entity or Meter

In EMM formulas, an entity or Meter will be identified by @Reference. The "@" symbol indicates that we are talking about an entity/Meter, and its unique reference allows us to identify it. This syntax allows direct access to an entity/Meter as long as the reference is known.

Alternative syntax: @(item.column("site_reference")). This expression allows you to retrieve an entity/Meter by passing directly through its reference, even if it is not known at the time of writing the code or if it must change dynamically according to the situation. It can be seen as the "@" function which receives a reference (text) as an argument, and which will return the object (entity/Meter) having this reference.

Another way to designate an entity/Meter is to use the keyword selection. This term will designate the entity/Meter (or entities/Meters) which is (are) currently included in the active selection. The selection depends on the tool you are working with. If you are working in a spreadsheet, it will be the result of the formula

Furthermore, the term all will return a collection containing all existing entities/Meters (not just those listed in the tree), regardless of the active selection in the channel selector.

Note: This term all will include both the entities/Meters visible in the entity selector view and any entities/Meters that are not displayed. Indeed, as a reminder, the views displayed in the selector can be customised and can therefore display all the existing entities/Meters, or a more restricted subset.

Note: The term all will however be limited to those entities/Meters to which the active user has read and/or write rights. Indeed, those to which he does not have access remain hidden, even if he uses the keyword all.

Finally, it is also possible to access all entities/Meters of a given type via the following syntax: #REF_DU_TYPE. The symbol "#" indicates that we wish to retrieve a list of entities/Meters, and the type reference attached to this symbol allows EMM to filter the entities/Meters to keep only those of the requested type.

Example: #SITE will return the list of entities of type "SITE

This syntax is much faster than using an "all.where(item.type="SITE")". Indeed, this last syntax recovers in DB the totality of the existing entities/Meters, and tests them all on their type, whereas the syntax #SITE directly recovers the good list of entities in DB.

- Functions associated with entities and Meters

.reference / .name / .icon / .creation / .lastupdate /.type

These functions in the EMM syntax are used to retrieve the identifying information of the entities/Meters, be it the name, the reference, or even the icon displayed in the selector.

.reference	Returns a string containing the unique reference of the entity/Meter (This function can also be applied to several other types of objects that exist in EMM, and will return the unique reference of the object in question)
.name	Returns a text containing the name of the entity/Meter
.icon	Returns a text containing the name of the entity/Meter icon (excluding the extension) Once on the icon, it is then possible to specify that you want the name of the icon or its colour via the following functions: .icon. **value** → Will return the name of the icon (which ".icon" already does by default) .icon. **color** → Will return the hexadecimal code of the icon's colour
.creation . lastupdate	These two functions will return the creation or last modification information of the object concerned. On this information, we can then recover the user or the date of this creation or update. item. creation. date → will return the date the item was created item. lastupdate.date → will return the date of the last update of the item Note that if you retrieve the "user" object associated with the item, you will then have to indicate what you want to retrieve as information about this user. item.creation. user.name → will return the name of the creator of the item item. lastupdate. user.mail → will return the email address of the user who last updated the relevant item. **Note**: These functions also apply to other object types, such as events, invoices, etc., as well as to tools such as DataSets. Dataset("DSET_LHT_TEST").creation.user.mail → will therefore correctly return the email address of the user who created this DataSet.
.type	Returns the type of the active entity/Meter

Examples:

@C1.**reference**	→	"C1" the reference of the requested entity/Meter
selection. **name**	→	returns the name of the selected entity/Meter
selection. **icon**	→	"The name of the entity/Meter in the case of a lightning bolt icon.

selection. **type** → "SITE" in case the active entity is of type "SITE".

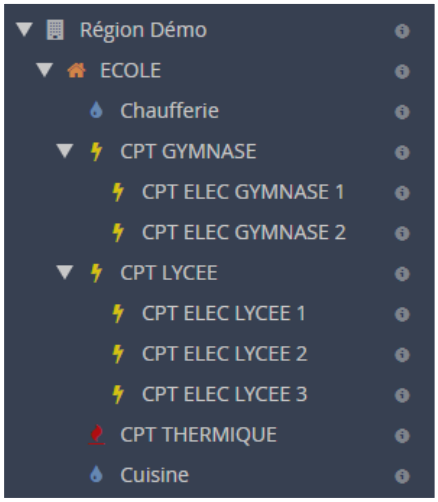
.parent / .children / .ancestors / .descendants

In the syntax, in order to access, from one entity/Meter, the other entities/Meters linked by kinship, one can use the following formulas:

.parent	Moves the active entity/Meter to the direct parent, immediately above the current entity/Meter in the structure tree. **Note:** An entity/Meter can only have one parent.
.children	Creates a list (collection) of all direct children of the current entity/Meter. **Note:** an entity/Meter can have several children, hence the interest of a list
.ancestors	Creates a list of all parents and parents of parents up to the root of the tree. **Note:** this list is made starting from the current entity/Meter, and moving away from it. The last entity/Meter in the list thus generated will then be the most distant ancestor, thus the root of the tree.
.descendants	Creates a list of all entities/Meters under the current selection, in the tree, starting with the direct children (.children) and then recursively continuing on all the children of the children and so on until the tree is exhausted.

Examples:

If the tree structure is as follows (assuming that the names and references of the entities/Meters are similar, and assuming that the actual tree structure corresponds to the active view)



@CPT _GYMNASE. **parent**.name → "SCHOOL", the name of the parent

@CPT_ELEC_GYMNASE_1.**parent**.reference → "CPT _GYMNASE"

@CPT_ELEC_GYMNASE_1.**parent**.parent.name → "SCHOOL"

@ECOLE. **children**

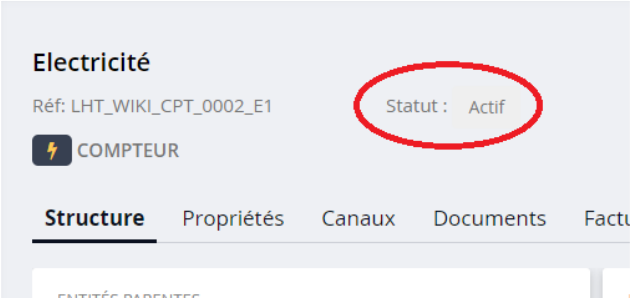
→Will return the list of entities directly under the entity "SCHOOL". In this case: boiler room; CPT_GYMNASE; CPT_LYCEE; CPT_THERMAL; kitchen

@CPT_ELEC_LYCEE_1.**ancestors**

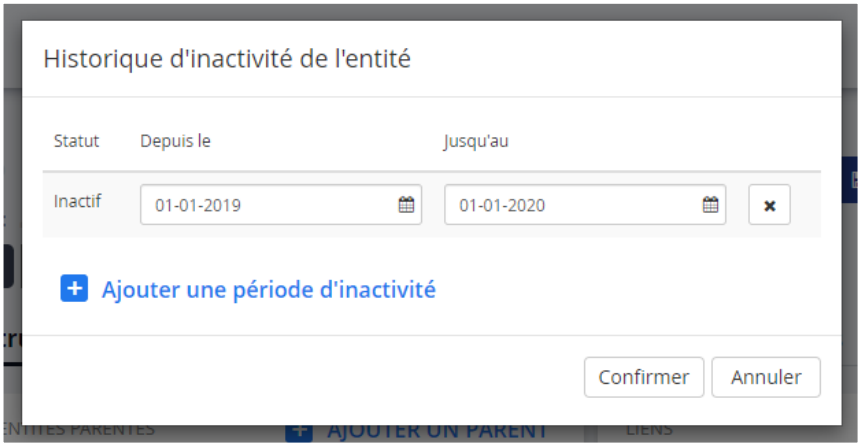
→Return the list of parents and grandparents... above the entity "CPT_ELEC_LYCEE_1". In this case: CPT_LYCEE; SCHOOL (in that order)

. status

On the record of each entity/Meter, there is a "Status" button indicating the current status of that entity/Meter.



In edit mode, this button allows you to add periods of inactivity if the entity/Meter must be deactivated for a certain period (contract on break, site under construction...)



This status information (active/inactive, and from when to when) can be retrieved in the DataSets, but also via the formulas of the parser thanks to the ".status" function. This function is used in conjunction with the ".isactive", ".from" or ".to" functions to obtain the status details. The ".status" function also accepts optional date parameters that allow you to target the time period for which you want status information.

Syntax :

@ENTITY. **status**. **isactive**

→Will return "true" if the entity/Meter is **currently** active, and "false" otherwise.

@ENTITY. **status**. **from**

→Return the start date of the current status. Since when the entity/Meter has been in its current status. (if no start date entered, will return 1/1/0001).

@ENTITY. **status**. **to**

→Will return the end date of the current status. Since when the entity/Meter has been in its current status. (if no end date is entered, will return 31/12/9999)

@ENTITY. **status**(from ; to). **isactive**

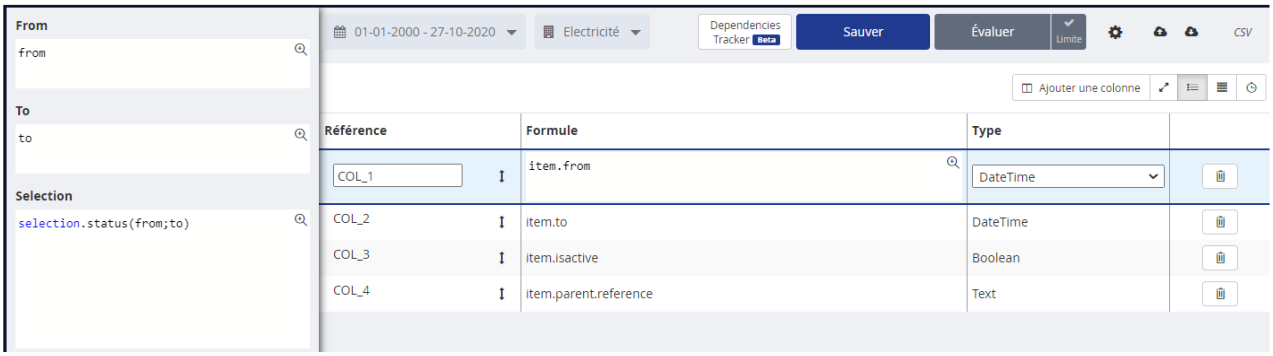
→Will return a Boolean indicating whether the status is active during the period (from/to) filled in. In case there are several different statuses during the indicated period, the formula will return a list of Booleans. Obviously, the dates (from/to) can be replaced by other formulas returning dates (dateeval for example).

@ENTITY. **status**. **parent**

→ Will return the entity/Meter to which the current status belongs.

Example of practical use

If one wants to retrieve all the status information of an entity/Meter, one can work as follows.



The selection therefore contains a list of objects of type "status", retrieved over the period covered by the active context (from/to). There will therefore be one row per status in the results table, and for each row, we have created columns containing the formulas "item.from", "item.to", "item.isactive", and "item.parent.reference". The keyword "item" here refers to a status taken by the entity/Meter in each row.

The result of such a worksheet could then be the following:

COL_1	COL_2	COL_3	COL_4
DateTime	DateTime	Boolean	Text
	01-01-2019 00:00:00	☑	LHT_WIKI_CPT_0002_E1
01-01-2019 00:00:00	01-01-2020 00:00:00	☐	LHT_WIKI_CPT_0002_E1
01-01-2020 00:00:00		☑	LHT_WIKI_CPT_0002_E1

Advanced note

It is possible to import massively the statuses of entities/Meters. The method is similar to the import of historical properties, with one line to import per status, and the fields "Status" (Boolean), "Status Since" and "Status Until" (dates).

.link

If you want to start from a selected entity/Meter and choose another entity/Meter associated with it (via a logical link) as the active selection, you can use the .link function

This function therefore starts from an entity/Meter, and takes as argument a text indicating which type of link to follow.

Note: As there may be multiple logical links (1:N links), it is possible that this syntax returns multiple entities/Meters as a result.

Note: Using the .link function without passing any arguments will return a list of all linked entities/Meters, regardless of the type of link.

Examples:

```
selection. link("WEATHER")
```

→Will allow access to a shared Meter which, in the tree structure, is connected to the selected entity by a link of type TEMPERATURE (parameter in text format).

```
selection. link
```

→Return all entities/Meters linked to the active selection, for all link types present.

CAUTION: The above syntax returns the linked entity/Meter itself as the active element. If you want to retrieve the reference of the target entity/Meter, you must specify it

```
selection. link("WEATHER").reference
```

.xlink - remote links

In many situations, it is useful to retrieve an entity/Meter associated with our active selection, but whose path is long or often variable.

The most common example is the retrieval of the weather Meter from a selected Meter. Generally the weather Meter is linked to the site entity, and not to each of its Meters. Therefore, starting from a Meter, the retrieval of the weather Meter would use the following syntax

```
selection.link("ENTITY_METER").link("WEATHER")
```

However, this formula is a bit too rigid. Indeed, it is only functional in the case of a Meter directly linked to the entity which is itself linked to the weather Meter. If we start with a sub-Meter or if the directly linked site is a sub-entity of the one linked to the weather Meter, we will have to adapt the formula. However, adapting such a formula so that it takes into account all possible cases can quickly become very cumbersome, both in terms of coding the formula and its execution.

```
selection. foreach(item; item.ancestors).foreach(item; item.link)
.foreach(item; item.ancestors). link("WEATHER").top(1)
```

...and even then, this syntax does not necessarily cover all possible cases.

To overcome this problem, the .xlink function was created. This function works like the .link function, except that if it does not find the expected link directly on the active selection, it will search on the entities/Meters linked to the selection, progressing by successive links until it finds the link it is looking for.

```
selection. xlink("WEATHER")
```

This formula will retrieve the weather Meter linked to the selection, even if the link is not direct. The xlink function will therefore check if the selection has a weather link itself. If so, it will retrieve it and stop there. If not, it will check if the entities/Meters linked to the selection have a weather link. If it finds such a link, it retrieves it and stops there, and if not, it continues on the following links...

Note: The order in which the links are browsed for the desired xlink is predetermined by the EMM administrator.

. ismaster - detection of subscriber entities/Meters

At times it may be useful to be able to differentiate an entity/Meter from a subscription from another entity/Meter that is only present in the customer database. The ".ismaster" function has been created for this purpose.

It simply applies to an entity/Meter and returns a Boolean indicating whether or not the entity/Meter in question originated from a subscription.

Syntax :

```
@REFERENCE. ismaster
```

Obviously, this function can be applied to one or more entities/Meters, either explicitly mentioned or resulting from a formula.

Example:

```
all.where(item. ismaster)
```

→Return the list of all subscribed entities/Meters in the database.

b. Notion of properties

A property is a piece of information stored in the database, and associated with an object. Properties can be associated with entities, Meters, channels, users... Properties can be of various types, such as text, numerical values, Boolean values...

It is therefore possible to use mathematical functions on the returned values, such as checking their values, or adding them together.

In the same way, Boolean properties can be used directly in formulas as conditions of an if function for example.

- Simple properties

So-called "simple" properties are those that have a unique and invariable value. An example is the year of construction of a building, which is a unique value that cannot change over time.

.properties

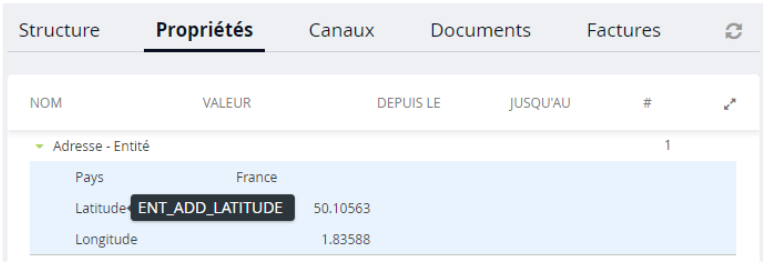
To retrieve the value of a simple property, simply use the ".properties" function, passing the reference of the property to be retrieved as an argument.

Syntax :

selection. **properties**("//PROPERTY_REFERENCE")

- The "//PROPERTY_REFERENCE" is in text format and will contain the unique reference of the requested property (Warning: sensitive box). The "//" is made necessary by the format of storage of the properties in the EMM DB. These characters have no impact on the syntax, except that they must not be forgotten.

To find out the reference of the desired property, it can be displayed in a tooltip on mouse-over in the "Properties" tab of the entity/Meter record.



The properties are organised in a tree structure and can be grouped by topic in property blocks. For example, the address property block ENT_ADD will contain the properties ENT_ADD_STREET, ENT_ADD_NUMBER... which represent the different parts of the address.

selection. **properties**("//ENT_ADD/ENT_ADD _STREET") → Will return the street name

selection. **properties**("//ENT_ADD_STREET") →Return the street name (ditto)

item. **properties**("//ENT_ADD_COUNTRY")

→Return the value of the ENT_ADD_COUNTRY property (the country in the address) associated with the current object (item).

It is possible to pass as an argument to this .properties function, the reference of the entire property block. The active object will then become the property block on which other operations can be performed later (.where...)

Once a block is selected, the specific properties in the block can be retrieved via a new call to .properties.

selection. **properties**("//ENT_ADD"). **properties**("//ENT_ADD_LATITUDE")

→Return the latitude of the site. Here we have no difference between this call, explicitly going through the property block, or a call that would fetch the final property directly.

The main advantage of being able to call the complete property block will be clearer in the case of multiple property blocks and is therefore to be able to filter the blocks based on a given property, and then retrieve another useful property from the blocks that emerge from the filter.

.xproperties - remote properties

In the same way as for remote links and the .xlink function, it may sometimes be useful to retrieve a property which is not necessarily on the selected entity/Meter, but which may be on the parent, a link, or the parent of a link.

A classic example is the retrieval of the area of a site to calculate consumption per square metre. The consumption data is on the meter, which must therefore be selected, but the property containing the area is stored on the associated site. However, the link may not be direct (starting a sub-meter, passing through sub-sites on the way, etc.) and may even vary depending on the sites concerned.

Like the .xlink function for remote links, the .xproperties function was created to retrieve remote properties.

```
selection. xproperties("//ENT_TECH_TOT_FLOORS_SURF")
```

This formula will therefore start by checking whether the "total area" property is filled for the active selection. If it is, it returns its value and stops there, if not, it will search the parent and direct links to find this property. Again, if it finds it, it returns its value and stops its execution. If not, it searches the following links until it finds the requested value.

Note: Once again, the priority in the order of links that the .xproperties function searches is predefined by the EMM administrator.

.parent (on a property)

Once on a property, it is possible to go back to the object from which the property comes (whether it is an entity, a Meter, a channel, a user...). To do this, you can use the ".parent" function, which will return, for example, the Meter from which the active property comes. This possibility will be very useful in certain cases of advanced syntax, as well as after recovery of a property via the .xproperties function for example.

- Translated properties

For some properties, it is interesting to display the value translated into the language of the active user. To do this, the EMM administrator will have taken care to apply an encoding constraint to the property in question, so that only a series of predefined values is available. He will also have associated this property with a fixed data table (Xtab) which will contain the list of allowed values and, for each of these values, the translations in all the languages used. The details of this configuration will be discussed at a later stage of the training, when Xtabs will be discussed in greater depth.

Finally, the information stored in the database will be a unique key designating the chosen value, and when the property is displayed, EMM will go to the XTab to retrieve the translation of the value whose key is stored in the language of the active user.

Example: the XTab can contain one line

KEY	EN	UK	NL
ELEC	Electricity	Electricity	Elektriciteit

In this case, the information stored in DB will be "ELEC", but when a French-speaking user displays the property, he will see "Electricité", while a Dutch-speaking user will see "Elektriciteit".

.key / .value

In the syntax, when working on these properties with translated values, it can sometimes be useful to work on the key specifically, as when we want to filter Meters according to their consumed resource for example.

The .key function applied to a property allows the key to be retrieved rather than the translated value. Indeed, if we filtered a list based on the (translated) value of the property, we would have different results depending on the language of the connected user.

Examples:

selection.properties("//CNL_DAC_RESOURCE"). **key**

→ Will return the key of the property. In the example above, this syntax would return "ELEC".

selection.properties("//CNL_DAC_RESOURCE"). **value**

→ Will return the translated value of the property. In the example, "Electricity"

selection.properties("//CNL_DAC_RESOURCE")

→ Default behaviour: will return the translated value of the property as well.

It should be noted that these functions can also be applied to simple properties (without translations) without any effect.

- Multiple property blocks

In addition to "simple" properties, for which there is only one, unchanging value, some property blocks can be defined as multiple.

A site can for example have several telephone numbers. In this case, there will be several copies of the same property in the entity's record.

In EMM, single properties cannot be defined as multiple, but they can be embedded in blocks which can be multiple. This makes it possible to have a coherent set of properties, characterising the same thing.

Example: If a site has multiple pieces of equipment, and these pieces of equipment have multiple properties, a multiple block can be created for the site, each instance of which will represent a given piece of equipment and contain the set of properties associated with that piece of equipment.

In the case of an isolated property which must be able to be multiple, it will then be necessary to create a multiple block containing only this single property. This may seem restrictive when creating the property (an extra level will have to be created, with the block), but in use, it has no negative impact since properties can be accessed directly via the syntax, without going through the "block" level.

In the database, the instances of the multiple property blocks are only differentiated by their ID. It is this ID that will be used to uniquely identify the instances of the blocks to be updated during bulk imports (see the chapter on bulk imports below).

When we call, via a formula, a property included in a multiple block, and of which several instances exist in practice, we will obtain a list of the different values taken by this property in the different existing blocks.

Example: If we take the previous example and we have two pieces of equipment on a site, for which a "serial number" property is stored, the following command :

item.properties("//N_SERIES") →will return the 2 serial numbers in a row.

.id

If one wants to retrieve the ID value of each instance of a multiple property block, one can use the ".id" function on the called property. This will return the unique ID of the selected instance of the property block, and can be very useful for export-reimport on multiple properties (id needed for massive updates).

Warning: In EMM, it is not possible to have a property block that is both multiple and historicised.

- Historical property blocks

In practice, it is also possible that certain properties change over time, for example the area of a building when an extension is built.

In this case, it may be useful not to overwrite the previous data but to keep it as historical data (e.g. the old surface area should be kept, otherwise the calculation of consumption per square metre on old consumption data will be disturbed by the new surface area value)

In order to keep the old data, it will then be possible to historicise blocks of properties that are likely to vary over time.

The blocks of properties thus historised may appear in several copies on the record of the object in question (entity, Meter, channel, user, etc.), but they will all be associated with a different validity period (start and end date columns in the record).

Apart from the start and end dates of validity, the operation of historicised blocks of properties does not differ from that of single or multiple blocks.

The retrieval of properties is perfectly similar to that of simple properties, except that the retrieval of values will be based on the active temporal context and will only retrieve values whose validity at least partially overlaps the context.

In the case of retrieving a property that has been historised over a period of time where its value has changed, the call to the property will then return a list of the different values that were valid during the active context.

In the case of a historicised block of properties, it is possible to provide the .properties function with a temporal context to work with. To do this, use the following syntax:

@C1.Properties("//SURFACE"; **from**; **to**)

→ Will return, the list of surfaces of the site @C1 between the dates of the context (here, no difference with not putting dates)

@C1.Properties("//SURFACE"; **from**; **from**)

→ Will return the area of the site at the start date of the active context (from).

@C1.Properties("//SURFACE"; **now**; **now**)

→ Will return the current area of the site (now).

@C1.Properties("//SURFACE"; **dateeval(2017)**; **dateeval(2018)**)

→ Will send back the list of surfaces since the beginning of 2017 until the beginning of 2018.

If we ask for a property that has been historised over a period of time during which it has evolved, we will therefore have a list of values. The user is free to do what he wants with them, such as summing (.sum), averaging (.avg), taking the largest (.max), or simply the first (.top(1))...

.from / .to

When you call up a historicised property, it can sometimes be interesting to know the validity dates of each value of this property. To do this, simply use the ".from" and ".to" functions on the property to retrieve the validity start and end dates.

.id

As with multiple property blocks, it is also possible to retrieve the value of the ID of each instance of a property block that is being historised. Again, the ".id" function can be used on the called property. This will return the unique ID of the selected instance of the property block, and this ID can also be used to perform export-reimports of historicised properties (see chapter on bulk imports for more details).

Warning: In EMM, it is not possible to have a property block that is both multiple and historicised.

c. Channel object (data source)

As a reminder, a channel is a sub-section of a Meter, which can contain a data profile. Starting from the Meter, we must therefore specify which of its channels we want to retrieve information from.

The ".channels" function therefore applies to a Meter, and can receive as an argument a character string indicating the reference or type of the desired channel. It will then return the requested "Channel" object, based on the type or reference provided.

Examples:

selection. **channels**("MAIN")

→ Returns, when starting from the Meter, the channel(s) of this Meter that have a type "MAIN"

selection. **channels**("CNL_001")

→ "CNL_001" is not a type, so EMM will understand that it is a reference and it will return the channel whose reference is "CNL_001"

selection. **channels**

→ the ".channels" function has no arguments, so here it will return the complete list of all channels in the selected Meter.

○ Functions associated with the channels

As with any object in EMM, there are functions applicable to channels:

.default: will filter the list of channels to keep only the one(s) marked "default". For example, starting from the Meter, the syntax "selection.channels. **default**" will therefore return the default channel of the selected Meter.

.isdefault: will return a Boolean (true / false) indicating whether or not the active channel is the default channel for its Meter.

In addition, several functions that exist elsewhere are also applicable to channels.

.reference: will return the channel reference

.name: will return the channel name if it has been configured (optional on channels)

.parent: will return the Meter of which the active channel is a part

.properties: accesses the specific properties of the channel. These properties will usually contain the bearing information, time step, resource, unit... The syntax is the same as for accessing all EMM properties: .properties("//REF_OF_PROPERTY")

.data: will return the data profile associated with the channel. Note that this .data function can also be applied to the Meter itself (not just a channel). In this case, it will return the profile of the channel marked "default" for the Meter.

. creation: will return a block of information containing the date and user associated with the creation of this channel. This function must be followed by ". date" to retrieve the creation date, or ". user" to retrieve the creator of this channel. (Note, the object will then be the user, whose information can be retrieved in the same way as for a normal user, with ". name", ". mail"...)

. lastupdate: works exactly like ".creation", but retrieves the last update information rather than the creation information.

4. Collections of objects

A collection of objects is a list of objects of the same type. Syntactically, a collection of objects is defined by listing them in parentheses, separated by ";".

The keywords selection and #SITE, for example, are reserved terms, each of which represents a collection of objects made up of all the active entities and Meters in the channel selector, or of all the entities of type "SITE" in the database.

Examples*:*

(1; 2; 3) →Collection of numerical values 1, 2 and 3

(@C1; @C2; @C3) →Collection of Meters C1, C2 and C3

Selection.descendants→ All descendant entities/Meters in the selection.

#SITE →All entities of type "SITE" present in the database.

all →Collection of all entities/Meters present in the database.

(limited to entities/Meters on which the active user has read and/or write rights)

Collections have a major interest in the creation of worksheets in EMM since in a worksheet, there will be one line per element of the selected collection, thus allowing to structure the data that we want to display.

A collection of objects can also be formed by joining several sub-collections, for example by using several consecutive formulas that each produce a list of objects.

Example*:*

(@C1.descendants; @C2.descendants)

→ Collection of objects consisting of the list of descendants of Meter C1 and the list of descendants of Meter C2.

Note*:* A collection of collections will be a global collection containing all the elements of the starting collections without keeping the structure of their origin.

A collection can come from a multiple selection in the entity selector, but also from multiple properties in an entity.

Selection.properties("//PROPERTY")

→ If we have an entity in selection with several instances of "PROPERTY", we will retrieve with this formula a collection consisting of all the values of "PROPERTY" of the selected entity.

Obviously, if one has multiple properties in a multiple selection, one will get a large collection containing all instances of the property for each entity in the selection.

○ Sorting and filtering a collection

When starting with a collection of objects, it will often be useful to filter this starting collection to retrieve only certain objects that meet certain conditions. For this purpose, there are several functions described here.

.distinct

It is possible in a collection to have the same object appear several times. In this case, a table based on this collection will copy each line that has been requested several times.

If the aim is not to have multiple rows, it is possible to use the .distinct function, which, when applied to a collection of objects, has the effect of generating a new collection from which the duplicates of the initial collection have been removed.

Example*:*

(@C1; @C2; @C2) →Collection of entities C1, C2 and C2 (duplicate so).

(@C1; @C2; @C2). **distinct** → C1 and C2 entities (the 2nd C2 has been deleted by the .distinct)

Example*:*

(selection.children; selection.descendants). **distinct**

1. selection.children creates the list (as a collection) of the direct children of the selection.
2. selection.descendants creates a list (as a collection) of all descendants, including direct children.
3. The combination of the two lists will form a collection, in which all direct children should be listed twice and grandchildren and other descendants once.
4. The .distinct function will then remove all multiple occurrences of direct children and the final collection will no longer contain duplicates.

.where

In the case where one is only interested in a subset of the objects in the initial collection, it is possible to filter the data directly in the collection to generate a sub-collection meeting a given criterion.

The .where function thus makes it possible to retain only those objects which meet the specific criterion passed to it as an argument.

Example*:*

(1; 2; 3). **where**(item >= 2)

→ Will return the list of values 2 and 3.

→ EMM receives the list 1; 2; 3, and then applies the .where filter. To do this, it will evaluate the condition in parentheses sequentially for each object in the list, and remove from the list those that do not meet the condition.

1 does not satisfy the condition (1 < 2) and is therefore rejected.

The other values fulfil the condition, and are therefore retained in the final list.

The item keyword in the condition is a generic term that will successively take all values from the initial collection to evaluate the condition on that item.

This . where function can be applied to collections of objects of any type, but mainly entities, Meters, channels or properties.

Examples:

selection. **where**(item.name.startswith("B"))

→ Will return all entities/channels in the selection whose name starts with B

all. **where**(item.name.startswith("B")=False)

→ Will return all entities/channels whose name does not start with B

selection.properties("//PROP"). **where**(item.startswith("B"))

→ Will return a list of "PROP" properties of the selected entities/channels for which the property value starts with B.

Note: in this case, the collection obtained as a result of this syntax is indeed a collection of properties. If they are displayed in an array, EMM will understand that it must display the property values by default. From the collection, it is possible to use these properties to go back to the entities from which they originate, by using ".parent" on this list of properties.

.where successive

You can of course combine several conditions in the .where arguments. Indeed, as seen in the chapter on Booleans, you can combine conditions with AND and OR.

In the case of the AND, however, it might be more efficient to simply use the ".where" function twice in a row.

The following 2 formulations will perform the same search and produce the same results.

all.where(item.properties("//SURF").isnotnull and item. properties("//UNIT").isnontull)

all.where(item.properties("//SURF").isnotnull).where (item. properties ("//UNIT").isnontull)

The only difference will be in the amount of computation EMM has to do, and therefore in the performance in terms of computation time.

The first version will take the total list of all entities/Meters (all) and will evaluate the two conditions (linked by an and) on all entities/Meters. Assuming 1000 entities, this version will therefore evaluate 2000 conditions.

The second version, on the other hand, will evaluate the first condition on all the entities/Meters and will constitute an intermediate list of those fulfilling this first condition, then only evaluate the 2nd condition on this intermediate list. On the same basis of 1000 entities/Meters, the first ".where" will evaluate condition 1 1000 times and keep, for example, 500 entities/Meters fulfilling this condition 1. Once this first pass has been made, it only has to evaluate condition 2 on the remaining 500 entities/Meters. Result: 1500 conditions evaluated instead of 2000.

As some conditions can be quite complex and therefore power and computationally intensive, it can be useful to think through your filters from the outset to avoid an unnecessary explosion in work time.

.where on multiple blocks

In the case of multiple (or historicised) blocks of properties, the ".where" function allows the blocks to be filtered according to a sub-property. For example, one could retrieve all the contact blocks of an entity/Meter, and among these contact blocks, keep only those which have a certain characteristic.

Example*:*

Let's imagine a multiple "EQUIPMENT" block. If we have two pieces of equipment for the same site, we will have the entire "EQUIPMENT" property block in duplicate. It is then possible, for example, to specifically find the serial number (in a property) of one of the pieces of equipment whose name is known.

item.properties("//N_SERIAL_NUMBER") →will return the 2 serial numbers in a row.

Item.properties("//EQUIPMENT").properties("//N_DE_SERIE")

→ Here we get the 2 serial numbers, exactly as before. The only difference is that we force the path of the property search as follows: we first search for the "EQUIPMENT" blocks, and within that, we'll search for the "N_DE_SERIE" property. (For the moment, no real difference)

The idea here is that you can use item.properties("//EQUIPMENT") as an object, and perform filters on it, before continuing down into its properties.

```
item.properties("//EQUIPMENT")
    .where(item.properties("//EQUIPMENT _NAME")="Generator")
    .properties("//N_DE_SERIE")
```

→ We take the different "EQUIPMENT" blocks. In these blocks, we filter via the .where function and keep only the blocks that contain the "EQUIPMENT_NAME" property with the value "Generator". Finally, from these specific blocks, the serial number is retrieved in the "SERIAL_NAME" property.

This way of doing things treats the property blocks as objects, allowing the ".where" function to be used to filter as required.

Note*:* Property blocks can be considered as objects but they do not have a place in the tree structure. Thus, if you use the ".parent" function on a property, you will not go back to the property block containing this property, but directly to the entity, Meter, channel or user containing the block (no way to go back to the property blocks from a property)

.orderby

When building a collection, it may be useful to sort the values in a certain order, which may not be present in the data initially.

Example*:* List all the meters of a site in alphabetical order, or in chronological order of creation, or by total decreasing consumption over the period of time considered...

Syntax*:*

selection. **orderby**(Criterion "desc")

- Criterion: The criterion according to which the sorting will be carried out
- "desc": indicates the sorting direction. "desc" = descending, "asc" = ascending ("asc" is the default value, so it is not necessary to write it)

It is also possible to do several cascading sorts, on several values.

Example*:*

selection. **orderby**(item.properties("//GROUP"); item.reference "desc")

→ Will return the entities/Meters in the selection sorted by group (here a "GROUP" property), in ascending order (since nothing is specified, EMM defaults to ascending order), and within each group they will then be sorted alphabetically by their reference, in descending order.

Note*:* it might be tempting to use the .orderby function twice in a row to do this kind of sorting, each time with only one argument) but in this case, the 2nd sort will overwrite the first. The correct method is therefore to use the .orderby function only once, but with several arguments.

Advanced example*:*

selection. **orderby**(item.data.sum "desc")

→ Will return the list of Meters in the selection of the selection, sorted in descending order of total consumption over the active time context.

.top / .flop

When you have a collection of items, you may sometimes only need the first 5 or the last 3 of this list, excluding all others. For example, you can search for the meter that consumes the most in a list, or the 5 customers who paid the least for their energy last month.

.top(n) will retrieve only the first n objects in a collection

.flop(n) will retrieve the last n objects in the collection

These two functions are often used after an .orderby, to make sure that the list is sorted according to a useful criterion and that you then take the first/last n.

These two functions can also be used together, for example to retrieve the 4th object in a list and no others.

selection. **top**(4). **flop**(1)

→ will create a sub-list of objects from the selection, containing only the first 4 objects, and then take only the last one from this sub-list

→ 4° of the original list.

Note: When performing a top/flop search of a list of constructs using the .ancestors function, the ancestors are searched from the bottom to the top of the tree. The last ancestor in the list will therefore be the one located at the very top of the hierarchical list, while the first will be the direct parent.

Example:

selection.orderby(item.data.sum desc). **top**(5)

→ As seen earlier, this code will return the Meters in the selection, sorted in order of decreasing total consumption over the time period defined by the active time context, and will keep only the top 5 in the list. In other words, this code returns the top 5 largest consumers over the time period considered.

Optimisation of filters

Some filters on a collection may take more or less time depending on how they are coded. It is strongly advised to think carefully about the sequence of filters to avoid multiplying the number of useless DB accesses, and thus save a significant amount of computing time.

We have already talked about making several successive .where filters rather than making a single .where with several conditions linked by an "AND". This reduces the number of tests to be carried out, and therefore speeds up the calculation.

Another recommended optimization is to avoid filtering Meters based on their properties. It is more efficient to go down to the properties, filter directly on them, and then go up to the Meters afterwards.

Example: If I want to retrieve the list of sites whose postcode is 1348, the first syntax that comes to mind would be the following:

```
all.where(item.properties("//ENT_ADD_ZIP")=1348)
```

This syntax will therefore make a first DB access to retrieve the list of entities/Meters present in the database. Then, in a sequential way, EMM will carry out, for each of the entities/Meters, a new DB request to recover the "ENT_ADD_ZIP" property and carry out the test on the value 1348. This represents a large number of DB accesses (one per DB entity), which can be reduced with a relatively simple modification.

Firstly, we can start from "#SITE" instead of "all". This reduces the number of objects studied, but it limits us to entities of type "SITE", to the exclusion of other types of entities, potentially useful. To be used in an enlightened way.

Another solution, which will work in all situations, is to use the following formula:

```
#SITE.properties("//ENT_ADD_ZIP"). where(item.value =1348). parent
```

This formula therefore starts from the list of sites, and retrieves (in a single DB access) all the "ENT_ADD_ZIP" properties, then filters on the values of these properties (already available in memory after the retrieval of the properties, so no need for a DB access per entity). Once the properties have been filtered, a final DB access is then made to massively retrieve the entities from which the remaining properties come (. parent), and in only two DB accesses, the desired list is obtained.

Finally, when we want to check that a given value belongs to a list of values, we might be tempted to code this as follows:

```
#SITE.properties("//ENT_ADD_ZIP").where(
    item.value = "1348" OR
```

```
item.value = "6043" OR
item.value = "4000" OR
item.value = "6210" OR
item.value = "5310" )
```

However, this requires EMM to perform 5 tests on each property value. It will be more efficient to search for the property value in the concatenated list of allowed values (separated by a special character, to avoid triggering on false values that straddle two allowed values). The following code is then obtained:

```
#SITE.properties("//ENT_ADD_ZIP")
    .where( ("1348|6043|4000|6210|5310").contains(item.value) )
```

In doing so, only one test per property value is performed to check that it belongs to the allowed list. This optimization may not seem important, but in the case of longer allowed lists, or slightly more complex tests, it quickly becomes extremely cost-effective in terms of computation time.

- Functions associated with collections

.count

The .count function is used to count the number of objects in a collection.

selection. **count** → will return the number of entities/Meters in the selection.

selection.data. **count**

→ will return the number of date/value pairs in the default channel data profiles of different Meters in the selection.

.min / .max / .sum / .avg / .std / .percentile

When given a collection of numerical values, EMM is able to perform a range of mathematical operations.

Below, the word "collection" is used to represent a collection of objects. It is not really a keyword used in EMM syntax. It represents a list of numerical values, a data profile... or any other relevant collection, which could be represented by a more complex real EMM syntax.

.max .min	Maximum Minimum	Returns the maximum or minimum of the values in the collection, in numeric (double) form collection. **max** → returns the maximum value of the collection collection. **min** → returns the minimum value of the collection
.sum	Sum	Returns the sum of the set of values in numerical form.

.avg	Average, arithmetic mean	Returns the average of these values in numerical form (double) collection. **avg** → returns the average value of the collection
.std	Standard deviation	Returns the standard deviation of the series of values in numerical form.
.percentile	Percentile	Returns the value of a collection corresponding to its Xth percentile, X being passed as a parameter between 0 and 1. collection. **percentile**(0.95) → will return the value of the collection corresponding to its 95° percentile. If the exact value is between two values in the series, EMM will return a linear interpolation between the nearby values. **Example:** (40; 58.2; 58.8; 60). **percentile**(0.4) The 0.4 percentile is between 58.2 and 58.8 - No. of intervals = No. of values - 1 → 3 - Position = 0.4 x 3 = 1.2 NB: The 1st value is the value n°0 The percentile is therefore 20% between values 1 and 2, i.e. between 58.2 and 58.8 → 58.2 + (58.8-58.2) x 0.2 = 58.2 + 0.12 = 58.32 Note: any NaN values present in the list are not ignored for the calculation.

.groupby

When it is necessary to group the objects of a collection according to certain criteria, one can use the .groupby function. This function will receive a collection of objects and generate a new collection of objects structured according to a grouping criteria.

Syntax *:*

collection. **groupby**(key)

- collection: The initial collection. This can be a selection of channels or other objects.
- key: indicates the value on which the grouping will be carried out.

This "key" value can take many different forms. For example, entities can be grouped by their initial, or by the value of one of their properties. You can also group rows in a table according to the value in one of their columns. Users can also be grouped according to their geographical area, or data according to their origin.

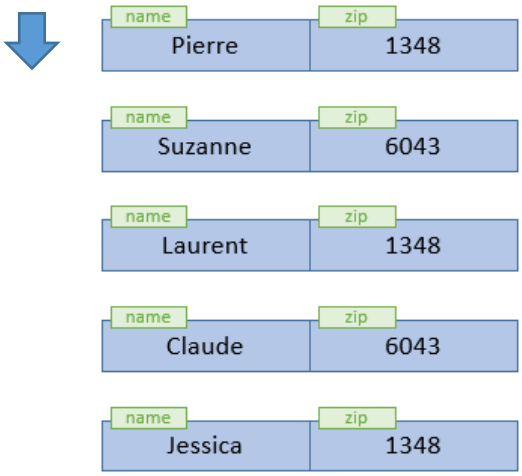
The result of this function will be a new collection, made up of key/elements pairs, containing, for each key (each possible value of the key passed as an argument) a collection of elements.

Each object in the resulting collection being a key/elements pair, it can be accessed using the following syntax:

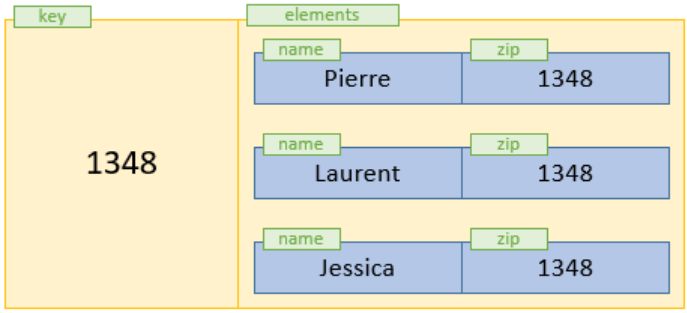
item. **key** → gives the value of the key for each group.

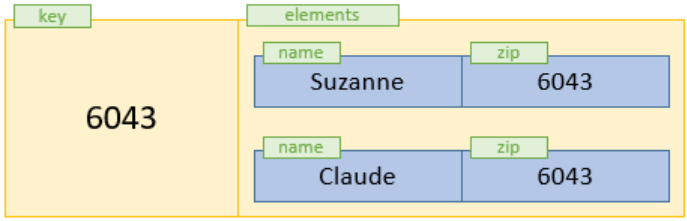
item. **elements** → represents the sub-collection of items included in the group.

Example*:* In the example shown below, the starting collection contains users and their postcodes. Grouping via . **groupby**(item.properties("//ZIP")) will generate a new collection, containing key/elements pairs. Each key being a possible value taken by the ZIP property, and each associated elements being a sub-collection containing all the elements of the starting collection meeting the grouping condition (here, the zip corresponding to the key)



user.**groupby**(item.properties("//ZIP"))





You can also perform several groupings by putting them in the same .groupby function

user. **groupby**(item.properties("//ZIP"); item.properties("//FUNCTION"))

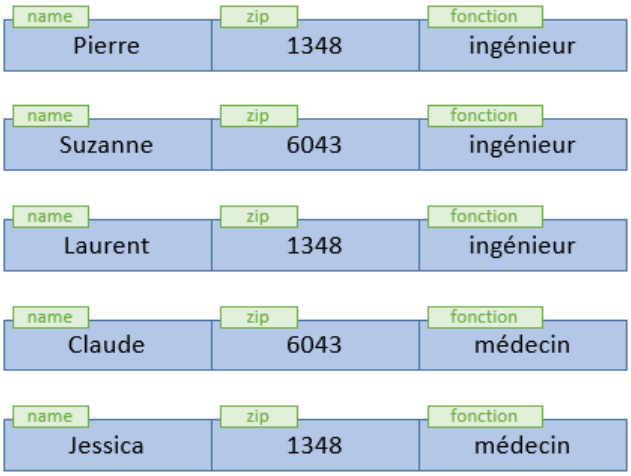
→ Will yield a collection of key/elements pairs, where users will be grouped according to their postcode **and** function.

The information in the new collection will always be accessed via .key to obtain the key for the group in question, and .elements to retrieve the data profile associated with the key.

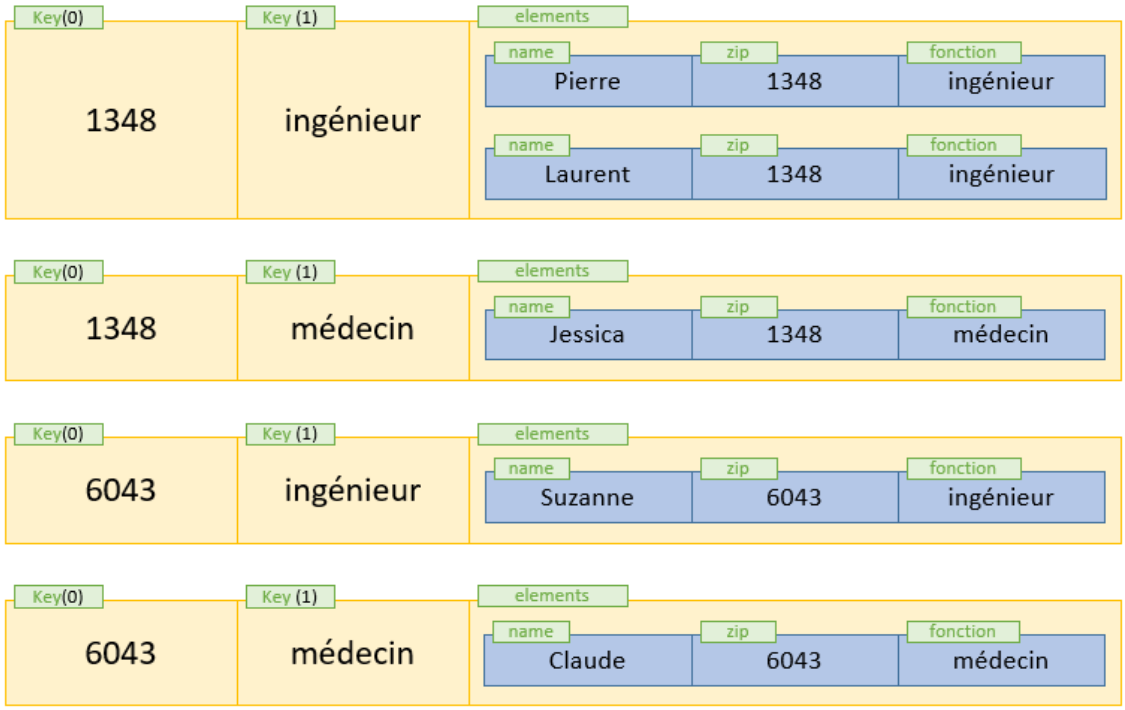
If there are two simultaneous groupings, the key will be the concatenation of the two subkeys key(0) and key(1), each containing the key associated with one of the groupings.

Example*:* If we use the same type of collection as in the previous example, but add information (in this case, the user's function), we can now group on the basis of both criteria simultaneously.

We will thus have 2 keys, the key(0) corresponding to the first grouping and the key(1) corresponding to the second grouping criterion. The key value then becomes the concatenation of the two subkeys.



user.**groupby**(item.properties("//ZIP"); item.properties("//FONCTION"))



.foreach

The .foreach function allows, from a given collection, to perform a defined operation for each element of the collection.

The function therefore takes a collection of objects as its source, and for each object in this collection, sequentially, EMM will perform the operation it receives as an argument. The keyword item can again be used here as a generic term sequentially replacing each object in the operation.

Examples*:*

selection. **foreach**(item.name)

→ Creates the list of entity/Meter names in the active selection

The system will therefore take the active selection, and for each of its elements, evaluate what is asked of it in the .foreach argument (here, retrieve its name).

Warning: in the case of the above example, the result no longer contains a series of entities/Meters, but a series of texts, which are the names of the entities/Meters of the initial selection.

Note*:* the .foreach applies its request sequentially to all the elements submitted to it in the selection, even if the same element appears several times (it will then perform the request several times on the same element)

Another interest of the .foreach is in the context of tasks to be carried out (such as sending an e-mail, generating a report...). In this case, the .foreach will allow to launch the task in a recursive way on all the

selected elements. For example, it can be used to generate a series of reports using the same template, but applied successively to each of the sites in the active selection. This point is detailed in the chapter on the EMM task manager.

Advanced example*:*

```
selection. foreach(SENDMAIL("dest@dapesco.com"; null ; "Confirmation mail"; "This mail is the confirmation that the entity " + item.name + " is present in the DB"))
```

→ Upon application of this code, EMM will send for each object in the selection, an email to the recipient "dest@dapesco.com", having as title "Confirmation email" and as content "This email is the confirmation that the entity NAMEFUL is present in the DB". This code can of course be adapted to send other things, such as PDF reports, or alarms.

.extend - Extension of a collection

It often happens that, starting from a collection of entities, we want to retrieve these entities themselves but also all their descendants for example. The natural syntax is then

```
Selection.foreach((item; item.descendants))
```

Unfortunately, this syntax performing a .foreach will generate a large number of calls to the database, which can dramatically increase the overall calculation time, especially if this kind of syntax is used several times in a row to retrieve, for example, a selection of entities and their descendants, as well as all their "ENTITY_METER" links and all the descendants of the latter... this syntax can be used, for example, in user rights to create perimeters based on a site.

To avoid the explosion of calculation time due to this syntax, a function has been created which allows the current selection to be extended by adding other entities directly.

```
selection. extend(item.descendants)
```

This syntax will have exactly the same result as the other one, given above. The advantage is that this function is optimised to minimise the number of accesses to the database, drastically speeding up the calculations.

The ".extend" function therefore applies to a collection of objects, and receives as an argument a formula which will return another collection of objects. The ".extend" function will ultimately return the initial collection, extended to include the result of the formula passed as an argument.

Functionally, the expression selection.**extend**(item.descendants) is in every way similar to the expression Selection.foreach(item; item.descendants). Both statements will return exactly the same collection of objects in the end. The only difference is that the former will go much faster than the latter.

5. Data profiles

A data profile is a special kind of collection of date/value pairs that allows the evolution of a data item to be visualised over time.

○ Channel profiles

In EMM, a channel can contain two separate but related profiles.

- RawData: this profile will contain the raw data injected into EMM as is, without transformation. This can be indexes or consumptions, from dataloggers or manual insertion, or the results of formulas creating virtual profiles (details later, in the section talking about channel formulas).

- Data: This profile is the cleaned, processed and standardised data profile for use in EMM. If the RawData were in index, the Data profile is made up of its successive differences to give a profile of consumptions, and these consumptions are then multiplied by a possible pulse weight (also called "pulse weight")

○ Recovery of channel profiles

.data / .rawdata

From a channel, its data profiles can be accessed via .data or .rawdata functions, which will produce the associated profiles. These profiles are therefore collections of objects, each of these objects being a date/value pair representing a consumption measurement at a given time.

Examples:

@C1.channels("MAIN"). **data**

→ Will return the Data profile associated with the "MAIN" channel of the C1 Meter.

selection.channels("MAIN"). **rawdata**

→ Will return the RawData profile associated with the "MAIN" channel of the Meter present in the selection

It is also possible to retrieve a data profile from a Meter, without specifying a specific channel. In this case, the data profile of the channel marked as "default" will be returned by the formula.

@C1.**data**

→ Will return the Data profile associated with the default channel of the C1 Meter.

By default, if a data profile is called (via .data) without specifying a time context, the context applied will be the context currently selected in the date selector (top left).

If we are now only interested in data within a certain time period, we can pass context start and end dates as arguments to the .data function.

Examples*:*

selection.data(**dateeval(2015; 03; 01)**; **now**)

→Creates a profile from March 1, 2015 to the present

selection.data(**from**; **to**)

→Consider the datas from the current context (no difference with selection.data)

selection.data(**now.synchro(1; "month")**; **now**)

→Create a profile from the first day of this month to the present

- Multiple selection profile(s)

If there is a selection of several channels, using the profile retrieval functions will retrieve a single global profile representing the sum of the profiles of all the selected channels.

selection.data

→If the selection contains multiple channels, this code will return a single data set, containing a date/value-by-date pair of the initial sets, summing the consumption values of the corresponding dates

If we want to avoid this and retrieve the separate profiles of each Meter in the selection, we should use the .foreach function seen earlier.

selection.foreach(item.data)

→Will create a list of date/value pairs for each Meter in the selection. This formula will keep the date/value pairs for each Meter quite distinct and list them as different objects.

- Generate a profile from a historical property

When a history property is called, and it varies over the active time context, EMM returns a list of values, each with its own from and to. In other words, EMM returns a data profile, consisting of a date-value pair for each value taken by the historised property.

To constitute a real regular profile, it will then suffice to apply an aggregation, as for example with the following formula: item.properties("//SURFACE").agg(1; "month"; "constant")

This formula will then return a monthly profile, containing for each month the value of the "SURFACE" property

Note: It is also possible to use the invoices of a site to create a profile. Indeed, the invoice properties are considered as historical (with the from/to dates of the invoice as validity dates). These historicised properties can then be used to build a profile exactly as described above

selection. invoices("//CONSO")

→ will therefore return a profile containing the various "CONSO" values of the invoices, with the validity dates of the associated invoices as dates.

- Accessing data (date/value pairs)

A data profile is therefore a collection of date/value objects containing various information. When you put a data profile in the "selection" box of a worksheet, EMM will interpret the profile as a collection, and it will therefore display one row in the worksheet for each date/value pair in the selected profile.

On these objects, it is then possible to retrieve information and make modifications via the following functions...

.value / .date / .from / .to

These functions allow you to directly retrieve information stored in date/value pairs.

item. **value→returns** the value of the item (e.g. consumption)

item. **from→** returns the start date of the measurement range

item. **to** → returns the end date of the measurement range

item. **date**

→ returns the date associated with the object. In the case of a profile, this will be the end date of the measurement period.

These functions can be applied directly to a date/value pair alone or directly to a profile (collection of these pairs). If applied to a single measure, it will return a single value, either numeric or a date. If applied to a profile, it will give a list of corresponding values for each of the measures in the profile.

Feed data back to the channel or Meter

When you are on a data item (date/value pair belonging to a profile), it is possible, via the syntax, to go back to find the channel or even the Meter from which the data comes. To do this, the following syntaxes can be used:

@C1.channels("MAIN").data. **parent**. **parent** → Will return the C1 Meter.

@C1.channels("MAIN").data. **parent**. **channels** → Will return the "MAIN" channel of cptr C1.

Basically, the principle is to go back once to the parent, which happens to be the data profile (an object that is not very usable as such in the syntax but useful for the computer code underlying EMM), then go back to

- to the .parent of the profile to get the Meter. This syntax is inherited from older versions of the spell, to ensure some backward compatibility with older formulas.

- to the profile's .channels, which will specifically return the channel as an active object

Note*: Although the* .parent function used here is similar to that used for channels, it is not exactly the same. This is because in an object-oriented language, a function is always defined in terms of the types of objects

to which it applies. So the .parent function for channels and the .parent function for date/value pairs are not exactly the same function, they just have the same name.

In our case, the two .parent functions are used in the same way, but this same subtlety of definition has implications for other functions.

The .ancestors function, for example, is defined to work exclusively from channels. So, starting from a date/value pair, you cannot use .ancestors directly to retrieve channels from the ascending tree, even though the existence of the .parent function on date/value pairs might have suggested this.

If you want to go back from a date/value pair and retrieve the ascending tree, you will have to use .parent.parent. to go back to the channel, and only then can you use .ancestors.

Example*:*

```
item.parent.parent.ancestors.flop(1)
```

→ From a date/value pair, will return the channel all the way up to the root of the tree where the starting data is located.

- Functions associated with data profiles

Once a profile has been retrieved, a range of transformations can be applied to it, such as aggregating its values, distributing them according to an SLP, or combining several profiles together.

.agg

It may happen that one needs the values of a profile not by tracking the actual data (which may be received per 10 minutes for example) but rather the data per hour, day or other. (e.g. the one-off consumption is useful, but for a bill check, the overall consumption of the month is sufficient information)

The .agg aggregation function allows an existing profile to be transformed into another profile containing the same data, aggregated by time slices of a defined size.

This function must be given a data profile, i.e. a collection of date/value pairs, otherwise it will be impossible to aggregate the data by time slice.

Syntax *:*

```
selection.data. agg(nbr_intervals; "type_interval"; "method")
```

- nbr_intervals: indicates the number of intervals over which each aggregation will take place.
- "type_interval": indicates the type of interval. Text format (i.e. between " "), choose from year, month, day, hour or minute
- "method" indicates the method by which the aggregation will be done (text format), to be chosen among :

avg	The average value over the interval (useful for a temperature profile for example)

sum	The sum of the values in the interval (useful for consumption for example)
max	The maximum value reached over the defined aggregation interval
min	The minimum value reached on the defined aggregation interval
count	Returns the number of data received over the time interval.
std	Calculates the standard deviation of the data present in the time interval under consideration (std=standard deviation)
sumprop	Performs a proportional sum on the interval (explained below)
constant	Takes the end value of each interval
persistent	Resumes the starting value of each interval

Note*:*

If you have several profiles at the same time (with a selection.foreach(item.data) for example), you can also use .agg to aggregate the values by time slice. They don't have to be in one profile with dates in order.

The aggregation will retrieve the data based on their dates, gather them by time slice and aggregate them using the aggregation method passed as an argument.

selection.foreach(item.data. **agg**(1; "hour"; "sum"))

→ will return a profile of summed data per hour; whether originally measured over 10 or 15 minute intervals will have no impact.

Adaptive time step aggregation

In addition to the possibility of aggregating data in a DataSet with an auto-adaptive time step, it is also possible to perform an auto-adaptive time step aggregation directly by formulas in the worksheets. The time step is then chosen by EMM according to the current observation period.

- If the duration of the time selection exceeds 500 days: 1 year
- If the duration of the time selection exceeds 300 : 1 month
- If the duration of the time selection exceeds 10 days: 1 day
- If the duration of the time selection exceeds 1 day: 1 hour
- Otherwise: No aggregation

To do this, the .agg function can accept, in addition to its usual set of 3 parameters (No. of time steps, Type of time steps, Aggregation method), another formulation that takes only one parameter: the aggregation method.

Classic formulation: Selection.data.agg(1; "month"; "sum")

Formulation for adaptive time step: Selection.data.agg("sum")

Limitations

- As the time step is dynamically defined according to the observation period, it is strongly discouraged to use this new functionality in channel formulas for example. Indeed, depending on the size of the pre-calculated time slot, the data will vary

- In HTML reports, following the same logic, it may be inadvisable to use this kind of functionality to avoid having reports showing highly variable consumption depending on the time context used.

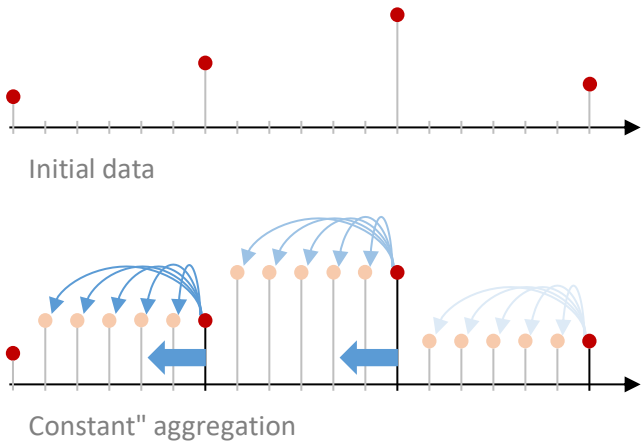
Constant or Persistent

Constant and persistent aggregations are very similar to each other and can be confusing. These two aggregation methods are mainly used to copy data from a large time step to several dates in a smaller time step.

There are several possible scenarios:

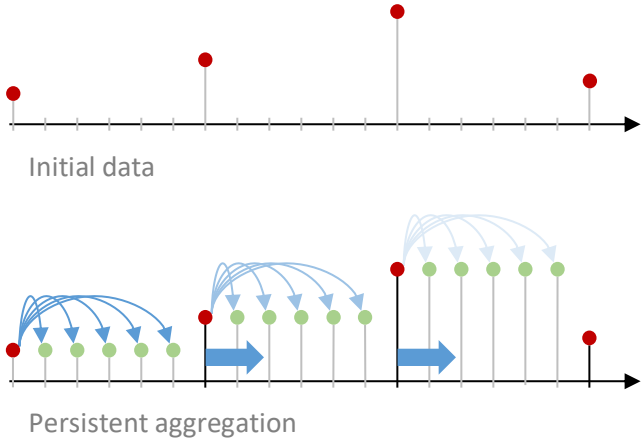
- A meter is read monthly and we want to obtain an hourly profile with a constant distribution of consumption (distribution of consumption in proportion to time)
- A historical property must be transformed into a numerical profile to perform calculations (e.g. the surface area of a building to calculate the specific consumption)
- An Xtab contains tariff values that should be transformed into a profile for invoice control.
- …

For the first cases (consumption data or historical property values), EMM considers that the numerical value is specifically associated with the end date of its validity interval. (This is consistent with consumption data, where the consumption of a time interval is known at the moment when the time interval ends. We rarely know the consumption of a period in advance...). The "constant" aggregation will then retrieve the **end** value of the validity interval of a consumption or a property and will copy it at each time step in this validity interval.



In the case of a tariff Xtab, the situation is the opposite. In fact, the value of a tariff is known at the beginning of its validity period and remains valid until a new value overwrites the previous one. Therefore, one should not stretch an end value to the beginning of an interval, but on the contrary, stretch a beginning value to

the next time steps. The "persistent" aggregation will therefore retrieve the **start** value of the validity



interval of a tariff for example, and copy it at each time step in this validity interval.

These two aggregation methods therefore do the same thing, but in opposite directions on the timeline.

SUM or SUMPROP

The "SUMPROP" aggregation method will perform a proportional sum of a profile over a chosen interval. The principle is not simply to add up the values present in the interval, but also to go and find the first values around this interval in order to recover the part of consumption that they cover in this interval and to count it in proportion to the duration of the period covered.

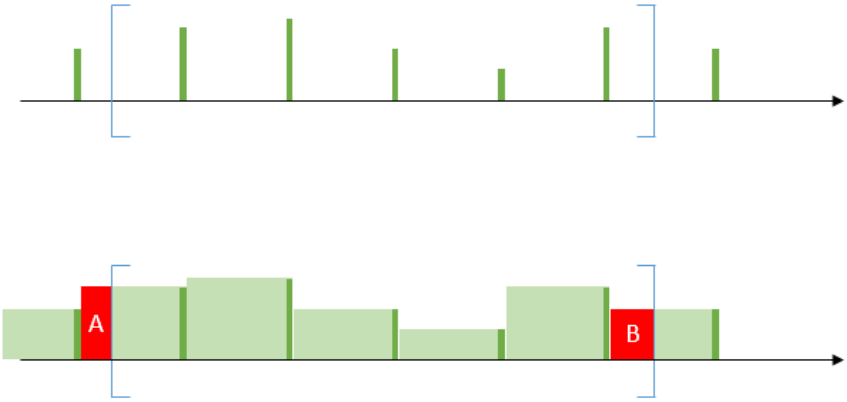
Example*:*

In the diagram below, we see the time interval (blue brackets) and the consumption measurements (green lines). A simple sum will take only the consumption values between the interval boundaries and add them up. However, each consumption measurement represents the actual consumption of the time interval preceding it, since the last measurement was made.

The first conso of the interval chosen in the scheme will therefore cover more than the chosen period, counting the A zone in excess of what is required.

Similarly, as the last measurement of the interval comes before the end of the interval, area B will not be counted, although it should.

The SUMPROP aggregation method will therefore fetch the first measurements before and after the chosen interval, in order to compensate for these errors in proportion to the time covered.



Numerical example:

Let's imagine a manual index reading giving the following values:

28/12/2015 00:00	800
4/1/2016 00:00	600
28/1/2016 00:00	1000
3/2/2016 00:00	750

An aggregation of these data by month should reconstitute the consumption for each month, from the first of the month to the first of the following month.

In this case, an aggregation by SUM for January will give 600 + 1000 = 1600

An aggregation by SUMPROP will make the following calculation:

- The consumption of 28/1 is entirely included in the month of January -> it is simply counted as 1000.
- The consumption of 4/1 covers part of January, but also part of the previous December. We will therefore only count the part of this consumption that relates to January, in proportion to the number of days covered.
  Here, the period from 28/12 to 4/1 covers 7 days (4/1 is excluded since we stop on 4/1 at 00:00), including 3 days in January. We therefore keep 600*3/7
- In the same way, the consumption of 3/2 covers part of January as well. We have a period of 6 days, 4 of which are in January -> We will keep 750*4/6

Consumption balance for January: 600*3/7 + 1000 + 750*4/6 = 1757.14

Note: In the case of consumption measured in indexes, 2 consecutive values are needed to reconstitute a consumption (by difference of consecutive indexes). The SUMPROP method will therefore require 2 external values on each side of the interval instead of one.

It is therefore imperative to have a value (A) before the 1st period for which a value is required and a value (B) after the last period for which a value is required (2 are required in the case of data encoded in indexes, since consumption is recovered by the difference between 2 consecutive indexes)

The problem with SUMPROP aggregation is that it is dependent on the time context in which one is working and can give rise to undesirable side effects.

An alternative to using SUMPROP is to use the .slp

.slp

The .slp function allows you to create another data profile with a shorter time step from one data profile. The idea is to be able to split a coarse measured value into more precise data, based on an existing standard profile (Synthetic Load Profile)

For example, we could transform a profile taken manually once a month into a daily profile. To do this, we will need to have injected a special channel into EMM, containing the standard profile that we will use for our distribution.

This profile can be constant to obtain an equitable distribution of consumption over the measurement period, or it can have any form depending on the desired distribution (e.g. 2X consumption during opening hours and X during closing hours. The SLP will ensure a correct distribution of the actual consumption according to the number of opening/closing hours).

The .slp function will then simply take this data profile as an argument.

Example:

@C1.data. **slp**(@SLP_S11_D. data)

→A standard value model profile (here, @SLP_S11_D) is used as an argument to the . slp function and the function will then produce a realistic consumption profile, based on the model profile, but with the correct total consumption for the period.

(Note: it is important to choose the model profile used to best fit the reality)

(Note: the data listed here for the Meter is that of its default channel since there are no additional details in the syntax)

Advanced remark*:*

Once this .slp function is used, the active object is the template profile. This can be a problem if, afterwards, we have to go and find a property of the active Meter, or its parent...

→The .slp function can be given a second argument, (optional) which will tell EMM which Meter should be the active object at the end of the function.

@C1.data. **slp**(@SLP_S11_D. data; @C1)

→So will give exactly the same as without the second argument, except that the active object at the end of the function is now the C1 Meter and no longer the SLP Meter containing the standard profile.

This .slp function can be applied to any data profile, whether it comes from a channel (.data), a table or an XTab (see below) or a series of invoices (see below). It is this .slp function that will allow us, for example, to distribute consumption stored in monthly invoices and to reconstitute consistent consumption profiles over the invoiced periods.

Note: Some of the allocation profiles are contractual and used by the energy suppliers themselves to allocate their monthly consumptions for billing purposes. It is therefore useful to know the SLP profiles used by the suppliers in their calculations in order to be able to reconstitute the invoices on the same basis.

.shift

The .shift function applies to a profile (collection of dates/values) and returns the same profile of values shifted in time by the period specified in the parameters.

@C1.data. **shift**(-1; "Month")

→ Will return the same profile as @C1.data but with dates shifted by one month, from the past to the present (negative value in the shift).

The interest of this function is to be able to display series of values from different periods on the same time graph (e.g. comparison between this year's consumption and last year's consumption).

Indeed, without this, the system would try to display the 2 consumptions by taking in X their real dates, which would lead to a very extended graph in X, with last year's data on the left and this year's data on the right of the graph.

.derivative

The .derivative function applies to a profile (collection of dates/values) and returns a new profile consisting of the differences of successive values of the initial profile (finite derivative).

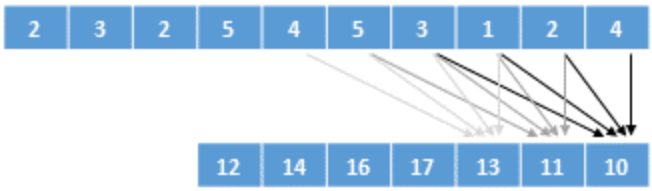
This is very useful when you have a profile of meter readings and you want to build a consumption profile. In this case, the consumption of a period corresponds to the difference between the index at the end of this period and the previous index (end of the previous period). The .derivative function makes these differences and constitutes the final consumption profile.

.queue

Applied to a data profile, this function provides a new profile where each value is the sum of the last n values of the original profile.

@C1.data. **queue**(4)

→ Returns a series of values, where each value is the sum of the last 4 values of the channel data profile



Example of use*:* One might want a table in which one is interested not only in the current value, but also in the average of the last 7 days.

(@C1.data. **queue**(7))/7 → Will return the list of averages of the last 7 data.

Caution: Since this function sums up the last "n" values, disturbing side effects may appear at the beginning of the series of values. Indeed, if we start from a truncated profile starting on January 1st and we make a ".queue" of 4 days, the first 3 values of the generated profile will be based on 1, 2 then 3 values and not the 4 expected ones. This will result in an incomplete profile start. To solve this problem, it is sometimes possible to trick the user into making the initial profile start a little earlier than expected (in this case, 3 days), then apply the ".queue" and make sure to keep only the data from the desired time range (with a .where filtering out of context data for example)

The initial situation would then be as follows:

Profile starting on

1/1	1
2/1	1
3/1	1
4/1	1
5/1	1
6/1	1
7/1	1
8/1	1

Profile

1/1	1
2/1	2
3/1	3
4/1	4
5/1	4
6/1	4
7/1	4
8/1	4

The corrected situation could then look like this:

3-day early profile

29/12	1
30/12	1
31/12	1
1/1	1
2/1	1
3/1	1
4/1	1
5/1	1

Profile .queue(4)

29/12	1
30/12	2
31/12	3
1/1	4
2/1	4
3/1	4
4/1	4
5/1	4

Truncated .tail(4) profile

1/1	4
2/1	4
3/1	4
4/1	4
5/1	4

This brings us back to the desired profile.

Automatic unit conversion

Some values are stored in profiles with their units (e.g. consumption measurements can be stored in Wh or kWh). This unit is defined in the measurement properties, usually in the properties of the relevant channel.

It is then possible to do an automatic unit conversion of these values via EMM, noting the new unit in braces.

selection.data→returns a profile in the unit in which the measurements are encoded.

selection.data **{kWh}→will return** the same data, but converted to kWh.

This conversion will obviously only be done if there is a possible link between the units. It is impossible to spontaneously convert degrees into square metres...

Note that the change of unit applies to the entire profile object, and not to the numerical values that would be recovered after a .value. Indeed, for certain conversions (kW to kWh for example), it is necessary to know the time step between the data in order to apply the correct transformation coefficients.

Filtering profiles (.where)

As profiles are collections of objects (date/value pairs), it is possible to filter them using the .where function based on a given criterion.

Examples:

selection.data.where(item.value > 0)

→ Only measurements with a positive value are recovered

selection.data.where(item.date > now.add(-1; "month")

→ Only data that are less than one month old are recovered.

.min / .max

Applied to a data profile (collection of date/value pairs), these functions will search for the smallest/largest values in the data (the search is therefore done on the "value" field of the data), and then return the data(s) with the smallest/largest value.

If the min/max is reached only once, only one data item will be returned as a result, but if there is a tie, the function will return all data items sharing that min/max value.

Examples:

selection.data. **max** →return the data(s) with the largest value

selection.data. **max**.flop(1)

→ will return the last instance of the data sharing the largest value

If the aim is only to retrieve the maximum value reached, without being interested in the number of times it is reached or the dates when it happened, the following syntax could be used, for example selection.data.value.max

In fact, in this case, the profile is first converted into a collection of numerical values, and then the .max function will find the largest value, which is unique this time, since we are in the case of the .max function applied to a collection of numbers.

.sum / .avg / .std / .percentile

These mathematical and statistical functions, when applied to a data profile, will convert the data profile into a collection of numbers (the data values in the profile) and perform their calculations on this collection. These formulas will in effect return a numerical value, reflecting the sum of the data, its mean, its standard deviation, or the requested percentile.

Examples:

selection.data. **sum**

→ Returns the sum of the selection data (for the active context)

selection.data. **avg**

→Returns the average of the selection data (for the active context)

selection.data. **std**

→Returns the standard deviation of the sample of data values

selection.data. **percentile**(0.95)

→Returns the value of the data sample corresponding to its 95° percentile.

Other functions

All other functions applicable to collections remain applicable to data profiles. For example, you can use the .orderby function to sort the data according to a given order (decreasing values for example). You could also count the number of data with a .count or keep only the last 5 data with a .flop(5)

Examples:

selection.data. **orderby(item.value)**

→Returns the data from the selection, but sorted by increasing values.

selection.data. **count**

→Returns the number of data items present on the selection, in the active time context.

selection.data. **flop(10)**

→Returns the last 10 values present on the selection profile.

selection.data.agg(1; "month"; "sum"). **orderby**(item.value). **flop**(3)

→Renders the aggregated data of the 3 months with the highest consumption in the active time context, for active selection.

- Combining profiles

As profiles are objects in EMM, they can be combined in various ways. For example, they can be simply added together:

@C1.data + @C2.data

→ will return a profile for which each value is the sum of the values of the two starting profiles.

In the same way, profiles can also be subtracted from each other or multiplied.

Warning: the syntax (@C1.data; @C2.data).sum will not work. Indeed, this syntax will first build a large collection of all the date-value pairs of the profiles of each Meter and will not know how to sum them up afterwards.

If you want to sum a variable number of channels, you will have to be cunning. For example, the following syntax can be used:

(@C1; @C2) .data

→ In this case, EMM will build up a collection of channels, then take all the profiles associated with those channels, and by default, sum them together.

This trick will only work for channels to be added together, however, as this is the default operation EMM does when it has multiple profiles.

○ Conditional profiles

Profiles can also be combined on a conditional basis. For example, a true/false profile can be created, which will be represented by a 0/1 profile based on a condition, such as a comparison between two channels.

Example*:*

(@C1.data > 0)

→Return a profile, which will consist of 1 when the data profile of the C1 Meter (default channel since no further details in the syntax) has a positive value, and 0 when it is negative.

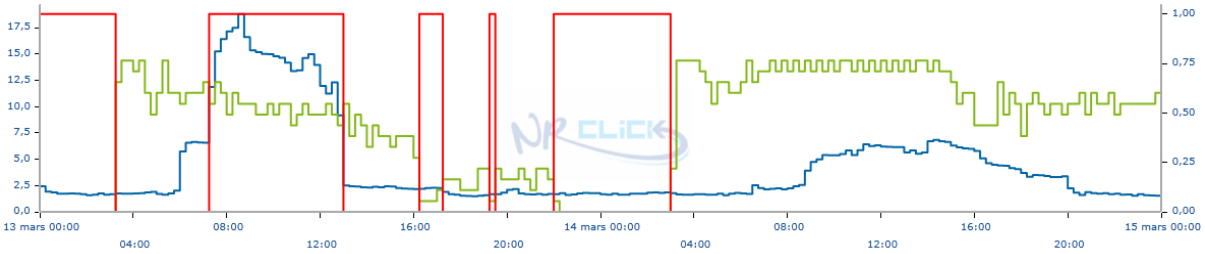
These conditions may also combine several channels

Example:

(@C1.data > @C2.data)

→The returned profile will consist of 1 when the data of the C1 Meter is greater than that of the C2 Meter, and 0 otherwise.

In the example below, the green and blue profiles are combined in this way to generate the 0/1 red profile, which is therefore 0 when the green profile is greater than the blue, and 1 otherwise.

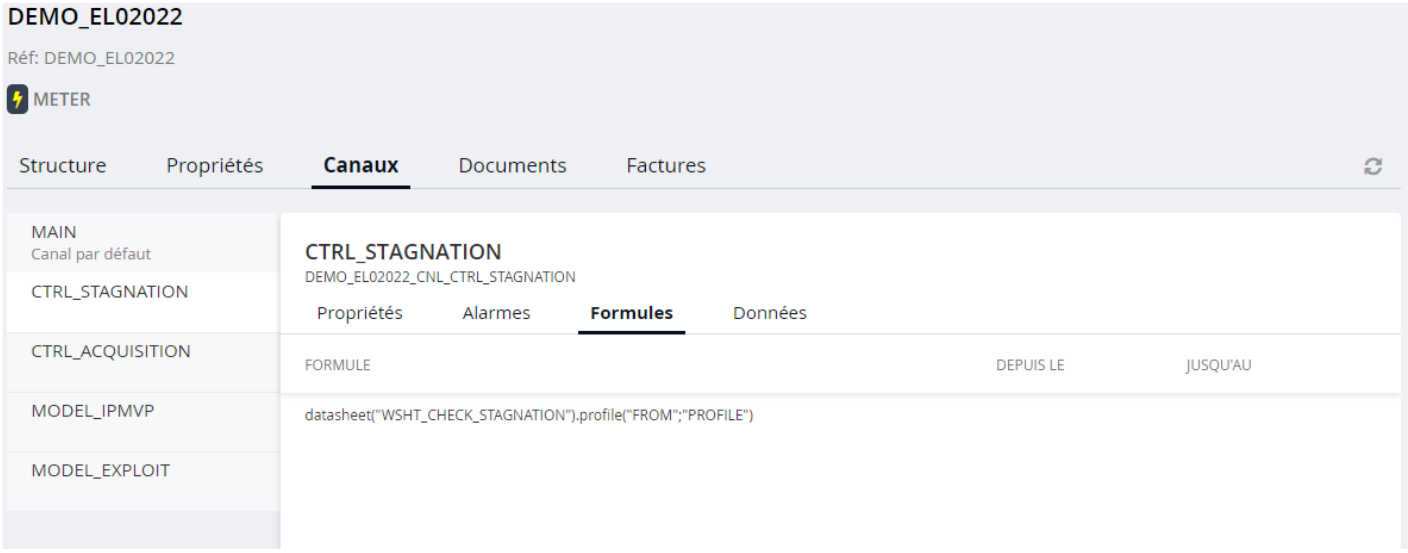


This kind of combination could be useful, for example, if you have several activation profiles (e.g. one ON/OFF for weekends, and another ON/OFF for school holidays) that you want to combine to obtain a general activation profile.

○ Virtual channel formulas

As already mentioned, there are several ways to feed a channel profile into EMM. A channel can receive data directly by external injection, from dataloggers or manual encodings, but it can also be the result of a formula calculation, based on other profiles or property values.

Channels calculated in this way are called "virtual" channels and contain, in the "Formula" tab of their channel record, a formula in EMM syntax, possibly with history, which describes the way in which the profile is calculated (example below).



These formulas can use any information present in EMM, they must return a data profile, and the result of the formula is stored in the RawData of the channel concerned. (be careful not to provide it with a pulse weight, otherwise it will be applied again when switching from RawData to Data)

Examples of formulas:

```
@C1.data + @C2.data
```

→ This formula indicates that the current channel is a calculated channel, summing the conso of the C1 meter (default channel) and the S2 meter (also default channel). This situation can happen when, for example, one wants to create a virtual head Meter on top of several sub-meters installed on a site.

```
@C1.data.agg(1; "day"; "sum")
```

→ The Data profile of the current channel will this time be the same as the C1 Meter, but with data aggregated (in sum) by month.

Obviously, all these methods can be combined to create the desired profile in the most efficient way.

Advanced example:

```
@C1.invoices("//TECH_CONS_TOTAL").slp(@C1.link("SLP").data; @C1)
```

→ We start with the C1 meter and build a data profile based on the TECH_CONS_TOTAL property values of its invoices. Once this profile has been built up, the .slp function is used to distribute this invoiced consumption according to the SLP profile, which is linked to the C1 meter via a link of type SLP. This method therefore requires that the SLP link type exists and that a dispatching SLP meter is linked to the current channel via a link of this type.

Based on consumption data from monthly invoices, a daily profile (or even a 10-minute profile) can easily be created.

.formula / .formula.from / .formula.to /.formula.value

If in the parser we need to retrieve the formula of a channel, we can use the ".formula" function. This function, when applied to a channel, will return all the formula blocks associated with that channel (in the case of a multiple selection, it will return all the formula blocks of each of the channels in the selection).

Once on a formula block, you can then use the .from / .to / .value functions to retrieve respectively the validity start date of the formula, its validity end date, and the text of the formula.

6. Data tables

A data array consists of rows and columns, and can be used as an object in the EMM syntax to apply various transformations to it.

A data table can be the result of a DataSet calculation, or the result of a Worksheet, or an Xtab, or even combinations of these.

Examples:

```
dataset("REF_DE_DATASET")

worksheet("REF_DE_WORKSHEET")

xtab("REF_DE_XTAB")
```

These three expressions will all return an array of data, each constructed based on the results of the requested dataset, workshet or Xtab.

In EMM syntax, a data array is considered a collection of rows, with each row consisting of as many fields as the array had columns.

Frequently, one will need to run a dataset, worksheet or Xtab based on another selection or context than the active selections or contexts. This can be done by passing additional optional arguments to the dataset, worksheet or xtab functions.

Example:

```
worksheet("REF_DE_WORKSHEET" ; selection ; from ; to)
```

This expression will explicitly pass the active selection and context to the worksheet function so that it uses them as the active selection and context when the worksheet is executed. In this case, this does not change the result, but it is of course possible to change the selection and context passed as arguments.

Example:

```
worksheet("REF_DE_WSHT" ; selection.children ; from.synchro(1; "year") ; to.synchro (1 ; "day"))
```

→This expression will run the "REF_DE_WSHT" spreadsheet based on the children of the active selection, for the period from the beginning of the year containing the start of the active context (from), to the beginning of the day containing the end of the active context (to).

Note that if you want to pass a modified context but without touching the active selection, you must still indicate the selection (unchanged) in the list of arguments. Indeed, it is the location of the arguments which gives them a meaning. The first argument is the reference of the worksheet to be executed, the second argument (optional) is the selection and the 3rd and 4th (also optional) are the from/to of the context to be used.

Note: Most of the time, a data table will be called in the selection box of a worksheet to serve as a source for the worksheet and to retrieve data for further work. However, it is also possible to call a data table directly in the formula of a worksheet column. For example, one could call a data table to retrieve the sum of the values in one of its columns and inject it into the cell of my new worksheet. This is not recommended, because it tends to be very resource-intensive and time-consuming. This is because in this situation the source table will be called up and presumably calculated for each row in the spreadsheet, with all that this can imply in terms of database access and calculation time.

Note: It is possible to apply the ".creation" and ".lastupdate" functions to DataSets, Worksheets and Xtabs. These functions will return a block of information containing the creation/update date and the user who created/updated it. The ".date" or ".user" functions must then be applied to retrieve the desired information.

Dataset("DSET_REF"). **creation**.user.mail
→will return the email address of the creator of this DataSet

worksheet("WSHT_REF"). **lastupdate**.date
→will return the date this spreadsheet was last updated.

a. Columns in a table

If you put a call to a table (such as those shown above) in the selection box of a spreadsheet, EMM will therefore interpret it as a collection of rows, and prepare a row in the spreadsheet for each row in the source table.

The active item for each row in the worksheet will then be a "row" object in the source table.

item.column

To access the information in this source row, the .column function can be used to access the various fields in the row in question based on the name of the corresponding column in the initial table.

Example:

Let's imagine that a DataSet "DSET_01" produces the following data table:

REFERENCE	NAME	SURFACE	VILLE
BXL_001	Site Grand Rue	1000	Bruxelles
BXL_002	Bourse	2000	Bruxelles
LLN_001	Halles	1000	Louvain-La-Neuve
LLN_002	Commune	500	Louvain-La-Neuve
LLN_003	Piscine	1500	Louvain-La-Neuve
LG_001	Hôtel de ville	800	Liège
LG_002	Bureaux	500	Liège

If in the selection box of a spreadsheet I call this DataSet with the following syntax: dataset("DSET_01"), my spreadsheet should contain 7 rows, the same number as the DataSet it uses as a source.

In the first column of my spreadsheet, which I will call "REF", I can then call up the values of the various columns of my source DataSet.

item. **column**("REFERENCE")

→ retrieves the "REFERENCE" column of my item (my source row), and displays the value in my spreadsheet, in the "REF" column.

In the spreadsheet, this column will then copy the "REFERENCE" column from my source DataSet. The resulting spreadsheet will therefore look like this:

REFERENCE
BXL_001
BXL_002
LLN_001
LLN_002
LLN_003
LG_001
LG_002

column

In the worksheet, rows of a source table can therefore be accessed via the expression item. **column**("REF_COL_SOURCE").

If now the goal is to access another column, not from the source table, but from the spreadsheet itself (retrieve the next column for example), we can use the function **column**("REF_COL_FEUILLE").

These two syntaxes are very similar, but conceptually quite different. The first one retrieves the columns of a source present in the active item, while the second one calls the result present in another column of the active sheet.

Example:

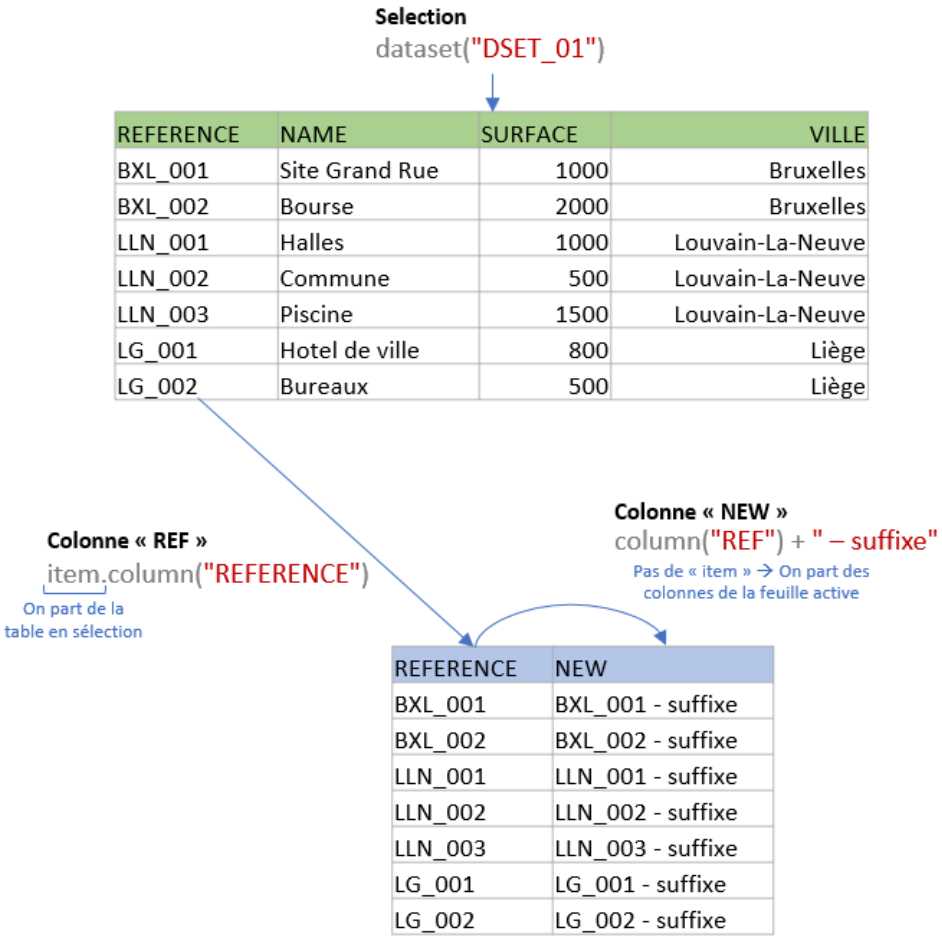
Continuing with the previous example, we could, for example, add a column to the spreadsheet and give it the following formula: column("REF") + " - suffix".

This syntax will then retrieve the information present in the "REF" column of the active worksheet (the column next to it), and add a "- suffix" to it

The following sheet is then obtained:

REFERENCE	NEW
BXL_001	BXL_001 - suffixe
BXL_002	BXL_002 - suffixe
LLN_001	LLN_001 - suffixe
...	...

The information goes through the following path:



REFERENCE	NAME	SURFACE	VILLE
BXL_001	Site Grand Rue	1000	Bruxelles
BXL_002	Bourse	2000	Bruxelles
LLN_001	Halles	1000	Louvain-La-Neuve
LLN_002	Commune	500	Louvain-La-Neuve
LLN_003	Piscine	1500	Louvain-La-Neuve
LG_001	Hotel de ville	800	Liège
LG_002	Bureaux	500	Liège

REFERENCE	NEW
BXL_001	BXL_001 - suffixe
BXL_002	BXL_002 - suffixe
LLN_001	LLN_001 - suffixe
LLN_002	LLN_002 - suffixe
LLN_003	LLN_003 - suffixe
LG_001	LG_001 - suffixe
LG_002	LG_002 - suffixe

accumulation

Quite regularly, it is useful to calculate a cumulative consumption. For example, we would have a column with the consumption, and we would want another column that would return for each row, the sum of the consumption of the previous rows.

The cumulate function allows you to cumulate values in this way.

Example:

COL_A = column of numerical values

COL_B = **cumul**("COL_A")

The column "COL_B" will return a column of numerical values representing the cumulative values of the column "COL_A".

Warning: In a spreadsheet, the column to be cumulated and the cumulated column must both be in numeric format, otherwise an error message will be displayed.

Advanced Use

In the case of a spreadsheet which displays the consos of several Meters one below the other, the accumulation should be done by Meter and not in a global way. It would therefore be necessary to be able to reset the accumulation to zero on the basis of a reference column. This would give the following situation:

COL_REF = column listing the meter references

COL_A = column of numerical values (several different values for each reference)

COL_B = **accumulation**("COL_A"; "COL_REF")

The "COL_B" column will return a column of cumulative values, but each time the value in the "COL_REF" column changes, the cumulative value will be reset to zero.

The 2nd argument of the cumulate function is optional and allows to limit the calculated cumulates to the lines of the same type. The type being, in the above case, the Meter considered. We will therefore have one accumulation per site. This reference column can obviously contain something other than references depending on the situation.

derivative

Similar to the cumulate function, but performs a derivative (successive deltas) instead of a cumulate. The syntax is similar.

Example:

COL_A = column of numerical values

COL_B = **derivative**("COL_A")

The "COL_B" column will return a column of numerical values representing the differences between successive values in the "COL_A" column

With the derivative function, the first value of the derived column will always be zero (NaN) since the previous value is not known, and therefore impossible to delta.

Advanced Use

In the case of a spreadsheet which displays the indexes of several Meters one below the other, the derivation should be done per Meter and not globally. The only impact of this is that successive values should not be taken into account when jumping between two different Meters. There is no practical significance to the difference between the last index of one Meter and the first index of another Meter.

COL_REF = column listing the meter references

COL_A = column of numerical values (several different values for each reference)

COL_B = **derivative**("COL_A"; "COL_REF")

The "COL_B" column will return a column of derived values (successive deltas) but each time the value in the "COL_REF" column changes, the derivative will have an empty cell since the delta sequence restarts at zero.

The 2nd argument of the derivative function is therefore optional and simply allows the calculated derivations to be cut off at lines of the same type. The type being, in the above case, the Meter considered. We will thus have a series of deltas per site, displaying an empty box at each change of value of the reference column (the first delta has no previous value, thus no way to calculate a difference). This reference column can obviously contain something other than references depending on the situation.

total

Still in the same vein as the accumulation and derivative functions, it is sometimes useful to retrieve the total value of another column, ideally without having to create additional worksheets. A classic example of use would be the constitution of a table in which one would have the consumptions of several meters (one per line for example), and that one would want a column displaying the percentage of the total conso associated with each meter. We would therefore have to divide the consumption of each meter by the total consumption, and we therefore need the sum of the consumption column to calculate this total consumption.

The total function, using the same syntax as the cumulative and derivative functions, takes this total from a numeric column and copies the value into each cell of the total column.

Example:

COL_A = column of numerical values

COL_B = **total**("COL_A")

The column "COL_B" will then contain in each of its cells the total value of the numerical column "COL_A".

Advanced Use

In some cases, one might want the TOTAL column not to contain an absolute total of another initial column, but subtotals by slices of that initial column. For example, one could have a table with several meters ("REF_CPT" column), divided into several groups of meters ("GROUP" column), and a column containing the consumption of each meter. In this case, it could be interesting to have not only the absolute total consumption of all the meters, but also the total consumption of each group

The situation would then be as follows:

GROUP = column listing the group references into which the meters are divided

COL_A = consumption column

COL_B = **total**("COL_A")

COL_C = **total**("COL_A"; "GROUP")

The column "COL_B" will again return a column containing the same value: the total consumption of the whole column "COL_A".

The "COL_C" column, on the other hand, will contain the total consumption of each group, identified by the values in the "GROUP" column.

The 2nd argument of the total function is therefore optional and allows the calculated totals to be limited to subtotals, based on rows of the same type. The type being, in the above case, the group considered.

linereg

Finally, it is possible to perform a linear regression between two numerical columns in the table, and to recover the parameters of this regression.

Example:

COL_A = column of numerical values

COL_B = column of numerical values

COL_C = **linereg**("COL_A"; "COL_AB"; "PARAMETER")

In this example, the column "COL_C" will perform a linear regression starting from the columns "COL_A" and "COL_B", respectively X and Y of the regression (so the order is important), and then it will return the values of the requested "PARAMETER".

The possible parameters are as follows:

- **DATA:** returns, for each row, the Y value of the linear regression line associated with the X of the current row.
- **FORMULA:** returns, for each row, the formula for the linear regression line. (in text format, and the same formula for all lines)
- **FORMULA_ROUNDED:** returns, for each line, the same formula as above, but rounded to 1 digit after the decimal point. (often more readable for a report)
- **INTERCEPT:** returns the intercept of the linear regression line. (same value for all lines)
- **SLOPE:** returns the slope of the linear regression line. (same value for all lines)
- **RMSE:** returns the root mean square error. (Square root of the average of the squared differences, same value for all lines)
- **CVRMSE:** returns the coefficient of variation of the root mean square error with respect to the mean of the points. (same value for all lines)
- **R2:** returns the coefficient of determination of the linear regression line. (same value for all rows)
- **MBE:** returns the mean bias error. (same value for all rows)
- **COV:** returns the covariance, the average of the product of the 2 values minus the product of the 2 averages. (same value for all rows)

Advanced Use

As with the "total", "derivative" and "cumulative" functions, it may sometimes be useful to perform not a complete linear regression on all the rows but on successive slices of data. For example, in the case of an

air conditioning system, one might want a linear regression of consumption against UDD. However, one might want two separate regression lines, one for heating consumption and the other for cooling consumption when the UDD exceeds a given threshold.

In this case, one or more columns can be used as additional argument(s) in the "linereg" function. Changes in the value of these columns will then serve as a trigger to cut a slice of data on which to perform the linear regression.

```
COL_C = linereg("COL_A"; "COL_B"; "PARAMETER"; "RESET_1"; "RESET_2")
```

In the case of the UDD example developed above, one could, for example, sort the data by increasing UDD, then create a column indicating whether one is below or above the threshold, and use this column as a reset parameter for the "linereg" function.

b. Create a range of values

Sometimes you need to create a series of values, evenly spaced, to serve as the basis for a calculation. For example, you may need to generate 10 rows numbered from 1 to 10, or generate one row for each day of the active context. The range function allows you to create such a collection of values in series.

This function actually generates an array of data that will contain just one column whose name is the first parameter to be given to the function, and with as many rows as necessary to start from the starting value and advance (without exceeding it) to the end value of the series.

Syntax for a numerical series :

```
Range("NAME"; Start; End; Step)
```

- "NAME" is a name to be given to the generated series (for future use)
- Start is the starting value of the series
- Fin is the maximum bound of this series
- Step is the step by which one progresses from the beginning to the end of the series

Syntax for a series of dates :

```
Range("NAME" ; Start_date ; End_date ; Step_number ; "Step_type" )
```

- "NAME" is a name to be given to the generated series (for future use)
- Start_date is the starting date of the time series
- End_date is the endpoint of the time series
- Step_number is the number of intervals for a time step
- "Step_type" is the interval type for a time step

Examples:

```
Range("A"; 1; 10; 1)
```

→ will return the series of values 1 2 3 4 5 6 7 8 9 10, in a table containing a single column called 'A', with one value per row.

```
Range("A"; 1; 10; 2)
```

→ will return the series of values 1 3 5 7 9. The series does not go any further since the next value (11) would be beyond the maximum bound provided to the range function (10).

```
Range("A" ; from ; to ; 1 ; "day")
```

→ will return a series of dates, starting with the start date of the active context, and progressing in steps of one day (1; "day"), until not exceeding the end date of the active context.

```
Range("A"; dateeval(2000); dateeval(2001); 1; "month")
```

→ will return a series of dates, from the first of January 2000 to the first of January 2001, in one-month steps.

In all the examples above, if we put this syntax in the selection of a worksheet, each row of this sheet will be associated with a row of the source table, and we will be able to retrieve the associated value via the function: item.column("A"), as if it were a completely classical table.

c. Filtering a table

As a selected array is considered a collection of rows, it is possible to apply the usual collection functions to it, using the values present in the columns of the source.

.orderby / .top / .flop

The .orderby function is used to sort the rows of the source table according to a given criterion. For example, one could decide to sort the rows of a consumption table by order of consumption. To do this, the selection box must contain a directly sorted source table.

```
dataset("DSET_CONSOS"). orderby(item.column("VALUE"))
```

→this selection will then constitute an array based on the DSET_CONSOS DataSet, ordered according to the values present in its "VALUE" column

It is also possible to keep only the first or last lines by using the .top / .flop functions.

```
dataset("DSET_CONSOS"). top(5)
```

→this selection will then form an array based on the DSET_CONSOS DataSet, and will keep only the first 5 rows of the array.

By combining these functions, it is possible to sort the rows and then keep only the first/last rows. This would allow you to keep only the highest consumption in the table, for example.

```
dataset("DSET_CONSOS"). orderby(item.column("VALUE")). flop(5)
```

→The DSET_CONSOS DataSet is used to build a source table, we then sort the rows according to the increasing value of the "VALUE" column, and finally, we keep only the last 5 rows, containing the worst consumptions in view of the sorting performed previously.

.where

It is also possible to filter the rows of a table by using the .where function to remove rows that do not meet certain criteria. For example, one could start with a table listing all the meters associated with a site, and filter in this table to keep only the electricity meters, by filtering on the column containing the resource measured by the meter.

Example:

```
dataset("DSET_CONSOS"). where(item.column("VALUE") <> 0)
```

→Construct a table based on the DataSet "DSET_CONSOS", and filter it to remove all rows where the consumption is zero (we keep those where the value of the column "VALUE" <> 0)

d. param - pass a variable value when calling an array

When calling an array of data, it can sometimes be convenient to add an (optional) value that would be used when calculating the array.

Example: If we want to retrieve all the information associated with all the electricity meters or all the gas meters via a DataSet, we would have to create two very similar DataSets. The first retrieving the information and filtering on the resource to keep only electricity, and the second doing exactly the same thing but keeping only gas.

The use of the additional parameter "param" makes it possible to create a single DataSet, which filters on the value of the parameter. The additional parameter "param" must be supplied when calling this DataSet, otherwise it will not work.

Syntax :

```
dataset("REFERENCE"; selection; from; to; param)
```

- The optional param argument must be in text format, but it can optionally come from an encapsulated formula that generates text.
- The selection, from and to arguments must be filled in if param is used. Indeed, it is always the order of the arguments that allows EMM to identify them and param must be the 5th argument.

Example: using the above example, the syntax to be used could be as follows:

```
dataset("DSET_GET_INFOS" ; selection ; from ; to ; "ELECTRICITY")

dataset("DSET_GET_INFOS" ; selection ; from ; to ; "NAT_GAS")
```

If within the DataSet, the term **[param]** (with square brackets) has been used as the filter value for the resource column (in key), the DataSet will filter on electricity or natural gas meters.

The use of the optional parameter param can also be applied to worksheet calls. The call is made in the same way, by adding a 5th argument to the worksheet call, and within the worksheet, the term **param** can then be used (without brackets this time) as if it were a variable which will receive its value when the worksheet is actually called.

Note that, in the same way as for selection and from/to, the value of the optional parameter param is considered a global variable from the moment it is defined, in that it is inherited by all successive arrays that might follow. If an array **A** calls an array **B** passing it a **param** value, and array **B** does nothing about it but calls an array **C**, without specifically passing it the **param** argument on the way, array **C** can still use the parameter value by simply using the term **param**.

e. Retrieving a value from a table

The first way to retrieve a specific value in a cell of a table is to filter it to keep only one row, then retrieve the desired column.

```
Xtab("REF_XTAB")
        .where(item.column("REF_COL_FILTER")=VALUE).top(1)
        .column("REV_COL_VALUE")
```

This is an accurate way of retrieving the value of a specific cell in the table, based on the column reference and the criteria used to filter the desired row correctly.

In the case of an Xtab where the first column is in date or numeric format, it is also possible to use an abbreviated syntax to retrieve a value.

```
Xtab("REF_ XTAB"; "REF_COL "; VALUE_1E_COL)
```

This syntax will retrieve, in the Xtab "REF_XTAB", the column "REF_COL", and will keep the value of the column in question corresponding to the row whose value in the first column is given by "VALUE_1E_COL".

Beware that if the value in the first column is not unique, this formula will return one of the answers it finds, but without any certainty that it will be the one we initially wanted. Ideally, therefore, the first column should only contain unique values if this method of retrieving values is to be used.

If the first column is in date format and its values are unique, it is also possible to retrieve a value by providing an approximate date of the date in the first column by telling EMM whether to retrieve the last date before the one provided or the first after.

Example:

Let's imagine that we have the following XTab as a working source

From	Abonnement	Terme Proportionnel	Terme de souscription
DateTime	Number	Number	Number
01/07/2014 08:00:00	33.24	26.32	0
01/07/2015 08:00:00	34.56	27.35	0
01/07/2016 08:00:00	34.2	28.72	0
01/07/2017 08:00:00	33.48	28.13	0
01/01/2018 07:00:00	40.32	28.13	0
01/07/2018 08:00:00	41.16	28.7	0

If this Xtab is called "XT_EPS_GF_T01" and we want to retrieve the value in the "Subscription" column as of 1 January 2017.

Note that there is no line in the XTab that gives a value for this specific date. The classic syntax

```
Xtab("XT_EPS_GF_T01"; "Subscription "; dateeval(2017))
```

will not return any result, as there are no lines with the correct date.

The syntax needs to be modified slightly to indicate that if EMM cannot find the right date line, it can look for the nearest one in the past, or in the future.

```
Xtab("XT_EPS_GF_T01"; "Subscription "; ~dateeval(2017))
```

```
Xtab("XT_EPS_GF_T01"; "Subscription "; dateeval(2017)~)
```

The symbol "**~**" placed before the date will indicate to EMM that it must go and retrieve the last value in the past (most frequent case, for the retrieval of a tariff for example)

If you place the symbol "**~**" after the date, EMM will retrieve the nearest value in the future.

f. Generate a profile from a table (.profile)

The .profile function allows, on the basis of a data table (DataSet, Worksheet, Xtab, complex combination...), to generate a data profile that can be used in the formula of a channel or in any other syntax in EMM.

This function will receive two or three arguments depending on what is available in the situation.

Syntax with 2 arguments :

```
DataSet("A"). profile("FROM" ; "VALUE")
```

This syntax was the first historically implemented. It therefore receives the reference of the column containing the "from" dates of the data profile, and another containing the reference of the column

containing the numerical values to be used for the profile. Each piece of data in the generated profile will then have as its "from" date the value available in the "FROM" column of the source table, and the "to" date of each piece of data is calculated by EMM using the "from" date of the next piece of data.

This works well when there is no "TO" column in the source table, such as in a tariff Xtab.

However, it can cause slight problems at the end of a profile. Indeed, as there is no next data, EMM does not find a "from" next, and therefore cannot constitute a "to" to the last data. Most of the time, this is not a problem, but in rare cases, it is information that must be taken into account.

Syntax with 3 arguments :

```
DataSet("A"). profile("FROM" ; "TO" ; "VALUE")
```

This new syntax works with three arguments, allowing data to be created with a clearly identified "from" and "to" in columns of the source table. This avoids the problem of profile endings (since the to's are identified without EMM having to calculate them), but it requires a "TO" column in the source table, which is not always possible (which is why the two syntaxes co-exist in EMM).

The first indicates the column of the source table that will be used as the "from" terminal for each value of the profile (this column must therefore obviously be in "date" format). The second parameter will be used as the "to" terminal for the corresponding value, and the last parameter is the reference of the column containing the numerical values of the profile.

In both cases, these formulas will return a data profile, dynamically built up either from the "FROM" and "VALUE" columns of the source table, or from the "FROM", "TO" and "VALUE" columns.

Note: This .profile function can be applied to any type of data table, whether it is from a DataSet, Worksheet, Xtab or any possible combination of these tables.

g. Grouping rows in a table

.groupby

A grouping of rows from a source table will be done in the selection of a worksheet, using the .groupby function. This grouping will generate a collection of objects of type key/elements, having as "key" the different values taken by the grouping key and as "elements" the list of rows of the source table sharing this grouping key.

Example:

Let's imagine that there is a DataSet "A", which, when calculated on the basis of the current selection and context, results in the following table (our source table)

Reference	Name	Groupe	Conso
REF_1	Elec général	1	25
REF_2	Elec éclairage	1	12
REF_3	Elec cuisine	1	6
REF_4	Gaz chauffage	2	32
REF_5	Eau générale	3	41
REF_6	Eau cuisine	3	13

If, in another table, this is passed in the selection box :

```
DataSet("A"). groupby(item.column("GROUP"))
```

The source of our active spreadsheet will then be the following table.

key	elements
1	Reference | Name | Groupe | Conso REF_1 | Elec général | 1 | 25 REF_2 | Elec éclairage | 1 | 12 REF_3 | Elec cuisine | 1 | 6
2	Reference | Name | Groupe | Conso REF_4 | Gaz chauffage | 2 | 32
3	Reference | Name | Groupe | Conso REF_5 | Eau générale | 3 | 41 REF_6 | Eau cuisine | 3 | 13

This table consists of 3 objects, 3 rows made up of 2 columns, the "key" column and the "elements" column, which itself contains a collection of sub-rows in each cell.

The spreadsheet will therefore have 3 rows, and each of these rows will have as its source one of the rows of the modified source table.

We can then use item.key to access the value of the first column of the source table, and item.elements to access the second column.

item.elements will itself be an array object, and its data can be retrieved via the .column function to produce a sum or an average, for example.

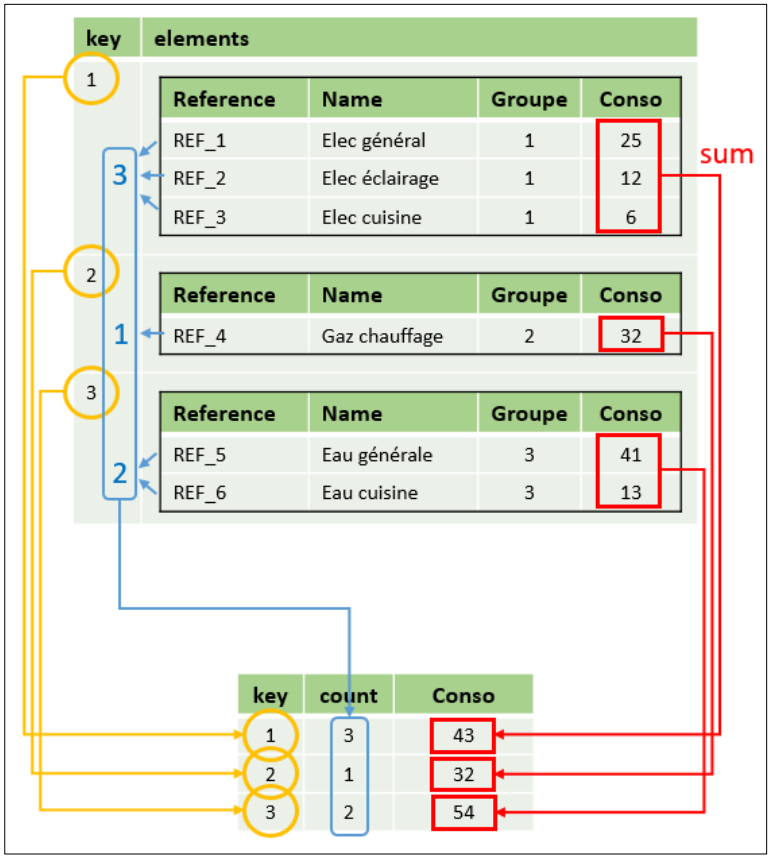
Example:

If the final table contains the following column definitions, the result will be as shown.

item.key

item.elements.count

item.elements.column("CONSO").sum



The first column simply lists the grouping key

The second column retrieves, for each key value, the list of associated elements, and counts their number.

The second column also retrieves, for each key value, the list of associated items, then looks at the "CONSO" column of each of these rows, and finally sums them.

h. Concatenate tables

Instead of retrieving information from a single table, it is also possible to call several (similar) tables at the same time by merging them (union procedure).

(DataSet("A"); DataSet ("B"))

→ This expression will fetch DataSet A and B and form a general array, in which the two arrays will simply be put end-to-end.

If the DataSets have columns with the same name, the columns will be merged and the column from B will continue the column started in A.

If a column does not exist in one of the DataSets, it will be created in the general table and left empty for the rows of the DataSet that does not contain it.

Attention: the column types must be respected.

This way of merging tables allows rows from several source tables to be combined, simply by calculating these source tables one after the other.

i. Grouping of tables (.join / .joinmulti)

In addition to being able to concatenate tables, it is also possible to merge them according to a defined key. For example, you could have a table with all the buildings of a customer, and a table with the list of customers and their information (name, telephone, email...)

In such a case, there is no need to copy all the information of a client into each line of his sites. Firstly for a question of database size, but also for a question of the number of calls to the DB. Indeed, if we copy the client's information into each line, the program will have to go and read them each time, whereas if we create separate tables, the reading will take place only once. Furthermore, as the information is stored in a single place, it simplifies any changes (change of telephone number, moving house, etc.)

.join

The .join function allows you to merge tables of different natures based on a specified key column.

Syntax :

(DataSet("A"); DataSet ("B")). **join**("reference"; "inner")

→Merge tables based on the "reference" column and use the "inner" method to do so.

The following methods are possible:

inner: Includes all the rows of the first table that can be associated with a row of the second table. Thus, if a row in the first table has no complement in the second, it will not be included in the final table. The same applies to a row in the second table which does not correspond to anything in the first table.

left: All the rows of the first table are taken over, and the information from the second table is attached to them (if there is no associated information, the columns remain empty)

right: All the rows of the second table are taken over, and the information from the first table is attached to them (if there is no associated information, the columns remain empty)

full : left AND right at the same time

Explanatory example:

For example, let's imagine that we have to join the following 2 tables on the basis of their common column "Reference":

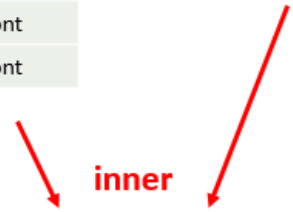
Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54

In the case of the "**inner**" junction, all the rows that can be completed will therefore be included. Line 003 of the first table does not have a Manager and will therefore not be included in the result, nor will the Manager Delieux, who is not associated with any company.

Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

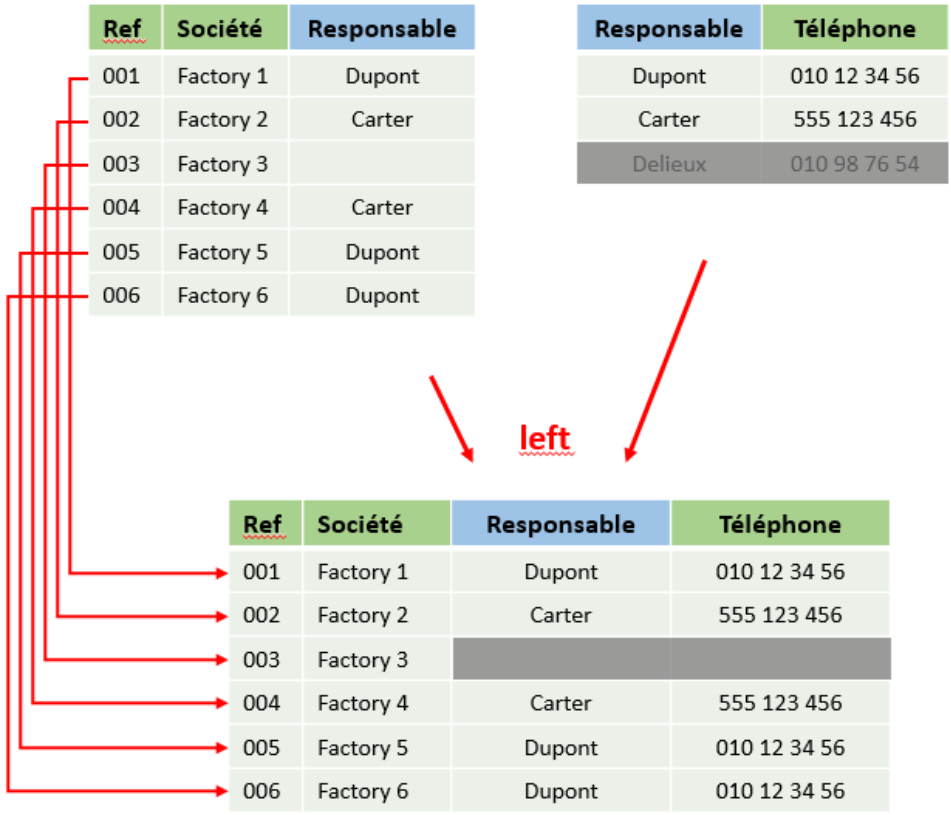
Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54



Ref	Société	Responsable	Téléphone
001	Factory 1	Dupont	010 12 34 56
002	Factory 2	Carter	555 123 456
004	Factory 4	Carter	555 123 456
005	Factory 5	Dupont	010 12 34 56
006	Factory 6	Dupont	010 12 34 56

The join via the "**left**" method will therefore take the list of rows of the first table, and will join the rows of the second table if possible, leaving empty those for which no join is possible.

Line 003 has no associated manager, so the rest of the line will remain empty, and as the Delieux manager is not associated with any company, it is never retrieved and therefore does not appear in the final result.



Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54

Ref	Société	Responsable	Téléphone
001	Factory 1	Dupont	010 12 34 56
002	Factory 2	Carter	555 123 456
003	Factory 3		
004	Factory 4	Carter	555 123 456
005	Factory 5	Dupont	010 12 34 56
006	Factory 6	Dupont	010 12 34 56

In the same way, the join via the "**right**" method will take the list of rows of the second table, and will join the rows of the first table if possible, leaving empty those for which no join is possible.

Note that in the case where several junctions are possible (as in the case of Dupont), the lines are duplicated to create all possible associations.

So here we start from the right, Smith is associated with 3 companies, so the line is multiplied to be associated with each company. Carter is associated with two companies, so there are two lines. As Smith is not associated with any company, the line is still created but the beginning will remain empty. Finally, it should be noted that company 003 is not linked to a manager, so it will never have been called in the junction, and will therefore not appear at all in the result

Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54

right

Ref	Société	Responsable	Téléphone
001	Factory 1	Dupont	010 12 34 56
005	Factory 5	Dupont	010 12 34 56
006	Factory 6	Dupont	010 12 34 56
002	Factory 2	Carter	555 123 456
004	Factory 4	Carter	555 123 456
		Delieux	010 98 76 54

Finally, the "**full**" join method will take all the rows from the inner method, then add the extra rows from the **left** and then the extra rows from the **right** so that no possibilities are left out.

Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54

Ref	Société	Responsable	Téléphone	
001	Factory 1	Dupont	010 12 34 56	inner
002	Factory 2	Carter	555 123 456	
004	Factory 4	Carter	555 123 456	
005	Factory 5	Dupont	010 12 34 56	
006	Factory 6	Dupont	010 12 34 56	
003	Factory 3			left
		Delieux	010 98 76 54	right

Warning: The tables joined by the .join function must all have the reference column (passed as an argument to the .join function), but they cannot have other columns in common. Indeed, the system would not know what to do in case of a conflict of values in different columns with the same name.

Note: it is possible to join more than two source arrays at once with the .join function. It is just necessary that all the joins are done on the same column and with the same method (left, inner...)

```
(DataSet("A"); DataSet ("B"); DataSet ("C"); DataSet ("D")). join("reference"; "inner")
```

→Merge all tables A, B, C and D based on the "reference" column and use the "inner" method each time to do so.

Otherwise, if junctions are to be made on different columns or if the joining methods differ, successive junctions will be necessary.

```
(
        (
                DataSet("A");
                DataSet ("B")
        ). join("reference"; "inner");
        DataSet ("C")
). join("consos"; "left")
```

This syntax will first join DataSet A and B based on the "reference" column and with the "inner" method, then it will join the result with DataSet C based on the "consos" column and with the "left" method.

.joinmulti

During the numerous uses of the ".join" function, we realised that we often had to make joins based not on a single common column but on a key composed of the concatenation of two columns.

Example: If you have two data tables with REFERENCE, DATE, VALUE columns and you want to join these two tables not only on the date but also on the reference, you would have to join on the concatenation of the REFERENCE and DATE columns. Using the current ".join", this would mean that each of the existing tables would have to be modified to include a "join key" column that would concatenate the REFERENCE and DATE columns. In the case of tables from DataSets, which cannot contain formulas, two intermediate worksheets would have to be created with the sole purpose of creating these two key columns.

To avoid the creation of unhelpful intermediate tables, the ".joinmulti" function has been implemented, allowing a join to be performed directly on a concatenation of columns.

Syntax :

(DataSet("A"); DataSet ("B")). **joinmulti**("method"; "reference_1"; "reference_2")

→Merge arrays based on the concatenation of columns "reference_1" and "reference_2" and use the "method" method to do so.

This function will therefore receive the join method ("inner", "left", "right" or "full"), followed by the list of column references to concatenate to form the join key. The number of column references is not limited to 2, so 3, 4 or 5 columns could be concatenated to form the join key.

Application example: let's imagine that we have the two tables on the left

The first is a consumption table containing several meters one below the other, with the meter reference, the date and the consumption value, to which a "REF_METEO" column has been added indicating the reference of the nearest weather station.

The second table is a weather data table, retrieving temperatures from several weather stations one below the other.

REF_CPT	REF_METEO	DATE	CONSO
CPT_001	W_BRUXELLES	1/1/2020	100
CPT_001	W_BRUXELLES	2/1/2020	95
CPT_001	W_BRUXELLES	3/1/2020	105
CPT_935	W_LLN	1/1/2020	44
CPT_935	W_LLN	2/1/2020	56
CPT_935	W_LLN	3/1/2020	48

REF_METEO	DATE	TEMP
W_BRUXELLES	1/1/2020	12
W_BRUXELLES	2/1/2020	10
W_BRUXELLES	3/1/2020	11
W_LLN	1/1/2020	14
W_LLN	2/1/2020	13
W_LLN	3/1/2020	15

REF_METEO	DATE	REF_CPT	CONSO	TEMP
W_BRUXELLES	1/1/2020	CPT_001	100	12
W_BRUXELLES	2/1/2020	CPT_001	95	10
W_BRUXELLES	3/1/2020	CPT_001	105	11
W_LLN	1/1/2020	CPT_935	44	14
W_LLN	2/1/2020	CPT_935	56	13
W_LLN	3/1/2020	CPT_935	48	15

A simple join based on the DATE column would be problematic. Indeed, this column can contain several times each date since the consumptions of several meters are included there. This would lead to cross-associations between the rows of a meter and the weather data of stations that do not correspond to it.

A simple join based on the weather station reference will obviously not work either, as the date concept is lost.

The solution here is therefore to make a connection based on the concatenation of the REF_METEO and DATE columns, so that each Meter data corresponds to the temperature of the associated weather station for the corresponding date.

Warning: the ".join" function and the ".joinmulti" function receive similar arguments (join key and join method), but **not in the same order**. Indeed, as the number of column references is potentially variable for ".joinmulti", the order of the parameters supplied to the function had to be reorganised to avoid computer ambiguities... and it is dangerous to swap the arguments of the historical ".join" function for questions of backward compatibility with the large number of existing configurations. So be careful to put the arguments in the right order for each of these functions.

.joinmulti dynamic (join columns with different references)

It can sometimes be complicated to have data tables with common column names for joins. Indeed, for example, the same DataSet could be used in several different situations, each requiring a specific name to

make joins. To solve this problem, it is possible to define correspondences between columns with different names in the ".joinmulti" function call

Let's say we have two DataSets

- A, containing (among other things) the column "COL_1" to be used to make a join.
- B, containing (among others) the column "COL_2" to be used for the same junction.

If it is impossible to rename these columns for some external reason, the join can still be made with the following syntax:

```
(dataset("A") ; dataset("B")).joinmulti("LEFT" ; "A.COL_1 = B.COL_2" )
```

This syntax indicates that, for the purposes of this join, column COL_1 in Table A should be considered as corresponding to column COL_2 in Table B. This 'common' column will then be used as the join column.

7. Invoices

In EMM, an invoice is seen as a set of property blocks, associated with a Meter, with a start date and an end date.

Starting from a Meter (to which invoices will generally be attached), the following syntax can then be used to obtain a list of all invoices associated with the Meter, for the active context.

```
selection. invoices
```

This syntax therefore returns a list of objects of type invoice. A selection box containing this syntax will generate a worksheet with one row per invoice.

Once the active object is an invoice (in the row of a table as described above for example), we can then retrieve the properties stored in the invoice with the usual syntax.

```
item.properties("//INV_ADM_TECH_CONS_TOTAL")
```

This formula will therefore return, for each invoice, the value of the "INV_ADM_TECH_CON_TOTAL" property, in this case the total consumption associated with the invoice, in the resource's unit of measurement.

As always, it is possible to filter invoices according to their properties with the ".where" function, so that only certain invoices are kept.

In addition to the properties contained in the invoice blocks, it is also possible to retrieve basic information associated with them:

item. **from** →will return the start date of the invoice

item. **to** →will return the end date of the invoice

item. **id** →will return the identifier of the invoice

item. **creation**.user.mail →will return the email address of the creator of this invoice

item. **lastupdate**.date →will return the date this invoice was last updated.

8. Users

In EMM, a user is an item that has certain basic characteristics and customisable property blocks. We are not talking here about user rights, which are configurable by an administrator, but which are not usable in the EMM syntax itself.

The complete list of users of a DB is accessible via the keyword "**users**". This keyword will return the list of items of type "user" present in the database. From this list, one can obviously carry out filters, sorting, and retrieve the information that one considers necessary.

Another function allows you to retrieve a specific user when you know his login.

user("abc") →will return as the active object the user who has login "abc"

Finally, the keyword "**me**" will return the logged-in user as the active item.

Once you have one or more users selected, you can retrieve their basic characteristics, such as their name, first name, email...

me. **firstname** →returns the first name of the active user

me. **lastname** → returns the last name of the active user

me. **name** → returns the full name (first name last name) of the active user

me. **mail** → returns the email address of the active user

user("abc"). **mail** → returns the mail address of the user with login "abc".

me. **login** → returns the login of the active user

user("abc"). **login** → returns the login of the user with the login "abc". Here, returns "abc".

users. **lastname** → returns the list of last names of all users

users.where(item.mail.endswith("@dapesco.com")). **name**

→ returns the list of names of all users whose email address ends with "@dapesco.com". Returns the list of users who are members of Dapesco.

In addition to the basic characteristics of users, an administrator can also assign custom property blocks to them, which can then be retrieved from the syntax via the ".properties" function

me. **properties**("//USR_MYSITE")

→ returns the "USR_MYSITE" property associated with the logged-in user.

users.where(item. **properties**("//USR_MYSITE")="BXL_0000").name

→ returns the list of names of all users whose "USR_MYSITE" property has the value "BXL_0000". This can be used to retrieve the list of users who are responsible for the Brussels site for example.

Note: When we are on a user property, we can use the ".parent" function to go back to the user concerned. The previous formula can then be optimised with the following syntax:

```
users. properties("//USR_MYSITE").where(item.value="BXL_0000")). parent.name
```

This syntax has the advantage of being more efficient than the previous one since it minimises the number of calls to the database. This is similar to the method developed in point 4: Object collection > Sorting and filtering a collection > Optimising filters.

9. Event management (.events)

- General

An event is a temporary property block that can be grafted onto an entity/Meter and which can provide information about an ongoing event concerning that entity/Meter.

Example: an event of type "WORK" can indicate to a manager that there is work in progress on a site, and that this impacts several Meters. The event must have a start date and may have an end date (some events open without knowing when they will close). It may also contain one or more property blocks detailing useful information about the event.

Finally, a discussion thread could be associated with each event, allowing the various participants to exchange information as required. For example, a WORK event could contain a thread where the works manager could discuss with the commune's energy manager and the EMM administrator.

In the EMM parser, it is possible to retrieve event information and associated threads via the following functions.

- Functions associated with events

.events

Starting from an entity/Meter, we can retrieve the linked events via the ".events" function

selection. **events→returns** the list of events of the selected entities/Meters

selection. **events**("RefTypeEvent")

→returns the list of events of the selected entities/Meters, but only those whose type matches the event type reference passed as argument.

Note: The events retrieved via these formulas are limited to those that cover all or part of the active time context. It is possible to force a context to the ".events" function by passing dates as parameters.

selection. **events**(from ; to)

→returns the list of events of the selected entities/Meters, limited to those appearing in the period from "from" to "to", these keywords can obviously be replaced by formulas returning dates.

selection. **events**("RefTypeEvent" ; from ; to)

→ same as the previous formulation, but limited to events of the type whose reference is passed as the first argument.

Events are therefore associated with entities/Meters, and whoever has the right to see the entity/Meter will also have the right to see the associated events. It is also possible to override users to give access to an event to a user who should not be able to see it (see EMM user manual for a description of the interface)

Events are therefore also, in a way, linked to users. It is therefore also possible to retrieve events from one or more users.

user. **events→returns** the list of events associated with all users

me. **events→returns** the list of events associated with the logged-in user

.from / .to / .reference / .name / .type / .parent / .creation / .lastupdate

An event therefore has a start date and potentially an end date. These can be retrieved via the ".from" and ".to" functions applied to the event in question.

selection.events.top(1). **from**

→will return the start date of the first event associated with the active selection.

In the same way, the main information of the event can be retrieved via the following functions:

.reference

→will return the reference (unique identifier) of the event. Usually this reference will be generated automatically by EMM and will only be used very rarely.

.type

→will return the reference of the associated event type. This is the type that will be used to sort the events according to the types of events to be followed in EMM (WORK, BREAK...)

.name

→will return the name of the associated event type. More humanly intelligible, it can be more explicit than the type reference.

.parent

→returns the list of entities/Meters linked to the event. Beware, an event can be linked to several entities/Meters, so the result of this function will indeed return a list of entities/Meters.

. creation

→returns the event creation information. Applying then .date will return the creation date, and applying .user will return the user of the event creator.

. lastupdate

→returns the last update information for the event. Applying then .date will return the update date, and applying .user will return the last user to have updated the event.

.comments / .comments.text / .comments.date / .comments.user

From an event, you can retrieve the list of comments that have been associated with it. To do this, starting from an event, use the ".comments" function which will return the list of all associated comments.

selection.events.flop(1). **comments**

→returns the list of comments associated with the last event associated with the selection.

Once on the comments, we can then use the following functions to retrieve the information for each comment:

.text→returns the text of the comment

.date→returns the date this comment was generated

.user

→returns the user object that wrote the comment. We can then follow up with a ".name" or ".mail" to retrieve the name or email of that user.

Advanced examples:

selection.events("WORK").orderby(item.from).flop(1).comments.orderby(item.date).flop(1).user.mail

→On the basis of the active selection, we retrieve all events of type "WORK" (event type reference = "WORK"). We then order them according to their start date, and keep only the last one. For this event, we recover the list of comments, which we order by date, and we keep only the last one. Finally, we find the user who wrote this comment, and we take his email which we finally display as the final result.

selection.events("WORK"). comments.orderby(item.date).flop(1).user.mail

→On the basis of the active selection, we retrieve all events of type "WORK" (event type reference = "WORK"). We then massively retrieve the list of all comments associated with all these events, which we order by date, and we keep the last one only. Finally, we find the user who wrote this comment, and we take his email which we finally display as the final result. This gives us a way to see who was the last person to comment on any event.

10. Alarms

General

In EMM, the alarm tasks are based on alarm properties, stored on the channels, to raise alarms when thresholds are exceeded. Once the alarms have been raised and stored in an alarm logbook, they can be retrieved via the DataSet (see EMM administrator's manual), or directly in the syntax.

Access to alarm properties

Accessing the alarm properties associated with a channel is done in exactly the same way as accessing a property, but via the ".alarm" function.

```
selection.channels("MAIN"). alarm("//CNL_ALA_MAX_ACTIVE")
```

This syntax will retrieve the "CNL_ALA_MAX_ACTIVE" alarm property associated with the "MAIN" channel of the selected Meter.

The method of operation is in every respect similar to that of the ".properties" function, both in terms of the management of single and multiple or historical values.

11. Execute code (Execute)

The EXECUTE function allows you to interpret a text as if it were EMM code. You can then dynamically build up a formula by joining various bits of text, which will finally be a statement that will be executed by EMM.

a. Converting a reference to an entity (sub-optimal syntax)

Although this is not optimal from a computational point of view, it is possible to transform a channel reference (text format) into a usable channel object.

Example:

```
execute("@" + "BXL_0000")
```

→ will return the entity whose reference is "BXL_0000". Any formula applicable to an entity can then be applied to it. This syntax will have exactly the same effect as "@BXL_0000", which may not be very interesting as is. @BXL_0000", which may not seem very interesting as it is. The following syntax is more interesting...

```
execute("@" + item.properties("//REF_SITE"))
```

→ This syntax is much more useful, since it starts from a property stored in text format in the active item, uses this text to retrieve the entity whose reference the text is. We can therefore transform a text property into an entity that can be used in the parser.

As mentioned above, this syntax is not optimal because it uses a heavy function (execute) to perform a rather simple task. The following syntax has been created to optimise this kind of thing: @(item.properties("//REF_SITE"))

This shorter syntax has been optimised mainly for transforming a reference into an entity, whereas the much more general execute function is not at all optimised for this case.

b. Temporary selection and context

In addition to this application, the "execute" function can also take as arguments a selection and a time period that will serve as a temporary context for the execution of the instruction text.

Syntax :

```
execute("TEXT TO BE EVALUATED"; selection; from; to )
```

→The "execute" function will execute the text to be evaluated but within the temporary selection and context passed as argument.

This can be extremely useful when, for example, one needs to retrieve data from a context other than the active time context, such as retrieving consumption over a reference period (outside the current context).

Example:

```
execute("dataset(""DSET_A_CALCULER"")"; @EntityRef; dateeval(2014); dateeval(2014; 2) )
```

→The "execute" function will execute the "DSET_A_CALCULER" array but passing it as a selection the "EntityRef" entity, and in the context from 1° January to 1° February 2014, and this, regardless of the current selection or context.

Note: be careful to double the " " " inside the text to be executed, otherwise EMM will consider that the text is closed.

Note: When calling a DataSet (or a spreadsheet for that matter...), it is also possible to pass the selection and context parameters directly to it, without necessarily passing through an execute.

The following syntax will therefore return exactly the same as the syntax above:

```
dataset("DSET_A_CALCULER"; @EntityRef; dateeval(2014); dateeval(2014; 2) )
```

This can improve the speed of calculation, and it is preferable to using a more resource-intensive execute. The use of the execute function will therefore be reserved for special cases, such as the case developed in the following paragraph.

c. Loop management

This "execute" function can also be used to retrieve arguments from code loops, for example when using a "foreach".

Explanation:

Let's imagine that we have as selection a list of Meters, and that we want to calculate in the cell of an array the consumption value of its descendants multiplied by a coefficient coming from the active Meter of the row.

You could try it like this:

```
item.descendants.foreach(item.data.sum * item.properties("//COEFF"))
```

The problem is that once in the foreach loop, the keyword "item" has a new meaning. The "item" at the beginning and the one in the loop do not represent the same thing. Once in the foreach, therefore, it is no longer possible to simply retrieve the starting Meter (outside the loop)

One solution is to use an "execute" to build the formula before executing it.

```
execute("item.descendants.foreach(item.data.sum * " + item.properties("//COEFF") + ")" )
```

This way of doing things makes it possible to build up the formula by inserting the calculated value out of the loop, and then to execute the formula, which will thus become

```
item.descendants.foreach(item.data.sum * 35)
```

...since the value of the coefficient will be calculated before it is inserted into the loop and executed.

12. HTML report creation

The easiest way to create reports in EMM is through the creation of dashboards (see the manual "Using EMM" for a full explanation of dashboard creation). However, there may be times when you need more flexibility in generating the report, such as when you want to dynamically integrate or collapse blocks in the report, or when you want to loop over a (previously unknown) number of objects to create a variable number of blocks in the report.

In these circumstances, the EMM HTML report editor should be used, which allows you to write a dynamically constructed HTML page based on a EMM selection and context.

In the editor, the basis is therefore HTML, into which the traditional style tags can be injected. To this, we add a special tag dedicated to injecting EMM code, the double braces {{ ... }}.

Between these double braces, EMM code can be injected to retrieve dynamic values based on the selection and context passed to the report. These tags can contain all the known EMM syntax, and the use of EMM code will make it possible, among other things, to create conditions or loops in a sometimes too rigid HTML language.

a. Recovery of dynamic values

It is possible to simply use the active selection when generating the report. In this case, the selection keyword is used and the desired transformations are applied.

Example : {{ selection.name }} → will return the name of the active selection (if there are several entities in the selection, this code will return the list of names of the entities in succession)

Similarly, it is possible to use the from and to keywords to retrieve the start and end dates of the report generation context.

Example: {{ from.format("d/MM/yyyy") }}→return the start date of the context, in the requested format.

In addition to these rather generic values, it is also possible to use the "selection" box in the report generator to retrieve information item by item. To do this, the @@itemtemplate@@ tag should be used. This tag, used in pairs in an HTML report, will tell EMM to copy the code between these tags as many times as there are objects in the selection box of the report. At each iteration of the code, the keyword "item" can be used to refer to the active object.

Example:

If the selection contains 3 sites, we could have the following code:

```
Report<br>
Period from <b> {{ from.format("d/MM/yyyy") }}</b> to <b> {{ to.format("d/MM/yyyy") }}</b>
<br>
<table>
```

```
<tr><td>Site</td><td>Reference</td><td>Number of Meters</td></tr>
@@itemtemplate@@
<tr>
        <td> {{ item.name }}</td>
        <td> {{ item.reference }}</td>
        <td> {{ item. link("ENTITY_METER").count }}</td>
</tr>
@@itemtemplate@@
</table>
```

This code would then return a result that looks like this:

Report

Period from **1/01/2018** to **1/01/2019**

Website	Reference	Number of meters
Site 1	SITE_001	4
Site 2	SITE_002	2
Site 3	SITE_003	5

In the title, we get the start and end dates of the context, then a table is built, with a title row (out of the loop), then @@itemtempalte@@ starts a loop which contains a table row displaying the name, reference, and number of Meters of each item in the selection. Once all the items in the selection have been processed, we then exit the loop and close the table cleanly.

Of course, it is also possible to provide a selection to the report which will be in the form of an array of values (result of the generation of a dataset, or of a spreadsheet, or of a junction between several tables. In this case, the table passed as a selection will be considered as a collection of rows, and "item.column("...")" can be used to retrieve the values from the source table.

b. Inclusion of report in another report

The generation of HTML reports can be done in a modular way. For example, one can create an HTML sub-report that analyses electricity consumption, another that specifically analyses gas consumption (the analyses displayed can be different), and a meta-report that will include these two sub-reports without the need to retype all the code of the sub-reports in the main report.

To do this, we will use the htmlreport function, which will retrieve the report that we pass to it as an argument and include it in the current report.

Syntax :

htmlreport("RefRapportHTML"; selection; from; to; "LANG"; selection_substitution)

- "RefRapportHTML" will be the reference of the "HTML Report" object that we want to generate.
- selection (optional) is the selection you want to use for the generation of the HTML report

- from; to (optional) represent the time context for which to generate the HTML report
- "LANG" (optional) is the code of the language in which to generate the report (for example: "FR")
- selection_substitution (optional) is another way to pass the selection to the report

If neither selection nor context is provided when calling this function (these three parameters are optional), the generating selection and context will be the active selection and context.

If no language code is provided, the language of the logged-in user will be used to generate the report.

With regard to the selections

The second argument selection is a selection in the same sense as for worksheet calls. This means that the selection passed in the second argument will be passed to the subreport and the keyword "selection" will be replaced there by the entities/Meters contained in this argument. This argument can obviously be in the form of a formula which would return one or more entities/Meters.

Unfortunately, there are situations where an HTML report is based on a table of data and it is interesting to pass part of this table to one of its sub-reports as a selection, rather than passing the list of entities and the sub-report being forced to recalculate the table in question. The situation is similar with grouped lists of items, for example... However, the syntax of the second argument requires that the items passed as a selection are entities/Meters.

To allow passing subsets of data (tables, subtables, sets resulting from a .groupby...), the optional 6th argument has been added. If one decides to pass a selection via this 6th argument, one can provide it with a collection of almost any type of objects, and this collection of objects passed as an argument will completely replace the result of the "selection" box of the called sub-report. This means that this selection box will not be used, and if it contained a formula, it would be overwritten by the selection substitution and would have no impact at call time.

Examples of syntax

htmlreport("RefRapportHTML")

→ Generates the requested report, passing it the active (default) selection and contexts, in the language of the logged-in user.

htmlreport("RefRapportHTML"; selection; from; to)

→ Generates the requested report, passing it the selection and context defined in the call, in the language of the logged-in user. Selection, from and to can be replaced by formulas returning a collection of entities/Meters, and dates respectively.

htmlreport("RefRapportHTML"; selection; from; to)

→ Generates the requested report, passing it the selection and context defined in the call, in the language of the logged-in user. Selection, from and to can be replaced by formulas returning a collection of entities/Meters, and dates respectively.

htmlreport("RefRapportHTML"; **null**; from; to)

→ In the case where we want to provide an argument but the previous arguments are not needed, we can replace them with "**null**". Thus, in this example, arguments 3 and 4 are indeed the from and to that we want to provide, without having to provide a selection.

htmlreport("RefRapportHTML"; **null**; from; to; **null** ; selection_substitution)

→ In this case, we want to pass a selection via substitution mode as an argument to the hmlreport function. To do this, the "classic" selection will be replaced with "**null**", and if we don't need to impose the generation language, the language code will also be replaced with "**null**" to keep the correct count in the argument positions.

The result of this syntax will be a text object, written in HTML, usable like any other EMM text. Injecting this piece of syntax (in double braces) into another HTML report will generate the sub-report, and inject the result into the main report, thus constituting a final report which may be composed of several sub-reports.

c. Conditional inclusion

The use of EMM code in HTML reports makes it possible to display or hide report blocks according to certain properties. For example, a report can be made to contain or not contain a gas consumption analysis block depending on whether or not there is gas on the site.

Example:

```
{{
        if(selection.children.channels.properties("//CNL_DAC_RESOURCE")
        .where(item.key="NAT_GAS").parent.count <> 0;
        htmlreport("HTML_SOUS_RAPPORT_GAZ"); "")
}}
```

With this syntax, we start with the children of the selection. We then go down to their channels, and get the property indicating what type of resource they are measuring. We filter to keep only the channels measuring natural gas, then we go back to the parents, and perform a count.

If there is a non-zero count, this means that there is at least one natural gas meter on the site. In this case, the "HTML_SOUS_RAPPORT_GAZ" sub-report is injected, and if not, nothing is injected. This is consistent with the fact that if there are gas meters, their analysis is displayed, otherwise the analysis is skipped so as not to display an empty block.

d. Embedding a widget in an HTML report

In many cases, it is useful to be able to integrate a widget into HTML. For example, a graph or a gauge can be integrated into the body of a report which can then be sent by e-mail, converted into a PDF.

Syntax :

{{ **widget**("Ref_du_Widget"; width; height; selection; from; to) }}

- "Ref_of_Widget" is of course the reference of the widget that we want to integrate.
- width; height are the width and height in pixels, in percentage or in "automatic" mode assigned to the widget in the HTML report
- selection is the selection (list of entities/Meters) to be used for the generation of the widget
- from; to are the bounds of the widget's execution context

All of these arguments can of course be calculated dynamically by formulas rather than hard-coded into the code.

Examples:

{{ **widget**("GRAPHE_DE_CONSO"; 200; 150; selection; from; to) }}

→ Returns a 200x150 pixel widget, calculated on the active selection and context.

{{ **widget**("GRAPHE_DE_CONSO"; "75%"; "50%"; selection; from; to) }}

→ Returns a widget 75% of its HTML container wide and 50% of the height of the same container. Note: the percentage is noted in text and the unit used must be indicated. This unit can also be "px" and in this case there is no difference between providing numbers or text in pixels.

{{ **widget**("GRAPHE_DE_CONSO"; "75%"; "auto"; selection; from; to) }}

→ Returns a widget 75% of its HTML container wide and in height, it will take up all the free space in its HTML container. If the argument is not supplied on the call, its default value will be set to "auto". It should be noted that it is more practical to use this "auto" mode than to impose 100%. Indeed, "100%" implies the entire size of the container, including any margins (padding). There is therefore a risk of overflowing the container if you impose 100% in conjunction with the presence of margins.

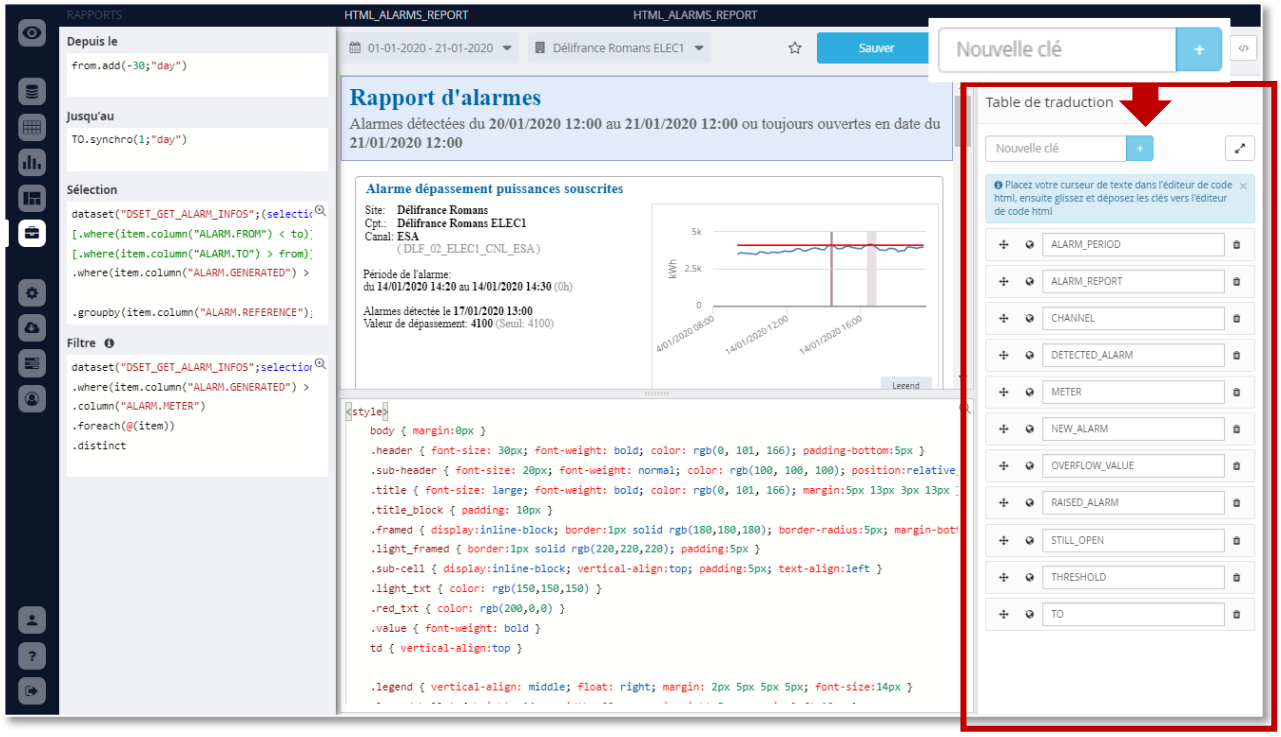
You can also combine dimensions in pixels and others in percentages, although this is sometimes more complicated to manage in a layout...

e. Translation module

In the case of a database frequented by multilingual users, many values will automatically adapt according to the language configured for the active user, whether on display or when sending reports by e-mail (each recipient is supposed to receive the report in his or her own language); widget titles (encoded in both languages when the widgets are defined), properties in multilingual text (chosen from drop-down menus whose values are taken from an Xtab)... but the "hard" text in the HTML report will not translate itself spontaneously.

A first solution is to create a clone of the report per language in the database, but this can quickly become a maintenance nightmare, especially if the number of reports increases a bit.

This is where the translation module can be useful. In the HTML editor, the translator can be accessed by clicking on the small planet at the top right of the page. This opens a pane on the right of the screen allowing you to set up translation keys, which can then be used in the body of the report.



When starting a translation table, you should start by adding the keys you want to use. This can be done by noting the desired key in the small field at the top left of the pane, and then clicking on the '+' next to it. This will add the new key to the list below it in the translation pane. A key should be seen as a reference, i.e. it should be unique (within the report), and it should not contain any special characters as this will interfere with the HTML code of the report.

Next to each key in the translator's list, there is a small cross to grab the key and drag and drop it onto the report, as well as a small planet to open a pop-up where you can give the translations of the key in the different languages of the database.

The small button with the 2 arrows at the top right of the translation pane allows you to deploy all the keys at once, directly in the translation pane.

Once the keys and translations are ready, they can be dragged directly into the HTML code of the report (the associated code will be inserted at the very top of the page), or the access code for this key can be typed directly into the report, where you want the translated text to be visible in the report.

Syntax :

[[translate.TRADUCTION_KEY]]

The text "TRANSLATION_KEY" is therefore one of the keys created in the translation pane, and when the HTML report is interpreted, this code will be replaced by the translation defined in the pane for the language of the logged-in user (when viewing in EMM), or of the recipient (in the case of an automatic report sending).

13. Generate PDF or CSV files

Whether to attach it to an email (see chapter on emailing) or to store it directly in an entity/Meter's documents, it is often useful to create PDF files from HTML reports, or CSV files from EMM data tables.

a. Retrieving an HTML report

Objects of type "HTML report" can be created in the report editor. The htmlreport function (detailed in a previous chapter) is used to retrieve the result of the generation of an HTML report object by EMM.

htmlreport("RefRapportHTML"; selection; from; to) will therefore return an object in text format, which can be used either directly in the body of the mail or as an attachment after conversion to PDF.

b. Converting an HTML report to PDF

The ".topdf" function applies to an HTML report (mostly generated at the time via the "htmlreport" function) and converts it into a PDF object. This PDF file can then be used as an attachment for an email or simply stored in the documents of one or more entities/Meters.

Syntax: htmlreport("RefRaHTML"). **topdf**("name.pdf")

This formula will result in a PDF object called "name.pdf", based on the HTML report "RefRapportHTML", which can then be attached to an outgoing email for example.

The ".topdf" function can also be given an additional optional parameter to save the generated PDF in the "Documents" tab of one or more entities/Meters.

htmlreport("RefRapportHTML"). topdf("name.pdf"; **@entity**)

→ The optional parameter must be an entity (hardcoded via an "@REFERENCE" or resulting from a formula like "item", or "item.parent" for example...) and the generated PDF will be stored in the "Documents" tab of this entity/Meter.

htmlreport("RefRapportHTML"). topdf("name.pdf" ; **(@entity_1 ; @entity_2)**)

→ the optional parameter can also be a list of entities/Meters. In this case, the list must be enclosed in brackets and the components of the list separated by ";".

The parentheses are mandatory so that EMM understands the list as a single parameter supplied to the ".topdf" function and not as a multitude of additional parameters.

It is also possible to store the generated file automatically, not only in the root of the "Documents" tab, but also in a directory located in this tab. To do this, the path must be indicated in the name of the PDF. If the given directory does not yet exist, it will be generated on the fly to store the PDF.

htmlreport("RefRapportHTML"). topdf("**DIRECTORY/name**.pdf"; @entity)

→ The PDF document will therefore be stored in the "Documents" tab of the chosen entity, but inside a directory, here named "DIRECTORY"

c. Converting a data table to CSV

Sometimes it makes more sense to send a raw data table directly as a CSV file than as a PDF report. For this purpose, the ".tocsv" function allows you to convert a data table into a CSV file, whether it is a dataset, a spreadsheet, the result of a join or a groupby... or any table present in EMM.

Syntax: dataset("RefDataSet"). **tocsv**("name. csv")

This code will return a CSV object, which can then be attached to an email to send as an attachment for example.

Like the ".topdf" function, this ".tocsv" function can also be given an additional optional argument if one wants to store the CSV file thus generated in the "Documents" tab of one or more entities/Meters. The parameter will once again have to be an entity/Meter (and not only its reference) and can therefore be indicated in hardcopy via an "@REFERENCE" or result from a formula like "item", or "item.parent" for example.

dataset("RefDataSet"). tocsv("name. csv"; **@entity**)

→ The optional parameter must be an entity (hardcoded via an "@REFERENCE" or resulting from a formula like "item", or "item.parent" for example...) and the generated PDF will be stored in the "Documents" tab of this entity/Meter.

dataset("RefDataSet"). tocsv("name. csv" ; **(@entity_1 ; @entity_2)**)

→ the optional parameter can also be a list of entities/Meters. In this case, the list must be enclosed in brackets and the components of the list separated by ";".

The parentheses are mandatory so that EMM understands the list as a single parameter supplied to the ".topdf" function and not as a multitude of additional parameters.

dataset ("RefDataSet "). tocsv ("**DIRECTORY/name**. csv "; **@entity**)

→ The CSV document will be stored in the "Documents" tab of the chosen entity, but inside a directory, here named "DIRECTORY"

o More optional parameters fot the .tocsv function

In addition to the mandatory name and the optional entity(ies) where to store the csv file, the ".tocsv" function accepts other optional parameters to precisely configure the format of the produces CSV file

Full syntax: dataset("RefDataSet"). **tocsv**("name. csv"; @ENTITY; ";" ; false ; "dd/MM/yyyy" ; ",")

- ";" : [optional] defines the column separator to use in the CSV file

Allowed values				
","	";"	" "	"\t"	"|"

- true / false : [optional] indicates if we keep (true) or remove (false) the column headers in the CSV file.

- "dd/MM/yyyy" : [optional] defines the date format to be used in the CSV file

- "," : [optional] choses the decimal separator for numbers in the CSV file ("." ou ",")

14. Manual mailings via the parser

In EMM, the management of the sending of dashboards and HTML reports by automatic emails is managed by the Reports manager. The use of the functions in this chapter will therefore be mainly limited to cases of re-generation of reports on demand, or debugging during their creation.

It is also possible that we just want to quickly send a simple e-mail to one or more recipients, with or without attachments. The following functions can help us with this.

The sendmail function is used to send an email. It receives a series of parameters defining all aspects of the email to be sent.

Syntax :

sendmail("abc@dapesco.com"; "def@mail.com"; "mail subject"; "mail content"; attachment)

- The first parameter is the recipient of the mail. The parameter must be in text format, but it can be a collection of texts.
  For example,

  "abc@dapesco.com will send an email to this address

  ("abc@dapesco.com"; "def@mail.com") will send an email with two recipients

- The second parameter is the address of the carbon copy recipient(s). The same syntax as above can be used to have multiple recipients.

- The third parameter is the subject of the email. Also in text format, it can of course be constructed via a more complex formula if required, but the formula must then produce text.

- The fourth parameter is the content of the mail body. It can be a simple text as well, possibly made up of EMM formulas, but it can also be the result of the generation of an HTML report. In this case the HTML report will be injected directly into the mail body.

- Finally, the fifth parameter (optional) is the attachment that you want to give to the email sent. This attachment will usually be an HTML report converted to PDF, or a CSV file allowing the export of a data table.

Examples:

sendmail("abc@dapesco.com"; "def@mail.com"; "test"; "this is a test")

→Send an email to "abc@dapesco.com", with a carbon copy to "def@mail.com", with the title "test" and simply containing the text "this is a test"

sendmail(("abc@dapesco.com"; "def@mail.com"); "archive@mail.com"; "test"; "this is a test")

→Send the same mail as before to "abc@dapesco.com" and "def@mail.com", with carbon copy to "archive.mail.com".

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "this is a test";
    htmlreport("RefRapportHTML"). topdf("name.pdf"))
```

→Send the same email as before to "abc@dapesco.com", with carbon copy to "archive.mail.com", and with an attachment named "name.pdf" which will be a PDF generated based on the HTML report "RefRapportHTML".

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "this is a test";
    dataset("RefDataSet"). tocsv("name. csv"))
```

→Send the same mail as before to "abc@dapesco.com", with carbon copy to "archive.mail.com", and with an attachment named "name.csv" which will be generated from the DataSet "RefDataSet".

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; htmlreport("RefRapportHTML"))
```

→Send the same mail as before except that the report will no longer be a PDF attached to the mail, but an HTML inserted directly into the body of the mail (the content parameter is directly the HTML report)

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; htmlreport("RefRapportHTML");
    htmlreport("RefRapportHTML"). topdf("name.pdf"))
```

→Send the same email as before with the report in the body of the email in HTML format, **and** the same report converted to PDF, attached to the email.

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "this is a test";
    htmlreport("RefRapportHTML"). topdf("name.pdf"; @SITE_1))
```

→Send the same email as before to "abc@dapesco.com", with carbon copy to "archive.mail.com", and with an attachment named "name.pdf" which will be a PDF generated based on the HTML report "RefRapportHTML". Furthermore, in the process, the generated PDF will be saved and stored in the "Documents" tab of the "SITE_1" entity.

15. Updates via the parser

It can be useful to update various things automatically in the EMM database. For example, a benchmarking Xtab could be updated every month, or a property or invoice could be updated based on a regular calculation. These updates can be triggered manually by running a spreadsheet, or automated via the task manager.

a. updatextab - Update a pre-existing Xtab

To modify the values present in an XTab, for example, for an XTab containing values to be changed every year, via a task that would update its values, there is the "updatextab" function, which can therefore be used in a spreadsheet or in the automatic task manager.

This function receives a single parameter which must be an array of data (dataset, worksheet...), containing the values to be added/updated in the XTab.

The first column of the source table in question will contain the reference of the XTab to be updated. (This will allow several different XTabs to be updated in the same action, since each row will update the XTab whose reference is stored in the first column)

The other columns of the source table must have the same references as the columns of the XTab to be modified. (It is on the basis of the column references that the update function will update)

The first column of the XTab will be used as a key column to identify which of its rows should be modified. If an attempt is made to import a row with a given key value and this key already exists in XTab, the corresponding row will be updated. If, on the other hand, the key in question does not exist, the function will create a new row for this new key value.

Example:

XTAB_TO_UPDATE

Key	Year	Value
001	2014	3
002	2015	24

WSHT_PREPA_XTAB_UPDATE

XTAB	Key	Year	Value
XTAB_TO_UPDATE	001	2015	12
XTAB_TO_UPDATE	003	2015	24

updatextab(datasheet("WSHT_PREPA_XTAB_UPDATE"))

→In this case, the function will look for the worksheet "WSHT_PREPA_XTAB_UPDATE" and look for each row at the XTab to be modified. In this example, it will always be the same: XTAB_TO_UPDATE.

→For the first row of the preparatory spreadsheet, the function will see that there is already the key 001 in the XTab and will therefore change its values

→For the 2nd row of the worksheet, the key is not already present in the XTab, so the function will create it and assign the given values to it.

XTAB_TO_UPDATE (after modifications)

Key	Year	Value
001	2015	12
002	2015	24
003	2015	24

Note that row 002 of the XTab has not been specified in the update worksheet and will not be affected by the function at all.

Reminder: The column references of the spreadsheet and XTab must be the same

b. updatextab - Creating Xtab on the fly

If the preparatory worksheet contains a non-existent Xtab reference in one of its rows, EMM will not be able to update anything for that row. Indeed, as the Xtab does not exist, there is nothing to update.

In this case, it is possible to ask EMM to create a new Xtab on the fly to inject the line in question. To do this, the "updatextab" function must be given an additional (and optional) parameter which will be the reference of an existing Xtab, to be used as a template for the creation of the new Xtab.

Syntax :

updatextab(datasheet("WSHT_PREPA_XTAB_UPDATE"); "XTAB_TEMPLATE")

- The first parameter is the preparatory spreadsheet, containing the rows of data to be updated.
- The second optional parameter is therefore the reference of another existing Xtab, to be used as a template for a possible creation of Xtab on the fly.

  If the preparatory spreadsheet contains a non-existent Xtab reference in the first column, EMM would then fetch the template Xtab (2nd parameter), clone its structure (columns) and assign it the non-existent reference. It would then inject the row of data from the preparatory sheet, as if the Xtab had always existed.

c. updateinvoice - update billing properties

The EMM syntax allows invoice properties to be updated via the parser. This can be very useful in case one wants to create virtual or allocation invoices. In this case, an automatic task could be used that would calculate virtual invoices every month, and enter the results of the calculation into real invoices stored on the relevant meter.

Syntax :

```
updateinvoice(
    METER;
    "InvoiceID";
    "Ref_Invoice_Type";
    "/FULL_PATH/TO_THE_PROPERTY";
    Property_Value;
    Invoice_FROM;
    Invoice_TO
)
```

- "METER" is the Meter on which the invoice should be located. The Meter object must be present here, i.e. not just its reference. (examples: selection, item, @(REFERENCE), @(item.column("REFERENCE")) ...)

- "InvoiceID" is the identifier of the invoice to be updated. This id is in text format, and can be a hard text or the result of a formula returning a text.

- "Ref_Invoice_Type" is the reference to the type of invoice concerned. Again, this is a text, which can be given in hard copy or as the result of a formula.

- "/FULL_PATH/TO_THE_PROPERTY" indicates the full path to the property. This means that we need to have the reference of the invoice property block in which the property is located, and then the reference of the property to be updated. Again, this parameter must be in text format.

- "Property_Value" gives the value of the property to be updated. This parameter can be of any type as long as it matches the type defined for the corresponding property.

- "Invoice_FROM" and "Invoice_TO" are the start and end dates of the invoice to be updated. So date format.

Attention: All parameters are mandatory for the function.
Warning: If the given invoice (meter, id, type, from, to) does not yet exist, the "updateinvoice" function will create it on the fly and insert the value it wanted to update.

Example:

```
updateinvoice(
    @LLN_001_ELEC;
    "VIRTUAL_ELEC_01";
    "VIRTUAL";
    "/ INV_EB/INV_EB_ADM_EAN";
    "55xxx";
    now.synchro(1; "month").add(-1; "month");
    now.synchro(1; "month")
)
```

→This function will find the invoice "VIRTUAL_ELEC_01", of type "VIRTUAL", starting at the beginning of last month and ending at the beginning of this month, and which is attached to the Meter "LLN_001_ELEC", and update the property "INV_EB_ADM_EAN" (in the block "INV_EB") to it and give it the value "55xxx".
In practice this is rare, as everything is hard-coded into the function. Instead, most of the time you will have spreadsheets preparing the values to be used for the update, and then a few columns performing the updates, one property at a time.

Example:

```
updateinvoice(
    @(item.column("REFERENCE"));
    item.column("ID");
    item.column("ADM_INVTYP");
    "/INV_ADM/INV_ADM_CONTACT_CUSTOMER";
    item.column("CONTACT_NAME");
    item.column("FROM");
    item.column("TO")

)
```

→In the case of a preparatory worksheet that calculates or prepares all the information needed to create an invoice, one might have this type of syntax. The useful information all comes from other columns in the worksheet, and the types match what is expected.

d. updateproperty - update properties

In certain circumstances, it may be useful to be able to update the properties of an entity, a Meter, or even a channel dynamically, via the parser. For this, the "updateproperty" function can be used.

Syntax :

```
updateproperty( Items_List ; "/FULL_PATH/TO_THE_PROPERTY" ; Prop_value )
updateproperty ( Items_List ; "/FULL_PATH/TO_THE_PROPERTY" ; Prop_value ; P_from ; P_to )
```

- "Items_List" is the entity, Meter or channel on which you want to update the property. It can be a list of entities if you want to update several at the same time (with the same value on the same property)

- "/FULL_PATH/TO_THE_PROPERTY" indicates the full path to the property in text format. We must therefore have the reference of the property block in which the property is located, then the reference of the property to be updated.

- "Prop_value" gives the value of the property to be updated. This parameter can be of any type as long as it matches the type defined for the corresponding property.

- Optional parameters, "P_from" and "P_to" give the start and end dates of the property's validity in the case of a property in a history block.

Caution: In the event of an update to a property which is historicised, the new value will be inserted with its validity dates in exactly the same way as during a bulk import. The values active during this period will be reduced by all or part of their validity period to make room for the new value and its validity period.

Caution: If you import a historicised property without giving it a date parameter, it will be considered as always and forever valid and will overwrite all current values of the property during history to replace it with the only new imported occurrence.

Caution: In case of an update of a property that would be multiple, a new instance of the property will be created with the new imported value.

e. updatealarm - update alarm properties

In the same way as the "updateproperty" function, the "updatealarm" function is used to update the alarm block properties of existing channels.

This feature can be very useful in the case of channels with automatic calibrations, and automatic successive refinements of alarm thresholds.

The syntax is exactly the same, except that the "updatealarm" function is used, and the items on which properties can be updated are only channels (the only object types on which alarms can be defined.

Syntax :

updatealarm(Channels_List ; "/FULL_PATH/TO_THE_PROPERTY" ; Prop_value)
updatealarm (Channels_List; "/FULL_PATH/TO_THE_PROPERTY" ; Prop_value ; P_from ; P_to)

- "Channels _List" is the **channel**(s) on which you want to update the property. Again, if you have multiple channels, the updated property will be the same for all of them, with the same value.

- "/FULL_PATH/TO_THE_PROPERTY" indicates the full path to the alarm property in text format. This means that we must have the reference of the property block in which the property is located, and then the reference of the property to be updated.

- "Prop_value" gives the value of the property to be updated. This parameter can be of any type as long as it matches the type defined for the corresponding property.

- Optional parameters, "P_from" and "P_to" give the start and end dates of the property's validity in the case of a property in a history block.

Caution: In the event of an update to a property which is historicised, the new value will be inserted with its validity dates in exactly the same way as during a bulk import. The values active during this period will be reduced by all or part of their validity period to make room for the new value and its validity period.

Caution: If you import a historicised property without giving it a date parameter, it will be considered as always and forever valid and will overwrite all current values of the property during history to replace it with the only new imported occurrence.

Caution: In case of an update of a property that would be multiple, a new instance of the property will be created with the new imported value.

16. Data correction - Application of the FWD via the parser

The "AVAL" process for "Acquisition Validation" is an automatic correction system for data entering EMM that can be activated or not, channel by channel.

Normally, AVAL is executed when data is inserted on a channel (if configured for the channel concerned). However, there may be occasions when you want to force a FWD correction pass on an already existing channel, without having to artificially re-import the data to trigger the FWD procedure.

The "processaval" function will force this FWD passage on the channel passed as an argument, for the specified period of time.

Syntax: **processaval**(Channel ; from ; to)

Example:

processaval(item.channels("MAIN");from;to)

→If the active item is a Meter (example: a selection containing a list of Meters), the "MAIN" channel of this Meter will be analysed by the FWD and the data between the "from" and "to" will be corrected.

In the case of a spreadsheet with one Meter per row, the MAIN channels of each of these Meters will be corrected by the FWD for the given period.

Obviously, channels such as start and end dates can be given in hard copy, or be the result of formulas calculated either on the fly or in another column of the current worksheet.