

## Le hash expliqué à ma fille

Ma fille vient me trouver pour m'expliquer qu'elle se met au hash. Elle a lu je ne sais où que c'était excellent, « surtout pour les joints ». Je n'avais que deux bémols avant d'acquiescer : d'abord qu'il ne faut pas en abuser, et qu'ensuite on ne dit pas joints mais jointures, en bon français.

C'est donc à une nouvelle manière de rédiger des jointures via l'étape DATA que le hash (également appelé *hash object* ou table de hachage) nous donne accès.

### La vie du hash & la programmation objet

Pour utiliser les tables de hachage dans l'étape DATA, on dispose d'une syntaxe relevant de la programmation objet. C'est-à-dire qu'une table de hachage est manipulée de la manière suivante :

1. on crée, en fonction de nos besoins spécifiques, une « version » (une instance) d'un objet générique appelé hash ; on baptise cette instance d'un nouveau de notre choix en suivant les mêmes normes que pour nommer une variable ;
2. on agit sur les caractéristiques de cette instance avec des fonctions appelées méthodes. La syntaxe pour appliquer une méthode à une instance est `nomInstance.nomMéthode(paramètresDeLaMéthode)` ;
3. si nécessaire, on détruit explicitement l'instance quand elle n'est plus utile. Toutes les instances sont automatiquement supprimées à la fin de l'étape DATA dans laquelle le hash a été créé.

Il découle du point 3 qu'on ne peut pas utiliser la même instance dans plusieurs étapes DATA et même qu'elle n'est pas sauvegardée dans la table au même titre qu'une variable. L'objet hash est temporaire, et ne permet que de manipuler de l'information le temps d'une étape DATA.

Pour le point 1, une seule instanciation suffit pour toute l'étape DATA. On inclura donc l'instruction correspondante (`DECLARE`) dans une condition `IF _N_=1` ou à l'extérieur d'une boucle entourant un `SET`.

```
/* syntaxe n°1 */
DATA ... ;
SET ... ;
IF _N_=1 THEN DO ;
  DECLARE HASH h ( ) ; /* le hash s'appelle "h" */
  ...
END ;
... /* opérations sur chaque observation lue */
RUN ;
```

```
/* syntaxe n°2 */
DATA ... ;
DECLARE HASH h ( ) ;
...
DO UNTIL(fin) ;
  SET ... END=fin ;
  ... /* opérations sur chaque observation lue */
END ;
RUN ;
```

## Que contient un hash ?

Une table de hachage contient en mémoire vive l'équivalent d'une table SAS, organisé en deux parties : les informations servant de clés d'accès (*key*) et les informations à restituer sans compréhension (*data*). Les clés d'accès servent à organiser une éventuelle jointure : il faut une correspondance avec une variable de même nom et de même type, lue dans la table SAS du SET. La logique de programmation est alors très proche du double SET avec l'option KEY – mais nul besoin d'index dans ce cas, la table de hachage jouant le même rôle d'accès direct à l'observation idoine.

On définit les deux groupes d'informations stockées avec 2 méthodes : DEFINEKEY et DEFINEDATA. On marque la fin des définitions avec la méthode DEFINEDONE sans arguments.

```
h.DEFINEKEY("variable1") ;
h.DEFINEKEY("variable2") ; /* clé = variable1 + variable2 */
h.DEFINEDATA("variable3") ;
h.DEFINEDATA("variable4") ;
h.DEFINEDONE() ;
```

On alimente la table de hachage soit en une seule étape, dès l'instruction DECLARE, soit au fil des observations à stocker avec la méthode ADD.

Dans l'instruction DECLARE, on alimente au moins un attribut DATASET avec le nom d'une table SAS. Ce dernier est une chaîne de caractères et peut donc être renvoyé par une fonction si nécessaire. Un second attribut facultatif permet d'indiquer le régime à appliquer aux répétitions sur les clés d'accès : erreur en cas de doublon (DUPLICATE:"E") ou remplacement de la valeur en doublon (DUPLICATE:"R"). On évitera d'utiliser des clés d'accès ayant des répétitions dont on souhaite garder la trace : l'utilisation de la table de hachage s'en trouve considérablement compliquée. Une clé unique ou dont on accepte qu'elle devienne unique par écrasement des doublons est préférable.

```
DECLARE HASH h (DATASET:"tableSAS"
                <, DUPLICATE:"R" >) ;
```

La méthode ADD s'applique, elle, sans aucun argument.

```
h.ADD() ;
```

## Le hash pour faire des jointures

On peut utiliser en premier lieu une table de hachage pour réaliser la jointure d'une petite table (qui sera stockée dans le hash) et une grosse table qu'on n'aura pas besoin de trier. Il est possible de récupérer les valeurs des variables stockées dans la table de hachage, mais pour un premier exemple, nous allons utiliser uniquement la clé d'accès.

Une table (SIN) contient une liste de sinistres survenus sur des contrats d'assurance automobile avec le numéro du contrat (variable CONTRAT). Un même contrat peut avoir plusieurs sinistres.

La table exhaustive des contrats (AUTO4R) est de grande taille, et on ne souhaite pas la trier ni l'indexer pour réaliser une jointure avec une instruction MERGE. On la parcourt dans son ordre actuel avec SET, en cherchant à chaque contrat s'il a son pendant dans la table de

hachage : s'il y a correspondance, la méthode `FIND` renvoie 0. Nous ne souhaitons conserver que les contrats sans sinistre, et nous écrivons donc...

```
DATA work.sans_sin ;
  IF _N_=1 THEN DO ;
    DECLARE HASH sin (DATASET:"livre.sin", DUPLICATE:"R") ;
    sin.DEFINEKEY("contrat") ;
    sin.DEFINEDONE() ;
  END ;
  SET livre.auto4r ;
  IF sin.FIND() NE 0 THEN OUTPUT ;
RUN ;
```

Pour récupérer les valeurs stockées dans la partie « data » de la table de hachage, la syntaxe se complique légèrement. En effet, si on se contente de trouver la correspondance entre une ligne de la table de hachage et l'observation en cours dans l'étape `DATA`, on n'écrit que ce qui est lu dans la table du `SET`. Toute autre variable sortirait de nulle part. On doit donc déclarer et initialiser à manquant (la routine `CALL MISSING` permet de le faire pour plusieurs variables, séparées par des virgules, quel que soit leur type) toutes les variables qui seront rapatriées de la table de hachage en cas de correspondance.

```
DATA work.dernier_sin ;
  ATTRIB dtSurSin
    LABEL = "Date dernier sinistre"
    LENGTH = 4
    FORMAT = DDMMYY10.
  ;
  IF _N_=1 THEN DO ;
    DECLARE HASH sin (DATASET:"livre.sin", DUPLICATE:"R") ;
    sin.DEFINEKEY("contrat") ;
    sin.DEFINEDATA("dtSurSin") ;
    sin.DEFINEDONE() ;
    CALL MISSING(dtSurSin) ;
  END ;
  SET livre.auto4r ;
  IF sin.FIND() = 0 THEN OUTPUT ;
RUN ;
```

Ici, avec les mêmes tables que précédemment, on récupère la date du dernier sinistre survenu en plus des caractéristiques de tous les contrats sinistrés.

Si on souhaite conserver tous les contrats, sinistrés ou non, avec en sus leur date de dernier sinistre, on doit éliminer l'instruction `OUTPUT` conditionnelle. Il faudra cependant conserver la méthode `FIND` qui assure la connexion entre l'observation lue par `SET` et la table de hachage.

```
DATA work.dernier_sin (DROP=resultat) ;
  ATTRIB dtSurSin
    LABEL = "Date dernier sinistre"
    LENGTH = 4
    FORMAT = DDMMYY10.
  ;
  IF _N_=1 THEN DO ;
    DECLARE HASH sin (DATASET:"livre.sin", DUPLICATE:"R") ;
    sin.DEFINEKEY("contrat") ;
```

```
sin.DEFINEDATA("dtSurSin") ;
sin.DEFINEDONE() ;
CALL MISSING(dtSurSin) ;
END ;
SET livre.auto4r ;
resultat = sin.FIND() ;
RUN ;
```

La variable numérique RESULTAT créée ici ne sert qu'un but : établir la connexion entre les données et la table de hachage. Elle peut donc être éliminée de la table finale.

## Le hash pour éclater une table SAS

Une facilité offerte par la table de hachage est de pouvoir en déverser le contenu dans une table SAS dont le nom est déterminé de manière dynamique. Avec une étape DATA classique, on doit énumérer dans l'instruction DATA le nom de toutes les tables qui seront créées (on peut utiliser pour cela le macro-langage si nécessaire). Avec la table de hachage, on utilisera la méthode OUTPUT qui a un attribut DATASET dynamique : on peut donc créer automatiquement autant de tables qu'une variable compte de valeurs, sans aucun élément de macro-langage.

Pour cela, on remplit la table de hachage avec toutes les observations partageant la même valeur (un tri préalable est indispensable pour s'assurer de blocs de valeurs délimités par des pseudo-variables FIRST et LAST). A la fin de chaque bloc, on déverse la table de hachage dans une table dont le nom est construit à partir de la valeur du bloc en cours.

```
PROC SORT DATA=sashelp.class OUT=work.class ;
  BY sex name ;
RUN ;
DATA _NULL_ ;
  DECLARE HASH h ( ) ;
  h.DEFINEKEY("i") ;
  h.DEFINEDATA("name", "age", "weight", "height") ;
  h.DEFINEDONE() ;
  DO i=1 BY 1 UNTIL(LAST.sex) ;
    SET work.class ;
    BY sex ;
    h.ADD() ;
  END ;
  h.OUTPUT(DATASET:CATS("WORK.",sex)) ;
RUN ;
```

On crée ici deux tables, F et M, contenant respectivement les observations de sexe féminin puis masculin. La boucle (d'aspect assez inhabituel) DO i=1 BY 1 UNTIL(LAST.sex) ; assure qu'on reste dans la boucle tant qu'on n'a pas atteint la fin d'un bloc. Au sein de chaque bloc, une variable i (compteur de boucle) assure le rôle de clé d'accès dont les valeurs sont uniques (le BY 1 assure que i est incrémenté à chaque passage dans la boucle, c'est-à-dire à chaque observation lue).