USERGUIDE

METRON-ENERGY MANAGEMENT MODULE (EMM (JOOL))

POWERED BY METRON

Laurent HANET

Version 1.03.9 – 24/6/2021

Table des matières

« EMM (JOOL) » est un programme permettant la collecte, l'organisation et le traitement d'un grand nombre de données liées aux énergies afin d'en tirer de l'information utile. Tous les droits intellectuels liés aux logiciels à ses illustrations et à sa documentation sont la propriété exclusive de Dapesco.

1. Notions de base du langage orienté objet

EMM (JOOL) utilise une syntaxe de programmation de type « Orienté objet ». Cela signifie que chaque élément sur laquelle on travaillera dans EMM (JOOL) sera un « objet » auquel on attribuera un type, des propriétés et des fonctions propres.

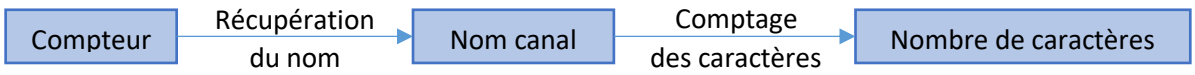
Un objet peut être une chaine de caractères, une date, ou même un compteur d'eau par exemple.

A chaque type d'objet est associée une série de fonctions qui peuvent s'y appliquer. Par exemple, la fonction .avg, qui effectue une moyenne arithmétique, est pertinente si l'on l'applique à une série de nombres. Elle l'est beaucoup moins si on tente de l'appliquer à une chaine de caractères.

Dans la pratique, EMM (JOOL) utilisera donc des objets, et les transformera en d'autres objets selon la fonction utilisée.

Exemple :

On pourrait partir d'un compteur, récupérer son nom, puis compter le nombre de lettres que contient le nom du compteur.



La fonction Récupération du nom transformera donc l'objet Compteur (de type « compteur ») en un autre objet « Nom du canal » (de type « chaine de caractères »). Pour ce faire, EMM (JOOL) ira récupérer dans les propriétés du compteur, la valeur de son « Nom », qui est une chaine de caractères.

Ensuite, la fonction Comptage des caractères ira compter le nombre de caractères dans la chaine fournie par l'étape précédente et fournira donc un objet de type « Nombre », qui sera le nombre de caractères dans le nom du compteur.

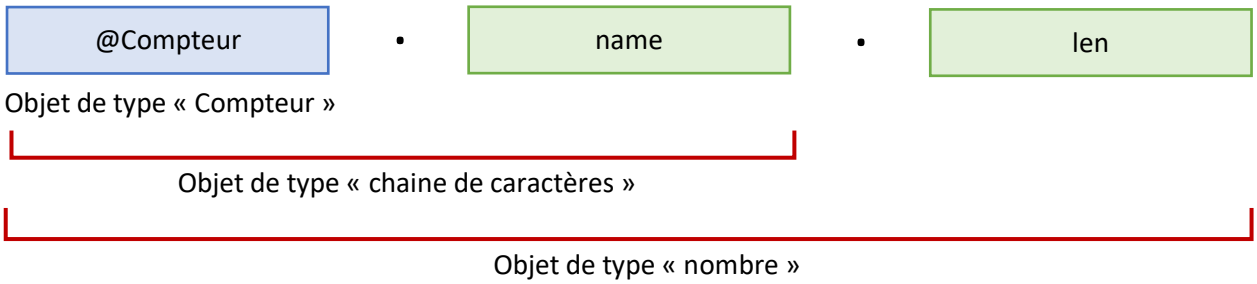
On a donc une suite d'opérations en cascade, partant toutes d'un objet et produisant un nouvel objet.

La syntaxe de EMM (JOOL) reprendra les différentes fonctions les unes derrière les autres, séparées par un point, dans le sens de la lecture.

.len étant l'opérateur qui compte la longueur d'un texte (venant de LENgth en anglais), l'exemple ci-dessus deviendra

```
@Compteur.name.len
```

Cette syntaxe fonctionnera de la manière suivante :



@Compteur = l'objet de départ, le compteur, un objet de type « Compteur »

@Compteur.name : la fonction .name s'applique à un objet de type Compteur et renvoie un objet de type « chaine de caractères », en l'occurrence le nom du compteur en question.

@Compteur.name.len : la fonction .len part quant à elle d'une chaine de caractères et renvoie un nombre, correspondant au nombre de caractères constituant la chaine.

Il est important, lors de la rédaction de codes EMM (JOOL), de toujours bien avoir à l'esprit quel est l'objet actif, pour s'assurer que la fonction suivante peut lui être appliquée, et pour suivre le raisonnement étape par étape jusqu'à la constitution de la valeur désirée.

Certaines fonctions sont simples et directes et s'appliqueront sur leur objet, sans avoir besoin d'informations complémentaires, comme les fonctions reprises ci-dessus (.name, .len …)

D'autres fonctions pourront ou devront recevoir un ou plusieurs paramètres supplémentaires pour fonctionner, comme par exemple la fonction .round qui effectue un arrondi d'un nombre décimal. Cette fonction doit recevoir un paramètre indiquant le nombre de chiffres après la virgule à garder dans l'arrondi. Ce paramètre sera alors passé à la fonction en argument. On lui indiquera la valeur du paramètre à utiliser entre parenthèses lors de son appel.

```
item.round(2)
```

→ cette expression renverra la valeur du nombre stocké dans la variable « item », arrondie à 2 chiffres après la virgule.

De nombreuses fonctions auront donc besoin d'arguments pour fonctionner. Certaines auront besoin de plusieurs arguments, qui seront alors fournis entre parenthèses, tous séparés par des points-virgules, dans l'ordre fixé par la définition de la fonction.

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

Certains paramètres peuvent en outre être optionnels et ne pas être strictement requis pour le bon fonctionnement de la formule.

On notera que les paramètres peuvent être de n'importe quel format (texte, numérique, date…), selon ce qui est requis par la fonction.

2. Fonctionnalités de base

A. Objet texte

a) Généralités

Les objets de type « texte » sont des chaines de caractères, qui peuvent être utilisées par le système pour diverses applications, comme identifier, trier, ou caractériser des compteurs par exemple.

Pour spécifier une valeur en format texte dans le système, on l'encadrera par des guillemets " ".

Si un texte doit contenir des guillemets (qui ne doivent pas être interprétés comme une clôture de texte), il faut le doubler dans le texte.

Ainsi, le texte hello "world" ! devra être encodé comme ceci : "hello ""world"" !"

Il est possible de concaténer des chaines de caractères via le symbole **+**

"hello" **+** " world" → hello world

Les textes peuvent être explicites ou être contenus dans une variable. Par exemple, item.name est un texte contenant le nom de l'élément (item) sélectionné, en format texte donc.

b) Fonctions associées aux textes

Plusieurs fonctions peuvent s'appliquer aux objets en format texte, qu'ils soient explicites, et notés entre " ", ou qu'ils soient contenus dans une variable de type texte.

.len

A partir d'une chaine de caractère, cette fonction renverra une valeur numérique indiquant la longueur de cette chaine (nombre de caractères la composant, y compris les espaces)

Exemples :

("test").**len** → 4

item.name.**len** → La longueur du nom de l'item active

.contains

Reçoit en argument une chaine de caractères et la compare au texte choisi. La fonction renverra TRUE ou FALSE (Booléen) selon que la chaine de caractère soit ou non comprise dans le texte.

On notera que la comparaison n'est pas sensible aux majuscules/minuscules.

Exemples :

("ceci est un test").**contains**("test") → TRUE

("ceci est un test").**contains**("TEST") → TRUE

("ceci est un test").**contains**("rouge") → FALSE

item.name.**contains**("rouge") → Renverra TRUE ou FALSE selon que le mot "rouge" soit ou non contenu dans le nom de l'item active

.startswith /.endswith

Reçoit en argument une chaine de caractères et la compare au texte choisi. La fonction renverra TRUE si le texte commence (.startswith) ou se termine (.endswith) par la chaine de caractère et elle renverra FALSE (Booléen) dans le cas contraire.

Une fois de plus, la comparaison est insensible aux majuscules/minuscules.

Exemples :

("ceci est un test").**startswith**("est") → FALSE

("ceci est un test").**endswith**("est") → TRUE

("ceci est un test").**startswith**("ceci") → TRUE

("ceci est un test").**startswith**("CECI") → TRUE

item.name.**startswith**("rouge") → Renverra TRUE ou FALSE selon que le nom de l'item active commence ou non par "rouge"

.left /.right

A partir du texte choisi, crée une chaine de caractères en recopiant à partir de la gauche (.left) ou de la droite (.right) le nombre donné de caractères du texte choisi.

Ces fonctions peuvent par exemple être utiles sur des références structurées qui contiendraient une référence de site concaténée avec une référence de compteur, afin de séparer les références.

Exemples :

("ceci est un test").**left** (6) → ceci e

("ceci est un test").**left** (10) → ceci est u

("ceci est un test").**right** (6) → n test

("ceci est un test").**right** (15) → eci est un test

Remarque : l'espace est toujours compté comme un caractère comme les autres.

.replace

Cette fonction sert à modifier une partie de texte et la remplacer par un autre texte.

Remarque : cette fonction **est** sensible aux majuscules/minuscules lors de la recherche de texte à remplacer.

Syntaxe :

("message à modifier").**replace**("texte à trouver/retirer"; "texte à insérer")

Exemples :

("message à modifier").**replace**("modifier"; "transformer") → message à transformer

("message à MODIFIER").**replace**("modifier"; "transformer") → message à MODIFIER

("ceci est un test").**replace**("e"; "X") → cXci Xst un tXst

B. Objet numérique

a) Généralités

Dans les bases de données de EMM (JOOL), les valeurs numériques sont stockées avec un point comme séparateur décimal, et sans séparateur de milliers, et ce quelles que soient les préférences culturelles de l'utilisateur.

On doit bien différencier les objets numériques tels qu'utilisés dans le programme, et qui sont stockés et utilisés avec un point et sans séparateurs de milliers, et les objets numériques finaux, présentés à l'utilisateur dans ses grids ou graphiques, et qui, eux, sont affichés selon ses préférences culturelles. Utiliser une valeur numérique décimale dans une formule de la syntaxe se fera donc en utilisant le point (.) comme séparateur décimal.

Les valeurs numériques dans EMM (JOOL) sont encodées au format « double » (format standard IT, où les nombres sont codés sur 64 bits).

EMM (JOOL) supporte les opérations mathématiques de base (+ - * /), qu'elles soient utilisées avec des valeurs numériques explicites ou via des variables contenant des valeurs de type numérique.

b) Valeurs prédéfinies

Certaines valeurs sont fréquemment utilisées et sont donc pré-encodées dans EMM (JOOL).

pi = 3.14159265358979

null = la valeur vide (souvent utilisée dans des formules pour tester si une donnée est vide)

rand = renvoie une valeur aléatoire comprise entre 0 et 1

rand(10) = renvoie une valeur aléatoire comprise entre 0 et 10. L'argument que l'on passe à la fonction lui indique la valeur maximale autorisée pour la génération du nombre aléatoire.

c) Fonctions associées aux valeurs numériques

Les fonctions suivantes s'appliqueront aux objets de type numérique, qu'ils soient explicitement indiqués, ou qu'ils soient contenus dans une variable de type numérique comme une valeur de consommation.

.round

Cette fonction s'applique sur une valeur numérique et renvoie cette même valeur, arrondie à un nombre de décimales indiqué par l'argument fourni.

L'argument doit être un nombre entier, et indiquera donc le nombre de chiffres gardés après la virgule.

Exemples :

pi.**round**(4)	→	3,1416
item.value.**round**(4)	→	Renverra la valeur de l'item arrondie à la 4° décimale

.floor /.ceiling

Ces fonctions reçoivent une valeur numérique, et fournissent la valeur arrondie à l'entier inférieur (floor) ou supérieur (ceiling)

Exemples :

(4.8439).**ceiling**	→	5
(-2.156).**ceiling**	→	-2
(4.8439).**floor**	→	4
pi.**floor**	→	3
item.value.**floor**	→	Renverra la valeur de l'item, arrondie à l'entier inférieur

Remarque : les parenthèses autour des valeurs numériques ne sont pas indispensables, mais elles peuvent clarifier la lecture, et elles évitent que l'on confonde le point décimal et le point séparant les fonctions successives du langage orienté objet.

Reste de la division entière (modulo)

La division modulo utilise simplement le symbole « % » à l a place de « / ».

Exemples :

5 % 3	→	2

.abs

Reçoit une valeur numérique et renvoie sa valeur absolue.

Exemples :

(-315.4563).**abs**	→	315,4563
(42.15) .**abs**	→	42,15

item.value.**abs** → Renverra la valeur absolue de la valeur de l'item

.sqrt

Reçoit une valeur numérique et renvoie sa racine carrée positive (square root).

Exemples :

16.**sqrt** → 4

(16.5).**sqrt** → 4,06201820231798

item.value.**sqrt** → Renverra la racine carrée positive de la valeur de l'item

.power

Elève une valeur numérique à la puissance passée en argument. L'exposant ne doit pas forcément être entier ni même positif.

Exemples :

3.**power**(2) → $3^2 = 9$

2.**power**(-1) → $2^{-1} = 0,5$

4.**power**(2.1) → $4^{2,1} = 18,3791736799526$

item.value.**power**(2) → renverra le carré de la valeur de l'item

.exp /.log /.log10

Ces fonctions renverront respectivement l'exponentielle, le logarithme népérien (ln) ou le logarithme en base 10 de la valeur numérique sur laquelle ils s'appliquent.

Exemples :

3.**exp** → $e^3 = 20,0855369231877$

12.**log** → $\ln(12) = 2,484906649788$

1000.**log10** → $\log_{10}(1000) = 3$

item.value.**exp** → renverra $e^{\text{la valeur de l'item}}$

Remarque avancée : Ces fonctions peuvent s'appliquer à des valeurs numériques mais aussi à des séries de valeurs, profils de données ou collections de numériques (voir définitions plus loin). Dans ce cas, elles fourniront une nouvelle série de valeur contenant les exp, log ou log10 des valeurs de la série initiale.

Collection de numériques (exemple avancé) :

(100; 10; 100000).**log10** → 2, 1, 5

Profil de données (exemple avancé) :

item.data.**exp** → Renvoie la liste des exponentielles de chaque valeur du profil

.sin /.cos /.tan

Ces fonctions renverront respectivement le sinus, le cosinus ou la tangente de la valeur numérique sur laquelle ils s'appliquent.

Exemples :

(2).**sin** → sin(2) = 0,909297426825682

(pi/3).**cos** → cos(pi/3) = 0,5

item.value.**tan** → renverra la tangente de la valeur de l'item

Remarque avancée : Ces fonctions peuvent s'appliquer à des valeurs numériques mais aussi à des séries de valeurs, profils de données ou collections de numériques. Dans ce cas, elles fourniront une nouvelle série de valeur contenant les sin, cos ou tan des valeurs de la série initiale.

Collection de numériques (exemple avancé) :

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

Profil de données (exemple avancé) :

item.data.**sin** → Renvoie la liste des sinus de chaque valeur du profil

.asin /.acos /.atan

Ces fonctions renverront respectivement l'arcsinus, l'arccosinus ou l'arctangente de la valeur numérique sur laquelle ils s'appliquent.

Les valeurs passées à asin et acos devront être comprises entre -1 et 1 sous peine d'être incalculables.

Exemples :

(0.5).**asin** → pi/6 = 0,5235987755982988

1.**acos** → 0

item.value.**atan** → renverra l'arc dont la valeur de l'item est la tangente

Remarque : Ces fonctions peuvent s'appliquer à des valeurs numériques mais aussi à des séries de valeurs, profils de données ou collections de numériques (voir définitions plus loin). Dans ce cas, elles fourniront une nouvelle série de valeur contenant les asin, acos ou atan des valeurs de la série initiale.

Collection de numériques (exemple avancé) :

(0.5; 0; 1).**acos** → 1.0471975511965979, 1.5707963267948966, 0

Profil de données (exemple avancé) :

item.data.**asin** → Renvoie la liste des arcsinus de chaque valeur du profil

.format

Cette fonction permet de donner une certaine forme aux nombres pour leur affichage final dans un tableau ou dans un rapport par exemple. Elle s'applique sur un numérique et renvoie une chaine de caractères (texte) dans le format demandé. Le code indiquant la forme du format est passé en argument à la fonction.

Attention : le résultat de cette fonction étant un objet de type « texte », il ne sera plus récupérable (sauf cas exceptionnels) pour effectuer des calculs ultérieurs.

Le code format passé en argument à cette fonction sera un texte, reflétant la structure que l'on devra utiliser pour afficher le nombre. Ce format utilise certains caractères bien définis pour symboliser les composants du nombre final.

- le point représente le séparateur décimal
- les virgules représentent les séparateurs de milliers
- " # " est utilisé pour matérialiser un chiffre, **SI** il y'en a un à afficher.
- " 0 " sert à forcer l'affichage d'un chiffre, et affichera un zéro s'il n'y a pas de valeur à afficher.
- Les autres caractères seront recopiés tels quels dans le résultat final.

Les symboles effectivement utilisés comme séparateurs décimaux et de milliers dans le résultat affiché seront ceux définis par les paramètres utilisateurs.

Exemples :

(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

Remarques :

- Les # avant la virgule ne limitent pas le nombre de chiffres affichés (voir avant-dernier exemple)
- Les # après la virgule, eux, définissent le nombre de chiffres affichés après la virgule (voir premiers exemples)

C. Objet date

a) Généralités

Dans le système EMM (JOOL), une date est un type d'objet reprenant une année, un mois, un jour, une heure, des minutes, des secondes et des millisecondes.

Ces objets seront très fréquemment utilisés dans EMM (JOOL), notamment dans les constitutions de profils de données, composés de couples « Date-Valeur ».

Dans la base de données EMM (JOOL), les dates sont stockées en GMT, et elles sont ensuite affichées à l'utilisateur une fois transposées dans leur fuseau horaire.

Il est possible d'encoder une date directement en dur dans une case, via le format suivant :

"2015-03-01 15:12:05.020" → premier mars 2015, 15h12 et 05 secondes, 20 millièmes.

Cependant, le format étant assez strict, et la date étant d'office considérée comme en GMT, cela rend cette utilisation assez rébarbative. On lui préfèrera l'utilisation de la fonction dateeval (décrite un peu plus loin) qui permettra de créer dynamiquement une date à partir de ses composants, et directement en heures locales.

Afficher un objet date sous forme de texte dans une feuille de calcul renverra la donnée stockée en dur dans la base de données, et son format n'est pas des plus clairs. On préfèrera donc à chaque fois utiliser le format « Date » pour visualiser des dates.

b) Dates prédéfinies

Il existe dans le système des valeurs de dates qui sont créées automatiquement à la demande.

from	Date de départ du contexte temporel actif
to	Date de fin du contexte temporel actif
now	Date actuelle (année, mois, jour, heure, minute, seconde, milliseconde)
today	Date d'aujourd'hui, à minuit ce matin (= now.synchro(1;"day"))

Pour rappel, le contexte temporel est la période de temps qui a été sélectionnée dans le sélecteur temporel, dans le haut de la fenêtre EMM (JOOL) utilisée. Dans le cas particulier d'une tâche automatique, tournant

sans que EMM (JOOL) ne soit ouvert, et donc sans contexte sélectionné, le contexte actif sera défini comme la période entre

- from = maintenant moins le pas de temps configuré pour la tâche
- to = maintenant, date de l'exécution = now.

Quand on se trouve dans la case From ou To d'un éditeur (feuille de calcul, rapport HTML…), les mots clés from et to renverront aux dates de début et de fin du contexte temporel affiché dans le sélecteur. Ces cases From ou To permettent le cas échéant de modifier le contexte actif via une formule, localement, pour les besoins de l'outil en cours. On pourra donc par exemple transposer les dates une semaine en arrière, pour pouvoir, dans un tableau de bord basé sur la semaine en cours, afficher dans l'un de ses widgets, la consommation de la semaine passée.

Une fois à l'intérieur de la définition de l'outil (tableau, rapportHTML…), les mots clés from et to renverront alors vers le début et la fin du contexte local, résultat des modifications éventuelles effectuées dans les cases From et To.

Remarque : Comme nous le verrons plus tard, un contexte temporel différent du contexte actif pourra être imposé lors de l'appel d'un tableau de données, ou via utilisation d'un execute ou lors de l'appel à un tableau. Ces contextes seront alors substitués au contexte du sélecteur. Tout le reste du raisonnement précédent restant similaire.

Remarque : Dans le même ordre d'idées, certaines fonctions peuvent recevoir des paramètres optionnels incluant un contexte from/to. Ces fonctions limiteront alors leurs résultats à ce qui est situé dans la période définie par ce contexte temporaire.

c) Fonctions associées aux dates

Les objets de type date peuvent être utilisés ou modifiés par des fonctions, afin de synchroniser des séries de données par exemple, ou de retrouver une autre date à partir de la date de départ.

dateeval

Construit un objet de type date à partir de ses composants numériques (année, mois…). Cette fonction permet de composer une date sans avoir à se soucier des détails du format. Il est fortement conseillé d'avoir recours à cette fonction quand il est nécessaire d'encoder une date manuellement en dur.

On remarquera que cette formule ne s'applique pas à un objet déjà existant. Elle construit une date sur base des paramètres que l'on lui fournit.

Syntaxe :

dateeval(année ; mois ; jour ; heure ; minute ; seconde)

Les périodes de temps sont encodées en valeurs numériques (pas besoin de "), et outre le premier paramètre obligatoire, tous les autres sont optionnels. Si l'on ne donne pas toutes les valeurs en paramètre (par exemple, si l'on s'arrête aux heures), la date constituée sera celle du début de la période associée.

Exemples :

dateeval(2015; 12; 20; 6)	→	Le 20 décembre 2015 à 06:**00**:**00**
dateeval(2015; 12; 20)	→	Le 20 décembre 2015 à **00**:**00**:**00**
dateeval(2015; 12)	→	Le **1°** décembre 2015 à **00**:**00**:**00**
dateeval(2015)	→	Le **1° Janvier** 2015 à **00**:**00**:**00**

.add

Ajoute une durée définie à une date donnée. La fonction reçoit 2 paramètres : le nombre d'intervalles de temps, et le type d'intervalle de temps, afin de lui indiquer la durée à ajouter, et fournit une valeur en format date.

Syntaxe :

.**add**(nbr_intervalles; "type_intervalle")

- nbr_intervalles : indique le nombre d'intervalles à ajouter (valeur numérique)
- "type_intervalle" : indique le type d'intervalle. Format texte (donc entre " "), à choisir parmi year, month, day, hour, minute ou second

Remarque : Au lieu de créer une autre fonction pour soustraire une durée, la fonction .add accepte des durées négatives et effectuera alors une soustraction de durée.

Exemples :

now.**add**(1; "day")	→	demain à la même heure (+1 jour)
now.**add**(1; "month")	→	le mois prochain, même jour, même heure (+1 mois)
from.**add**(-1; "year")	→	Un an avant la date de départ du contexte actuel
to.**add**(2; "month")	→	Deux mois après la date de fin du contexte actuel

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

→ renverra la date du 12 février 2015, 00:00:00 (10 février + 2 jours)

.synchro

Modifie une date pour la ramener au début d'une période donnée. Permet par exemple d'obtenir la date de début du mois ou de début de l'année contenant la date sélectionnée dans le contexte.

Syntaxe :

.**synchro**(taille_intervalle ; "type_intervalle")

- taille_intervalle : indique la taille de l'intervalle considéré (valeur numérique)
- "type_intervalle" : indique le type d'intervalle. Format texte (donc entre " "), à choisir parmi year, month, day, hour, minute ou second

Exemples :

now.**synchro**(1; "day") → Ce matin, 00 :00:00

now.**synchro**(1; "month") → Minuit, le matin du premier jour du mois en cours

now.**synchro**(3; "month") → Minuit, le matin du premier jour du trimestre en cours

now.**synchro**(1; "year") → Minuit, le matin du premier de l'an de cette année

to.**synchro**(1; "month") → Minuit, le matin du 1° jour du mois de fin du contexte

dateeval(2015; 2; 10).**synchro**(1; "month") → Le premier février 2015, 00 :00:00

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

Renvoie le **nombre** (valeur numérique) de l'année, mois, jour, heure, minute ou seconde de la date donnée (aucun paramètre supplémentaire)

Exemples :

now.**hour** → Le chiffre de l'heure actuelle

to.**year** → L'année de la fin du contexte

from.**month** → Le chiffre du mois du début du contexte

dateeval(2015; 12; 20).**month** → 12

.weekday / .day365

Similaire à la fonction .day, la fonction .weekday renverra le numéro du jour de semaine associé à la date donnée : 1 pour lundi, 2 pour mardi … et 0 pour dimanche.

La fonction .day365 renverra quant à elle le numéro d'ordre du jour donné dans l'année (sur 365/366 jours)

Attention : Dans le cas des années bissextiles, la fonction .day365 comptera le 29 février. Le 1° mars n'aura donc pas la même valeur que l'on soit dans une année bissextile ou non. Ce détail a de l'importance si l'on essaie de comparer l'avancement relatif dans l'année entre deux années différentes par exemple.

.format

Cette fonction part d'une date et génère un texte, traduisant la date donnée selon le format que l'on passe en argument.

Le format de time/date passé en paramètre est de type texte (donc entre " ") et suit les formats de c#.

Code	Période	Explication	Exemple
G	Ère	L'ère	A.D.
yy	Année	Les deux derniers chiffres de l'année	15
yyyy	Année	Les 4 chiffres de l'année	2015
M	Mois	Le mois	1 - 12

MM	Mois	Le mois sur 2 chiffres	01 - 12
MMM	Mois	Le mois en texte abrégé (localisé selon l'utilisateur)	Oct
MMMM	Mois	Le nom du mois en entier (localisé selon l'utilisateur)	Octobre
d	Jour	Jour du mois	1 - 31
dd	Jour	Jour du mois avec 2 chiffres	01 - 31
ddd	Jour	Nom du jour abrégé (localisé selon l'utilisateur)	lun.
dddd	Jour	Nom du jour en entier (localisé selon l'utilisateur)	Lundi
h	Heure	L'heure en format 12h	1 - 12
hh	Heure	L'heure en format 12h sur 2 chiffres	01 - 12
H	Heure	L'heure en format 24h	0 - 23
HH	Heure	L'heure en format 24h sur 2 chiffres	00 - 23
m	Minute	La minute	0 - 59
mm	Minute	La minute sur 2 chiffres	00 - 59

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

Exemples :

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-MMMM-yyyy") → ven. 16-Octobre-2015

datediff

Cette fonction reçoit deux dates et un type de pas de temps (texte) en argument et elle construit une valeur numérique comptant le nombre de pas de temps de ce type entièrement entre les deux dates.

On remarquera que cette formule ne s'applique pas à un objet déjà existant. Elle se construit uniquement sur base de ses paramètres.

Syntaxe :

datediff(date1; date2; "day")

→ Renverra un numérique = le nombre de jours entiers entre la date1 et la date2.

Les dates peuvent être entrées en dur en format texte (déconseillé), ou en utilisant dateeval, ou sur base d'une formule : from, to, now, from.synchro(1; "day") …

Exemple :

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

→ Renverra le nombre de jours entiers depuis le début du mois en cours.

daysinyear / daysinmonth

Fonctions recevant une date en argument, elles renverront le nombre de jours présents dans l'année / le mois contenant la date donnée.

On remarquera que cette formule ne s'applique pas à un objet déjà existant. Elle se construit uniquement sur base de ses paramètres.

Exemples :

daysinyear(now)

→ Renverra un numérique indiquant le nombre de jours dans l'année contenant la date « now ». Basiquement, cela renverra 365 ou 366 selon que l'on soit ou non dans une année bissextile.

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

→ Renverra 365. En effet, l'année 2015, contenant la date du 20/12/2015 6:00:00, n'est pas une année bissextile et contient donc 365 jours.

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

daysinmonth(dateeval(2017; 2)) → 28

daysinmonth(column("FROM"))

→ Renverra le nombre de jours contenus dans le mois qui contient la date présente dans la colonne « FROM ». Si la colonne contient autre chose qu'une date, la fonction renverra une erreur.

D. Objet logique (Booléens)

a) Généralités

Les valeurs true et false représentent les valeurs logiques « vrai » et « faux » et peuvent être utilisées directement dans le système. Ces valeurs sont souvent le résultat de tests logiques (avec if, contains, startswith… par exemple), et sont rarement écrites explicitement dans une formule.

b) Fonctions et connecteurs logiques

Sont reprises ici toutes les fonctions qui génèrent, utilisent, ou combinent une ou plusieurs valeurs logiques.

> < = >= <= <>

Les opérations logiques peuvent être utilisées directement dans la syntaxe. Elles effectuent la comparaison entre leurs arguments et renvoient une valeur logique utilisable par la suite dans une fonction logique comme un if par exemple.

Les opérations disponibles sont les suivantes :

>	plus grand

<	plus petit
=	égal
<=	plus petit ou égal
>=	plus grand ou égal
<>	différent

Exemples :

2 < 1 → false

2 > 1 → true

item.value = 1 → Renvoie true si l'item vaut 1 et false dans le cas contraire

.not

Très basique, cette fonction permet d'inverser la valeur d'un Booléen. Ainsi, une valeur true deviendra false, et une valeur false deviendra true

(2 < 1).**not** → true

(2 > 1).**not** → false

(item.value = 1).**not** → Renvoie false si l'item vaut 1 et true dans le cas contraire

.isnull / .isnotnull

Ces fonctions reçoivent une valeur et vérifient si elle existe ou si elle est vide, ce qui correspond à la valeur informatique null.

Rappel : Null ≠ 0. Null signifie que la variable n'existe pas, alors que 0 indique que la variable existe, et que sa valeur est de zéro. Nous avons ici affaire à la même nuance qu'entre une consommation dont on n'a pas reçu la valeur (null) et une consommation effectivement à zéro sur une période donnée (0).

Exemples :

(null).**isnull** → true

(0).**isnull** → false (0 n'est pas "vide", c'est la valeur numérique zéro)

(item.value).**isnotnull**

→ Renvoie true si la valeur de l'item existe et false dans le cas contraire

and / or

Les mots clés and et or sont reconnus par le système pour combiner des tests ou des conditions. Ils correspondent au "et" et "ou" logiques et produiront une valeur logique à partir de deux valeurs logiques de départ ou de deux conditions.

Exemples :

true **and** true	→	true
true **and** false	→	false
(2 < 1 **or** 4 > 3)	→	true

Exemple avancé :

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

→ Renverra les entités sélectionnées dont le nom commence par A ou B

if

Dans le cas où l'on veut récupérer conditionnellement une valeur ou une autre en fonction d'une expression, on peut utiliser une condition « if ».

L'expression conditionnelle doit fournir comme valeur true ou false, mais ces valeurs peuvent être le résultat d'une formule plus complexe.

Syntaxe :

if(condition ; valeur_si_oui ; valeur_si_non)

Exemples :

if (true; 1; 0)	→	1
if (2 < 1; 1; 0)	→	2 < 1 = false, donc la condition if va renvoyer 0

Exemples avancés :

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

→ L'expression conditionnelle vérifie si la valeur de l'item en cours est plus grande que zéro. Si oui, la condition if renverra le texte "ok" et dans le cas contraire, elle renverra le texte "zero".

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

→ L'expression conditionnelle vérifie si les données du compteur bxl sont vides. Dans ce cas, on prend un profil par défaut, et dans le cas contraire (si les données du compteur ont bien été reçues par exemple), on prend les données réelles.

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

→ La condition va chercher la valeur du paramètre d'alarme "//MAX_ACTIVE", qui est un Booléen (la propriété a été définie de la sorte). On utilise alors directement cette valeur Booléenne comme résultat de la condition et la fonction renverra "Alarm Max ON" ou "Alarm Max OFF" selon que l'alarme soit active ou pas.

Dans cette condition, les deux expressions suivantes auront donc le même effet :

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

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

Remarque avancée : Si la valeur Booléenne provient d'une propriété, l'utilisation de la fonction .not doit être bien comprise. En effet, une propriété peut avoir comme valeur, non seulement true et false, mais aussi null, dans le cas où la propriété n'est pas définie.

Dans ce cas, il convient de ne pas oublier que true.not ≠ false

En effet, true.not indique simplement que la valeur n'est pas true. Elle peut donc être false **ou** null.

3. Entités / Compteurs / Canaux et Propriétés

A. Objets entités et compteurs

a) Accéder à une entité ou un compteur

Dans les formules de EMM (JOOL), une entité ou un compteur sera identifié par @Reference. Le symbole « @ » signale que l'on parle d'une entité/compteur, et sa référence unique permet de l'identifier. Cette syntaxe permet d'accéder directement à une entité/compteur pour autant que l'on en connaisse la référence.

Syntaxe alternative : @(item.column("site_reference")). Cette expression permet de récupérer une entité/compteur en passant directement par sa référence, même si elle n'est pas connue au moment de la rédaction du code ou si elle doit changer dynamiquement en fonction de la situation. On peut la voir comme la fonction « @ » qui reçoit en argument une référence (texte), et qui renverra l'objet (entité/compteur) ayant cette référence.

Une autre manière de désigner une entité/compteur est d'utiliser le mot clé selection. Ce terme désignera l'entité/compteur (ou les entités/compteurs) qui est (sont) actuellement reprise(s) dans la sélection active. La sélection dépend de l'outil dans lequel on travaille. Si l'on est dans une feuille de calcul, il s'agira du résultat de la formule

Par ailleurs, le terme all renverra une collection contenant l'ensemble des entités/compteurs existantes (pas uniquement ceux repris dans l'arborescence), et ce, quelle que soit la sélection active dans le sélecteur de canaux.

Remarque : Ce terme all inclura aussi bien les entités/compteurs visibles dans la vue du sélecteur d'entités que les éventuelles entités/compteurs qui ne seraient pas affichées. En effet, pour rappel, les vues affichées dans le sélecteur peuvent être personnalisées et peuvent donc afficher l'intégralité des entités/compteurs existantes, ou bien un sous-ensemble plus restreint.

Remarque : Le terme all se bornera cependant aux entités/compteurs sur lesquelles l'utilisateur actif a des droits en lecture et/ou en écriture. En effet, celles auxquels il n'a pas accès restent cachées, même s'il utilise le mot clé all.

Enfin, il est aussi possible d'accéder à l'ensemble des entités/compteurs d'un type donné via la syntaxe suivante : #REF_DU_TYPE. Le symbole « # » indique que l'on désire récupérer une liste d'entités/compteurs, et la référence de type que l'on accole à ce symbole permet à EMM (JOOL) de filtrer les entités/compteurs pour ne garder que celles qui sont du type demandé.

Exemple : #SITE renverra la liste des entités de type « SITE »

Cette syntaxe est beaucoup plus rapide que si l'on passait par un « all.where(item.type="SITE") ». En effet, cette dernière syntaxe récupère en DB la totalité des entités/compteurs existantes, et les teste toutes sur leur type, alors que la syntaxe #SITE récupère directement la bonne liste d'entités en DB.

b) Fonctions associées aux entités et compteurs

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

Ces fonctions de la syntaxe EMM (JOOL) permettent de récupérer les informations d'identification des entités/compteurs, qu'il s'agisse du nom, de la référence, ou même de l'icône affichée dans le sélecteur.

.reference	Renvoie un texte (string) contenant la référence unique de l'entité/compteur (Cette fonction peut également s'appliquer à plusieurs autres types d'objets existant dans EMM (JOOL), et renverra la référence unique de l'objet en question)
.name	Renvoie un texte contenant le nom de l'entité/compteur
.icon	Renvoie un texte contenant le nom de l'icône de l'entité/compteur (hors extension) Une fois sur l'icône, il est alors possible de spécifier que l'on veut le nom de l'icône ou sa couleur via les fonctions suivantes : .icon.**value** → Renverra le nom de l'icône (ce que fait déjà « .icon » par défaut) .icon.**color** → Renverra le code hexadécimal de la couleur de l'icône
.creation .lastupdate	Ces deux fonctions renverront les informations de création ou de dernière modification de l'objet concerné. Sur ces informations, on peut alors récupérer l'utilisateur ou la date de cette création ou de cette mise à jour. item.creation.date → renverra la date de la création de l'item item.lastupdate.date → renverra la date de la dernière mise à jour de l'item Attention, si l'on récupère l'objet « utilisateur » associé à l'item, il faudra ensuite indiquer ce que vous voulez récupérer comme information sur cet utilisateur. item.creation.user.name → renverra le nom du créateur de l'item item.lastupdate.user.mail → renverra l'adresse email de l'utilisateur qui a effectué la dernière mise à jour de l'item concernée. **Remarque** : ces fonctions s'appliquent également sur d'autres types d'objets, comme les événements, les factures… ainsi qu'à des outils comme les DataSets. Dataset("DSET_LHT_TEST").creation.user.mail → renverra donc bien l'adresse email de l'utilisateur qui a créé ce DataSet.
.type	Renvoie le type de l'entité/compteur active

Exemples :

@C1.**reference**	→	"C1" la référence de l'entité/compteur demandée
selection.**name**	→	renvoie le nom de l'entité/compteur en sélection
selection.**icon**	→	"lightning" dans le cas où l'entité/compteur a pour icone un éclair

selection.**type** → "SITE" dans le cas où l'entité active est de type « SITE »

.parent / .children / .ancestors / .descendants

Dans la syntaxe, afin d'accéder, à partir d'une entité/compteur, aux autres entités/compteurs reliées par des liens de parenté, on peut utiliser les formules suivantes :

.parent	Déplace l'entité/compteur active sur le parent direct, immédiatement au-dessus de l' entité/compteur actuelle dans l'arborescence structurelle. **Remarque :** une entité/compteur ne peut avoir qu'un seul parent.
.children	Crée une liste (collection) de tous les enfants directs de l'entité/compteur actuelle. **Remarque :** une entité/compteur peut avoir plusieurs enfants, d'où l'intérêt d'une liste
.ancestors	Crée la liste de tous les parents et parents de parents jusqu'à la racine de l'arborescence. **Remarque :** cette liste se fait en partant de l'entité/compteur actuelle, et en s'en éloignant. La dernière entité/compteur de la liste ainsi générée sera alors l'ancêtre le plus éloigné, donc la racine de l'arborescence.
.descendants	Crée une liste de toutes les entités/compteurs sous la sélection actuelle, dans l'arborescence, en commençant par les enfants directs (.children) puis en continuant récursivement sur tous les enfants des enfants et ainsi de suite jusqu'à épuisement de l'arborescence.

Exemples :

Si l'arborescence est la suivante (en imaginant que les noms et les références des entités/compteurs soient similaires, et en supposant que l'arborescence réelle corresponde à la vue active)

@CPT _GYMNASE.**parent**.name → "ECOLE", le nom du parent

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

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

@ECOLE.**children**

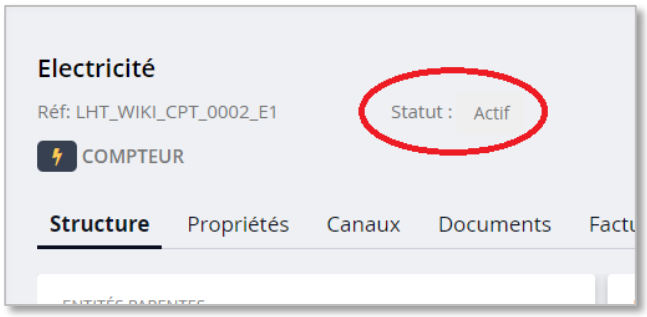
→ Renverra la liste des entités directement sous l'entité « ECOLE ». En l'occurrence : Chaufferie ; CPT_GYMNASE ; CPT_LYCEE ; CPT_THERMIQUE; Cuisine

@CPT_ELEC_LYCEE_1.**ancestors**

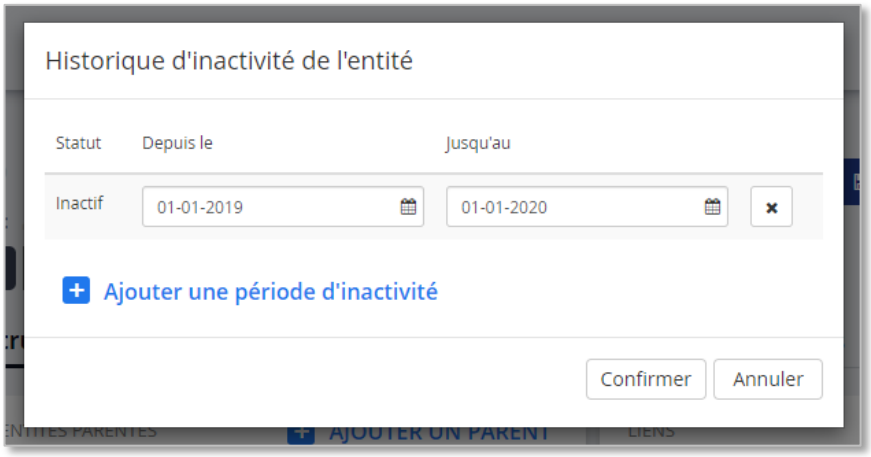
→ Renverra la liste des parents et grands-parents… au-dessus de l'entité "CPT_ELEC_LYCEE_1". En l'occurrence : CPT_LYCEE ; ECOLE (dans cet ordre-là)

.status

Sur la fiche de chaque entité/compteur, on peut trouver un bouton « Statut » indiquant le statut actuel de cette entité/compteur.



En mode édition, ce bouton permet d'ajouter des périodes d'inactivité si l'entité/compteur doit être désactivé(e) pour une certaine période (contrat en pause, site en travaux…)



Ces informations de statut (actif/inactif, et de quand à quand) sont récupérables dans les DataSets, mais aussi via les formules du parseur grâce à la fonction « .status ». Cette fonction s'utilise en association avec les fonctions « .isactive », « .from » ou « .to » pour avoir les détails du statut. La fonction « .status » accepte en outre des paramètres optionnels de date permettant de cibler la période sur laquelle on veut des informations de statut.

Syntaxe :

@ENTITY.**status.isactive**

→ Renverra « true » si l'entité/compteur est **actuellement** active, et « false » dans le cas contraire.

@ENTITY.**status.from**

→ Renverra la date de démarrage du statut actuel. Depuis quand l'entité/compteur se trouve dans son statut actuel. (si aucune date de départ renseignée, renverra le 1/1/0001)

@ENTITY.**status.to**

→ Renverra la date de fin du statut actuel. Depuis quand l'entité/compteur se trouve dans son statut actuel. (si aucune date de fin renseignée, renverra le 31/12/9999)

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

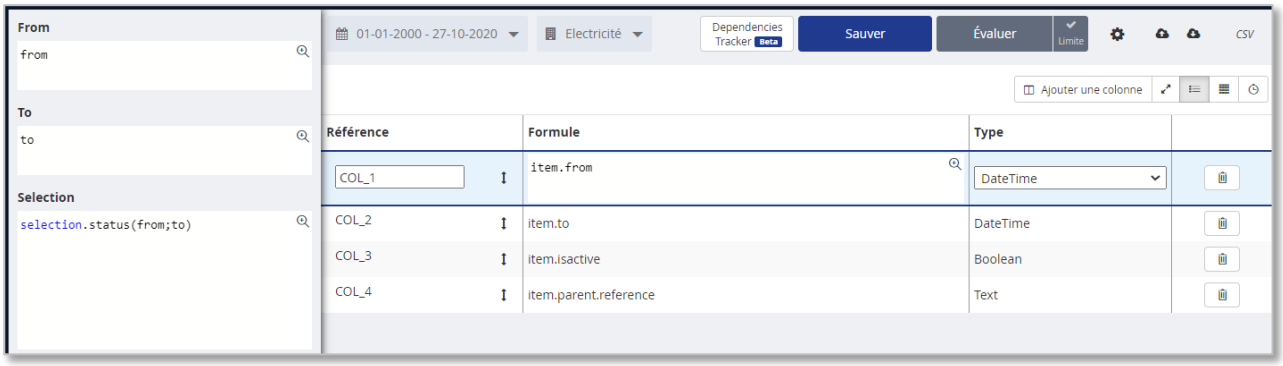
→ Renverra un Booléen indiquant si le statut est actif pendant la période (from/to) renseignée. Dans le cas où on aurait plusieurs statuts différents pendant la période indiquée, la formule renverra une liste de Booléens. Evidemment, les dates (from/to) peuvent être remplacées par d'autres formules renvoyant des dates (dateeval par exemple) .

@ENTITY.**status.parent**

→ Renverra l'entité/compteur à laquelle appartient le statut en cours.

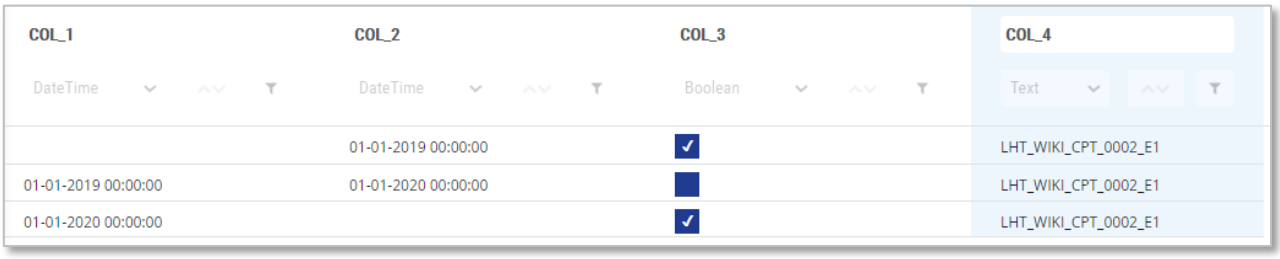
Exemple d'utilisation pratique

Si l'on veut récupérer l'ensemble des informations de statut d'une entité/compteur, on peut travailler comme suit.



La sélection contient donc une liste d'objets de type « statut », récupérés sur la période couverte par le contexte actif (from/to). On aura donc une ligne par statut dans le tableau de résultats, et pour chaque ligne, on a créé des colonnes contenant les formules « item.from », « item.to », « item.isactive », et « item.parent.reference ». Le mot clé « item » concernant ici à chaque ligne un statut pris par l'entité/compteur.

Le résultat d'un tel worksheet pourrait alors être le suivant :



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

Remarque avancée

Il est possible d'importer massivement les statuts d'entités/compteurs. La méthode est similaire aux imports de propriétés historisées, avec une ligne à importer par statut, et les champs « Statut » (Booléen), « Statut depuis » et « Statut jusqu'au » (dates).

.link

Si l'on veut, en démarrant d'une entité/compteur sélectionnée, choisir une autre entité/compteur qui lui est associée (via un lien logique) comme sélection active, on peut utiliser la fonction .link

Cette fonction part donc d'une entité/compteur, et prend comme argument un texte indiquant quel type de lien suivre.

Remarque : les liens logiques pouvant être multiples (liens 1:N), il est possible que cette syntaxe renvoie plusieurs entités/compteurs comme résultat.

Remarque : Utiliser la fonction .link sans lui passer aucun argument renverra la liste de toutes les entités/compteurs liées, quel que soit le type de lien.

Exemples :

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

→ Permettra d'accéder à un compteur partagé qui, dans l'arborescence, est relié à l'entité sélectionnée par un lien de type TEMPERATURE (paramètre en format texte).

```
selection.link
```

→ Renverra la totalité des entités/compteurs reliées à la sélection active, et ce pour tous les types de liens présents.

ATTENTION : La syntaxe ci-dessus renvoie comme élément actif l'entité/compteur liée elle-même. Si l'on veut récupérer la référence de l'entité/compteur cible, on doit alors le préciser

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

.xlink – liens distants

Dans de nombreuses situations, il est utile de récupérer une entité/compteur associée à notre sélection active, mais dont le chemin d'accès est long ou souvent variable.

L'exemple le plus fréquent est la récupération du compteur météo à partir d'un compteur sélectionné. Généralement le compteur météo est relié à l'entité site, et non à chacun de ses compteurs. Du coup, en partant d'un compteur, la récupération du compteur météo passerait par la syntaxe suivante

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

Cependant, cette formule est un peu trop rigide. En effet, elle est uniquement fonctionnelle dans le cas d'un compteur directement lié à l'entité qui est elle-même liée au compteur météo. Si l'on part d'un sous-compteur ou que le site directement lié est une sous-entité de celle qui est liée au compteur météo, il va falloir adapter la formule. Or, adapter une telle formule pour qu'elle prenne en compte tous les cas de figure possible peut rapidement devenir très lourd, tant au niveau codage de la formule que de son exécution.

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

…et encore, cette syntaxe ne couvre pas forcément tous les cas possibles.

Pour pallier à ce problème, la fonction .xlink a été créée. Celle-ci fonctionne comme la fonction .link, à ceci près que si elle ne trouve pas le lien attendu directement sur la sélection active, elle va chercher sur les entités/compteurs liées à la sélection, progressant par liens successifs jusqu'à récupérer le lien recherché.

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

Cette formule ira récupérer le compteur météo lié à la sélection, même si le lien n'est pas direct. La fonction xlink vérifiera donc si la sélection a un lien météo elle-même. Si oui, elle le récupère et s'arrête là. Dans le cas contraire, elle va aller vérifier si les entités/compteurs liées à la sélection ont un lien météo. Si elle trouve un tel lien, elle le récupère et s'arrête là, et sinon, elle continue sur les liens suivants…

Remarque : l'ordre de parcours des liens pour rechercher le xlink désiré est prédéterminé par l'administrateur EMM (JOOL).

.ismaster – détection des entités/compteurs abonnés

Il peut par moment être utile de pouvoir différencier une entité/compteur provenant d'un abonnement d'une autre entité/compteur présente uniquement dans la base de données client. La fonction « .ismaster » a été créée dans ce but.

Elle s'applique simplement à une entité/compteur et renvoie un Booléen indiquant si oui (true) ou non (false) l'entité/compteur en question est originaire d'un abonnement ou pas.

Syntaxe :

```
@REFERENCE.ismaster
```

Evidemment, cette fonction peut s'appliquer à une ou plusieurs entités/compteurs, citées explicitement ou résultant d'une formule.

Exemple :

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

→ Renverra la liste de toutes les entités/compteurs abonnées dans la base de données.

B. Notion de propriétés

Une propriété est une information stockée dans la base de données, et associée à un objet. On peut associer des propriétés aux entités, aux compteurs, aux canaux, aux utilisateurs… Les propriétés peuvent être de types variés, comme du texte, des valeurs numériques, des valeurs Booléennes…

On pourra donc éventuellement utiliser des fonctions mathématiques sur les valeurs renvoyées, comme vérifier leurs valeurs, ou les additionner entre elles.

De la même manière, les propriétés de type Booléen peuvent être utilisées directement dans les formules comme conditions d'une fonction if par exemple.

a) Propriétés simples

Les propriétés dites « simples » sont celles qui ont une valeur unique et invariable. On peut penser par exemple à l'année de construction d'un bâtiment, qui est donc une valeur unique et non susceptible de changer avec le temps.

.properties

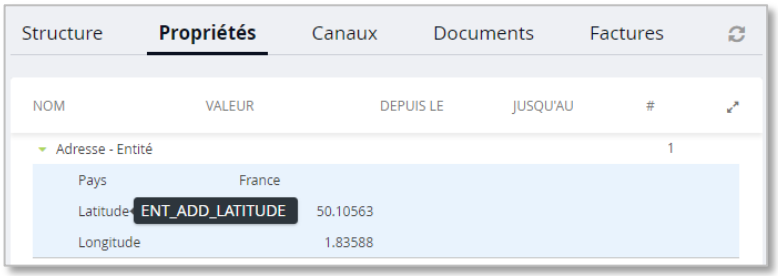
Pour récupérer la valeur d'une propriété simple, il suffit d'utiliser la fonction « .properties » en lui passant comme argument la référence de la propriété à récupérer.

Syntaxe :

```
selection.properties("//REFERENCE_DE_PROPRIETE")
```

- "//REFERENCE_DE_PROPRIETE" est en format texte et reprendra la référence unique de la propriété demandée (Attention : case sensitive). Le « // » est rendu nécessaire par le format de stockage des propriétés dans la DB EMM (JOOL). Ces caractères n'ont aucune incidence sur la syntaxe sinon qu'il ne faut pas les oublier.

Pour connaitre la référence de la propriété désirée, on peut l'afficher dans une infobulle au survol de la souris dans l'onglet « Propriétés » de la fiche entité/compteur.



Les propriétés sont organisées en arborescence et peuvent être regroupées par thèmes dans des blocs de propriétés. Par exemple, le bloc de propriété d'adresses ENT_ADD contiendra les propriétés ENT_ADD_STREET, ENT_ADD_NUMBER… reprenant les différentes parties de l'adresse.

selection.properties("//ENT_ADD/ENT_ADD _STREET") → Renverra le nom de la rue

selection.**properties**("//ENT_ADD_STREET") → Renverra le nom de la rue (idem)

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

→ Renverra la valeur de la propriété ENT_ADD_COUNTRY (le pays dans l'adresse) associée à l'objet en cours (item).

Il est possible de passer comme argument à cette fonction .properties, la référence du bloc entier de propriétés. L'objet actif deviendra alors le bloc de propriétés sur lequel on pourra faire d'autres opérations par la suite (.where…)

Une fois un bloc sélectionné, on peut récupérer les propriétés spécifiques dans le bloc via un nouvel appel à .properties.

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

→ Renverra la latitude du site. Ici, nous n'avons aucune différence entre cet appel, passant explicitement par le bloc de propriétés, ou un appel qui irait directement chercher la propriété finale.

L'intérêt principal de pouvoir appeler le bloc de propriété complet sera plus clair dans le cas de blocs de propriétés multiples et c'est donc de pouvoir filtrer les blocs sur base d'une propriété donnée, pour ensuite récupérer une autre propriété utile dans les blocs ressortant du filtre.

.xproperties – propriétés distantes

Dans le même ordre d'idées que pour les liens distants et la fonction .xlink, il peut parfois être utile de récupérer une propriété qui n'est pas forcément sur l'entité/compteur sélectionnée, mais qui peut être sur le parent, un lien, ou le parent d'un lien.

Un exemple classique est la récupération de la surface d'un site pour calculer des consommations par mètre carré. Les données de consommation sont sur le compteur qui doit donc être passé en sélection, mais la propriété contenant la surface est stockée sur le site associé. Or le lien n'est peut-être pas direct (démarrage d'un sous-compteur, passage par des sous-sites en chemin…) et peut même souvent varier en fonction des sites concernés.

Tout comme la fonction .xlink pour les liens distants, la fonction .xproperties a été créée afin de récupérer des propriétés distantes.

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

Cette formule va donc commencer par vérifier si la propriété « surface totale » est remplie pour la sélection active. Si oui, elle renvoie sa valeur et s'arrête là, sinon, elle va chercher sur le parent et les liens directs pour trouver cette propriété. De nouveau, si elle la trouve, elle renvoie sa valeur et arrête son exécution. Dans le cas contraire, elle cherche sur les liens suivants, jusqu'à trouver la valeur demandée.

Remarque : une fois de plus, la priorité dans l'ordre des liens sur lesquels la fonction .xproperties effectue ses recherches est prédéfinie par l'administrateur EMM (JOOL).

.parent (sur une propriété)

Une fois sur une propriété, il est possible de remonter à l'objet d'où vient la propriété (que cela soit une entité, un compteur, un canal, un user…). Pour cela, on peut utiliser la fonction « .parent », qui renverra donc par exemple le compteur d'où vient la propriété active. Cette possibilité sera bien utile dans certains cas de syntaxe avancée, ainsi qu'après récupération d'une propriété via la fonction .xproperties par exemple.

b) Propriétés traduites

Pour certaines propriétés, il est intéressant d'en afficher la valeur traduite dans la langue de l'utilisateur actif. Pour cela, l'administrateur EMM (JOOL) aura pris soins d'appliquer une contrainte d'encodage sur la propriété en question, afin que seule une série de valeurs prédéfinies soit disponible. Il aura en outre associé cette propriété à une table de données fixes (Xtab) qui contiendra la liste des valeurs autorisées et, pour chacune de ces valeurs, les traductions dans toutes les langues utilisées. Les détails de cette configuration seront abordés à un stade plus avancé de la formation, quand on parlera plus en profondeur des Xtabs.

Finalement, l'information stockée en base de données sera une clé unique désignant la valeur choisie, et à l'affichage de la propriété, EMM (JOOL) ira dans le XTab récupérer la traduction dans la langue de l'utilisateur actif, de la valeur dont la clé est stockée.

Exemple : le XTab peut contenir une ligne

KEY	FR	UK	NL
ELEC	Electricité	Electricity	Elektriciteit

Dans ce cas, l'information stockée en DB sera « ELEC », mais quand un utilisateur francophone affichera la propriété, il verra « Electricité », alors qu'un utilisateur néerlandophone verra « Elektriciteit ».

.key / .value

Dans la syntaxe, lorsque l'on travaille sur ces propriétés à valeurs traduites, il peut parfois être utile de travailler sur la clé spécifiquement, comme quand on veut filtrer des compteurs selon leur ressource consommée par exemple.

La fonction .key appliquée à une propriété permet d'en récupérer la clé plutôt que la valeur traduite. En effet, si l'on filtrait une liste en se basant sur la valeur (traduite) de la propriété, on aurait des résultats différents selon la langue de l'utilisateur connecté.

Exemples :

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

→ Renverra la clé de la propriété. Dans l'exemple ci-dessus, cette syntaxe renverrait « ELEC »

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

→ Renverra la valeur traduite de la propriété. Dans l'exemple, « Electricité »

selection.properties("//CNL_DAC_RESOURCE")

→ Comportement par défaut : Renverra la valeur traduite de la propriété également.

Il est à noter que ces fonctions peuvent s'appliquer également aux propriétés simples (sans traductions), et ce sans le moindre effet.

c) Blocs de propriétés multiples

En plus des propriétés « simples », pour lesquelles il n'y a qu'une seule valeur, invariable, certains blocs de propriétés peuvent être définis comme multiples.

Un site peut par exemple avoir plusieurs numéros de téléphones. Dans ce cas, on aura une même propriété en plusieurs exemplaires dans la fiche de l'entité.

Dans EMM (JOOL), les propriétés isolées ne peuvent être définies comme multiples, mais elles peuvent être intégrées dans des blocs qui, eux, peuvent être multiples. Cela permet d'avoir un ensemble de propriétés cohérentes, caractérisant la même chose.

Exemple : Si un site comporte plusieurs équipements, et que ces équipements ont de multiples propriétés, on pourra créer un bloc multiple pour le site, dont chaque instance représentera un équipement donné et contiendra le jeu de propriétés associées à l'équipement en question.

Dans le cas d'une propriété isolée qui doit pouvoir être multiple, il faudra alors créer un bloc multiple ne contenant que cette unique propriété. Cela peut paraître contraignant lors de la création de la propriété (il faudra créer un niveau de plus, avec le bloc), mais à l'utilisation, cela n'a aucun impact négatif puisque l'on peut accéder directement aux propriétés via la syntaxe, sans passer par le niveau « bloc ».

Dans la base de données, les instances des blocs de propriétés multiples ne sont différenciées entre elles que par leur ID. C'est cet ID qui sera utilisé pour identifier de façon univoque les instances des blocs à mettre à jour lors des imports massifs (voir plus loin le chapitre sur les imports massifs).

Quand on appelle, via une formule, une propriété comprise dans un bloc multiple, et dont plusieurs instances existent en pratique, on obtiendra une liste des différentes valeurs prises par cette propriété dans les différents blocs existants.

Exemple : Si l'on reprend l'exemple précédent et que l'on a deux équipements sur un site, pour lesquels une propriété « numéro de série » est stockée, la commande suivante :

item.properties("//N_SERIE") → renverra les 2 numéros de série à la suite.

.id

Si l'on veut récupérer la valeur de l'ID (identifiant) de chaque instance d'un bloc de propriétés multiple, on peut utiliser la fonction « .id » sur la propriété appelée. Cela renverra l'identifiant unique de l'instance

choisie du bloc de propriétés, et cela peut être bien utile pour un export-réimport sur des propriétés multiples (id nécessaire pour les mises à jour massives).

Attention : Dans EMM (JOOL), il n'est pas possible d'avoir un bloc de propriétés qui serait à la fois multiple et historisé.

d) Blocs de propriétés historisés

En pratique, il est aussi possible que certaines propriétés évoluent au cours du temps, comme par exemple la surface d'un bâtiment quand on construit une annexe.

Dans ce cas, il peut être utile de ne pas écraser la donnée précédente mais de la garder comme une donnée historique. (L'ancienne surface par exemple doit être conservée sinon, le calcul de consommation au mètre carré sur de vieilles données de consommations sera perturbé par la nouvelle valeur de la surface)

Pour garder les anciennes données, on pourra alors historiser les blocs de propriétés qui sont susceptibles de varier dans le temps.

Les blocs de propriétés ainsi historisés pourront apparaitre en plusieurs exemplaires sur la fiche de l'objet considéré (entité, compteur, canal, user…), mais ils seront tous associés à une période de validité différente (colonnes de dates de début et de fin dans la fiche).

Outre les dates de début et de fin de validité, le fonctionnement des blocs historisés de propriétés ne diffère pas de celui des blocs simples ou multiples.

La récupération des propriétés est parfaitement similaire à celle des propriétés simples, à ceci près que la récupération des valeurs se basera sur le contexte temporel actif pour ne récupérer que les valeurs dont la validité chevauche au moins partiellement le contexte.

Dans le cas où l'on veut récupérer une propriété historisée sur une période où sa valeur a évolué, l'appel à la propriété renverra alors la liste des différentes valeurs qui ont été valides pendant le contexte actif.

Dans le cas d'un bloc historisé de propriétés, il est possible de fournir à la fonction .properties un contexte temporel sur lequel travailler. Pour cela, on utilisera la syntaxe suivante :

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

→ Renverra, la liste des surfaces du site @C1 entre les dates du contexte (ici, aucune différence avec le fait de ne pas mettre de dates)

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

→ Renverra la surface du site à la date de départ du contexte actif (from).

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

→ Renverra la surface actuelle du site (now).

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

→ Renverra la liste des surfaces depuis début 2017 jusque début 2018.

Si l'on demande une propriété historisée sur une période pendant laquelle elle a évolué, on aura donc une liste de valeurs. Libre à l'utilisateur d'en faire ce qu'il veut, comme en faire la somme (.sum), la moyenne (.avg), prendre la plus grande (.max), ou simplement la première (.top(1))…

.from / .to

Quand on appelle une propriété historisée, il peut parfois être intéressant de connaitre les dates de validité de chaque valeur de cette propriété. Pour cela, il suffit d'utiliser les fonctions « .from » et « .to » sur la propriété pour en récupérer la date de début et de fin de validité.

.id

Tout comme pour les blocs de propriétés multiples, il est également possible de récupérer la valeur de l'ID (identifiant) de chaque instance d'un bloc de propriétés historisé. On peut à nouveau utiliser la fonction « .id » sur la propriété appelée. Cela renverra l'identifiant unique de l'instance choisie du bloc de propriétés, et cet ID pourra également être utilisé pour effectuer des exports-réimports de propriétés historisées (voir chapitre sur les imports massifs pour plus de détails).

Attention : Dans EMM (JOOL), il n'est pas possible d'avoir un bloc de propriétés qui serait à la fois multiple et historisé.

C. Objet canal (source de données)

Pour rappel, un canal est une sous-section d'un compteur, qui peut contenir un profil de données. En partant du compteur, on devra donc préciser sur lequel de ses canaux on veut récupérer des informations.

La fonction « .channels » s'applique donc à un compteur, et peut recevoir comme argument une chaine de caractère indiquant la référence ou le type du canal désiré. Elle renverra alors l'objet « Canal » demandé, sur base du type ou de la référence fournie.

Exemples :

```
selection.channels("MAIN")
```

→ Renvoie, en démarrant du compteur, le ou les canaux de ce compteur qui ont un type « MAIN »

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

→ « CNL_001 » n'étant pas un type, EMM (JOOL) va comprendre qu'il s'agit d'une référence et il renverra le canal dont la référence est « CNL_001 »

selection.**channels**

→ la fonction « .channels » n'ayant pas d'argument, elle renverra ici la liste complète de tous les canaux du compteur en sélection.

a) Fonctions associées aux canaux

Comme pour tout objet dans EMM (JOOL), il existe des fonctions applicables aux canaux :

.default : filtrera la liste des canaux pour ne garder que celui/ceux marqué(s) « par défaut ». Par exemple, en partant du compteur, la syntaxe « selection.channels.**default** » renverra donc le canal par défaut du compteur sélectionné.

.isdefault : renverra un Booléen (true / false) indiquant si le canal actif est ou non le canal par défaut de son compteur.

En outre, plusieurs fonctions existantes par ailleurs sont également applicables aux canaux.

.reference : renverra la référence du canal

.name : renverra le nom du canal si il a été configuré (optionnel sur les canaux)

.parent : renverra le compteur dont le canal actif est une partie

.properties : accède aux propriétés spécifiques du canal. Ces propriétés contiendront en général les informations de relève, le pas de temps, la ressource, l'unité… La syntaxe est la même que pour accéder à toutes les propriétés de EMM (JOOL): .properties("//REF_DE_PROPRIETE")

.data : renverra le profil de données associé au canal. Il est à noter que cette fonction .data peut également s'appliquer au compteur en tant que tel (et pas uniquement à un canal). Dans ce cas, elle renverra le profil du canal marqué « par défaut » pour le compteur.

.creation : renverra un bloc d'informations contenant la date et l'utilisateur associés à la création de ce canal. Cette fonction doit être suivie de « .date » pour récupérer la date de création, ou de « .user » pour récupérer le créateur de ce canal. (Note, l'objet sera alors l'utilisateur, dont on peut récupérer les informations comme pour un user normal, avec « .name », « .mail »…)

.lastupdate : fonctionne exactement comme le «.creation », mais récupère les informations de dernière mise à jour plutôt que les informations de création.

4. Collections d'objets

Une collection d'objets est une liste d'objets de même type. Syntaxiquement, on définit une collection d'objets en les listant entre parenthèses, séparés par des « ; »

Les mots clés selection et #SITE par exemple, sont des termes réservés, qui représentent chacun une collection d'objets constituée par l'ensemble des entités et compteurs actifs dans le sélecteur de canaux, ou de l'ensemble des entités de type « SITE » dans la base de données.

Exemples *:*

(1; 2; 3) → Collection des valeurs numériques 1, 2 et 3

(@C1 ; @C2 ; @C3) → Collection des compteurs C1, C2 et C3

Selection.descendants → Toutes les entités/compteurs descendantes de la sélection.

#SITE → Toutes les entités de type « SITE » présentes dans la base de données.

all → Collection de toutes les entités/compteurs présentes dans la base de données

(limitée aux entités/compteurs sur lesquels l'utilisateur actif à des droits de lecture et/ou d'écriture)

Les collections ont un intérêt majeur dans la création de worksheets dans EMM (JOOL) puisque dans un worksheet, on aura une ligne par élément de la collection sélectionnée, permettant ainsi de structurer les données que l'on veut afficher.

On peut également constituer une collection d'objets en accolant plusieurs sous-collections, comme par exemple en utilisant plusieurs formules consécutives qui produisent chacune une liste d'objets.

Exemple *:*

(@C1.descendants ; @C2.descendants)

→ Collection d'objets constituée de la liste des descendants du compteur C1 et de la liste des descendants du compteur C2.

Remarque *:* Une collection de collections sera une collection globale reprenant tous les éléments des collections de départ sans garder la structure de leur provenance.

Une collection peut provenir d'une sélection multiple dans le sélecteur d'entités, mais aussi de propriétés multiples dans une entité.

Selection.properties("//PROPRIETE")

→ Si l'on a une entité en sélection ayant plusieurs instances de « PROPRIETE », on récupèrera avec cette formule une collection constituée de toutes les valeurs de « PROPRIETE » de l'entité sélectionnée.

Evidemment, si l'on a des propriétés multiples dans une sélection multiple, on obtiendra une vaste collection contenant toutes les instances de la propriété pour chaque entité de la sélection.

a) Trier et filtrer une collection

En démarrant avec une collection d'objets, il sera souvent utile de filtrer cette collection de départ pour ne récupérer que certains objets répondant à certaines conditions. Pour cela, il existe plusieurs fonctions décrites ici.

.distinct

Il est possible dans une collection de voir apparaitre plusieurs fois un même objet. Dans ce cas, un tableau basé sur cette collection recopiera chaque ligne qui aura été demandée plusieurs fois.

Si le but n'est pas d'avoir des lignes en multiples exemplaires, il est possible d'utiliser la fonction .distinct, qui, appliquée à une collection d'objets, a pour effet de générer une nouvelle collection dont ont été retirés les doublons de la collection initiale.

Exemple *:*

(@C1; @C2; @C2) → Collection des entités C1, C2 et C2 (en double donc)

(@C1; @C2; @C2).**distinct** → Entités C1 et C2 (le 2° C2 a été supprimé par le .distinct)

Exemple*:*

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

1. selection.children crée la liste (sous forme de collection) des enfants directs de la sélection.
2. selection.descendants crée la liste (sous forme de collection) de tous les descendants, y compris donc les enfants directs.
3. L'association des deux listes constituera une collection, dans laquelle devraient être repris en double tous les enfants directs et une seule fois les petits enfants et autres descendants.
4. La fonction .distinct va alors supprimer toutes les multiples occurrences des enfants directs et la collection finale ne contiendra plus de doubles.

.where

Dans le cas où l'on ne s'intéresse qu'à un sous-ensemble des objets de la collection initiale, il est possible de filtrer les données directement dans la collection pour générer une sous-collection répondant à un critère donné.

La fonction .where permet ainsi de ne retenir que les objets remplissant le critère spécifique qui lui est passé en argument.

Exemple *:*

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

→ Renverra la liste de valeurs 2 et 3.

→ EMM (JOOL) reçoit la liste 1 ; 2 ; 3, puis applique le filtre du .where. Pour ce faire, il va évaluer la condition entre parenthèses séquentiellement pour chaque objet de la liste, et retirer de la liste ceux qui ne répondent pas à la condition.

1 ne remplit pas la condition (1 < 2) et est donc rejeté.

Les autres valeurs remplissent la condition, et sont donc conservées dans la liste finale.

Le mot clé item dans la condition est un terme générique qui prendra successivement toutes les valeurs de la collection initiale pour évaluer la condition sur cet élément.

Cette fonction .where peut s'appliquer à des collections d'objets de tous types, mais il s'agira principalement d'entités, de compteurs, de canaux ou de propriétés.

Exemples *:*

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

→ Renverra toutes les entités/canaux de la sélection dont le nom commence par B

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

→ Renverra toutes les entités/canaux dont le nom ne commence pas par B

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

→ Renverra une liste des propriétés « PROP » des entités/canaux sélectionnées pour lesquels la valeur de la propriété commence par B.

Remarque : dans ce cas-ci, la collection obtenue comme résultat de cette syntaxe est donc bien une collection de propriétés. Si on les affiche dans un tableau, EMM (JOOL) comprendra qu'il doit afficher par défaut les valeurs des propriétés. A partir de la collection, il est possible d'utiliser ces propriétés pour remonter aux entités d'où elles proviennent, en utilisant « .parent » sur cette liste de propriétés.

.where successifs

On peut bien entendu combiner plusieurs conditions dans les arguments du .where. En effet, comme vu dans le chapitre sur les Booléens, on peut combiner des conditions avec AND et OR.

Dans le cas du AND par contre, il pourrait être plus efficace de simplement utiliser 2 fois la fonction « .where » d'affilée.

Les 2 formulations suivantes effectueront la même recherche et produiront les mêmes résultats.

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

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

La seule différence sera au niveau de la quantité de calculs à effectuer par EMM (JOOL), et donc de la performance en termes de temps de calcul.

La première version va prendre la liste totale de toutes les entités/compteurs (all) et va évaluer les deux conditions (liées par un and) sur toutes les entités/compteurs. En imaginant 1000 entités, cette version va donc évaluer 2000 conditions.

La seconde version, en revanche, va évaluer la première condition sur l'ensemble des entités/compteurs et va constituer une liste intermédiaire de celles remplissant cette première condition, puis seulement évaluer la 2° condition sur cette liste intermédiaire. Sur la même base de 1000 entités/compteurs, le premier « .where » va évaluer 1000 fois la condition 1 et garder par exemple 500 entités/compteurs remplissant

cette condition 1. Une fois ce premier passage effectué, il n'a plus qu'à évaluer la condition 2 sur les 500 entités/compteurs restantes. Bilan : 1500 conditions évaluées au lieu de 2000.

Etant donné que certaines conditions peuvent être assez complexes et donc gourmandes en puissance et en temps de calcul, il peut être utile de bien penser ses filtres dès le départ pour éviter une explosion inutile du temps de travail.

.where sur blocs multiples

Dans le cas de blocs multiples (ou historisés) de propriétés, la fonction « .where » permet de filtrer les blocs selon une sous-propriété. On pourra par exemple récupérer l'ensemble des blocs contacts d'une entité/compteur, et parmi ces blocs contacts, ne garder que ceux qui ont une certaine caractéristique.

Exemple *:*

Imaginons un bloc multiple « EQUIPMENT ». Si l'on a 2 équipements pour un même site, on aura donc tout le bloc de propriétés « EQUIPMENT » en double. Il est alors possible par exemple, de retrouver spécifiquement le numéro de série (dans une propriété) de l'un des équipements dont on connait le nom.

item.properties("//N_DE_SERIE") → renverra les 2 numéros de série à la suite.

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

→ On récupère ici les 2 numéros de série, exactement comme précédemment. La seule différence est que l'on force le chemin de la recherche de la propriété de la façon suivante : on cherche d'abord les blocs « EQUIPMENT », et dedans, on va chercher la propriété « N_DE_SERIE ». (Pour l'instant, aucune réelle différence)

L'idée est ici que l'on peut utiliser item.properties("//EQUIPMENT") comme un objet, et effectuer des filtres dessus, avant de continuer de descendre dans ses propriétés.

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

→ On prend les différents blocs « EQUIPMENT ». Dans ces blocs, on filtre via la fonction .where et on ne garde que les blocs qui contiennent la propriété « EQUIPMENT_NAME » avec la valeur « Générateur ». Enfin, parmi ces blocs spécifiques, on récupère le numéro de série dans la propriété « N_DE_SERIE ».

Cette manière de faire considère les blocs de propriétés comme des objets, ce qui permet d'utiliser la fonction « .where » pour effectuer des filtres au besoin.

Remarque *:* Les blocs de propriété peuvent être considérés comme des objets mais ils n'ont pas de place pour autant dans l'arborescence. Ainsi, si l'on utilise la fonction « .parent » sur une propriété, on ne remontera pas au bloc de propriétés comprenant cette propriété, mais bien directement à l'entité, compteur, canal ou user comprenant le bloc. (Pas moyen donc de remonter sur les blocs de propriétés au départ d'une propriété)

.orderby

Lors de la constitution d'une collection, il peut être utile de trier les valeurs dans un certain ordre, qui n'est peut-être pas présent initialement dans les données.

Exemple *:* Lister l'ensemble des compteurs d'un site par ordre alphabétique, ou par ordre chronologique de création, ou encore par consommation totale décroissante sur la période de temps considérée…

Syntaxe*:*

```
selection.orderby(Critère "desc")
```

- Critère : Le critère suivant lequel s'effectuera le tri
- "desc": indique le sens du tri. "desc" = décroissant, "asc" = croissant. ("asc" est la valeur par défaut et il n'est donc pas indispensable de l'écrire)

Il est aussi possible de faire plusieurs tris en cascade, sur plusieurs valeurs.

Exemple *:*

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

→ Renverra les entités/compteurs de la sélection triées par groupe (ici une propriété « GROUP »), en ordre croissant (puisque rien n'est indiqué, EMM (JOOL) prend l'ordre croissant par défaut), et dans chaque groupe, elles seront alors triées par ordre alphabétique de leur référence, en ordre décroissant.

Remarque *:* on pourrait être tenté d'utiliser la fonction .orderby deux fois de suite pour faire ce genre de tri, avec chaque fois un seul argument) mais dans ce cas, le 2° tri écrasera le premier. La bonne méthode est donc bien d'utiliser la fonction .orderby une seule fois, mais avec plusieurs arguments.

Exemple avancé *:*

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

→ Renverra la liste des compteurs de la sélection de la sélection, triée par ordre décroissant de consommation totale sur le contexte temporel actif.

.top / .flop

Quand on a une collection d'items, on peut parfois n'avoir besoin que des 5 premiers ou des 3 derniers de cette liste, à l'exclusion de tous les autres. On peut par exemple rechercher le compteur qui consomme le plus dans une liste, ou encore les 5 clients qui paient le moins cher leur énergie pour le mois dernier.

.top(n) permettra de ne récupérer que les n premiers objets d'une collection

.flop(n) permettra de récupérer les n derniers objets de la collection

Ces deux fonctions sont souvent utilisées après un .orderby, afin d'être certain que la liste est triée selon un critère utile et que l'on prend ensuite les n premiers/derniers.

On peut aussi utiliser ces deux fonctions ensemble, pour récupérer par exemple le 4° objet d'une liste et aucun autre.

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

→ créera une sous-liste d'objets de la sélection, contenant uniquement les 4 premiers objets, puis prendra uniquement le dernier de cette sous-liste

→ le 4° de la liste initiale.

Remarque : Lors d'une recherche top/flop d'une liste de construite à partir de la fonction .ancestors, le parcours des ancêtres se fait du bas vers le haut de l'arborescence. Le dernier ancêtre de la liste sera donc celui situé tout en haut de la liste hiérarchique, alors que le premier sera le parent direct.

Exemple :

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

→ Comme vu précédemment, ce code nous renverra les compteurs de la sélection, triés par ordre de consommation totale décroissante sur la période de temps définie par le contexte temporel actif, et ne gardera que les 5 premiers de la liste. Autrement dit, ce code nous renvoie le top 5 des plus gros consommateurs sur la période de temps considérée.

Optimisation des filtres

Certains filtres sur une collection peuvent prendre plus ou moins de temps selon la manière dont ils seront codés. Il est fortement conseillé de bien penser l'enchainement de filtres pour éviter de multiplier le nombre d'accès DB inutiles, et économiser ainsi un temps de calcul conséquent.

Nous avons déjà parlé du fait de faire plusieurs filtres .where successifs plutôt que de faire un seul .where avec plusieurs conditions reliées par un « AND ». En effet, cela diminue le nombre de tests à faire, et accélère donc le calcul.

Une autre optimisation conseillée est d'éviter de filtrer des compteurs sur base de leurs propriétés. Il est plus efficace de descendre aux propriétés, de filtrer directement dessus, puis de remonter aux compteurs par la suite.

Exemple : Si je veux récupérer la liste des sites dont le code postal est 1348, la première syntaxe qui vient à l'esprit serait la suivante :

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

Cette syntaxe va donc faire un premier accès DB pour récupérer la liste des entités/compteurs présentes dans la base de données. Ensuite, de façon séquentielle, EMM (JOOL) va effectuer, pour chacune des entités/compteurs, une nouvelle requête DB pour récupérer la propriété « ENT_ADD_ZIP » et effectuer le test sur la valeur 1348. Cela représente donc un grand nombre d'accès DB (un par entité de la DB), qu'il est possible de diminuer avec une modification relativement simple.

Premièrement, on peut démarrer de « #SITE » au lieu de « all ». Cela diminue le nombre d'objets étudiés, mais cela nous limite aux entités de type « SITE », à l'exclusion d'autres types d'entités, potentiellement utiles. A utiliser de façon éclairée donc.

Une autre solution, qui fonctionnera dans toutes les situations, sera d'utiliser la formule suivante :

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

Cette formule démarre donc de la liste des sites, et récupère (en un seul accès DB) toutes les propriétés « ENT_ADD_ZIP », puis filtre sur les valeurs de ces propriétés (déjà disponibles en mémoire après la récupération des propriétés, donc pas besoin d'un accès DB par entité). Une fois les propriétés filtrées, on effectue alors un dernier accès DB pour récupérer massivement les entités d'où viennent les propriétés restantes (.parent), et en seulement deux accès DB, on obtient la liste désirée.

Enfin, quand on veut vérifier qu'une valeur donnée appartient à une liste de valeurs, on pourrait être tentés de coder cela comme suit :

```
#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" )
```

Or, cette façon de faire impose à EMM (JOOL) d'effectuer 5 tests sur chaque valeur de propriété. Il sera plus efficace de rechercher la valeur de propriété dans la liste concaténée des valeurs autorisées (séparées par un caractère spécial, pour éviter de déclencher sur de fausses valeurs à cheval sur deux valeurs autorisées). On obtient alors le code suivant :

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

Ce faisant, on n'effectue qu'un seul et unique test par valeur de propriété pour vérifier qu'elle appartient bien à la liste autorisée. Cette optimisation peut sembler peu importante, mais dans le cas de listes autorisées plus longues, ou de tests un peu plus complexes, elle devient très vite extrêmement rentable au niveau du temps de calcul.

b) Fonctions associées aux collections

.count

La fonction .count sert à compter le nombre d'objets présents dans une collection.

selection.**count** → renverra le nombre d'entités/compteurs dans la sélection.

selection.data.**count**

→ renverra le nombre couples date/valeur dans les profils de données des canaux par défaut de différents compteurs de la sélection.

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

Quand on lui fournit une collection de valeurs numériques, EMM (JOOL) est capable d'effectuer toute une série d'opérations mathématiques.

Ci-dessous, le mot « collection » est utilisé pour représenter une collection d'objets. Il ne s'agit pas vraiment d'un mot clé utilisé dans la syntaxe EMM (JOOL). Il représente une liste de valeurs numériques, un profil de données… ou toute autre collection pertinente, qui pourrait être représentée par une véritable syntaxe EMM (JOOL) plus complexe.

.max .min	Maximum Minimum	Renvoient le maximum ou le minimum des valeurs de la collection, sous forme numérique (double) collection.**max** → renvoie la valeur maximale de la collection collection.**min** → renvoie la valeur minimale de la collection
.sum	Somme	Renvoie la somme de la série de valeurs sous forme numérique.
.avg	Average, moyenne arithmétique	Renvoie la moyenne de ces valeurs sous forme numérique (double) collection.**avg** → renvoie la valeur moyenne de la collection
.std	Déviation standard	Renvoie la déviation standard de la série de valeurs sous forme numérique.
.percentile	Percentile	Renvoie la valeur d'une collection correspondant à son Xème percentile, X étant passée en paramètre entre 0 et 1. collection.**percentile**(0.95) → renverra la valeur de la collection correspondant à son 95° percentile. Si la valeur exacte est entre deux valeurs de la série, EMM (JOOL) renverra une interpolation linéaire entre les valeur proches. **Exemple:** (40 ; 58.2 ; 58.8 ; 60).**percentile**(0.4) Le percentile 0.4 est entre 58.2 et 58.8 - Nb d'intervalles = nb de valeurs - 1 → 3 - Position = 0.4 x 3 = 1.2 NB: La 1è valeur st la valeur n°0 Le percentile est donc à 20% entre les valeurs 1 et 2, donc entre 58.2 et 58.8 → 58.2 + (58.8-58.2) x 0.2 = 58.2 + 0.12 = 58.32

		Note : les éventuelles valeurs NaN présentes dans la liste non ignorées pour le calcul.

.groupby

Quand il est nécessaire de regrouper les objets d'une collection selon certains critères, on peut utiliser la fonction .groupby. Cette fonction va donc recevoir une collection d'objets et générer une nouvelle collection d'objets structurée selon un critère de regroupement.

Syntaxe *:*

```
collection.groupby(key)
```

- collection : La collection initiale. Cela peut être une sélection de canaux ou d'autres objets.
- key : indique la valeur sur laquelle s'effectuera le regroupement.

Cette valeur « key » peut avoir des formes très variées. On peut par exemple grouper des entités par rapport à leur initiale, ou par rapport à la valeur de l'une de leurs propriétés. On peut aussi grouper des lignes dans un tableau en fonction de la valeur présente dans l'une de leurs colonnes. On peut aussi regrouper des utilisateurs suivant leur zone géographique, ou des données suivant leur provenance.

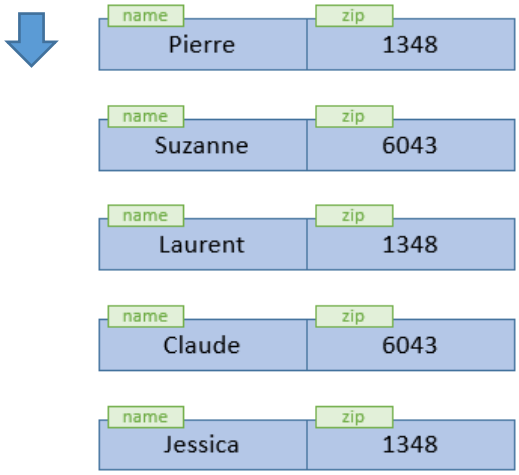
Le résultat de cette fonction sera une nouvelle collection, constituée de couples key/elements, contenant, pour chaque clé (chaque valeur possible de la clé passée en argument) une collection d'éléments.

Chaque objet de la collection obtenue étant un couple key/elements, on pourra y accéder via la syntaxe suivante :

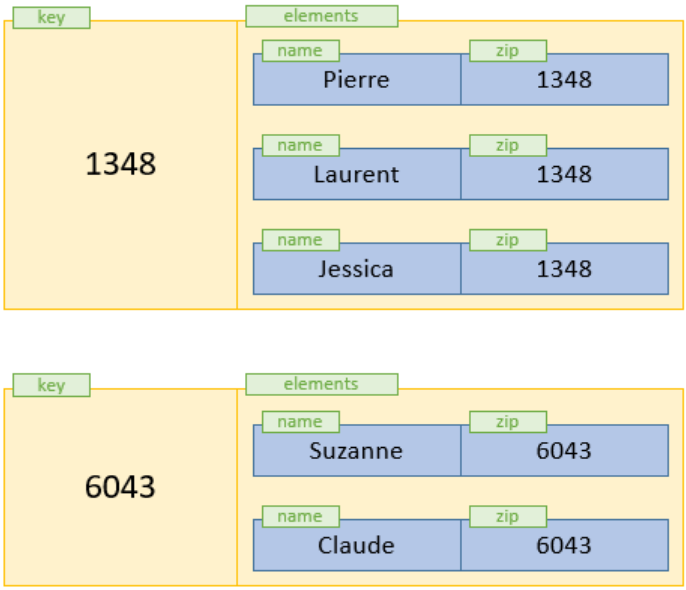
item.**key** → donne la valeur de la clé pour chaque groupe.

item.**elements** → représente la sous-collection des objets inclus dans le groupe.

Exemple *:* Dans l'exemple illustré ci-dessous, la collection de départ contient des utilisateurs et leur code postal. Effectuer un groupement via .**groupby**(item.properties("//ZIP")) génèrera une nouvelle collection, contenant des couples key/elements. Chaque key étant une valeur possible prise par la propriété ZIP, et chaque elements associé étant une sous-collection contenant tous les éléments de la collection de départ remplissant la condition de groupement (ici, le zip correspondant à la key)



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



On peut également effectuer plusieurs groupements, en les mettant dans la même fonction .groupby

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

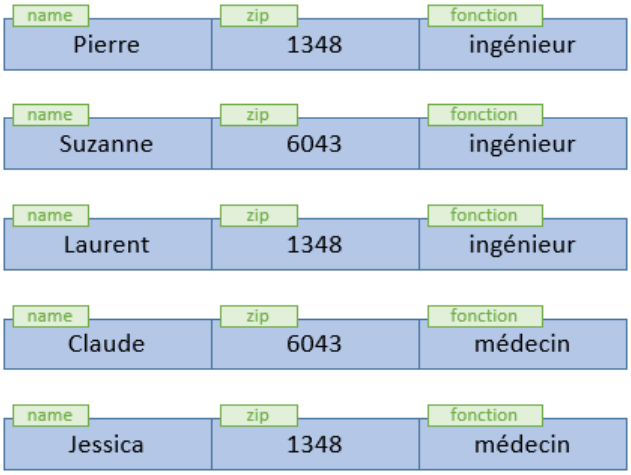
→ Donnera une collection de couples key/elements, où les utilisateurs seront regroupés selon leur code postal **et** leur fonction.

L'accès aux informations de la nouvelle collection se fera toujours via .key pour obtenir la clé du groupe considéré, et .elements pour récupérer le profil de données associées à la clé.

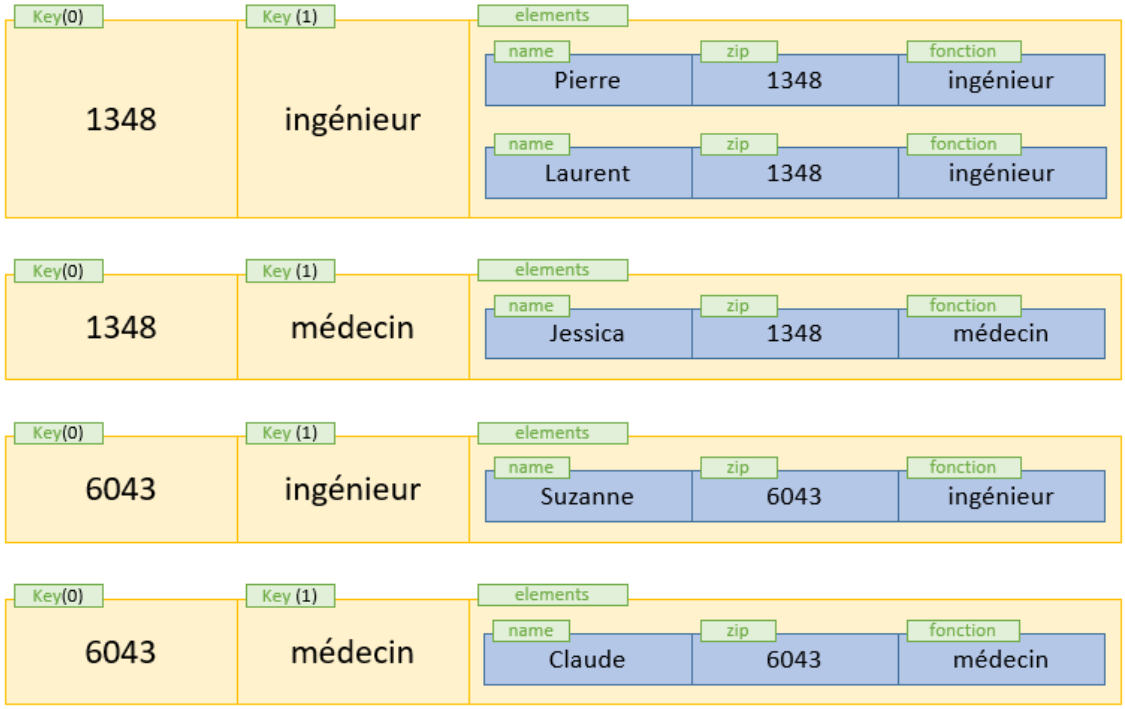
S'il y a deux groupements simultanés, la clé sera la concaténation des deux sous-clés key(0) et key(1), contenant chacune la clé associée à l'un des groupements.

Exemple *:* Si l'on reprend le même genre de collection que pour l'exemple précédent, mais que l'on ajoute une information (ici, la fonction de l'utilisateur), on peut maintenant effectuer un regroupement sur base des deux critères simultanément.

On aura donc 2 keys, la key(0) correspondant au premier groupement et la key(1) correspondant au deuxième critère de regroupement. La valeur key devenant alors la concaténation des deux sous-clés.



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



.foreach

La fonction .foreach permet, à partir d'une collection donnée, d'effectuer une opération définie pour chacun des éléments de la collection.

La fonction prend donc une collection d'objets comme source, et pour chaque objet de cette collection, séquentiellement, EMM (JOOL) va effectuer l'opération qu'elle reçoit en argument. Le mot clé item peut une nouvelle fois servir ici de terme générique remplaçant séquentiellement chaque objet dans l'opération.

Exemples *:*

```
selection.foreach(item.name)
```

→ Crée la liste des noms d'entités/compteurs dans la sélection active

Le système va donc prendre la sélection active, et pour chacun de ses éléments, évaluer ce qui lui est demandé dans l'argument du .foreach (ici, récupérer son nom).

Attention : dans le cas de l'exemple ci-dessus, le résultat ne contient donc plus une série d'entités/compteurs, mais bien une série de textes, qui sont les noms des entités/compteurs de la sélection initiale.

Remarque *:* le .foreach applique sa demande séquentiellement sur tous les éléments qu'on lui a soumis dans la sélection, et ce, même si un même élément apparait plusieurs fois (il effectuera alors la demande plusieurs fois sur ce même élément)

Un autre intérêt du .foreach est dans le cadre de tâches à effectuer (comme envoyer un mail, générer un rapport…). Dans ce cas, le .foreach permettra de lancer la tâche de façon récursive sur tous les éléments sélectionnés. Par exemple, il permettra de générer en série des rapports utilisant un même template, mais appliqué successivement à chacun des sites de la sélection active. Ce point est détaillé dans le chapitre parlant du gestionnaire de tâche de EMM (JOOL).

Exemple avancé *:*

```
selection.foreach(SENDMAIL("dest@dapesco.com"; null ; "Mail de confirmation"; "Ce mail est la confirmation que l'entité " + item.name + " est bien présente dans la DB"))
```

→ A l'application de ce code, EMM (JOOL) enverra pour chaque objet de la sélection, un email au destinataire "dest@dapesco.com", ayant pour titre "Mail de confirmation" et comme contenu "Ce mail est la confirmation que l'entité NOMDELENTITE est bien présente dans la DB". Ce code peut bien entendu être adapté pour envoyer autre chose, comme des rapports en PDF, ou des alarmes.

.extend – Extension d'une collection

Il arrive bien souvent qu'en partant d'une collection d'entités, on veuille récupérer ces entités elles-mêmes mais aussi tous leurs descendants par exemple. La syntaxe naturelle est alors

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

Malheureusement, cette syntaxe effectuant un .foreach va générer un grand nombre d'appels à la base de données, ce qui peut rallonger dramatiquement le temps de calcul global, surtout si ce genre de syntaxe est utilisé plusieurs fois d'affilée pour récupérer par exemple une sélection d'entités et ses descendants, ainsi que tous leurs liens « ENTITY_METER » et tous les descendants de ces derniers… cette syntaxe pouvant être utilisée par exemple dans les droits utilisateurs pour créer des périmètres sur base d'un site.

Pour éviter l'explosion du temps de calcul dû à cette syntaxe, une fonction a été créée qui permet d'étendre la sélection actuelle en lui rajoutant directement d'autres entités.

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

Cette syntaxe aura exactement le même résultat que l'autre, donnée ci-dessus. L'avantage étant que cette fonction-ci est optimisée pour minimiser le nombre d'accès à la base de données, accélérant drastiquement les calculs.

La fonction « .extend » s'applique donc à une collection d'objets, et reçoit en argument une formule qui va renvoyer une autre collection d'objets. La fonction « .extend » renverra au final la collection initiale, étendue pour inclure en plus le résultat de la formule passée en argument.

Fonctionnellement, l'expression selection.**extend**(item.descendants) est en tout point similaire à l'expression Selection.foreach(item; item.descendants). Ces deux instructions renverront exactement la même collection d'objets au final. La seule différence est que la première ira beaucoup plus vite que la seconde.

5. Profils de données

Un profil de données est une collection d'un genre particulier, constituée de couples date/valeur permettant de visualiser l'évolution d'une donnée dans le temps.

a) Profils associés aux canaux

Dans EMM (JOOL), un canal va pouvoir contenir deux profils distincts mais reliés.

- RawData : ce profil contiendra les données brutes injectées dans EMM (JOOL) telles quelles, sans transformation. Il peut s'agir d'index ou de consommations, en provenance de dataloggers ou d'insertion manuelle, ou encore des résultats de formules créant des profils virtuels (détails plus loin, dans la section parlant des formules de canaux).

- Data : ce profil est le profil de données nettoyé, traité et standardisé pour être utilisé dans EMM (JOOL). Si les RawData étaient en index, le profil Data est constitué de ses différences successives pour donner un profil de consommations, et ces consommations sont alors multipliées par un éventuel poids d'impulsion (aussi appelé « poids de pulse »)

b) Récupération des profils des canaux

.data / .rawdata

A partir d'un canal, on peut accéder à ses profils de données via les fonctions .data ou .rawdata, qui produiront les profils associés. Ces profils sont donc des collections d'objets, chacun de ces objets étant un couple date/valeur représentant une mesure de consommation à un instant donné.

Exemples :

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

→ Renverra le profil de données Data associé au canal "MAIN" du compteur C1.

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

→ Renverra le profil de données brutes RawData associé au canal "MAIN" du compteur présent dans la sélection

On peut aussi récupérer un profil de données en partant d'un compteur, sans préciser un canal spécifique. Dans ce cas, c'est le profil de données du canal marqué comme « par défaut » qui seront renvoyées par la formule.

@C1.**data**

→ Renverra le profil de données Data associé au canal par défaut du compteur C1.

Par défaut, si l'on appelle un profil de données (via .data) sans préciser de contexte temporel, le contexte appliqué sera le contexte actuellement sélectionné dans le sélecteur de dates (en haut à gauche).

Si maintenant, nous ne sommes intéressés que par les données dans une certaine période de temps, on peut passer des dates de départ et de fin de contexte en argument à la fonction .data.

Exemples *:*

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

→ Crée un profil allant du premier mars 2015 jusqu'à maintenant

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

→ Considèrera les datas du contexte actuel (pas de différence avec selection.data)

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

→ Crée un profil allant du premier jour de ce mois-ci jusqu'à maintenant

c) Profil(s) de sélection multiple

Si l'on a une sélection de plusieurs canaux, l'utilisation des fonctions récupérant les profils ira récupérer un seul profil global représentant la somme des profils de tous les canaux sélectionnés.

selection.data

→ Si la sélection contient plusieurs canaux, ce code renverra une seule série de données, contenant un couple date/valeur par date des séries initiales, en sommant les valeurs de consommation des dates correspondantes

Si l'on veut éviter cela et récupérer les profils séparés de chaque compteur de la sélection, on devra utiliser la fonction .foreach vue précédemment.

selection.foreach(item.data)

→ Créera une liste des couples date/valeur pour chaque compteur de la sélection. Cette formule gardera les couples date/valeur de chaque compteur bien distincts et les listera comme des objets différents.

d) Générer un profil à partir d'une propriété historisée

Quand on appelle une propriété historisée, et que celle-ci varie sur le contexte temporel actif, EMM (JOOL) renvoie une liste de valeurs ayant chacune leurs propres from et to. Autrement dit, EMM (JOOL) renvoie un profil de données, composé d'un couple date-valeur par valeur prise par la propriété historisée.

Pour constituer un véritable profil régulier, il suffira alors de lui appliquer une agrégation, comme par exemple avec la formule suivante : item.properties("//SURFACE").agg(1; "month"; "constant")

Cette formule renverra alors un profil mensuel, contenant pour chaque mois la valeur de la propriété « SURFACE »

Remarque: Il est aussi possible d'utiliser les factures d'un site pour créer un profil. En effet, les propriétés de factures sont considérées comme historisées (avec comme dates de validité les dates from/to de la

facture). Ces propriétés historisées sont alors utilisables pour construire un profil exactement comme décrit-ci-dessus

```
selection.invoices("//CONSO")
```

→ renverra donc un profil contenant les différentes valeurs « CONSO » des factures, avec comme dates les dates de validité des factures associées.

e) Accéder aux données (couples date/valeur)

Un profil de données est donc une collection d'objets date/valeur contenant diverses informations. Quand on met un profil de données dans la case « selection » d'une feuille de calcul (Worksheet), EMM (JOOL) va interpréter le profil comme une collection, et il affichera donc, dans le worksheet, une ligne par couple date/valeur présent dans le profil en sélection.

Sur ces objets, il est alors possible de récupérer des informations et d'effectuer des modifications via les fonctions suivantes…

.value / .date / .from / .to

Ces fonctions permettent de récupérer directement des informations stockées dans les couples date/valeur.

item.**value** → renvoie la valeur de l'élément (la consommation par exemple)

item.**from** → renvoie la date de début de la tranche de mesure

item.**to** → renvoie la date de fin de la tranche de mesure

item.**date**

→ renvoie la date associée à l'objet. Dans le cas d'un profil, il s'agira de la date de fin de période de mesure.

Ces fonctions peuvent s'appliquer directement à un couple date/valeur seul ou directement à un profil (collection de ces couples). Si on l'applique à une mesure seule, on obtiendra une valeur unique, soit numérique, soit une date. Si on l'applique à un profil, on obtiendra la liste des valeurs correspondantes pour chacune des mesures du profil.

Remonter d'une donnée vers le canal ou le compteur

Quand on est sur une donnée (couple date/valeur faisant partie d'un profil), il est possible, via la syntaxe, de remonter pour retrouver le canal ou même le compteur d'où la donnée provient. Pour cela, on utilisera les syntaxes suivantes :

@C1.channels("MAIN").data.**parent.parent** → Renverra le compteur C1.

@C1.channels("MAIN").data.**parent.channels** → Renverra le canal « MAIN » du cptr C1.

En gros, le principe est de remonter une première fois au parent, qui se trouve être le profil de données (un objet peu utilisable en tant que tel dans la syntaxe mais utile pour le code informatique sous-jacent à EMM (JOOL)), puis on remonte

- au .parent du profil pour obtenir le compteur. Cette syntaxe est héritée des anciennes versions du sort, pour assurer une certaine rétrocompatibilité aux anciennes formules.

- au .channels du profil, ce qui renverra spécifiquement le canal comme objet actif

Remarque *:* Bien que la fonction .parent ici utilisée ressemble à celle utilisée pour les canaux, elle n'est pas exactement la même. En effet, dans un langage orienté objet, une fonction est toujours définie en fonction des types d'objets sur lesquels elle s'applique. Ainsi, la fonction .parent s'appliquant aux canaux et la fonction .parent s'appliquant aux couples date/valeur ne sont pas exactement les mêmes fonctions, elles ont juste le même nom.

Dans notre cas, les deux fonctions .parent s'utilisent de la même manière, mais cette même subtilité de définition a des implications sur d'autres fonctions.

La fonction .ancestors par exemple est définie pour fonctionner exclusivement à partir de canaux. Donc, en partant d'un couple date/valeurs, on ne peut pas utiliser directement .ancestors pour récupérer les canaux de l'arborescence montante, et ce même si l'existence de la fonction .parent sur les couples date/valeur aurait pu le laisser penser.

Si l'on veut remonter depuis un couple date/valeur et récupérer l'arborescence montante, on devra donc utiliser .parent.parent. pour remonter au canal, et là seulement, on pourra utiliser .ancestors.

Exemple *:*

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

→ A partir d'un couple date/valeur, renverra le canal tout en haut à la racine de l'arborescence où se trouve la donnée de départ.

f) Fonctions associées aux profils de données

Une fois un profil récupéré, on peut lui appliquer toute une série de transformations, comme agréger ses valeurs, les répartir selon un SLP, ou combiner plusieurs profils entre eux.

.agg

Il peut arriver que l'on ait besoin des valeurs d'un profil non pas en suivant les données réelles (qui peuvent être reçues par 10 minutes par exemple) mais plutôt les données par tranche d'heure, de jour ou autre. (ex: La consommation ponctuelle est utile, mais pour une vérification de facture, la consommation globale du mois est une information suffisante)

La fonction d'agrégation .agg permet de transformer un profil existant en un autre profil contenant les mêmes données, agrégées par tranches de temps d'une taille définie.

Cette fonction doit bien recevoir un profil de données, autrement dit une collection de couples date/valeurs, sans quoi il lui sera impossible d'agréger les données par tranche de temps.

Syntaxe *:*

```
selection.data.agg(nbr_intervalles; "type_intervalle"; "méthode")
```

- nbr_intervalles : indique le nombre d'intervalles sur lesquels se fera chaque agrégation.

- "type_intervalle" : indique le type d'intervalle. Format texte (donc entre " "), à choisir parmi year, month, day, hour ou minute

- "méthode" indique la méthode selon laquelle se fera l'agrégation (format texte), à choisir parmi :

avg	La moyenne de la valeur sur l'intervalle (utile pour un profil de températures p.ex.)
sum	La somme des valeurs de l'intervalle (utile pour une consommation par exemple)
max	La valeur maximale atteinte sur l'intervalle d'agrégation défini
min	La valeur minimale atteinte sur l'intervalle d'agrégation défini
count	Renvoie le nombre de données reçues sur l'intervalle de temps considéré.
std	Calcule l'écart type sur les données présentes dans l'intervalle de temps considéré (std=standard deviation)
sumprop	Effectue une somme proportionnelle sur l'intervalle (explications ci-dessous)
constant	Reprend la valeur de fin de chaque intervalle
persistant	Reprend la valeur de départ de chaque intervalle

Remarque *:*

Si l'on a plusieurs profils en même temps (avec un selection.foreach(item.data) par exemple), on peut aussi utiliser .agg pour agréger les valeurs par tranche de temps. Pas besoin qu'elles soient dans un seul profil avec des dates dans l'ordre.

L'agrégation ira récupérer les données sur base de leurs dates, les rassemblera par tranche de temps et les agrégera en utilisant la méthode d'agrégation passée en argument.

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

→ renverra un profil de données sommées par heure ; qu'elles soient à l'origine mesurées sur des intervalles de 10 ou de 15 minutes n'aura aucun impact.

Agrégation à pas de temps adaptatif

En plus de la possibilité d'agréger des données dans un DataSet avec un pas de temps auto-adaptatif, il est également possible d'effectuer une agrégation à pas auto-adaptatif directement par formules dans les worksheets. Le pas de temps est alors choisi par EMM (JOOL) en fonction de la période d'observation actuelle.

- Si la durée de la sélection temporelle dépasse les 500 jours : 1 year
- Si la durée de la sélection temporelle dépasse les 300 : 1 month
- Si la durée de la sélection temporelle dépasse les 10 jours : 1 day
- Si la durée de la sélection temporelle dépasse le 1 jour : 1 hour
- Sinon : Pas d'agrégation

Pour cela, la fonction .agg peut accepter, en plus de son set de 3 paramètres habituels (Nb Pas de temps, Type pas de temps, Méthode d'agrégation), une autre formulation ne reprenant qu'un seul paramètre : la méthode d'agrégation.

Formulation classique : Selection.data.agg(1 ; "month" ; "sum")

Formulation pour pas de temps adaptatif : Selection.data.agg("sum")

Limitations

- Le pas de temps étant défini dynamiquement en fonction de la période d'observation, il est fortement déconseillé d'utiliser cette nouvelle fonctionnalité dans des formules de canaux par exemple. En effet, en fonction de la taille de la tranche de temps précalculée, les données varieront

- Dans les rapports HTML, suivant la même logique, il peut être déconseillé d'utiliser ce genre de fonctionnalité pour éviter d'avoir des rapports indiquant des consommations très variables selon le contexte temporel utilisé.

Constant ou Persistant

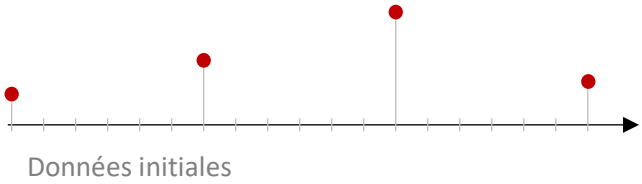
Les agrégations constantes et persistantes sont fortement similaires entre elles et peuvent être source de confusions. Ces deux méthodes d'agrégation servent principalement à recopier une donnée d'un pas de temps important sur plusieurs dates d'un pas de temps plus court.

Plusieurs cas de figure peuvent se présenter :

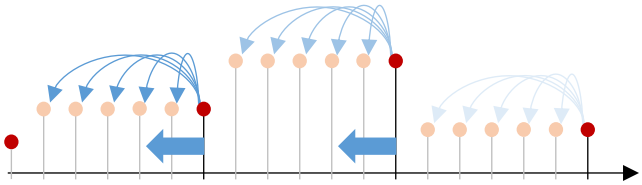
- Un compteur est relevé mensuellement et on veut obtenir un profil horaire avec une répartition constante des consommation (répartition de la conso au prorata du temps)
- Une propriété historisée doit être transformée en profil numérique pour effectuer des calculs (par exemple, la surface d'un bâtiment pour calculer la conso spécifique)
- Un Xtab contient des valeurs tarifaires qui devraient être transformées en profil pour faire du contrôle de facture.
- …

En ce qui concerne les premiers cas (données de consommation ou valeurs de propriétés historisées), EMM (JOOL) considère que la valeur numérique est associée spécifiquement à la date de fin de son intervalle de

validité. (cela est cohérent pour les consommations, on connait la conso d'un intervalle de temps au



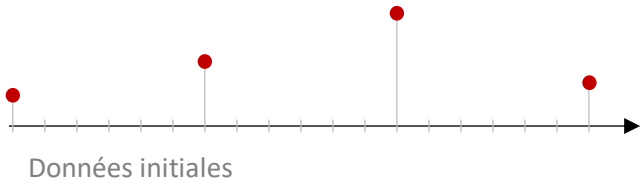
Données initiales



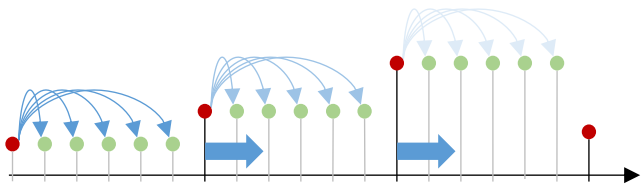
Agrégation « constant »

moment où cet intervalle de temps se termine. On ne connait que rarement la conso d'une période à l'avance…). L'agrégation « constant » ira alors récupérer la valeur de **fin** de l'intervalle de validité d'une conso ou d'une propriété et la recopiera à chaque pas de temps dans cet intervalle de validité.

Dans le cas d'un Xtab de tarification, on a une situation contraire. En effet, la valeur d'un tarif est connue au début de sa période de validité et reste valable jusqu'à ce qu'une nouvelle valeur vienne écraser la précédente. On ne doit donc pas étirer une valeur de fin vers le début d'un intervalle, mais au contraire, étirer une valeur de début vers les pas de temps à venir. L'agrégation « persistant » effectuera donc une



Données initiales



Agrégation « persistant »

récupération de la valeur de **début** de l'intervalle de validité d'un tarif par exemple, et la recopiera à chaque pas de temps dans cet intervalle de validité.

Ces deux méthodes d'agrégation font donc la même chose, mais dans des sens opposés sur la ligne du temps.

SUM ou SUMPROP

La méthode d'agrégation « SUMPROP » va effectuer une somme proportionnelle d'un profil sur un intervalle choisi. Le principe est de ne pas simplement additionner les valeurs présentes dans l'intervalle, mais également d'aller chercher les premières valeurs autour de cet intervalle afin de récupérer la partie de consommation qu'elles couvrent dans cet intervalle et la comptabiliser au prorata de la durée de la période couverte.

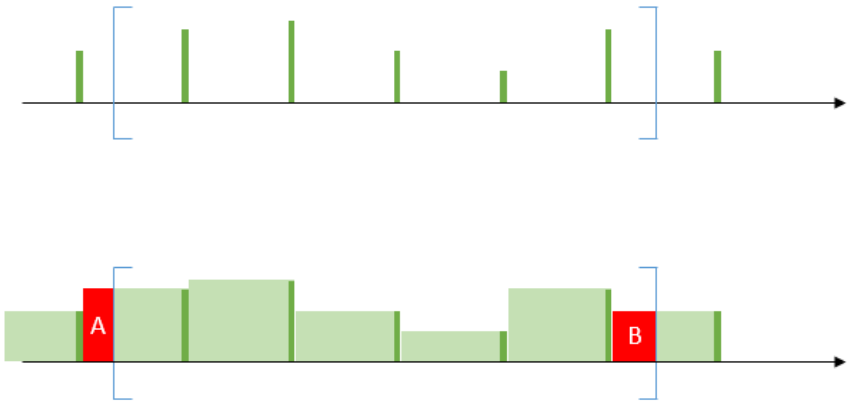
Exemple *:*

Dans le schéma ci-dessous, on voit l'intervalle de temps (crochets bleus) et les mesures de consommation (lignes vertes). Une simple somme va prendre uniquement les valeurs de consommation entre les bornes de l'intervalle et les sommer. Or, chaque mesure de consommation représente la consommation réelle de l'intervalle de temps la précédant, depuis la dernière mesure effectuée.

La première conso de l'intervalle choisi dans le schéma va donc couvrir plus que la période choisie, comptant la zone A en trop par rapport à ce qui est demandé.

De la même manière, la dernière mesure de l'intervalle arrivant avant la fin de l'intervalle, la zone B ne sera pas comptabilisée, alors qu'elle le devrait.

La méthode d'agrégation en SUMPROP va donc aller chercher les premières mesures avant et après l'intervalle choisi, afin de compenser ces erreurs au prorata du temps couvert.



Exemple numérique *:*

Imaginons une relève manuelle d'index donnant les valeurs suivantes :

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

Une agrégation de ces données par mois devra reconstituer les consommations pour chaque mois, du premier du mois au premier du mois suivant.

Dans ce cas, une agrégation par SUM pour janvier donnera 600 + 1000 = 1600

Une agrégation par SUMPROP fera le calcul suivant :

- La consommation du 28/1 est entièrement comprise dans le mois de janvier -> on la comptabilise simplement 1000.
- La conso du 4/1 couvre une partie de janvier, mais aussi une partie du décembre précédent. On va donc comptabiliser uniquement la partie de cette consommation qui se rapporte à janvier, au prorata du nombre de jours couverts.
  Ici donc, la période du 28/12 au 4/1 couvre 7 jours (le 4/1 est exclu puisque l'on s'arrête le 4/1 à 00:00), dont 3 jours en janvier. On garde donc 600*3/7
- De la même manière, la consommation du 3/2 couvre une partie de janvier également. On a là une période de 6 jours, dont 4 en janvier -> On gardera 750*4/6

Bilan de la consommation pour janvier : 600*3/7 + 1000 + 750*4/6 = 1757.14

Remarque : Dans le cas d'une consommation mesurée en index, il faut 2 valeurs consécutives pour reconstituer une consommation (par différence des index consécutifs). Il faudra donc à la méthode SUMPROP, 2 valeurs extérieures de chaque côté de l'intervalle au lieu d'une seule.

Il est donc impératif d'avoir une valeur (A) avant la 1ère période pour laquelle on souhaite une valeur et une valeur (B) après la dernière période pour laquelle on souhaite une valeur. (il en faudra même 2 dans le cas de données encodées en index, puisque les consommations sont récupérées par différence de 2 index consécutifs)

Le problème de l'agrégation en SUMPROP est qu'elle dépend du contexte temporel dans lequel on travaille et qu'elle peut donner lieu à des effets de bord indésirables.

Une solution alternative à l'utilisation de SUMPROP sera l'utilisation de la fonction .slp

.slp

La fonction .slp permet, à partir d'un profil de données, de créer un autre profil de données avec un pas de temps plus court. L'idée est de pouvoir répartir une valeur mesurée grossièrement en données plus précises, sur base d'un profil standard existant (**S**ynthetic **L**oad **P**rofile)

Par exemple, on pourrait transformer un profil relevé manuellement une fois par mois en un profil journalier. On aura pour cela besoin d'avoir préalablement injecté dans EMM (JOOL) un canal spécial, contenant le profil standard que nous allons utiliser pour effectuer notre répartition.

Ce profil peut être constant pour obtenir une répartition équitable des consommations sur la période de mesure, ou il peut avoir n'importe quelle forme en fonction de la répartition désirée (par exemple, on consommera 2X pendant les heures d'ouverture et X pendant les heures de fermeture. Le SLP assurera une répartition correcte de la consommation réelle selon le nombre d'heures d'ouverture/fermeture).

La fonction .slp prendra alors simplement ce profil de données comme argument.

Exemple :

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

→ On utilise comme argument de la fonction .slp un profil modèle de valeurs standard (ici, @SLP_S11_D) et la fonction produira alors un profil de consommation réaliste, basé sur le profil modèle, mais avec la bonne consommation totale pour la période.

(Remarque : il faut bien choisir le profil modèle utilisé pour coller au mieux à la réalité)

(Remarque : les données reprises ici pour le compteur sont celles de son canal par défaut puisque l'on n'a pas de précisions supplémentaires dans la syntaxe)

Remarque avancée *:*

Une fois cette fonction .slp utilisée, l'objet actif est le profil modèle. Cela peut être un problème si par la suite, on doit aller chercher une propriété du compteur actif, ou son parent…

→ On peut donner à la fonction .slp un deuxième argument, (optionnel) qui indiquera à EMM (JOOL) quel compteur doit être l'objet actif à la fin de la fonction.

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

→ Donnera donc exactement la même chose que sans le deuxième argument, à ceci près que l'objet actif à la fin de la fonction est maintenant le compteur C1 et plus le compteur SLP contenant le profil standard.

Cette fonction .slp peut s'appliquer à n'importe quel profil de données, qu'il vienne d'un canal (.data), d'un tableau ou d'un XTab (voir plus loin) ou d'une série de factures (voir plus loin). C'est cette fonction .slp qui nous permettra par exemple de répartir des consommations stockées dans des factures mensuelles et de reconstituer des profils de consommations cohérents sur les périodes facturées.

Remarque : Certains profils de répartition sont contractuels et utilisés par les fournisseurs d'énergie eux-mêmes pour effectuer leurs répartitions de consos mensuelles afin de réaliser leur facturation. Dans le cadre d'un éventuel contrôle de facture, il est donc utile de connaitre les profils SLP utilisés par les fournisseurs dans leurs calculs, afin de pouvoir reconstituer les factures sur les mêmes bases.

.shift

La fonction .shift s'applique à un profil (collection de date/valeurs) et renvoie le même profil de valeurs décalées dans le temps de la période indiquée en paramètres.

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

→ Renverra le même profil que @C1.data mais avec des dates décalées d'un mois, depuis le passé vers le présent (valeur négative dans le shift).

L'intérêt de cette fonction est de pouvoir afficher des séries de valeurs provenant de périodes différentes sur un même graphe temporel (ex : comparaison entre les consommations de cette année avec celles de l'année dernière).

En effet, sans cela, le système essaierait d'afficher les 2 consommations en prenant en X leurs dates réelles, ce qui conduirait à un graphe très étendu en X, avec les données de l'an dernier tout à gauche et les données de cette année tout à droite du graphe.

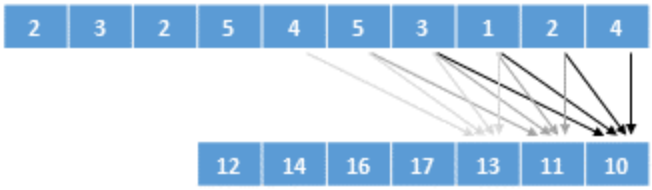
.derivative

La fonction .derivative s'applique à un profil (collection de date/valeurs) et renvoie un nouveau profil constitué des différences des valeurs successives du profil initial (dérivée finie).

Cela est très utile quand on a un profil d'index relevés et que l'on veut construire un profil de consommations. Dans ce cas, la consommation d'une période correspond à la différence entre l'index à la fin de cette période et l'index précédent (fin de la période précédente). La fonction .derivative effectue ces différences et constitue le profil de consommations finales.

.queue

Appliquée à un profil de données, cette fonction fournit un nouveau profil où chaque valeur correspond à la somme des valeurs des n dernières valeurs du profil initial.



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

→ Renvoie une série de valeurs, où chaque valeur est la somme des 4 dernières valeurs du profil data du canal

Exemple d'utilisation *:* On pourrait vouloir un tableau dans lequel on s'intéresse non seulement à la valeur actuelle, mais aussi à la moyenne des 7 derniers jours.

(@C1.data.**queue**(7))/7 → Renverra la liste des moyennes des 7 dernières données.

Attention : Etant donné que cette fonction effectue une somme des « n » dernières valeurs, on peut voir apparaitre des effets de bord perturbants en début de série de valeurs. En effet, si l'on part d'un profil tronqué démarrant au 1° janvier et que l'on fait une « .queue » de 4 jours, les 3 premières valeurs du profil généré se baseront sur 1, 2 puis 3 valeurs et non les 4 attendues. On aura donc un démarrage de profil incomplet. Pour résoudre ce problème, on peut parfois ruser, et faire en sorte que le profil initial démarre un peu plus tôt que prévu (ici, 3 jours), pour ensuite appliquer le « .queue » puis s'assurer de ne garder que les données de la plage de temps désirée (avec un .where filtrant les données hors contexte par exemple)

La situation initiale serait alors la suivante :

Profil démarrant le 1/1

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

Profil .queue(4)

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

La situation corrigée pourrait alors ressembler à ceci :

Profil anticipé de 3 jours

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

Profil .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

Profil .queue(4) tronqué

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

Ce qui nous ramène au profil espéré.

Conversion automatique des unités

Certaines valeurs sont stockées en profils avec leurs unités (les mesures de consommation par exemple, peuvent être stockées en Wh ou en kWh). Cette unité est définie dans les propriétés de mesure, généralement dans les propriétés du canal concerné.

Il est alors possible de faire une conversion automatique des unités de ces valeurs via EMM (JOOL), en notant la nouvelle unité entre accolades.

selection.data → renvoie un profil dans l'unité dans laquelle les mesures sont encodées.

selection.data **{kWh}** → renverra les mêmes données, mais converties en kWh.

Cette conversion ne se fera évidement que s'il y a un lien possible entre les unités. Impossible de convertir spontanément des degrés en mètres carrés…

On remarquera que le changement d'unité s'applique sur l'objet profil tout entier, et pas sur les valeurs numériques qui seraient récupérées après un .value. En effet, pour certaines conversions (kW vers kWh par exemple), il est nécessaire de connaitre le pas de temps entre les données afin d'appliquer les bons coefficients de transformation.

Filtrer des profils (.where)

Les profils étant des collections d'objets (couples date/valeur), il est possible de les filtrer en utilisant la fonction .where sur base d'un critère donné.

Exemples :

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

→ On ne récupère que les mesures dont la valeur est positive

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

→ On ne récupère que les données qui ont moins d'un mois d'âge.

.min / .max

Appliquées à un profil de données (collection de couples date/valeur), ces fonctions vont chercher les plus petites/grandes valeurs dans les données (la recherche se fait donc bien sur le champ « valeur » des données), puis renvoyer la ou les donnée(s) qui ont la plus petite/grande valeur.

Si le min/maximum est atteint une seule fois, on aura une seule donnée comme résultat, mais s'il y'a exæquo, la fonction renverra toutes les données partageant cette valeur min/maximum.

Exemples :

selection.data.**max** → renverra la/les donnée(s) ayant la plus grande valeur

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

→ renverra la dernière instance des données partageant la plus grande valeur

Si le but est d'uniquement récupérer la valeur maximale atteinte, sans s'intéresser au nombre de fois où elle est atteinte ou aux dates où cela est arrivé, on pourra par exemple utiliser la syntaxe suivante selection.data.value.max

En effet, dans ce cas, le profil est d'abord converti en collection de valeurs numériques, puis la fonction .max retrouvera la plus grande valeur, unique cette fois, puisque l'on est dans le cas de la fonction .max appliquée à une collection de numériques.

.sum / .avg / .std / .percentile

Ces fonctions mathématiques et statistiques, appliquées à un profil de données, convertiront le profil de données en collection de numériques (les valeurs des données du profil) et exécuteront leurs calculs sur cette collection. Ces formules vont en effet renvoyer une valeur numérique, reflétant la somme des données, leur moyenne, leur déviation standard, ou le percentile demandé.

Exemples :

selection.data.**sum**

→ Renvoie la somme des données de la sélection (pour le contexte actif)

selection.data.**avg**

→ Renvoie la moyenne des données de la sélection (pour le contexte actif)

selection.data.**std**

→ Renvoie la déviation standard de l'échantillon de valeurs des données

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

→ Renvoie la valeur de l'échantillon de données correspondant à son 95° percentile.

Autres fonctions

Toutes les autres fonctions applicables aux collections restent applicables aux profils de données. On pourra par exemple utiliser la fonction .orderby pour trier les données selon un ordre donné (valeurs décroissantes par exemple). On pourrait aussi compter le nombre de données avec un .count ou ne garder que les 5 dernières données avec un .flop(5)

Exemples :

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

→ Renvoie les données de la sélection, mais triées par valeurs croissantes.

selection.data.**count**

→ Renvoie le nombre de données présentes sur la sélection, dans le contexte temporel actif.

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

→ Renvoie les 10 dernières valeurs présentes sur le profil de la sélection

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

→ Renvoie les données agrégées des 3 mois ayant eu la plus grosse consommation dans le contexte temporel actif, pour la sélection active.

g) Combiner des profils

Les profils étant des objets dans EMM (JOOL), il est possible de les combiner de diverses manières. On peut par exemple les additionner simplement :

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

→ Renverra un profil pour lequel chaque valeur sera la somme des valeurs des deux profils de départ.

De la même manière, on pourra également soustraire des profils entre eux ou les multiplier.

Attention : la syntaxe (@C1.data; @C2.data).sum quant à elle, ne fonctionnera pas. En effet, cette syntaxe va commencer par constituer une large collection reprenant tous les couples date-valeur des profils de chaque compteur et ne saura pas comment les additionner par la suite.

Dans le cas où l'on voudrait sommer un nombre variable de canaux, il faudra donc ruser. On pourra par exemple utiliser la syntaxe suivante :

```
(@C1; @C2) .data
```

→ Dans ce cas, EMM (JOOL) va constituer une collection de canaux, puis prendre tous les profils associés à ces canaux, et par défaut, les additionner entre eux.

Cette astuce ne fonctionnera cependant que pour des canaux à additionner, puisque c'est l'opération par défaut que fait EMM (JOOL) quand il a plusieurs profils.

h) Profils conditionnels

On peut également combiner des profils sur base conditionnelle. On peut ainsi créer un profil vrai/faux, qui sera matérialisé par un profil 0/1 sur base d'une condition comme par exemple une comparaison entre 2 canaux.

Exemple *:*

```
(@C1.data > 0)
```

→ Renverra un profil, qui sera constitué de 1 quand le profil de données du compteur C1 (canal par défaut puisque pas de précisions supplémentaires dans la syntaxe) aura une valeur positive, et de 0 quand il sera négatif.

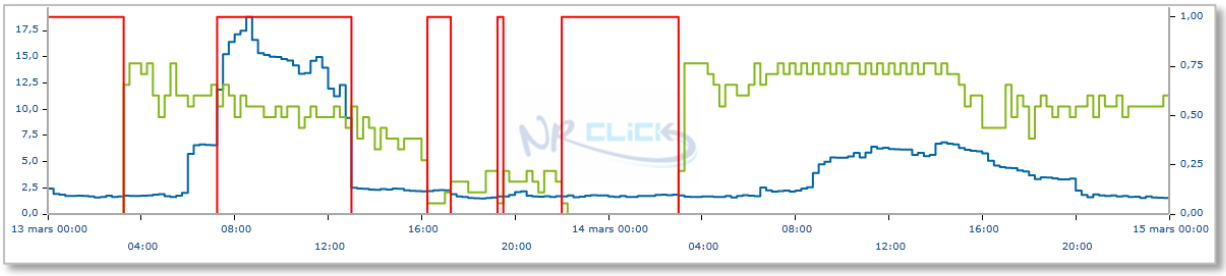
Ces conditions peuvent également combiner plusieurs canaux

Exemple :

```
(@C1.data > @C2.data)
```

→ Le profil renvoyé sera constitué de 1 quand les données du compteur C1 seront supérieures à celles du compteur C2, et de 0 dans le cas contraire.

Dans l'exemple ci-dessous, les profils verts et bleus sont combinés de la sorte pour générer le profil 0/1 rouge qui est donc à 0 quand le profil vert est plus grand que le bleu, et à 1 dans le cas contraire.

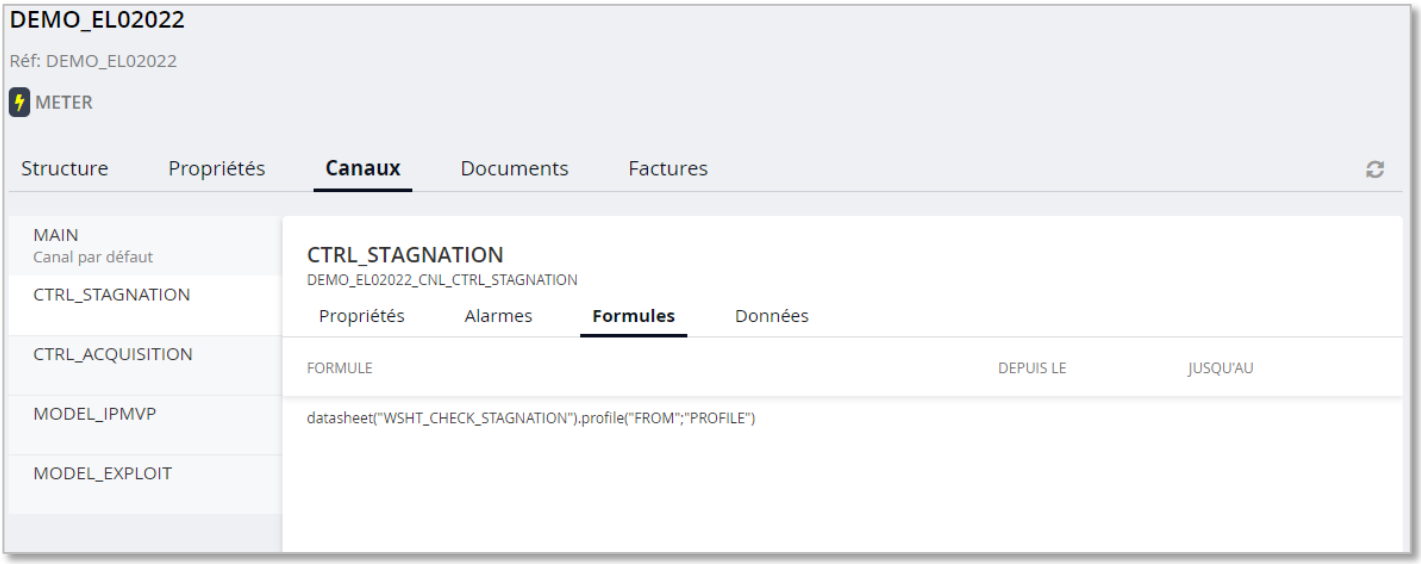


Ce genre de combinaison pourra être utile par exemple dans le cas où l'on aurait plusieurs profils d'activation (p.ex. un ON/OFF pour les week-ends, et un autre ON/OFF pour les vacances scolaires) que l'on voudrait combiner pour obtenir un profil d'activation général.

i) Formules des canaux virtuels

Comme cela a déjà été évoqué, il y'a plusieurs manières d'alimenter le profil d'un canal dans EMM (JOOL). Un canal peut recevoir des données directement par injection externe, en provenance de dataloggers ou d'encodages manuels, mais il peut aussi être le résultat du calcul d'une formule, basée sur d'autres profils ou sur des valeurs de propriétés.

Les canaux ainsi calculés sont appelés des canaux « virtuels » et ils contiennent, dans l'onglet « Formule » de leur fiche canal, une formule en syntaxe EMM (JOOL), éventuellement historisée, et qui décrit la manière dont le profil est calculé (exemple ci-dessous).



Ces formules peuvent faire appel à n'importe quelle information présente dans EMM (JOOL), elles doivent renvoyer un profil de données, et le résultat de la formule est stocké dans les RawData du canal concerné. (attention à ne pas lui fournir en plus un poids de pulse sinon celui-ci sera appliqué à nouveau lors du passage de RawData à Data)

Exemples de formules :

@C1.data + @C2.data

→ Cette formule indique que le canal actuel est un canal calculé, faisant la somme de la conso du compteur C1 (canal par défaut) et celle du compteur S2 (canal par défaut également). Cette situation peut arriver par exemple on veut créer un compteur de tête virtuel au-dessus de plusieurs sous-compteurs installés sur un site.

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

→ Le profil Data du canal actuel sera cette fois le même que celui du compteur C1, mais avec des données agrégées (en somme) par mois.

On pourra évidement mixer toutes ces façons de faire pour constituer le profil désiré de la manière la plus efficace.

Exemple avancé :

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

→ On démarre avec le compteur C1 et on constitue un profil de données sur base des valeurs de propriété TECH_CONS_TOTAL de ses factures. Une fois ce profil constitué, on utilise la fonction .slp pour répartir cette consommation facturée en suivant le profil SLP qui est relié au compteur C1 via un link de type SLP. Cette façon de faire requiert donc que le type de lien SLP existe et qu'un compteur SLP de répartition soit lié au canal actuel via un lien de ce type.

En partant des données de consommations reprises sur des factures mensuelles, on peut donc aisément créer un profil quotidien (voire par pas de 10 minutes).

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

Si dans le parseur, on a besoin de récupérer la formule d'un canal, on peut utiliser la fonction « .formula ». Cette fonction, quand elle est appliquée sur un canal, renverra l'ensemble des blocs de formule associés à ce canal (en cas de sélection multiple, elle renverra la totalité des blocs de formules de chacun des canaux de la sélection).

Une fois sur un bloc de formule, on peut alors utiliser les fonctions .from / .to / .value pour récupérer respectivement la date de début de validité de la formule, sa date de fin de validité, et le texte de la formule.

6. Tableaux de données

Un tableau de données est constitué de lignes et de colonnes, et que l'on peut utiliser comme un objet dans la syntaxe EMM (JOOL) pour lui appliquer diverses transformations.

Un tableau de données peut être le résultat du calcul d'un DataSet, ou le résultat d'une feuille de calcul (Worksheet), ou encore un Xtab, ou même de combinaisons de ces derniers.

Exemples :

```
dataset("REF_DE_DATASET")

worksheet("REF_DE_WORKSHEET")

xtab("REF_DE_XTAB")
```

Ces trois expressions renverront toutes un tableau de données, chacun construit sur base des résultats du dataset, du workshet ou du Xtab demandés.

Dans la syntaxe EMM (JOOL), un tableau de données est considéré comme une collection de lignes, chaque ligne étant constituée d'autant de champs que le tableau n'avait de colonnes.

Fréquemment, on aura besoin d'exécuter un dataset, une feuille de calcul ou un Xtab sur base d'une autre sélection ou d'un autre contexte que les sélections ou contextes actifs. Pour cela, on peut passer des arguments optionnels supplémentaires aux fonctions dataset, worksheet ou xtab.

Exemple :

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

Cette expression va passer explicitement la sélection et le contexte actifs à la fonction worksheet pour qu'elle les utilise comme sélection et contextes actifs lors de l'exécution de la feuille de calcul. En l'occurrence, cela ne change rien au résultat, mais il est évidemment possible de modifier la sélection et le contexte passés en arguments.

Exemple :

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

→ Cette expression va exécuter la feuille de calcul "REF_DE_WSHT" sur base des enfants de la sélection active, pour la période allant du début de l'année contenant le début du contexte actif (from), jusqu'au début de la journée contenant la fin du contexte actif (to).

On notera que si l'on veut passer un contexte modifié mais sans toucher à la sélection active, il faudra quand même indiquer la sélection (inchangée du coup) dans la liste des arguments. En effet, c'est l'emplacement des arguments qui leur attribue une signification. Le premier argument est la référence de la feuille de calcul à exécuter, le deuxième argument (optionnel) est la sélection et les 3° et 4° (optionnels également) sont les from/to du contexte à utiliser.

Remarque : la plupart du temps, on appellera un tableau de données dans la case sélection d'une feuille de calcul pour servir de source à cette feuille et y récupérer des données pour les travailler. Il est cependant possible également d'appeler un tableau de données directement dans la formule d'une colonne d'une feuille de calcul. On pourrait par exemple appeler un tableau de données pour récupérer la somme des valeurs de l'une de ses colonnes et l'injecter dans la case de ma nouvelle feuille de calcul. Cette façon de faire est peu recommandée, parce qu'elle a tendance à être très gourmande en ressources et en temps de calcul. En effet, dans cette situation, le tableau source sera appelé et vraisemblablement calculé pour chaque ligne de la feuille de calcul, avec tout ce que cela peut impliquer en termes d'accès à la base de données et en temps de calcul.

Remarque : il est possible d'appliquer les fonctions « .creation » et « .lastupdate » aux DataSets, Worksheets et Xtabs. Ces fonctions renverront un bloc d'informations contenant la date de création / mise à jour et l'utilisateur qui a effectué cette création / mise à jour. Il faudra alors appliquer les fonctions « .date » ou « .user » pour récupérer alors l'information désirée.

```
Dataset("DSET_REF").creation.user.mail
```
→ renverra l'adresse email du créateur de ce DataSet

```
worksheet("WSHT_REF").lastupdate.date
```
→ renverra la date de dernière mise à jour de cette feuille de calcul

A. Colonnes d'un tableau

Si l'on met un appel à un tableau (comme ceux présentés ci-dessus) dans la case sélection d'une feuille de calcul, EMM (JOOL) l'interprètera donc comme une collection de lignes, et préparera une ligne dans la feuille de calcul pour chaque ligne dans le tableau source.

L'item active pour chaque ligne de la feuille de calcul sera alors un objet « ligne » du tableau source.

item.column

Pour accéder alors aux informations présentes dans cette ligne source, on pourra utiliser la fonction .column qui accèdera aux différents champs de la ligne en question en se basant sur le nom de la colonne correspondante dans le tableau initial.

Exemple :

Imaginons qu'un DataSet « DSET_01 » produise le tableau de donnée suivant :

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

Si dans la case sélection d'une feuille de calcul, j'appelle ce DataSet avec la syntaxe suivante : dataset("DSET_01"), ma feuille de calcul devrait contenir 7 lignes, le même nombre que le DataSet qu'elle utilise comme source.

Dans la première colonne de ma feuille de calcul, que j'appellerai « REF », je peux alors faire appel aux valeurs des différentes colonnes de mon DataSet source.

```
item.column("REFERENCE")
```

→ récupère la colonne « REFERENCE » de mon item (la ligne de ma source), et en affiche la valeur dans ma feuille de calcul, dans la colonne « REF ».

Dans la feuille de calcul, cette colonne recopiera alors la colonne « REFERENCE » de mon DataSet source. La feuille de calcul résultante sera donc la suivante :

REFERENCE
BXL_001
BXL_002
LLN_001
LLN_002
LLN_003
LG_001
LG_002

column

Dans la feuille de calcul, on peut donc accéder aux lignes d'un tableau source via l'expression item.**column**("REF_COL_SOURCE").

Si maintenant le but est d'accéder à une autre colonne, non plus du tableau source, mais de la feuille de calcul elle-même (récupérer la colonne d'à-côté par exemple), on peut utiliser la fonction **column**("REF_COL_FEUILLE").

Ces deux syntaxes sont très proches, mais conceptuellement bien différentes. La première récupère les colonnes d'une source présente dans l'item active, alors que la seconde fait appel au résultat présent dans une autre colonne de la feuille active.

Exemple :

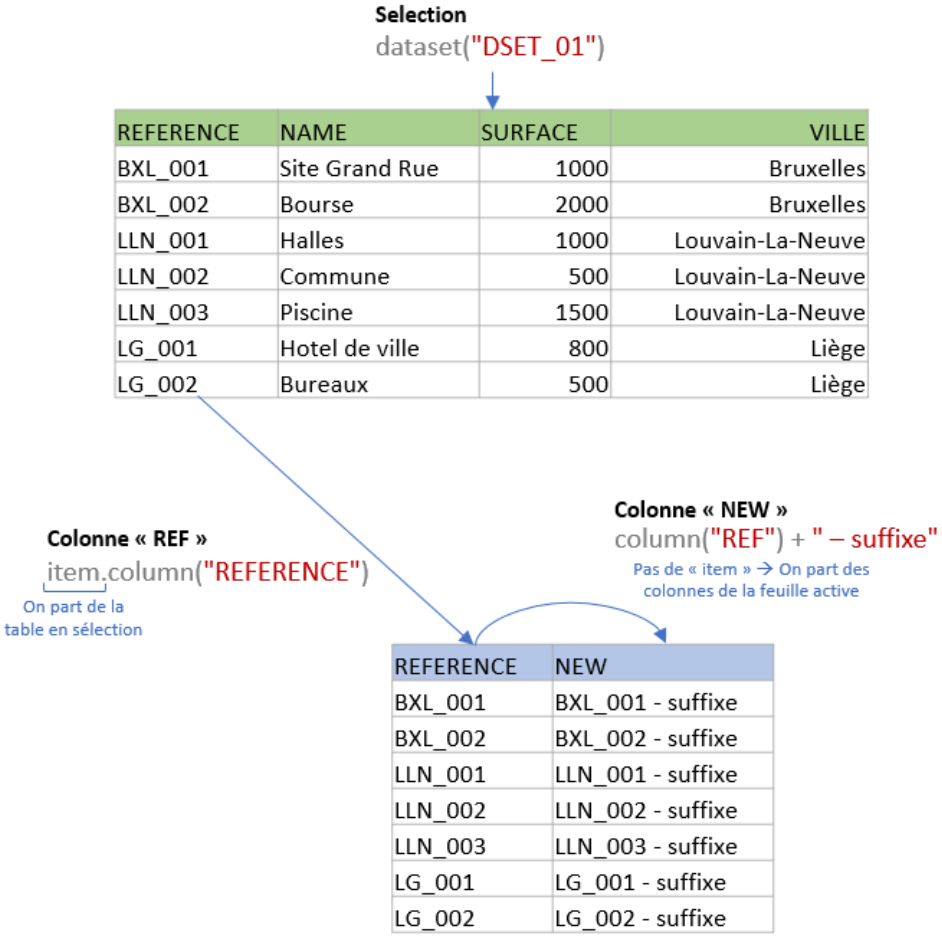
Si l'on continue l'exemple précédent, on pourrait par exemple ajouter une colonne à la feuille de calcul et lui donner comme formule l'expression suivante : column("REF") + " – suffixe"

Cette syntaxe ira alors récupérer les informations présentes dans la colonne « REF » de la feuille de calcul active (la colonne d'à-côté), et y ajoute un suffixe « – suffixe »

On obtient alors la feuille suivante :

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

L'information parcourt le chemin suivant :



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

cumul

Assez régulièrement, il est utile de calculer une consommation cumulée. On aurait par exemple une colonne avec la consommation, et on voudrait une autre colonne qui renverrait pour chaque ligne, la somme des consommations des lignes précédentes.

La fonction cumul permet d'effectuer ce genre de cumul de valeurs.

Exemple :

COL_A = colonne de valeurs numériques

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

La colonne « COL_B » renverra une colonne de valeurs numériques représentant les valeurs cumulées de la colonne « COL_A ».

Attention : Dans une feuille de calcul, la colonne à cumuler et la colonne cumulée doivent toutes deux être en format numérique sans quoi on aura un message d'erreur.

Utilisation Avancée

Dans le cas d'une feuille de calcul qui afficherait les consos de plusieurs compteurs les unes en-dessous des autres, le cumul devrait se faire par compteur et non de façon globale. Il faudrait donc pouvoir remettre à zéro le cumul sur base d'une colonne de référence. On aura la situation suivante :

COL_REF = colonne listant les références des compteurs

COL_A = colonne de valeurs numériques (plusieurs valeurs différentes pour chaque référence)

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

La colonne « COL_B » renverra cette fois-ci une colonne de valeurs cumulées mais à chaque changement de valeur dans la colonne « COL_REF », le cumul reprendra à zéro.

Le 2° argument de la fonction cumul est optionnel et il permet de limiter les cumuls calculés aux lignes de même type. Le type étant, dans le cas ci-dessus, le compteur considéré. On aura donc un cumul par site. Cette colonne de référence peut évidemment contenir autre chose que des références en fonction des situations.

derivative

Similaire à la fonction cumul, mais qui effectue une dérivée (deltas successifs) au lieu de faire un cumul. La syntaxe est similaire.

Exemple :

COL_A = colonne de valeurs numériques

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

La colonne « COL_B » renverra une colonne de valeurs numériques représentant les différences entre valeurs successives de la colonne « COL_A »

Avec la fonction derivative, la première valeur de la colonne dérivée sera toujours nulle (NaN) puisque l'on ne connait pas la valeur précédente, et donc impossible de faire le delta.

Utilisation Avancée

Dans le cas d'une feuille de calcul qui afficherait les index de plusieurs compteurs les uns en-dessous des autres, la dérivée devrait se faire par compteur et non de façon globale. Cela a comme seul impact qu'il ne faut pas tenir compte des valeurs successives lors d'un saut entre deux compteurs différents. Il n'y a en effet aucune signification pratique à la différence entre le dernier index d'un compteur et le premier index d'un autre compteur.

COL_REF = colonne listant les références des compteurs

COL_A = colonne de valeurs numériques (plusieurs valeurs différentes pour chaque référence)

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

La colonne « COL_B » renverra cette fois-ci une colonne de valeurs dérivées (delta successifs) mais à chaque changement de valeur dans la colonne « COL_REF », la dérivée aura une case vide puisque la suite de delta redémarre à zéro.

Le 2° argument de la fonction derivative est donc bien optionnel et il permet simplement de couper les dérivations calculées aux lignes de même type. Le type étant, dans le cas ci-dessus, le compteur considéré. On aura donc une suite de deltas par site, affichant alors une case vide à chaque changement de valeur de la colonne de référence (le premier delta n'a pas de valeur précédente, donc pas moyen de calculer une différence). Cette colonne de référence peut évidemment contenir autre chose que des références en fonction des situations.

total

Toujours dans le même ordre d'idée que les fonctions cumul et derivative, il est parfois utile de récupérer la valeur totale d'une autre colonne, idéalement sans avoir à créer des feuilles de calcul supplémentaires. Un exemple classique d'utilisation serait par exemple la constitution d'une table dans laquelle on aurait les consommations de plusieurs compteurs (un par ligne p.ex.), et que l'on voudrait une colonne affichant le pourcentage de la conso totale associé à chaque compteur. Il faudrait donc diviser la conso de chacun par la conso totale et on a donc besoin de la somme de la colonne des consos pour calculer cette conso totale.

La fonction total, utilisant la même syntaxe que les fonctions cumul et derivative, effectue ce total d'une colonne numérique et en recopie la valeur dans chaque case de la colonne total.

Exemple :

COL_A = colonne de valeurs numériques

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

La colonne « COL_B » contiendra alors dans chacune de ses cases la valeur totale de la colonne numérique « COL_A ».

Utilisation Avancée

Dans certains cas, on pourrait vouloir que la colonne TOTAL pourrait ne pas contenir un total absolu d'une autre colonne initiale, mais des sous-totaux par tranches de cette colonne initiale. Par exemple, on pourrait avoir une table avec plusieurs compteurs (colonne « REF_CPT »), répartis en plusieurs groupes de compteurs (colonne « GROUPE »), et une colonne contenant la consommation de chaque compteur. Dans ce cas, il pourrait être intéressant d'avoir non seulement la consommation totale absolue de tous les compteurs, mais aussi pourquoi pas la consommation totale de chaque groupe

La situation serait alors la suivante :

GROUPE = colonne listant les références de groupes dans lesquels sont répartis les compteurs

COL_A = colonne des consommations

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

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

La colonne « COL_B » renverra cette fois-ci encore une colonne contenant toujours la même valeur : la conso totale de toute la colonne « COL_A ».

La colonne « COL_C » en revanche, contiendra quant à elle les consommations totales de chaque groupe, identifiées par les valeurs de la colonne « GROUPE ».

Le 2° argument de la fonction total est donc bien optionnel et il permet de limiter les totaux calculés à des sous-totaux, basés sur des lignes de même type. Le type étant, dans le cas ci-dessus, le groupe considéré.

linereg

Enfin, il est possible d'effectuer une régression linéaire entre deux colonnes numériques présentes dans le tableau, et de récupérer les paramètres de cette régression.

Exemple :

COL_A = colonne de valeurs numériques

COL_B = colonne de valeurs numériques

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

Dans cet exemple, la colonne « COL_C » effectuera une régression linéaire en partant des colonnes « COL_A » et « COL_B », respectivement X et Y de la régression (l'ordre est donc important), puis elle renverra les valeurs du « PARAMETRE » demandé.

Les paramètres possibles sont les suivants :

- **DATA :** renvoie, pour chaque ligne, la valeur Y de la droite de régression linéaire associée au X de la ligne en cours.
- **FORMULA :** renvoie, pour chaque ligne, la formule de la droite de régression linéaire. (en format texte, et la même formule pour toutes les lignes)
- **FORMULA_ROUNDED :** renvoie, pour chaque ligne, la même formule que ci-dessus, mais arrondie à 1 chiffre après la virgule. (souvent plus lisible pour un rapport)
- **INTERCEPT :** renvoie l'intercept de la droite de régression linéaire. (valeur identique pour toutes les lignes)
- **SLOPE :** renvoie la pente de la droite de régression linéaire. (valeur identique pour toutes les lignes)
- **RMSE :** renvoie l'erreur quadratique moyenne. (Racine carrée de la moyenne des différences au carré, valeur identique pour toutes les lignes)
- **CVRMSE :** renvoie le coefficient de variation de l'erreur quadratique moyenne par rapport à la moyenne des points. (valeur identique pour toutes les lignes)
- **R2 :** renvoie le coefficient de détermination de la droite de régression linéaire. (valeur identique pour toutes les lignes)
- **MBE :** renvoie la mean bias error. (valeur identique pour toutes les lignes)
- **COV :** renvoie la covariance, la moyenne du produit des 2 valeurs moins le produit des 2 moyennes. (valeur identique pour toutes les lignes)

Utilisation Avancée

Comme pour les fonctions « total », « derivative » et le « cumul », il peut parfois être utile d'effectuer non pas une régression linéaire complète sur la totalité des lignes mais bien sur des tranches de données successives. Exemple : dans le cas d'un système de conditionnement d'air, on pourrait vouloir une régression linéaire des consommations par rapport aux DJU. Cependant, on pourrait vouloir deux droites de régression distincte, l'une pour la consommation de chauffage et l'autre pour la consommation de climatisation quand les DJU dépassent un seuil donné.

Dans ce cas, on peut utiliser une ou plusieurs colonne(s)comme argument(s) complémentaire(s) dans la fonction « linereg ». Les changements de valeur dans ces colonnes serviront alors de déclencheur pour couper une tranche de données sur laquelle effectuer la régression linéaire.

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

Dans le cas de l'exemple des DJU développé ci-dessus, on pourrait par exemple trier les données par DJU croissant, puis créer une colonne indiquant si l'on est en-dessous ou au-dessus du seuil, et se servir de cette colonne comme paramètre de réinitialisation pour la fonction « linereg ».

B. Créer une série de valeurs (range)

On a parfois besoin de créer une série de valeurs, régulièrement espacées, pour servir de base à un calcul. On peut par exemple avoir besoin de générer 10 lignes numérotées de 1 à 10, ou de générer une ligne par jour du contexte actif. La fonction range permet de créer de telles collection de valeurs en série.

Cette fonction génère en fait un tableau de données qui contiendra juste une colonne dont le nom est le premier paramètre à donner à la fonction, et avec autant de lignes que nécessaire pour partir de la valeur de départ et avancer (sans la dépasser) la valeur de fin de la série.

Syntaxe pour une série numérique :

```
Range("NOM" ; Début ; Fin ; Step)
```

- "NOM" est un nom à donner à la série ainsi générée (pour utilisation ultérieure)
- Début est la valeur de départ de la série
- Fin est la borne maximale de cette série
- Step est le pas par lequel on progresse du début vers la fin de série

Syntaxe pour une série de dates :

```
Range("NOM" ; Date_début ; Date_fin ; Step_number ; "Step_type" )
```

- "NOM" est un nom à donner à la série ainsi générée (pour utilisation ultérieure)
- Date_début est la date de départ de la série temporelle
- Date_fin est la borne de fin de la série temporelle
- Step_number est le nombre d'intervalles pour un pas de temps
- "Step_type"est le type d'intervalle pour un pas de temps

Exemples :

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

→ renverra la série de valeurs 1 2 3 4 5 6 7 8 9 10, dans une table contenant une seule colonne appelée « A », avec une valeur par ligne.

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

→ renverra la série de valeurs 1 3 5 7 9. La série ne va pas plus loin puisque la valeur suivante (11) serait au-delà de la borne maximale fournie à la fonction range (10).

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

→ renverra une série de dates, démarrant avec la date du début de contexte actif, et progressant par pas d'une journée (1 ; "day"), jusqu'à ne pas dépasser la date de fin de contexte actif.

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

→ renverra une série de dates, allant du premier janvier 2000 au premier janvier 2001, par pas d'un mois.

Dans tous les exemples ci-dessus, si l'on met cette syntaxe dans la sélection d'une feuille de calcul, chaque ligne de cette feuille sera associée à une ligne de la table source, et on pourra récupérer la valeur associée via la fonction : item.column("A"), comme s'il s'agissait d'une table tout à fait classique.

C. Filtrer un tableau

Un tableau en sélection étant considéré comme une collection de lignes, il est possible de lui appliquer les fonctions habituelles des collections, en utilisant les valeurs présentes dans les colonnes de la source.

.orderby / .top / .flop

La fonction .orderby permet de trier les lignes du tableau source selon un critère donné. On pourrait par exemple décider de trier les lignes d'un tableau de consommation par ordre de conso. Pour cela, la case sélection doit contenir un tableau source directement trié.

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

→ cette sélection constituera alors un tableau basé sur le DataSet DSET_CONSOS, ordonné selon les valeurs présentes dans sa colonne « VALUE »

Il est également possible de ne garder que les premières ou les dernières lignes en utilisant les fonctions .top / .flop.

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

→ cette sélection constituera alors un tableau basé sur le DataSet DSET_CONSOS, et ne gardera que les 5 premières lignes du tableau.

En combinant ces fonctions, il est possible de trier les lignes, puis de ne garder que les premières/dernières lignes. On pourrait ainsi ne garder que les plus fortes consommations du tableau par exemple.

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

→ Le DataSet DSET_CONSOS est utilisé pour construire un tableau de source, on en trie alors les lignes suivant la valeur croissante de la colonne « VALUE », et enfin, on ne garde que les 5 dernières lignes, contenant les pires consommations au vu du tri effectué précédemment.

.where

Il est également possible de filtrer les lignes d'un tableau en utilisant la fonction .where pour supprimer les lignes ne remplissant pas certains critères. On pourrait par exemple démarrer d'un tableau listant tous les compteurs associés à un site, et filtrer dans ce tableau pour ne garder que les compteurs d'électricité, en filtrant sur la colonne contenant la ressource mesurée par le compteur.

Exemple :

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

→ Constitue un tableau sur base du DataSet « DSET_CONSOS », et le filtre pour en supprimer toutes les lignes où la consommation est nulle (on garde ceux où la valeur de la colonne « VALUE » <> 0)

D. param – passer une valeur de variable à l'appel d'un tableau

Lors de l'appel à un tableau de données, il peut parfois être pratique d'y adjoindre une valeur (optionnelle) qui serait utilisée lors du calcul du tableau.

Exemple : Si l'on veut récupérer via un DataSet toutes les informations associées à l'ensemble de compteurs d'électricité ou l'ensemble des compteurs de gaz, il faudrait créer deux DataSets fortement similaires. Le premier récupérant les informations et filtrant sur la ressource pour ne garder que l'électricité, et le second faisant exactement la même chose mais en ne gardant que le gaz.

L'utilisation du paramètre complémentaire « param » permet de créer un unique DataSet, qui filtre sur la valeur du paramètre. On devra alors fournir le paramètre complémentaire « param » lors d'un appel à ce DataSet sans quoi il ne fonctionnera pas.

Syntaxe :

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

- L'argument optionnel param doit être en format texte, mais il peut éventuellement provenir d'une formule encapsulée qui génère un texte.
- Les arguments selection, from et to doivent absolument être renseignés si l'on utilise param. En effet, c'est toujours l'ordre des arguments qui permet à EMM (JOOL) de les identifier et param doit être le 5° argument.

Exemple : en reprenant l'exemple ci-dessus, la syntaxe à utiliser pourrait être la suivante :

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

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

Si à l'intérieur du DataSet, on a utilisé le terme **[param]** (avec les crochets) comme valeur de filtre pour la colonne ressource (en key), le DataSet effectuera le filtre sur les compteurs d'électricité ou de gaz naturel.

L'utilisation du paramètre optionnel param peut également être appliquée aux appels de feuilles de calcul. L'appel se fait de la même manière, en ajoutant un 5° argument à l'appel de la feuille de calcul, et à l'intérieur de cette feuille de calcul, on peut alors utiliser le terme **param** (sans crochet cette fois) comme s'il s'agissait d'une variable qui recevra sa valeur lors de l'appel effectif de la feuille de calcul.

Attention : il est à noter que, de la même manière que pour la selection et le from/to, la valeur du paramètre optionnel param est considéré comme une variable globale à partir du moment où il est défini, en ce sens qu'elle est héritée par tous les tableaux successifs qui pourraient s'enchainer. Si un tableau **A** appelle un tableau **B** en lui passant une valeur de **param**, et que ce tableau **B** n'en fait rien mais qu'il appelle un tableau **C**, sans spécifiquement lui passer l'argument **param** en chemin, le tableau **C** peut quand même utiliser la valeur du paramètre en utilisant simplement le terme **param**.

E. Récupérer une valeur dans un tableau

La première possibilité pour récupérer une valeur précise dans une case d'un tableau, c'est de le filtrer pour ne garder qu'une seule ligne, puis de récupérer la colonne désirée.

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

Cette façon de faire est précise, et permet de récupérer spécifiquement la valeur d'une case du tableau, sur base de la référence de colonne et des critères utilisés pour filtrer correctement la ligne désirée.

Dans le cas où l'on a un Xtab dont la première colonne est sous format date ou numérique, il est également possible d'utiliser une syntaxe abrégée pour récupérer une valeur.

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

Cette syntaxe va aller récupérer, dans le Xtab « REF_XTAB », la colonne « REF_COL », et gardera la valeur de la colonne en question correspondant à la ligne dont la valeur en première colonne est donnée par « VALUE_1E_COL »

Attention, dans le cas où la valeur de 1° colonne donnée n'est pas unique, cette formule va renvoyer l'une des réponses qu'elle trouvera, mais sans aucune certitude qu'il s'agira de celle que l'on voulait initialement. Idéalement, la première colonne ne doit donc contenir que des valeurs uniques si l'on veut utiliser cette méthode de récupération de valeurs.

Si la première colonne est en format date et que ses valeurs sont uniques, il est également possible de récupérer une valeur en fournissant une date approchée de la date en première colonne en indiquant à EMM (JOOL) s'il doit aller récupérer la dernière date avant celle fournie ou la première après.

Exemple :

Imaginons que l'on ait le XTab suivant comme source de travail

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

Si ce Xtab s'appelle « XT_EPS_GF_T01 » et que l'on cherche à récupérer la valeur dans la colonne « Abonnement » en date du 1° janvier 2017.

On remarque qu'il n'y a aucune ligne dans le XTab qui donne une valeur pour cette date précise. La syntaxe classique

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

ne renverra donc rien comme résultat, puisqu'il n'y a aucune ligne portant la bonne date.

La syntaxe doit être légèrement modifiée pour indiquer que, si EMM (JOOL) ne trouve pas pile la bonne ligne de date, il peut aller chercher la plus proche dans le passé, ou dans l'avenir.

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

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

Le symbole « ~ » placé avant la date indiquera à EMM (JOOL) qu'il doit aller récupérer la dernière valeur en date dans le passé (cas le plus fréquent, pour la récupération d'un tarif par exemple)

Si l'on place le symbole « ~ » après la date, EMM (JOOL) ira récupérer la valeur la plus proche dans l'avenir.

F. Générer un profil à partir d'un tableau (.profile)

La fonction .profile permet, sur base d'un tableau de données (DataSet, Worksheet, Xtab, combinaison complexe…), de générer un profil de données utilisable dans la formule d'un canal ou dans n'importe quelle autre syntaxe dans EMM (JOOL).

Cette fonction recevra deux ou trois arguments selon ce qui est disponible en situation.

Syntaxe à 2 arguments :

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

Cette syntaxe fut la première mise en place historiquement. Elle reçoit donc la référence de la colonne contenant les dates « from » du profil de données, et une autre contenant la référence de la colonne contenant les valeurs numériques à utiliser pour le profil. Chaque donnée du profil généré aura alors comme date « from » la valeur disponible dans la colonne "FROM" du tableau source, et la date « to » de chaque donnée est calculée par EMM (JOOL) en utilisant la date « from » de la donnée suivante.

Cette façon de faire marche bien quand on n'a pas de colonne "TO" dans le tableau source, comme dans un Xtab de tarifs par exemple.

Elle peut cependant causer de légers soucis en fin de profil. En effet, vu qu'il n'y a pas de donnée suivante, EMM (JOOL) ne trouve pas de « from » suivant, et ne peut donc pas constituer de « to » à la dernière donnée. La plupart du temps, cela ne pose pas de problème, mais dans de rares cas, c'est une information à prendre en compte.

Syntaxe à 3 arguments :

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

Cette nouvelle syntaxe fonctionne avec trois arguments, permettant de créer des données avec un « from » et un « to » clairement identifiés dans des colonnes du tableau source. Cette façon de faire évite le problème des fins de profils (puisque les « to » sont identifiés sans que EMM (JOOL) n'ait à les calculer), mais elle impose d'avoir une colonne "TO" dans le tableau de source, ce qui n'est pas toujours possible (raison pour laquelle les deux syntaxes cohabitent dans EMM (JOOL)).

Le premier indique la colonne du tableau source qui servira de borne « from » à chaque valeur du profil (cette colonne doit donc évidement être en format « date »). Le deuxième servira de borne « to » à la valeur correspondante, et le dernier paramètre est quant à lui la référence de la colonne contenant les valeurs numériques du profil.

Dans les deux cas, ces formules renverront un profil de données, constitué dynamiquement sur base soit des colonnes "FROM" et "VALUE" du tableau de source, soit à partir des colonnes "FROM", "TO" et "VALUE".

Remarque : Cette fonction .profile peut s'appliquer à n'importe quel type de tableau de données, qu'il provienne d'un DataSet, d'un Worksheet, d'un Xtab ou de toute combinaison possible de ces tableaux.

G. Grouper les lignes d'un tableau

.groupby

Un groupement de lignes à partir d'un tableau source se fera dans la sélection d'une feuille de calcul, en utilisant la fonction .groupby. Ce groupement génèrera une une collection d'objets de type key/elements, ayant comme « key » les différentes valeurs prises par la clé de groupement et comme « elements » la liste des lignes du tableau source partageant cette clé de groupement.

Exemple :

Imaginons qu'il existe un DataSet « A », qui, une fois calculé sur base de la sélection et du contexte actuels donne comme résultat la table suivante (notre tableau source)

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

Si, dans un autre tableau, on passe ceci dans la case sélection :

```
DataSet("A").groupby(item.column("GROUPE"))
```

On obtiendra alors comme source de notre feuille de calcul active le tableau suivant.

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

Ce tableau se compose de 3 objets, 3 lignes composées de 2 colonnes, la colonne « key » et la colonne « elements », contenant elle-même une collection de sous-lignes dans chaque case.

La feuille de calcul comportera donc 3 lignes, et chacune de ces lignes aura comme source une des lignes du tableau source modifié.

On pourra alors utiliser item.key pour accéder à la valeur de la première colonne du tableau source, et item.elements pour accéder à la seconde colonne.

item.elements sera donc ici lui-même un objet de type tableau, et on pourra aller récupérer ses données via la fonction .column pour en faire une somme ou une moyenne par exemple.

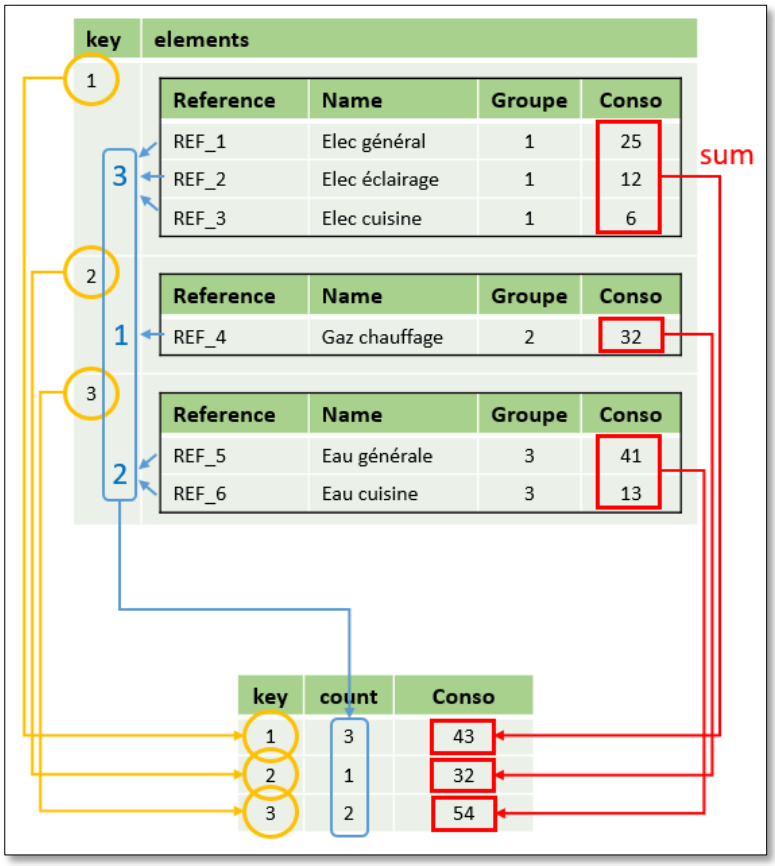
Exemple :

Si le tableau final contient les définitions de colonnes suivantes, on obtiendra le résultat développé ci-contre.

item.key

item.elements.count

item.elements.column("CONSO").sum



La première colonne reprend simplement la clé du groupement

La deuxième colonne récupère, pour chaque valeur de clé, la liste des éléments associés, et en compte le nombre.

La deuxième colonne récupère également, pour chaque valeur de clé, la liste des éléments associés, puis regarde la colonne « CONSO » de chacune de ces lignes, et enfin en fait la somme.

H. Concaténer des tableaux

Au lieu de récupérer les infos d'un seul tableau, il est aussi possible d'en appeler plusieurs (similaires) en même temps en les fusionnant (procédure d'union).

(DataSet("A"); DataSet ("B"))

→ Cette expression va aller chercher les DataSet A et B et constituera un tableau général, dans lequel seront simplement mis bout à bout les deux tableaux.

Si les DataSet ont des colonnes de même nom, les colonnes seront fusionnées et la colonne en provenance de B continuera la colonne démarrée dans A.

Si une colonne n'existe pas dans l'un des DataSet, elle sera créée dans le tableau général et laissée vide pour les lignes du DataSet qui ne la contient pas.

Attention : les types de colonnes doivent être respectés.

Cette façon de fusionner des tableaux permet de rassembler des lignes en provenance de plusieurs tableaux sources, en calculant simplement ces tableaux sources l'un après l'autre.

I. Groupements de tableaux (.join / .joinmulti)

En plus de pouvoir concaténer des tableaux, il est aussi possible de les fusionner selon une clé définie. On pourra ainsi par exemple avoir un tableau avec tous les bâtiments d'un client, et un tableau avec la liste des clients et de leurs informations (nom, téléphone, email…)

Dans un cas pareil, inutile de recopier toutes les infos d'un client dans chaque ligne de ses sites. Premièrement pour une question de taille des bases de données, mais aussi pour une question de nombre d'appels à la DB. En effet, si l'on recopie les infos du client dans chaque ligne, le programme devra aller les relire chaque fois, alors que si l'on crée des tableaux séparés, la lecture aura lieu une seule fois. Par ailleurs, l'information étant stockée à un seul endroit, cela simplifie les modifications éventuelles (changement de numéro de téléphone, déménagement…)

.join

La fonction .join permet de fusionner des tableaux de natures différentes sur base d'une colonne clé spécifiée.

Syntaxe :

```
(DataSet("A"); DataSet ("B")).join("reference"; "inner")
```

→ Fusionnera les tableaux sur base de la colonne « référence » et utilisera pour ce faire la méthode « inner ».

Les méthodes possibles sont les suivantes :

inner : Reprend toutes les lignes du premier tableau qu'il est possible d'associer à une ligne du second. Ainsi, si une ligne du premier tableau n'a pas de complément dans le second, elle ne sera pas reprise dans le tableau final. Idem pour une ligne du second qui ne correspond à rien dans le premier tableau.

left : Toutes les lignes du premier tableau sont reprises, et on y attache les infos venant du second tableau qui s'y rattachent (si pas d'infos associées, les colonnes restent vides)

right : Toutes les lignes du second tableau sont reprises, et on y attache les infos venant du premier tableau qui s'y rattachent (si pas d'infos associées, les colonnes restent vides)

full : left ET right en même temps

Exemple explicatif :

Imaginons par exemple devoir faire une jonction entre les 2 tableaux suivants sur base de leur colonne commune « Référence » :

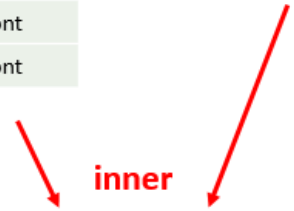
Ref	Société	Responsable
001	Factory 1	Dupont
002	Factory 2	Carter
003	Factory 3	
004	Factory 4	Carter
005	Factory 5	Dupont
006	Factory 6	Dupont

Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54

Dans le cas de la jonction « **inner** », on reprendra donc toutes les lignes qu'il est possible de compléter. La ligne 003 du premier tableau n'ayant pas de Responsable, elle ne sera pas reprise dans le résultat, tout comme le responsable Delieux, qui n'est associé à aucune société.

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

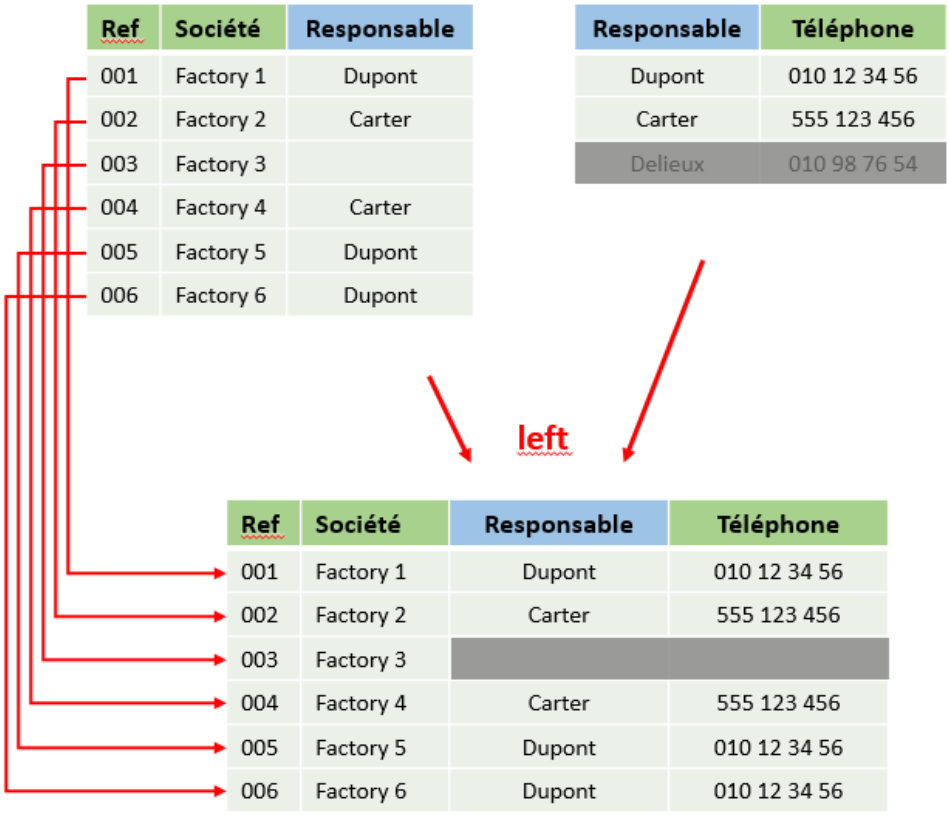
Responsable	Téléphone
Dupont	010 12 34 56
Carter	555 123 456
Delieux	010 98 76 54



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

La jonction via la méthode « **left** » prendra donc quant à elle la liste des lignes du premier tableau, et viendra y accoler les lignes de la seconde table si cela est possible, laissant vides celles pour lesquelles aucune jonction n'est possible.

La ligne 003 n'a pas de responsable associé, donc le reste de la ligne restera vide, et e responsable Delieux n'étant associé à aucune société, il n'est jamais récupéré et n'apparait donc pas dans le résultat final.



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

De la même manière, la jonction via la méthode « **right** » reprendra la liste des lignes du second tableau, et viendra y accoler les lignes de la première table si cela est possible, laissant vides celles pour lesquelles aucune jonction n'est possible.

On remarque que dans le cas où plusieurs jonctions sont possibles (comme pour le cas de Dupont), les lignes sont dupliquées pour créer toutes les associations possibles.

On part donc ici de droite, Dupont est associé à 3 sociétés, donc la ligne est démultipliée pour être associée à chaque société. Carter étant associé à deux sociétés, on retrouvera donc 2 lignes. Delieux n'étant associé à aucune société, on crée tout de même la ligne mais le début restera vide. Enfin, on remarque que la société 003 n'étant pas reliée à un responsable, elle n'aura jamais été appelée dans la jonction, et n'apparaitra donc pas du tout dans le résultat

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

Enfin, la méthode de jonction « **full** » prendra toutes les lignes de la méthode inner, puis y ajoutera les lignes supplémentaires du **left** puis celles du **right** pour n'abandonner aucune possibilité.

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

full

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

Attention : Les tableaux joints par la fonction .join devront tous posséder la colonne de référence (passée en argument de la fonction .join), par contre, ils ne peuvent avoir d'autres colonnes en commun. En effet, le système ne saurait que faire en cas de conflit de valeurs dans des colonnes différentes portant le même nom.

Remarque : il est possible de joindre plus de deux tableaux source en une fois avec la fonction .join. Il faut juste que toutes les jonctions se fassent sur la même colonne et avec la même méthode (left, inner…)

```
(DataSet("A"); DataSet ("B"); DataSet ("C"); DataSet ("D")).join("reference"; "inner")
```

→ Fusionnera tous les tableaux A, B, C et D sur base de la colonne « référence » et utilisera pour ce faire la méthode « inner » à chaque fois.

Dans le cas contraire, s'il faut effectuer des jonctions sur des colonnes différentes ou si les méthodes de jonction diffèrent, il faudra passer par des jonctions successives.

```
(
        (
                DataSet("A");
                DataSet ("B")
        ).join("reference"; "inner");
        DataSet ("C")
).join("consos"; "left")
```

Cette syntaxe effectuera une première jonction entre les DataSet A et B sur base de la colonne « reference » et avec la méthode « inner », puis elle joindra le résultat avec le DataSet C sur base de la colonne « consos » et avec la méthode « left ».

.joinmulti

Lors des nombreuses utilisations de la fonction « .join », nous nous sommes rendu compte qu'il fallait souvent faire des jonctions sur base non pas d'une seule colonne commune mais bien sur base d'une clé composée de la concaténation de deux colonnes.

Exemple : si l'on a deux tables de données avec des colonnes REFERENCE, DATE, VALEUR et que l'on veut effectuer une jonction de ces deux tables non seulement sur la date mais aussi sur la référence, on devrait effectuer une jonction sur la concaténation des colonnes REFERENCE et DATE. Avec l'utilisation du « .join » actuel, cela voudrait dire qu'il faudrait que chacune des tables existantes soit modifiée pour inclure une colonne « clé de jonction » qui concatènerait les colonnes REFERENCE et DATE. Dans le cas de tables provenant de DataSets, ne pouvant donc pas contenir de formules, on devra alors créer deux feuilles de calcul intermédiaires dont le seul but serait de créer ces deux colonnes clés.

Pour éviter cette création de tables intermédiaires peu utiles, la fonction « .joinmulti » a été mise en place, permettant d'effectuer directement une jonction sur une concaténation de colonnes.

Syntaxe :

```
(DataSet("A"); DataSet ("B")).joinmulti("method" ; "reference_1" ; "reference_2")
```

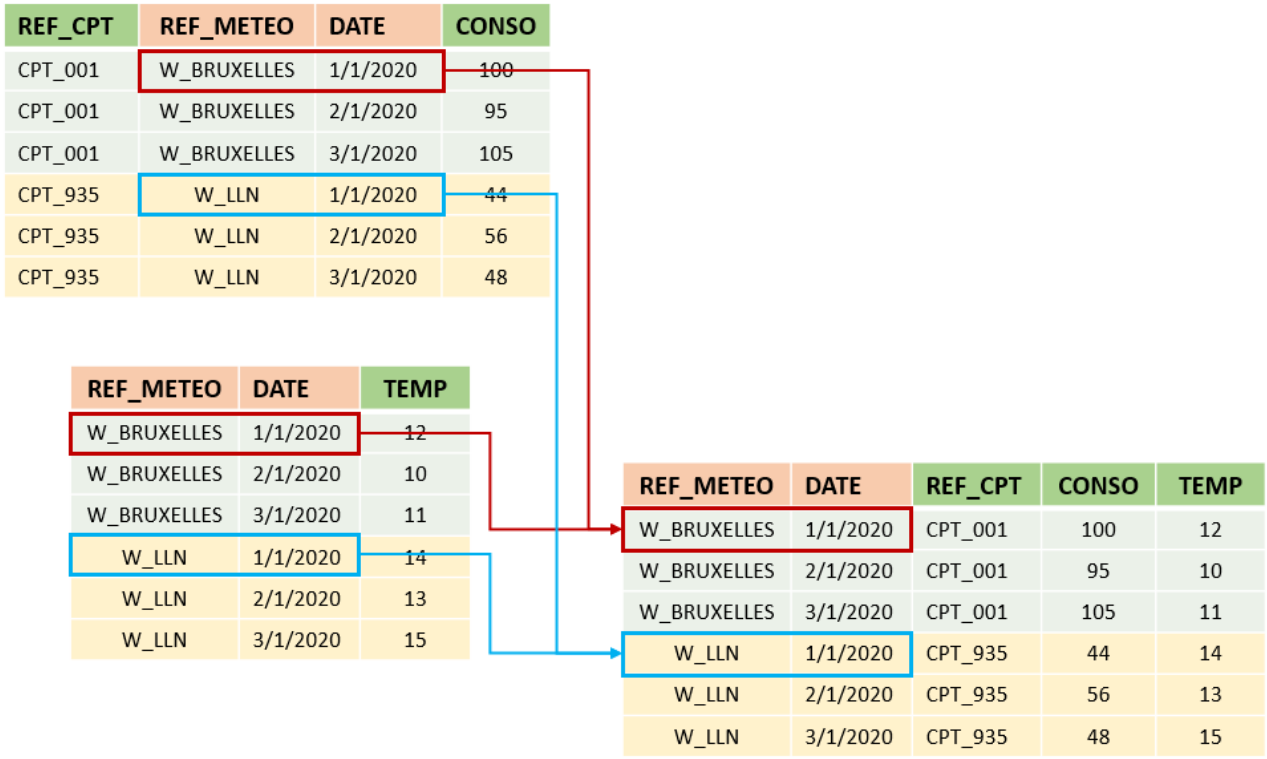
→ Fusionnera les tableaux sur base de la concaténation des colonnes "reference_1" et "reference_2" et utilisera pour ce faire la méthode "method".

Cette fonction recevra donc la méthode de jonction ("inner", "left", "right" ou "full"), suivie de la liste des références de colonnes à concaténer pour constituer la clé de jonction. Le nombre de référence de colonnes n'est pas limité à 2, et on pourrait donc concaténer 3, 4 ou 5 colonnes pour constituer la clé de jonction.

Exemple applicatif : imaginons que nous ayons les deux tables de gauche

La première est une table de consommation contenant plusieurs compteurs les uns en-dessous des autres, avec la référence du compteur, la date et la valeur de consommation, à laquelle on a ajouté une colonne « REF_METEO » indiquant la référence de la station météo la plus proche.

La seconde table est une table de données météo, récupérant les températures de plusieurs stations météo les unes en-dessous des autres.



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

Une simple jonction sur base de la colonne DATE poserait un problème. En effet, cette colonne peut contenir plusieurs fois chaque date puisque les consommations de plusieurs compteurs y sont reprises. On aurait alors des associations croisées entre les lignes d'un compteur et les données météo des stations ne lui correspondant pas.

Une jonction simple sur base de la référence de la station météo ne fonctionnera évidemment pas non plus, puisque l'on perd la notion de date.

La solution ici est donc bien de faire une jonction sur base de la concaténation des colonnes REF_METEO et DATE, afin que l'on fasse bien correspondre à chaque donnée d'un compteur, la température de la station météo associée, pour la date correspondante.

Attention : la fonction « .join » et la fonction « .joinmulti » reçoivent des arguments similaires (clé de jonction et méthode de jonction), mais **pas dans le même ordre**. En effet, le nombre de références de colonnes étant potentiellement variable pour le « .joinmulti », il a fallu réorganiser l'ordre des paramètres fournis à la fonction pour éviter des ambiguïtés informatiques… et il est dangereux de permuter les arguments de la fonction « .join » historique pour des questions de rétrocompatibilité avec le grand nombre de configurations déjà existantes. Attention donc à bien mettre les arguments dans le bon ordre pour chacune de ces fonctions.

.joinmulti dynamiques (colonnes de jonction avec références différentes)

Il peut parfois être compliqué d'avoir des tableaux de données ayant des noms de colonne communs pour faire les jonctions. En effet par exemple, un même DataSet pourrait être utilisé dans plusieurs situations différentes, demandant chacune d'avoir un nom spécifique pour faire des jonctions. Pour régler ce problème, il est possible de définir des correspondances entre colonnes de noms différentes dans l'appel de la fonction « .joinmulti »

Imaginons que l'on ait deux DataSets

- A, contenant (entre autres) la colonne « COL_1 » devant être utilisée pour faire une jonction.
- B, contenant (entre autres) la colonne « COL_2 » devant être utilisée pour cette même jonction.

S'il est impossible de renommer ces colonnes pour une raison extérieure, on peut malgré tout effectuer la jonction avec la syntaxe suivante :

```
(dataset("A") ; dataset("B")).joinmulti("LEFT" ; "A.COL_1 = B.COL_2" )
```

Cette syntaxe indique que, pour les besoins de cette jonction, la colonne COL_1 du tableau A doit être considérée comme correspondant à la colonne COL_2 du tableau B. C'est alors cette colonne « commune » qui sera utilisée comme colonne de jonction.

7. Factures

Dans EMM (JOOL), une facture est considérée comme un ensemble de blocs de propriétés, associé à un compteur, avec une date de début et une date de fin de validité.

En partant d'un compteur (auxquels généralement on attachera les factures), on peut alors utiliser la syntaxe suivante pour obtenir la liste de toutes les factures associées au compteur, pour le contexte actif.

```
selection.invoices
```

Cette syntaxe renvoie donc une liste d'objets de type facture. Une case sélection contenant cette syntaxe génèrera un worksheet avec une ligne par facture.

Une fois que l'objet actif est une facture (dans la ligne d'un tableau comme décrit ci-dessus par exemple), on pourra alors récupérer les propriétés stockées dans la facture avec la syntaxe habituelle.

```
item.properties("//INV_ADM_TECH_CONS_TOTAL")
```

Cette formule renverra donc pour chaque facture, la valeur de la propriété « INV_ADM_TECH_CON_TOTAL », en l'occurrence il s'agit de la consommation totale associée à la facture, dans l'unité de mesure de la ressource.

Comme toujours, il est possible de filtrer les factures selon leurs propriétés avec la fonction « .where », afin de n'en garde que certaines.

En plus des propriétés contenues dans les blocs de la facture, il est également possible de récupérer des informations basiques qui y sont associées :

- item.**from** → renverra la date de début de la facture
- item.**to** → renverra la date de fin de la facture
- item.**id** → renverra l'identifiant de la facture
- item.**creation**.user.mail → renverra l'adresse email du créateur de cette facture
- item.**lastupdate**.date → renverra la date de dernière mise à jour de cette facture

8. Utilisateurs (users)

Dans EMM (JOOL), un utilisateur est un item qui dispose de certaines caractéristiques de base ainsi que de blocs de propriétés personnalisables. Nous ne parlons pas ici des droits utilisateurs, qui sont configurables par un administrateur, mais qui ne sont pas utilisables dans la syntaxe EMM (JOOL) proprement dite.

La liste complète des utilisateurs d'une DB est accessible via le mot clé « **users** ». Ce mot clé renverra la liste des items de type « utilisateur » présents dans la base de données. A partir de cette liste, on pourra évidemment effectuer des filtres, des tris, et récupérer les informations que l'on jugera nécessaires.

Une autre fonction permet quant à elle de récupérer un utilisateur spécifique quand on connait son login.

user("abc") → renverra comme objet actif l'utilisateur qui a pour login « abc »

Enfin, le mot clé « **me** » renverra comme item actif l'utilisateur connecté.

Une fois que l'on a un ou plusieurs utilisateurs en sélection, on peut aller récupérer ses caractéristiques de base, comme son nom, prénom, mail…

me.**firstname** → renvoie le prénom de l'utilisateur actif

me.**lastname** → renvoie le nom de famille de l'utilisateur actif

me.**name** → renvoie le nom complet (prénom nom) de l'utilisateur actif

me.**mail** → renvoie l'adresse mail de l'utilisateur actif

user("abc").**mail** → renvoie l'adresse mail du user ayant pour login « abc »

me.**login** → renvoie le login de l'utilisateur actif

user("abc").**login** → renvoie le login du user ayant le login « abc ». Ici, renvoie « abc »

users.**lastname** → renvoie la liste des noms de famille de tous les utilisateurs

users.where(item.mail.endswith("@dapesco.com")).**name**

→ renvoie la liste des noms de tous les utilisateurs dont l'adresse mail se termine par « @dapesco.com ». Renvoie donc la liste des utilisateurs membres de Dapesco.

Outre les caractéristiques de base des utilisateurs, un administrateur peut également leur attribuer des blocs de propriétés personnalisés, qu'il est alors possible de récupérer dans la syntaxe via la fonction « .properties »

me.**properties**("//USR_MYSITE")

→ renvoie la propriété « USR_MYSITE » associée à l'utilisateur connecté.

```
users.where(item.properties("//USR_MYSITE")="BXL_0000").name
```

→ renvoie la liste des noms de tous les utilisateurs dont la propriété « USR_MYSITE » a pour valeur « BXL_0000 ». Cela peut être utilisé pour récupérer la liste des utilisateurs qui sont responsables du site de Bruxelles par exemple.

Remarque : Quand on est sur une propriété d'utilisateur, on peut utiliser la fonction « .parent » pour remonter à l'utilisateur concerné. La formule précédente peut alors être optimisée avec la syntaxe suivante :

```
users.properties("//USR_MYSITE").where(item.value="BXL_0000")).parent.name
```

Cette syntaxe a l'avantage d'être plus performante que la précédente puisqu'elle minimise le nombre d'appels à la base de données. Cette façon de faire est similaire à celle développée au point 4. Collection d'objets > Trier et filtrer une collection > Optimisation des filtres.

9. Gestion des événements (.events)

a) Généralités

Un événement, ou event, est un bloc de propriétés temporaire que l'on peut venir greffer à une entité/compteur et qui peut fournir des informations sur un événement en cours concernant cette entité/compteur.

Exemple : un event de type « TRAVAUX » peut indiquer à un gestionnaire qu'il y'a des travaux en cours sur un site, et que cela impacte plusieurs compteurs. L'event doit avoir une date de début et peut avoir une date de fin (certains events s'ouvrent sans savoir quand ils se clôtureront). Il peut en outre contenir un ou plusieurs blocs de propriétés détaillant les informations utiles de cet événement.

Enfin, un fil de discussion eut être associé à chaque event, permettant aux différents intervenants d'échanger des informations au besoin. Ainsi, un event de type TRAVAUX pourra contenir un fil de discussion où le conducteur des travaux pourra discuter avec le responsable énergie de la commune ainsi qu'avec l'administrateur EMM (JOOL).

Dans le parseur EMM (JOOL), il est possible de récupérer les informations des events ainsi que les fils de discussion associés via les fonctions suivantes.

b) Fonctions associées aux events

.events

En partant d'une entité/compteur, on peut récupérer les events liés via la fonction « .events »

`selection.events` → renvoie la liste des events des entités/compteurs en sélection

`selection.events("RefTypeEvent")`

→ renvoie la liste des events des entités/compteurs en sélection, mais uniquement ceux dont le type correspond à la référence de type d'event passée en argument.

Remarque : Les events récupérés via ces formules sont limités à ceux qui couvrent tout ou partie du contexte temporel actif. Il est possible de forcer un contexte à la fonction « .events » en lui passant des dates en paramètre.

`selection.events(from ; to)`

→ renvoie la liste des events des entités/compteurs en sélection, limités à ceux apparaissant dans la période allant de « from » à « to », ces mots-clés pouvant être évidemment remplacés par des formules renvoyant des dates.

`selection.events("RefTypeEvent" ; from ; to)`

→ idem que la formulation précédente, mais limité aux events du type dont la référence est passée en premier argument.

Les événements sont donc associés aux entités/compteurs, et quiconque a droit de voir l'entité/compteur aura également le droit de voir les events associés. Il est en plus possible de surcharger les utilisateurs pour

donner accès à un event à un user qui ne devrait pas pouvoir le voir (voir manuel d'utilisation de EMM (JOOL) pour une description de l'interface)

Les events sont donc également, d'une certaine manière, liés aux utilisateurs. Il est donc également possible de récupérer les events en partant d'un ou plusieurs utilisateurs.

user.**events** → renvoie la liste des events associés à tous les users

me.**events** → renvoie la liste des events associés à l'utilisateur connecté

.from / .to / .reference / .name / .type / .parent / .creation / .lastupdate

Un event a donc une date de début et potentiellement une date de fin. Celles-ci sont récupérables via les fonctions « .from » et « .to » appliquées à l'event en question.

selection.events.top(1).**from**

→ renverra la date de départ du premier event associé à la sélection active.

Dans le même ordre d'idée, on peut récupérer les informations principales de l'event via les fonctions suivantes :

.reference

→ renverra la référence (identifiant unique) de l'event. Généralement, cette référence sera générée automatiquement par EMM (JOOL) et ne sera que très rarement utilisée.

.type

→ renverra la référence du type d'event associé. C'est ce type qui servira à trier les events en fonction des types d'événements à suivre dans EMM (JOOL) (TRAVAUX, COUPURE…)

.name

→ renverra le nom du type d'event associé. Plus intelligible humainement, il peut être plus explicite que la référence de type.

.parent

→ renvoie la liste des entités/compteurs lié(s) à l'event. Attention, un event pouvant être lié à plusieurs entités/compteurs, le résultat de cette fonction renverra donc bien une liste d'entités/compteurs.

.creation

→ renvoie les informations de création de l'événement. Appliquer ensuite .date renverra la date de création, et appliquer .user renverra le user du créateur de l'event.

.lastupdate

→ renvoie les informations de dernière mise à jour de l'événement. Appliquer ensuite .date renverra la date de mise à jour, et appliquer .user renverra le dernier user à avoir mis l'event à jour.

.comments / .comments.text / .comments.date / .comments.user

A partir d'un event, on peut récupérer la liste des commentaires qui y ont été associés. Pour cela, en partant d'un event, on utilisera la fonction « .comments » qui renverra la liste de tous les commentaires associés.

```
selection.events.flop(1).comments
```

→ renvoie la liste des commentaires associé au dernier event associé à la sélection.

Une fois sur les commentaires, on peut alors utiliser les fonctions suivantes pour récupérer les informations de chaque commentaire :

.text → renvoie le texte du commentaire

.date → renvoie la date de génération de ce commentaire

.user

→ renvoie l'objet utilisateur qui a écrit le commentaire. On pourra alors enchainer avec un « .name » ou un « .mail » pour récupérer le nom ou l'email de cet utilisateur.

Exemples avancés :

```
selection.events("TRAVAUX").orderby(item.from).flop(1).comments.orderby(item.date).flop(1).user.mail
```

→ Sur base de la sélection active, on récupère tous les events de type « TRAVAUX » (référence de type d'event = « TRAVAUX »). On les ordonne alors selon leur date de début, et on ne garde que le dernier. Pour cet event, on récupère la liste des commentaires, que l'on ordonne par date, et on garde le dernier uniquement. Enfin, on trouve alors l'utilisateur qui a écrit ce commentaire, et on prend son email que l'on affiche enfin comme résultat final.

```
selection.events("TRAVAUX"). comments.orderby(item.date).flop(1).user.mail
```

→ Sur base de la sélection active, on récupère tous les events de type « TRAVAUX » (référence de type d'event = « TRAVAUX »). On récupère alors massivement la liste de tous les commentaires associés à tous ces events, que l'on ordonne par date, et on garde le dernier uniquement. Enfin, on trouve alors l'utilisateur qui a écrit ce commentaire, et on prend son email que l'on affiche enfin comme résultat final. Cela nous donne donc une manière de voir qui est la dernière personne à avoir commenté un event quelqu'il soit.

10. Alarmes

a) Généralités

Dans EMM (JOOL), les tâches d'alarmes se basent sur des propriétés d'alarmes, stockées sur les canaux, pour lever des alarmes en cas de dépassement de seuils. Une fois les alarmes levées et stockées dans un logbook d'alarmes, celles-ci sont récupérables via les DataSet (voir manuel EMM (JOOL) de l'administrateur), ou directement dans la syntaxe.

b) Accès aux propriétés d'alarmes

L'accès aux propriétés d'alarmes associées à un channel se font exactement comme pour un accès à une propriété, mais via la fonction « .alarm ».

```
selection.channels("MAIN").alarm("//CNL_ALA_MAX_ACTIVE")
```

Cette syntaxe ira récupérer la propriété d'alarme « CNL_ALA_MAX_ACTIVE » associée au canal « MAIN » du compteur présent en sélection.

La méthode de fonctionnement est en tout point similaire à celle de la fonction « .properties », tant au niveau de la gestion des valeurs simples que multiples ou historisées.

11. Exécuter du code (Execute)

La fonction EXECUTE permet d'interpréter un texte comme s'il s'agissait de code EMM (JOOL). On peut alors constituer dynamiquement une formule en accolant divers bouts de texte, et qui finalement sera une instruction que l'on fera exécuter par EMM (JOOL).

A. Convertir une référence en entité (syntaxe sous-optimale)

Bien que cela ne soit pas optimal informatiquement, on peut par exemple transformer une référence de canal (format texte) en objet de type Canal utilisable.

Exemple :

```
execute("@" + "BXL_0000")
```

→ renverra l'entité dont la référence est « BXL_0000 ». On pourra ensuite y appliquer toute formule applicable à une entité. Cette syntaxe aura exactement le même effet que « @BXL_0000 », ce qui peut paraitre peu intéressant tel quel. La syntaxe suivante est plus intéressante…

```
execute("@" + item.properties("//REF_SITE"))
```

→ Cette syntaxe est beaucoup plus utile, puisqu'elle part d'une propriété stockée en format texte dans l'item active, luis utilise ce texte pour récupérer l'entité dont le texte est la référence. On peut donc transformer une propriété texte en entité exploitable dans le parseur.

Comme signalé plus haut, cette syntaxe n'est pas optimale parce qu'elle fait appel à une fonction lourde (execute) pour effectuer une tâche plutôt simple. La syntaxe suivante a été créée pour optimiser ce genre de chose : @(item.properties("//REF_SITE"))

Cette syntaxe, plus courte, a surtout été optimisée pour effectuer la transformation d'une référence en entité, alors que la fonction execute, bien plus générale n'est pas du tout optimisée pour ce cas.

B. Sélection et contexte temporaires

En plus de cette application, la fonction « execute » peut également prendre en argument une sélection et une période de temps qui serviront de contexte temporaire à l'exécution du texte d'instruction.

Syntaxe :

```
execute("TEXTE A EVALUER"; selection; from; to )
```

→ La fonction « execute » va exécuter le texte à évaluer mais dans le cadre de la sélection et du contexte temporaires passés en argument.

Cela peut être extrêmement pratique quand on doit par exemple aller récupérer une donnée en provenance d'un autre contexte que le contexte temporel actif, comme par exemple récupérer une consommation sur une période de référence (hors contexte actuel).

Exemple :

```
execute(""dataset(""DSET_A_CALCULER"")""; @EntityRef; dateeval(2014); dateeval(2014; 2) )
```

→ La fonction « execute » va exécuter le tableau « DSET_A_CALCULER » mais en lui passant comme sélection l'entité « EntityRef », et dans le contexte allant du 1° Janvier au 1° Février 2014, et ce, quels que soient la sélection ou le contexte actuels.

Remarque : attention à bien doubler les « " » à l'intérieur du texte à exécuter sans quoi EMM (JOOL) considèrera que l'on clôture le texte.

Remarque : Lors de l'appel à un DataSet (ou à une feuille de calcul d'ailleurs…), il est également possible de lui passer directement les paramètres de sélection et de contexte, sans forcément passer par un execute.

La syntaxe suivante renverra donc exactement la même chose que la syntaxe reprise ci-dessus :

```
dataset("DSET_A_CALCULER"; @EntityRef; dateeval(2014); dateeval(2014; 2) )
```

Cela peut améliorer la vitesse de calcul, et on la préfèrera à l'utilisation d'un execute plus gourmand en ressources. L'utilisation de la fonction execute sera donc réservée aux cas particuliers, comme par exemple le cas développé dans le paragraphe suivant.

C. Gestion des boucles

Cette fonction « execute » permet également de récupérer des arguments hors de boucles de code, par exemple dans le cas de l'utilisation d'un « foreach ».

Explication :

Imaginons que l'on ait comme sélection une liste de compteurs, et que l'on veuille calculer dans la case d'un tableau la valeur de conso de ses descendants multipliée par un coefficient en provenance du compteur actif de la ligne.

On pourrait essayer comme ceci :

```
item.descendants.foreach(item.data.sum * item.properties("//COEFF"))
```

Le problème est qu'une fois dans la boucle du « foreach », le mot clé « item » a une nouvelle signification. Le « item » du début et celui dans la boucle ne représentent pas la même chose. Une fois dans le « foreach » donc, on ne peut plus récupérer simplement le compteur de départ (hors boucle)

Une solution est donc d'utiliser un « execute » pour construire la formule avant de l'exécuter.

```
execute("item.descendants.foreach(item.data.sum * " + item.properties("//COEFF") + ")" )
```

Cette façon de faire permet de constituer la formule en insérant la valeur calculée hors boucle, puis d'exécuter la formule qui deviendra donc

```
item.descendants.foreach(item.data.sum * 35)
```

…puisque la valeur du coefficient sera calculée avant son insertion dans la boucle et son exécution.

12. Création de rapport HTML

La manière la plus simple de créer des rapports dans EMM (JOOL) est de passer par la création de tableaux de bord (voir manuel « Utiliser EMM (JOOL) » pour une explication complète sur la création de tableaux de bord). Cependant, il peut arriver que l'on ait besoin de plus de flexibilité dans la génération du rapport, comme par exemple si l'on veut intégrer ou escamoter dynamiquement des blocs du rapport, ou si l'on veut boucler sur un nombre d'objets (inconnu au préalable) pour créer plusieurs blocs en nombre variable dans le rapport.

Dans ces conditions, il convient d'utiliser l'éditeur de rapport HTML de EMM (JOOL), qui permet de rédiger une page HTML construite dynamiquement sur base d'une sélection et d'un contexte EMM (JOOL).

Dans l'éditeur, la base est donc le HTML, dans lequel on peut injecter les balises traditionnelles de style. A cela, on rajoute une balise spéciale dédiée à injecter du code EMM (JOOL), les doubles accolades {{ … }}.

Entre ces doubles accolades, on peut injecter du code EMM (JOOL) pour récupérer des valeurs dynamiques sur base de la sélection et du contexte passés au rapport. Ces balises peuvent contenir toute la syntaxe EMM (JOOL) connue, et l'emploi du code EMM (JOOL) permettra entre autres de créer des conditions ou des boucles dans un langage HTML parfois trop rigide.

A. Récupération de valeurs dynamiques

Il est possible de se baser simplement sur la sélection active lors de la génération du rapport. On utilisera alors le mot clé selection et on lui appliquera les transformations désirées.

Exemple : {{ selection.name }} → renverra le nom de la sélection active (si il y'a plusieurs entités dans la sélection, ce code renverra la liste des noms des entités à la suite)

De la même manière, il est possible d'utiliser les mots-clés from et to pour récupérer les dates de début et de fin du contexte de génération du rapport.

Exemple : {{ from.format("d/MM/yyyy") }} → renverra la date de début de contexte, dans le format demandé.

Outre ces valeurs plutôt génériques, il est également possible de se baser sur la case « sélection » du générateur de rapport pour récupérer des informations item par item. Pour cela, on devra utiliser la balise @@itemtemplate@@. Cette balise, utilisée par paire dans un rapport HTML, indiquera à EMM (JOOL) qu'il doit recopier le code présent entre ces balises autant de fois qu'il n'y a d'objet présent dans la case sélection du rapport. A chaque itération du code, le mot clé « item » pourra être utilisé pour référer à l'objet actif.

Exemple :

Si la sélection contient 3 sites, on pourrait avoir le code suivant :

```
Rapport<br>
Période du <b>{{ from.format("d/MM/yyyy") }}</b> au <b>{{ to.format("d/MM/yyyy") }}</b><br>
```

```
<table>
	<tr><td>Site</td><td>Référence</td><td>Nombre de compteurs</td></tr>
	@@itemtemplate@@
	<tr>
		<td>{{ item.name }}</td>
		<td>{{ item.reference }}</td>
		<td>{{ item.link("ENTITY_METER").count }}</td>
	</tr>
	@@itemtemplate@@
</table>
```

Ce code renverrait alors un résultat ressemblant à ceci :

Rapport

Période du **1/01/2018** au **1/01/2019**

Site	Référence	Nombre de compteurs
Site n°1	SITE_001	4
Site n°2	SITE_002	2
Site n°3	SITE_003	5

Dans le titre, on récupère les dates de début et de fin de contexte, puis une table est construite, avec une ligne de titre (hors boucle), puis @@itemtempalte@@ démarre une boucle qui contient une ligne de tableau affichant nom, référence et nombre de compteurs de chaque item de la sélection. Une fois tous les items de la sélection traités, on ressort alors de la boucle et on clôture la table proprement.

Bien entendu, il est également possible de fournir une sélection au rapport qui sera sous forme de tableau de valeurs (résultat de génération d'un dataset, ou d'une feuille de calcul, ou encore d'une jonction entre plusieurs tableaux. Dans ce cas, le tableau passé en sélection sera considéré comme une collection de lignes, et on pourra utiliser « item.column("…") » pour récupérer les valeurs du tableau source.

B. Inclusion de rapport dans un autre rapport

La génération de rapport HTML peut se faire de façon modulaire. On peut par exemple créer un sous-rapport HTML qui analyse la consommation d'électricité, un autre analysant spécifiquement la consommation de gaz (les analyses affichées peuvent être différentes), et un méta-rapport qui ira inclure ces deux sous-rapports sans avoir besoin de retaper tout le code des sous-rapports dans le rapport principal.

Pour cela, on utilisera la fonction htmlreport qui ira récupérer le rapport qu'on lui passera en argument et qui l'inclura dans le rapport actuel.

Syntaxe :

htmlreport("RefRapportHTML"; selection; from; to; "LANG" ; selection_substitution)

- "RefRapportHTML" sera la référence de l'objet « Rapport HTML » que l'on veut générer.
- selection (optionnel) est la sélection que l'on veut utiliser pour la génération du rapport HTML
- from; to (optionnels) représentent le contexte temporel pour lequel générer le rapport HTML
- "LANG" (optionnel) est le code de la langue dans laquelle générer le rapport (par exemple : "FR")
- selection_substitution (optionnel) est une autre manière de passer la sélection au rapport

Si l'on ne fournit ni sélection ni contexte lors de l'appel de cette fonction (ces trois paramètres sont optionnels), la sélection et le contexte de génération seront les sélection et contexte actifs.

Si l'on ne fournit pas de code de langage, c'est la langue de l'utilisateur connecté qui sera utilisée pour la génération du rapport.

En ce qui concerne les sélections

Le deuxième argument selection est une sélection au même sens que pour les appels de worksheets. Cela signifie que la sélection passée en deuxième argument sera transmise au sous-rapport et le mot-clé « selection » y sera remplacé par les entités/compteurs contenus dans cet argument. Cet argument peut évidemment avoir la forme d'une formule qui renverrait un ou plusieurs entités/compteurs.

Malheureusement, il arrive des situations où un rapport HTML se base sur un tableau de données et qu'il soit intéressant de passer à un de ses sous-rapports une partie de ce tableau comme sélection, plutôt que de lui passer la liste des entités et que le sous-rapport soit forcé de recalculer le tableau en question. La situation est similaire avec des listes d'items groupées par exemple… Or, la syntaxe du deuxième argument impose que les items passés en sélection soient des entités/compteurs.

Pour permettre de passer des sous-ensembles de données (tables, sous-tables, ensembles résultant d'un .groupby..), le 6° argument optionnel a été rajouté. Si l'on décide de passer une sélection via ce 6° argument, on peut lui fournir une collection d'à peu près n'importe quel type d'objets, et cette collection d'objets passée en argument ira remplacer complètement le résultat de la case « selection » du sous-rapport appelé. Cela signifie que cette case « selection » ne sera pas utilisée, et que si elle contenait une formule, elle serait écrasée par la sélection de substitution et n'aurait aucun impact au moment de l'appel.

Exemples de syntaxe

```
htmlreport("RefRapportHTML")
```

→ Génère le rapport demandé, en lui passant la sélection et le contextes actifs (par défaut), dans la langue de l'utilisateur connecté.

```
htmlreport("RefRapportHTML"; selection; from; to)
```

→ Génère le rapport demandé, en lui passant la sélection et le contexte définis dans l'appel, dans la langue de l'utilisateur connecté. Selection, from et to peuvent être remplacés par des formules renvoyant respectivement une collection d'entités/compteurs, et des dates.

```
htmlreport("RefRapportHTML"; selection; from; to)
```

→ Génère le rapport demandé, en lui passant la sélection et le contexte définis dans l'appel, dans la langue de l'utilisateur connecté. Selection, from et to peuvent être remplacés par des formules renvoyant respectivement une collection d'entités/compteurs, et des dates.

```
htmlreport("RefRapportHTML"; null; from; to)
```

→ Dans le cas où 'on veut fournir un argument mais que les arguments précédents ne sont pas nécessaires, on peut alors les remplacer par « **null** ». Ainsi, dans cet exemple, les arguments 3 et 4 sont bien les from et to que l'on veut fournir, sans pour autant avoir à fournir une selection.

```
htmlreport("RefRapportHTML"; null; from; to; null ; selection_substitution)
```

→ Dans ce cas, on veut passer une sélection via le mode de substitution en argument à la fonction hmlreport. Pour cela, la sélection « classique » sera remplacée par « **null** », et si l'on n'a pas besoin d'imposer la langue de génération, le code de langage sera également remplacé par « **null** » pour garder le bon compte dans les positions des arguments.

Le résultat de cette syntaxe sera un objet de type texte, rédigé en langage HTML, utilisable comme n'importe quel autre texte de EMM (JOOL). Injecter ce morceau de syntaxe (entre doubles accolades donc) dans un autre rapport HTML génèrera le sous-rapport, et injectera le résultat dans le rapport principal, constituant donc un rapport final pouvant être composé de plusieurs sous-rapports.

C. Inclusion conditionnelle

L'utilisation de code EMM (JOOL) dans les rapports HTML permet d'afficher ou d'escamoter des blocs de rapport en fonction de certaines propriétés. On pourra par exemple faire en sorte qu'un rapport contienne ou pas un bloc d'analyse de la consommation en gaz selon qu'il y'ait ou non du gaz sur le site.

Exemple :

```
{{
        if(selection.children.channels.properties("//CNL_DAC_RESOURCE")
        .where(item.key="NAT_GAS").parent.count <> 0;
        htmlreport("HTML_SOUS_RAPPORT_GAZ"); "")
}}
```

Avec cette syntaxe, on part des enfants de la sélection. On descend alors sur leurs channels, et on récupère la propriété indiquant quel type de ressource ils mesurent. On filtre pour ne garder que les canaux mesurant du gaz naturel, puis on remonte aux parents, et on effectue un comptage.

Si l'on a un compte différent de zéro, cela signifie que l'on a au moins un compteur de gaz naturel sur le site. Dans ce cas, on effectue l'injection du sous-rapport « HTML_SOUS_RAPPORT_GAZ », et dans le cas contraire, on n'injecte rien. Cela est cohérent, si l'on a des compteurs de gaz, on affiche leur analyse, dans le cas contraire, on escamote l'analyse pour ne pas afficher un bloc vide.

D. Intégrer un widget dans un rapport HTML

Dans de nombreux cas, il est utile de pouvoir intégrer un widget dans un HTML. On pourra alors par exemple intégrer un graphique ou une jauge dans le corps d'un rapport que l'on pourra ensuite envoyer par mail, converti en PDF.

Syntaxe :

{{ **widget**("Ref_du_Widget"; largeur; hauteur; selection; from; to) }}

- " Ref_du_Widget " est donc bien entendu la référence du widget que l'on veut intégrer.
- largeur; hauteur sont la largeur et la hauteur en pixel, en pourcentage ou en mode « automatique » attribuées au widget dans le rapport HTML
- selection est la sélection (liste d'entités/compteurs) à utiliser pour la génération du widget
- from; to sont les bornes du contexte d'exécution du widget

Tous ces arguments peuvent évidemment être calculés dynamiquement par des formules plutôt que passés en dur dans le code.

Exemples :

{{ **widget**("GRAPHE_DE_CONSO"; 200; 150; selection; from; to) }}

→ Renvoie un widget de 200x150 pixels, calculés sur la sélection et le contexte actifs.

{{ **widget**("GRAPHE_DE_CONSO"; "75%"; "50%"; selection; from; to) }}

→ Renvoie un widget de 75% de son contenant HTML de large et de 50% de la hauteur de ce même contenant. Remarque : le pourcentage est noté en texte et on doit indiquer l'unité utilisée. Cette unité peut aussi être « px » et dans ce cas, on n'a aucune différence entre le fait de fournir des nombres ou un texte en pixels.

{{ **widget**("GRAPHE_DE_CONSO"; "75%"; "auto"; selection; from; to) }}

→ Renvoie un widget de 75% de son contenant HTML de large et en hauteur, il prendra toute la place libre de son contenant HTML. Si l'argument n'est pas fourni à l'appel, sa valeur par défaut sera sur « auto ». Il est à noter qu'il est plus pratique d'utiliser ce mode « auto » que de lui imposer 100%. En effet, « 100% » implique la totalité de la taille du contenant, en ce compris les marges éventuelles (padding). On risque donc de déborder du contenant si l'on impose 100% conjointement avec la présence de marges.

On peut également combiner des dimensions en pixels et d'autres en pourcentage, bien que cela soit parfois plus compliqué à gérer dans une mise en page…

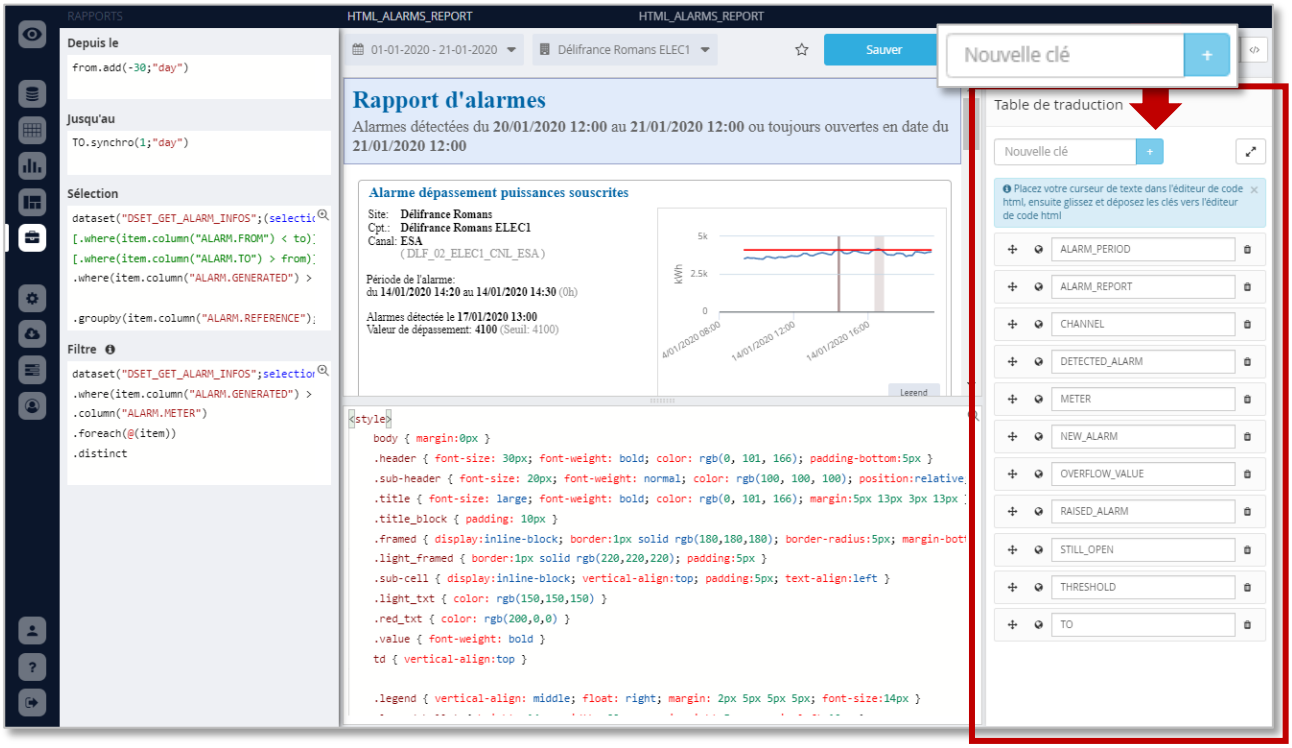
E. Module de traduction

Dans le cas d'une base de données fréquentée par des utilisateurs multilingues, de nombreuses valeurs s'adapteront automatiquement en fonction de la langue configurée pour l'utilisateur actif, que ce soit à

l'affichage ou lors de l'envoi de rapports par mail (chaque destinataire est censé recevoir le rapport dans sa langue) ; les titres de widgets (encodés dans les deux langues lors de la définition des widgets), les propriétés en texte multilingue (choisies dans des menus déroulants dont les valeurs sont issues d'un Xtab)… mais le texte « en dur » dans le rapport HTML ne va pas se traduire tout seul spontanément.

Une première solution est de créer un clone du rapport par langue dans la base de données, mais cela peut vite devenir un cauchemar au niveau de la maintenance, surtout si le nombre de rapport augmente un peu.

C'est là que le module de traduction peut être utile. Dans l'éditeur HTML, le traducteur est accessible en cliquant sur la petite planète tout en haut à droite de la page. Cela ouvre un volet sur la droite de l'écran permettant de configurer des clés de traductions, utilisables par la suite dans le corps du rapport.



En commençant une table de traduction, on doit commencer par ajouter les clés que l'on veut utiliser. Pour cela, on peut noter la clé désirée dans le petit champ tout en haut à gauche du volet, puis cliquer sur le « + » juste à côté. Cela rajoutera la nouvelle clé dans la liste en-dessous, dans le volet de traduction. Une clé doit être vue comme une référence, c'est-à-dire qu'elle doit être unique (à l'intérieur du rapport), et qu'elle ne doit pas contenir de caractères spéciaux sous peine d'interférer avec le code HTML du rapport.

A côté de chaque clé dans la liste du traducteur, on a une petite croix qui servira à attraper la clé pour la faire glisser (drag&drop) sur le rapport, ainsi qu'une petite planète qui permet d'ouvrir un pop-up où l'on peut donner les traductions de la clé dans les différentes langues de la base de données.

Le petit bouton avec les 2 flèches en haut à droite du volet de traduction permettent de

déployer toutes les clés en une fois, directement dans le volet de traduction.

Une fois les clés et les traductions prêtes, on peut les faire glisser directement dans le code HTML du rapport (le code associé viendra s'insérer tout en haut de la page), ou on peut directement taper le code d'accès à cette clé dans le rapport, là où l'on veut que le texte traduit soit visible dans le rapport.

Syntaxe :

[[translate.CLE_DE_TRADUCTION]]

Le texte « CLE_DE_TRADUCTION » est donc bien l'une des clés créées dans le volet de traduction, et à l'interprétation du rapport HTML, ce code sera remplacé par la traduction définie dans le volet pour la langue de l'utilisateur connecté (lors d'une visualisation dans EMM (JOOL)), ou du destinataire (dans le cas d'un envoi de rapport automatique).

13. Générer des fichiers PDF ou CSV

Que cela soit pour l'attacher à un email (voir chapitre sur les envois de mails) ou pour le stocker directement dans les documents d'une entité/compteur, il est souvent utile de créer des fichiers PDF à partir de rapports HTML, ou des fichiers CSV à partir de tables de données de EMM (JOOL).

A. Récupération d'un rapport HTML

Les objets de type « Rapport HTML » peuvent être créés dans l'éditeur de rapports. La fonction htmlreport (détaillée dans un chapitre précédent) sert à récupérer le résultat de la génération d'un objet de type rapport HTML par EMM (JOOL).

htmlreport("RefRapportHTML"; selection; from; to) renverra donc un objet en format texte, utilisable soit directement dans le corps de mail, soit en attachement après conversion en PDF.

B. Conversion d'un rapport HTML en PDF

La fonction « .topdf » s'applique à un rapport HTML (la plupart du temps généré sur le moment via la fonction « htmlreport ») et le convertit en objet de type PDF. Ce fichier PDF pourra alors être utilisé comme attachement pour un email ou simplement stocké dans les documents d'une ou plusieurs entités/compteurs.

Syntaxe : htmlreport("RefRapportHTML").**topdf**("nom.pdf")

Cette formule aura pour résultat un objet PDF appelé « nom.pdf », basé sur le rapport HTML « RefRapportHTML », et que l'on pourra alors par exemple attacher à un email en partance.

La fonction « .topdf » peut en outre recevoir un paramètre optionnel supplémentaire permettant de sauver le PDF généré dans l'onglet « Documents » d'une ou plusieurs entités/compteurs.

htmlreport("RefRapportHTML").topdf("nom.pdf" ; **@entité**)

→ Le paramètre optionnel doit être une entité (indiquée en dur via un « @REFERENCE » ou résultant d'une formule comme « item », ou « item.parent » par exemple…) et le PDF généré sera stocké dans l'onglet « Documents » de cette entité/compteur.

htmlreport("RefRapportHTML").topdf("nom.pdf" ; **(@entité_1 ; @entité_2)**)

→ le paramètre optionnel peut également être une liste d'entités/compteurs. Dans ce cas, la liste doit être entre parenthèses et les composants de la liste séparés par des « ; ».

Les parenthèses sont obligatoires pour que EMM (JOOL) comprenne la liste comme un seul paramètre fourni à la fonction « .topdf » et non comme une multitude de paramètres supplémentaires.

Il est également possible de stocker automatiquement le fichier généré, non seulement dans la racine de l'onglet « Documents », mais aussi dans un répertoire se trouvant dans cet onglet. Pour cela, on devra indiquer le chemin d'accès dans le nom du PDF. Dans le cas où le répertoire donné n'existe pas encore, il sera généré à la volée pour y stocker le PDF.

htmlreport("RefRapportHTML").topdf("**REPERTOIRE/**nom.pdf" ; @entité)

→ Le document PDF sera donc stocké dans l'onglet « Documents » de l'entité choisie, mais à l'intérieur d'un répertoire, ici nommé « REPERTOIRE »

C. Conversion d'un tableau de données en CSV

Parfois, il est plus pertinent d'envoyer directement un tableau de données brut en CSV qu'un rapport en PDF. Pour cela, la fonction « .tocsv » permet de convertir une table de données en un fichier CSV, qu'il s'agisse d'un dataset, d'une feuille de calcul, du résultat d'un join ou d'un groupby… ou toute table présente dans EMM (JOOL).

Syntaxe : dataset("RefDataSet").**tocsv**("nom.csv")

Ce code renverra un objet de type CSV, que l'on peut alors attacher à un email pour envoi en pièce jointe par exemple.

Comme la fonction « .topdf », cette fonction « .tocsv » peut aussi recevoir un argument optionnel supplémentaire si l'on veut stocker le fichier CSV ainsi généré dans l'onglet « Documents » d'une ou plusieurs entités/compteurs. Le paramètre devra un fois de plus être une entité/compteur (et pas seulement sa référence) et peut donc être indiqué en dur via un « @REFERENCE » ou résulter d'une formule comme « item », ou « item.parent » par exemple.

dataset("RefDataSet").tocsv("nom.csv" ; **@entité**)

→ Le paramètre optionnel doit être une entité (indiquée en dur via un « @REFERENCE » ou résultant d'une formule comme « item », ou « item.parent » par exemple…) et le PDF généré sera stocké dans l'onglet « Documents » de cette entité/compteur.

dataset("RefDataSet").tocsv("nom.csv" ; **(@entité_1 ; @entité_2)**)

→ le paramètre optionnel peut également être une liste d'entités/compteurs. Dans ce cas, la liste doit être entre parenthèses et les composants de la liste séparés par des « ; ».

Les parenthèses sont obligatoires pour que EMM (JOOL) comprenne la liste comme un seul paramètre fourni à la fonction « .topdf » et non comme une multitude de paramètres supplémentaires.

dataset ("RefDataSet "). tocsv ("**REPERTOIRE/**nom. csv " ; **@entité**)

→ Le document CSV sera stocké dans l'onglet « Documents » de l'entité choisie, mais à l'intérieur d'un répertoire, ici nommé « REPERTOIRE »

- Plus de paramètres optionnels pour la fonction .tocsv

Outre le nom obligatoire et la ou les entités facultatives où stocker le fichier csv, la fonction ".tocsv" accepte d'autres paramètres facultatifs pour configurer précisément le format du fichier CSV produit.

Syntaxe complète : dataset("RefDataSet"). **tocsv**("nom. csv"; @ENTITY; ";" ; false ; "jj/MM/aaaa" ; ",")

- ";" : [optionnel] indique le séparateur de colonnes à utiliser dans le fichier CSV

Valeurs autorisées				
","	";"	" "	"\t"	"|"

- true / false : [optionnel] indique si l'on garde (true) ou retire (false) les noms de colonnes dans le CSV.
- "dd/MM/yyyy" : [optionnel] précise le format de date à utiliser dans le fichier CSV.
- "," : [optionnel] définit le séparateur decimal à utiliser dans le fichier CSV ("." ou ",")

Si les paramètres optionnels ne sont pas indiqués, le système utilisera la configuration de l'utilisateur connecté pour générer le fichier CSV (comportement habituel)

14. Envois manuels mails via le parseur

Dans EMM (JOOL), la gestion des envois de tableaux de bord et de rapports HTML par mails automatiques est gérée par le gestionnaire de Rapports. L'utilisation des fonctions reprises dans ce chapitre sera donc principalement limitée aux cas de re-génération de rapports à la demande, ou de débugging lors de leurs créations.

Il est aussi possible que l'on veuille simplement envoyer rapidement un mail simple à un ou plusieurs destinataires, avec ou sans attachements. Les fonctions qui suivent peuvent nous y aider.

La fonction sendmail sert à envoyer un email. Elle reçoit une série de paramètres définissant tous les aspects du mail à envoyer.

Syntaxe :

sendmail("abc@dapesco.com"; "def@mail.com"; "sujet du mail"; "contenu du mail"; attachement)

- Le premier paramètre est le destinataire du mail. Le paramètre doit être en format texte, mais il peut être une collection de textes.
  Par exemple,

"abc@dapesco.com" enverra un mail à cette adresse

("abc@dapesco.com"; "def@mail.com") enverra un mail avec deux destinataires

- Le deuxième paramètre est l'adresse du (ou des) destinataires en copie carbone. La même syntaxe que ci-dessus peut être utilisée pour avoir plusieurs destinataires.

- Le troisième paramètre est le sujet du mail. Toujours en format texte également, il peut évidemment être construit via une formule plus complexe au besoin, mais la formule doit alors produire un texte.

- Le quatrième paramètre est le contenu du corps de mail. Il peut être un simple texte également, éventuellement constitué à partir de formules EMM (JOOL), mais il peut également être le résultat de la génération d'un rapport HTML. Le rapport HTML sera dans ce cas injecté directement dans le corps de mail.

- Enfin, le cinquième paramètre (optionnel) est l'attachement que l'on veut donner au mail envoyé. Cet attachement sera la plupart du temps un rapport HTML converti en PDF, ou un fichier CSV permettant l'export d'une table de données.

Exemples :

```
sendmail("abc@dapesco.com"; "def@mail.com"; "test"; "ceci est un test")
```

→ Enverra un mail à « abc@dapesco.com », avec copie carbone à « def@mail.com », dont le titre sera « test » et qui contiendra simplement le texte « ceci est un test »

```
sendmail(("abc@dapesco.com"; "def@mail.com"); "archive@mail.com"; "test"; "ceci est un test")
```

→ Enverra le même mail que précédemment à « abc@dapesco.com » et à « def@mail.com », avec copie carbone à « archive.mail.com ».

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "ceci est un test";
    htmlreport("RefRapportHTML").topdf("nom.pdf"))
```

→ Enverra le même mail que précédemment à « abc@dapesco.com », avec copie carbone à « archive.mail.com », et avec un attachement nommé « nom.pdf » qui sera un PDF généré sur base du rapport HTML « RefRapportHTML ».

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "ceci est un test";
    dataset("RefDataSet").tocsv("nom.csv"))
```

→ Enverra le même mail que précédemment à « abc@dapesco.com », avec copie carbone à « archive.mail.com », et avec un attachement nommé « nom.csv » qui sera généré à partir du DataSet « RefDataSet ».

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; htmlreport("RefRapportHTML"))
```

→ Enverra le même mail que précédemment à ceci près que le rapport ne sera plus un PDF attaché au mail, mais bien un HTML inséré directement dans le corps du mail (le paramètre de contenu est directement le rapport HTML)

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; htmlreport("RefRapportHTML");
    htmlreport("RefRapportHTML").topdf("nom.pdf"))
```

→ Enverra le même mail que précédemment avec le rapport dans le corps du mail en format HTML, **et** le même rapport converti en PDF, attaché au mail.

```
sendmail("abc@dapesco.com"; "archive@mail.com"; "test"; "ceci est un test";
    htmlreport("RefRapportHTML").topdf("nom.pdf"; @SITE_1))
```

→ Enverra le même mail que précédemment à « abc@dapesco.com », avec copie carbone à « archive.mail.com », et avec un attachement nommé « nom.pdf » qui sera un PDF généré sur base du rapport HTML « RefRapportHTML ». Par ailleurs, dans la foulée, le PDF ainsi généré sera enregistré et stocké dans l'onglet « Documents » de l'entité « SITE_1 ».

15. Mises à jour via le parseur

Il peut être utile de mettre diverses choses à jour de façon automatique dans la base de données EMM (JOOL). On pourra par exemple mettre à jour un Xtab de benchmarking tous les mois, ou mettre à jour une propriété ou une facture en fonction d'un calcul régulier. Ces mises à jour pourront être déclenchées manuellement via l'exécution d'une feuille de calcul, ou automatisées via le gestionnaire de tâches.

A. updatextab – Mise à jour d'un Xtab pré-existant

Pour modifier les valeurs présentes dans un XTab, comme par exemple, pour un XTab contenant des valeurs à changer tous les ans, via une tâche qui updaterait ses valeurs, il existe la fonction « updatextab », que l'on pourra donc utiliser dans une feuille de calcul ou dans le gestionnaire de tâches automatiques.

Cette fonction reçoit un seul paramètre qui doit être un tableau de donnée (dataset, worksheet…), contenant les valeurs à ajouter/updater dans le XTab.

La première colonne du tableau source en question contiendra la référence du XTab à mettre à jour. (Cela permettra de faire, dans une même action, la mise à jour de plusieurs XTab différents, puisque chaque ligne mettra à jour le XTab dont la référence est stockée en première colonne)

Les autres colonnes du tableau source doivent avoir les mêmes références que les colonnes du XTab à modifier. (C'est sur base des références de colonnes que la fonction update fera la mise à jour)

La première colonne du XTab servira de colonne clé pour identifier laquelle de ses lignes doit être modifiée. Si l'on essaie d'importer une ligne avec une valeur de clé donnée et que cette clé existe déjà dans le XTab, la ligne correspondante sera mise à jour. Si en revanche la clé en question n'existe pas, la fonction créera une nouvelle ligne pour cette nouvelle valeur de clé.

Exemple :

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"))
```

→ Dans ce cas, la fonction va chercher la feuille de calcul « WSHT_PREPA_XTAB_UPDATE » et regarde pour chaque ligne le XTab à modifier. Dans cet exemple, ce sera toujours le même: XTAB_TO_UPDATE.

→ Pour la première ligne de la feuille de calcul préparatoire, la fonction va voir qu'il existe déjà la clé 001 dans le XTab et va donc modifier ses valeurs

→ Pour la 2° ligne de la feuille de calcul, la clé n'est pas déjà présente dans le XTab, donc la fonction va la créer et lui attribuer les valeurs données.

XTAB_TO_UPDATE (après modifications)

Key	Year	Value
001	2015	12
002	2015	24
003	2015	24

On remarquera que la ligne 002 du XTab n'ayant pas été indiquée dans de la feuille de calcul de mise à jour, ne sera absolument pas affectée par la fonction.

Rappel : Les références de colonnes de la feuille de calcul et de l'XTab doivent être les mêmes

B. updatextab – Création d'Xtab à la volée

Si la feuille de calcul préparatoire contient, dans l'une de ses lignes, une référence de Xtab inexistante, EMM (JOOL) ne pourra rien mettre à jour pour cette ligne. En effet, le Xtab n'existant pas, il n'y a rien à updater.

Dans ce cas, il est possible au besoin de demander à EMM (JOOL) de créer à la volée un nouvel Xtab pour y injecter la ligne en question. Pour cela, il faudra fournir à la fonction « updatextab », un paramètre supplémentaire (et optionnel) qui sera la référence d'un Xtab existant, à utiliser comme modèle pour la création du nouveau Xtab.

Syntaxe :

```
updatextab(datasheet("WSHT_PREPA_XTAB_UPDATE"); "XTAB_TEMPLATE")
```

- Le premier paramètre est la feuille de calcul préparatoire, contenant donc les lignes de données à mettre à jour.
- Le deuxième paramètre, optionnel est donc la référence d'un autre Xtab existant, à utiliser comme modèle pour une éventuelle création de Xtab à la volée.

  Dans le cas où la feuille de calcul préparatoire contiendrait en première colonne une référence de Xtab inexistant, EMM (JOOL) irait alors chercher le Xtab modèle (2° paramètre), clonerait sa structure (colonnes) et lui attribuerait la référence inexistante. Il irait ensuite injecter la ligne de données de la feuille préparatoire, comme si le Xtab avait toujours existé.

C. updateinvoice – mise à jour de propriétés de facturation

La syntaxe de EMM (JOOL) permet de mettre à jour des propriétés de factures via le parseur. Cela peut être très utile dans le cas où l'on veut créer des factures virtuelles ou de répartition. Dans ce cas, une tâche

automatique pourrait être utilisée qui irait calculer les factures virtuelles tous les mois, et entrer les résultats du calcul dans de véritables factures stockées sur le compteur concerné.

Syntaxe :

```
updateinvoice(
    METER;
    "InvoiceID";
    "Ref_Type_de_Facture";
    "/FULL_PATH/TO_THE_PROPERTY";
    Property_Value;
    Facture_FROM;
    Facture_TO
)
```

- « METER » est le compteur sur lequel la facture doit se trouver. On doit ici avoir l'objet compteur, autrement dit, pas simplement sa référence. (exemples : selection, item, @(REFERENCE), @(item.column("REFERENCE")) …)

- « InvoiceID » est l'identifiant de la facture à mettre à jour. Cet id est en format texte, et pourra être un texte en dur ou le résultat d'une formule renvoyant un texte.

- « Ref_Type_de_Facture » est la référence du type de la facture concernée. De nouveau, il s'agit d'un texte, pouvant être donné en dur ou comme le résultat d'une formule.

- « /FULL_PATH/TO_THE_PROPERTY » indique le chemin d'accès complet à la propriété. On doit donc avoir la référence du bloc de propriétés de facture dans lequel se trouve la propriété, puis la référence de la propriété à mettre à jour. Une fois de plus, ce paramètre doit être en format texte.

- « Property_Value » donne la valeur de la propriété à mettre à jour. Ce paramètre pourra être de n'importe quel type tant qu'elle correspond au type défini pour la propriété correspondante.

- « Facture_FROM » et « Facture_TO » sont les dates de début et de fin de la facture à mettre à jour. Format date donc.

Attention : Tous les paramètres sont obligatoires pour la fonction.
Attention : Si la facture donnée (meter, id, type, from, to) n'existe pas encore, la fonction « updateinvoice » la créera à la volée et y insèrera la valeur qu'il voulait mettre à jour.

Exemple :

```
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")
)
```

→ Cette fonction va trouver la facture « VIRTUAL_ELEC_01 », de type « VIRTUAL », commençant au début du mois dernier et finissant au début de ce mois-ci, et qui est attachée au compteur « LLN_001_ELEC », et y mettre à jour la propriété « INV_EB_ADM_EAN » (dans le bloc « INV_EB ») et lui donner la valeur « 55xxx ».
En pratique ce genre de situation est rare, puisque tout est passé en dur dans la fonction. La plupart du temps, on aura plutôt des feuilles de calcul préparant les valeurs à utiliser pour la mise à jour, puis quelques colonnes effectuant les mises à jour, une propriété à la fois.

Exemple :

```
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")

)
```

→ Dans le cas d'un worksheet préparatoire qui calcule ou prépare toutes les informations nécessaires à la création d'une facture, on pourrait avoir ce type de syntaxe. Les informations utiles proviennent toutes d'autre colonnes de la feuille de calcul, et les types correspondent à ce qui est attendu.

D. updateproperty – mise à jour de propriétés

Dans certaines circonstances, il peut être utile de pouvoir mettre à jours les propriétés d'une entité, d'un compteur, ou même d'un canal de façon dynamique, via le parseur. Pour cela, la fonction « updateproperty » peut être utilisée.

Syntaxe :

```
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 » est l'entité, le compteur ou le canal sur lequel on veut mettre la propriété à jour. Il peut s'agir d'une liste d'entités si l'on veut en mettre plusieurs à jour en même temps (avec la même valeur sur la même propriété du coup)

- « /FULL_PATH/TO_THE_PROPERTY » indique le chemin d'accès complet à la propriété en format texte. On doit donc avoir la référence du bloc de propriétés dans lequel se trouve la propriété, puis la référence de la propriété à mettre à jour.

- « Prop_value » donne la valeur de la propriété à mettre à jour. Ce paramètre pourra être de n'importe quel type tant qu'elle correspond au type défini pour la propriété correspondante.

- Paramètres optionnels, « P_from » et « P_to » donnent les dates de début et fin de validité de la propriété dans le cas d'une propriété dans un bloc historisé.

Attention : En cas de mise à jour d'une propriété qui serait historisée, la nouvelle valeur viendra s'insérer avec ses dates de validité exactement de la même manière que lors d'un import massif. Les valeurs actives pendant cette période seront amputées de tout ou partie de leur période de validité pour laisser place à la nouvelle valeur et sa période de validité.

Attention : Si l'on importe une propriété historisée sans lui donner de paramètres de date, elle sera considérée comme valable depuis toujours et pour toujours et elle viendra écraser toutes les valeurs actuelles de la propriété au cours de l'histoire pour la remplacer par la seule nouvelle occurrence importée.

Attention : En cas de mise à jour d'une propriété qui serait multiple, une nouvelle instance de la propriété sera créée avec la nouvelle valeur importée.

E. updatealarm – mise à jour de propriétés d'alarme

De la même manière que la fonction « updateproperty », la fonction « updatealarm » permet de mettre à jour les propriétés des blocs d'alarmes des canaux existants.

Cette fonctionnalité peut être très utile dans le cas de canaux avec des calibrations automatiques, et des affinements successifs automatiques des seuils d'alarmes.

La syntaxe est exactement la même, à ceci près que l'on utilise la fonction « updatealarm », et que les items sur lesquelles on peut mettre à jour les propriétés sont uniquement des canaux (seuls types d'objets sur lesquels on peut définir des alarmes.

Syntaxe :

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 » est le ou les **canaux** sur lesquels on veut mettre la propriété à jour. De nouveau, si l'on a plusieurs canaux, la propriété mise à jour sera la même pour tous, avec la même valeur.

- « /FULL_PATH/TO_THE_PROPERTY » indique le chemin d'accès complet à la propriété d'alarme en format texte. On doit donc avoir la référence du bloc de propriétés dans lequel se trouve la propriété, puis la référence de la propriété à mettre à jour.

- « Prop_value » donne la valeur de la propriété à mettre à jour. Ce paramètre pourra être de n'importe quel type tant qu'elle correspond au type défini pour la propriété correspondante.

- Paramètres optionnels, « P_from » et « P_to » donnent les dates de début et fin de validité de la propriété dans le cas d'une propriété dans un bloc historisé.

Attention : En cas de mise à jour d'une propriété qui serait historisée, la nouvelle valeur viendra s'insérer avec ses dates de validité exactement de la même manière que lors d'un import massif. Les valeurs actives pendant cette période seront amputées de tout ou partie de leur période de validité pour laisser place à la nouvelle valeur et sa période de validité.

Attention : Si l'on importe une propriété historisée sans lui donner de paramètres de date, elle sera considérée comme valable depuis toujours et pour toujours et elle viendra écraser toutes les valeurs actuelles de la propriété au cours de l'histoire pour la remplacer par la seule nouvelle occurrence importée.

Attention : En cas de mise à jour d'une propriété qui serait multiple, une nouvelle instance de la propriété sera créée avec la nouvelle valeur importée.

16. Correction de données - Application de l'AVAL via le parseur

Le processus « l'AVAL » pour « Acquisition Validation » est un système de correction automatique des données entrant dans EMM (JOOL) qui peut être activé ou pas, canal par canal.

Normalement, l'AVAL est exécuté à l'insertion des données sur un canal (s'il est configuré pour le canal concerné). Cependant, il peut arriver que l'on veuille forcer un passage de la correction de l'AVAL sur un canal déjà existant, sans devoir artificiellement réimporter les données pour déclencher la procédure de l'AVAL.

La fonction « processaval » va forcer ce passage de l'AVAL sur le canal passé en argument, pour la période de temps précisée.

Syntaxe : **processaval**(Channel ; from ; to)

Exemple:

```
processaval(item.channels("MAIN");from;to)
```

→ Si l'item active est un compteur (exemple : une sélection contenant une liste de compteurs), le canal « MAIN » de ce compteur sera analysé par l'AVAL et les données entre le « from » et le « to » seront corrigées.

Dans le cas d'une feuille de calcul avec un compteur par ligne, les canaux « MAIN » de chacun de ces compteurs seront corrigés par l'AVAL pour la période donnée.

Evidemment, les canaux comme les dates de début et de fin peuvent être données en dur, ou être le résultat de formules calculées soit à la volée soit dans une autre colonne de la feuille de calcul actuelle.