Chapitre 5

Mise à jour des hiérarchies de dimension

5.1 Introduction

Dans les bases de données, l'évolution de schéma peut être vue comme la mise en œuvre d'opérations élémentaires d'ajout, de suppression et de modification. Par exemple, dans le modèle objet, ces opérations sont appliquées soit au niveau des classes, soit au niveau des propriétés des classes. Dans le modèle relationnel, ces opérations sont appliquées au niveau des tables ou des attributs. Ainsi les concepts subissant les opérations ont tous le même rôle. Or ce n'est pas le cas dans la modélisation des entrepôts de données.

Si les travaux de recherche dans le domaine des entrepôts de données s'inspirent des travaux réalisés dans celui des bases de données, il n'en demeure pas moins que les entrepôts de données ont des spécificités qu'il faut prendre en compte. C'est le cas de l'évolution de schéma, comme le soulignent les différents travaux de recherche relatifs à cette problématique (chapitre 3). La sémantique portée par le schéma de l'entrepôt induit des considérations différentes sur ces concepts et sur leur évolution a fortiori. De ce fait, lorsque l'on parle d'évolution de schéma dans un entrepôt de données, on ne peut pas se baser sur les distinctions adoptées dans les bases de données. Il nous faut distinguer la mise à jour des tables de faits, de celle des tables de dimension, ou encore de celle des hiérarchies de dimension.

Une hiérarchie étant composée d'un ensemble de niveaux de granularité, la mise à jour concerne donc directement les niveaux de granularité. Ainsi, dans ce chapitre, nous proposons différents algorithmes pour réaliser la mise à jour des hiérarchies. Nous proposons un algorithme de création et de suppression de niveaux de granu-

larité. Nous proposons également un algorithme qui permet de gérer la propagation des mises à jour dans la hiérarchie. En effet, lorsqu'un niveau est créé ou supprimé en milieu de hiérarchie, il est nécessaire d'effectuer une propagation qui consiste à définir les liens d'agrégation requis.

Ensuite, nous proposons un modèle d'exécution qui gère les modules nécessaires à la réalisation des mises à jour des hiérarchies de dimension. Il permet donc de gérer les modules d'acquisition, d'intégration et d'évolution. Ce modèle d'exécution s'inscrit dans une approche relationnelle, il s'agit d'exploiter des structures relationnelles dans l'exécution des différents processus. Il comprend un ensemble d'algorithmes qui se basent sur le formalisme que nous avons proposé dans [FBB07c] et qui permet de représenter les différents concepts manipulés dans la mise en œuvre de notre approche.

Ce chapitre est organisé comme suit. Tout d'abord, la section 5.2 est consacrée à la présentation du processus de mises à jour de hiérarchies de dimension et de leur propagation dans le schéma de l'entrepôt. Puis, dans la section 5.3, nous déployons notre approche dans un contexte relationnel. La section 5.4 est consacrée à une discussion. Enfin, nous concluons dans la section 5.5.

5.2 Création et suppression de niveaux de hiérarchie

5.2.1 Création d'un nouveau niveau de hiérarchie

La création d'un niveau de granularité dans une hiérarchie consiste à créer ce niveau et le lien qu'il a avec le niveau inférieur sur lequel il est basé selon les règles d'agrégation exprimées. Cette création doit bien évidemment satisfaire les contraintes présentées dans le modèle *R-DW*, en l'occurrence : les contraintes liées aux concepts de partition d'une part et de bijection d'autre part (section 4.4).

Un niveau peut être créé au milieu d'une hiérarchie ou à la fin d'une hiérarchie. Lorsque nous parlons de création en fin de hiérarchie, cela correspond en fait soit à la création au-dessus du dernier niveau d'une hiérarchie existante, soit à la création au-dessus de la dimension elle-même, définissant ainsi une nouvelle hiérarchie.

Pour créer un niveau au milieu d'une hiérarchie, il faut que cela ait un sens d'agréger les données du niveau que l'on crée vers un niveau supérieur déjà existant, autrement dit les données sont agrégeables sémantiquement du niveau créé vers le niveau existant supérieur.

Pour illustrer cette création, nous prenons le cas de la hiérarchie de la dimension `AGENCY` issue de l'exemple LCL. Considérons le schéma de dimension de la

figure 5.1(a). Une unité commerciale correspond à un regroupement d'agences, selon leur localisation géographique. Il est alors possible de créer, entre les niveaux `AGENCY` et `COMMERCIAL UNIT`, un niveau qui correspondrait à un regroupement d'agences par quartier (`AGENCIES GROUP`), où une unité commerciale correspondrait donc à un regroupement de ces groupes d'agences, comme c'est le cas dans la figure 5.1(b). Dans ce cas, il faut définir le lien d'agrégation entre `AGENCIES GROUP` et `COMMERCIAL UNIT`.

En revanche, si l'on crée un nouveau niveau qui permet d'agréger les données des agences selon leur taille (petite, moyenne et grande), ce niveau `AGENCY SIZE` sera créé au-dessus de la dimension `AGENCY`, définissant ici une nouvelle hiérarchie pour cette dimension comme c'est illustré dans la figure 5.1(c).

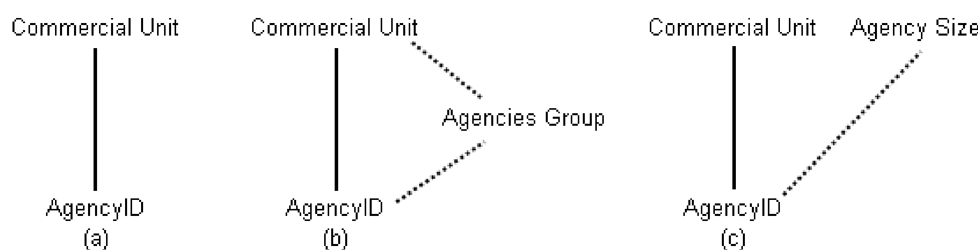

FIG. 5.1 – Schémas de la dimension `AGENCY` pour illustrer la création de niveaux

5.2.2 Suppression d'un niveau de hiérarchie existant

La suppression d'un niveau de granularité implique la suppression du niveau lui-même et du lien qu'il a avec le niveau au-dessus duquel il est construit. Cette suppression peut être réalisée sur un niveau placé au milieu ou à la fin d'une hiérarchie.

Par exemple, considérons le schéma de dimension de la figure 5.2(a). Une unité commerciale (`COMMERCIAL UNIT`) correspond à un regroupement d'agences par quartier (`AGENCIES GROUP`). Il est possible de supprimer le niveau `AGENCIES GROUP`. Dans ce cas, cette suppression implique de redéfinir le lien d'agrégation entre `AGENCY` et `COMMERCIAL UNIT` comme le montre la figure 5.2(b). Il est également possible de supprimer le niveau `COMMERCIAL UNIT`, auquel cas il suffit de supprimer le niveau et le lien qui le relie avec le niveau `AGENCIES GROUP` tel que le montre la figure 5.2(c).

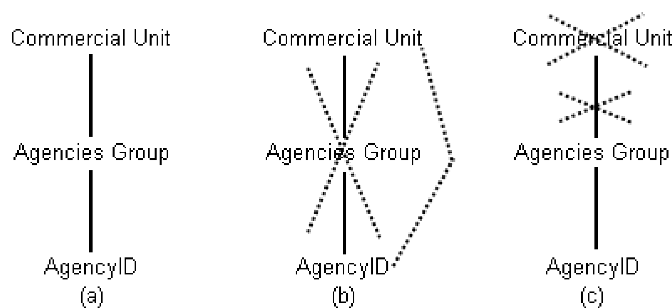

FIG. 5.2 – Schémas de la dimension AGENCY pour illustrer la suppression de niveaux

Ainsi, si un niveau est supprimé et que ce niveau est situé entre deux autres niveaux, il faut assurer le lien entre les deux niveaux en question. Ceci est réalisé lors de la propagation des mises à jour.

5.2.3 Propagation des mises à jour

Lorsqu'un niveau est supprimé ou créé et que ce niveau ne se situe pas en fin de hiérarchie, il est nécessaire de propager cette mise à jour dans la hiérarchie. Cette propagation consiste à assurer les liens requis entre les niveaux concernés de la hiérarchie mise à jour.

Dans le cas de la création d'un niveau de granularité au milieu d'une hiérarchie, la propagation consiste à définir le lien d'agrégation entre le niveau créé et le niveau supérieur. Par exemple, considérons le schéma de dimension de la figure 5.1(a). Lors de la création du niveau AGENCIES GROUP entre AGENCY et COMMERCIAL UNIT, la propagation consiste à définir le lien entre AGENCIES GROUP et COMMERCIAL UNIT tel que c'est illustré dans la figure 5.1(b).

Dans le cas de la suppression d'un niveau de granularité au milieu d'une hiérarchie, la propagation consiste à définir le lien d'agrégation entre le niveau inférieur et le niveau supérieur du niveau supprimé. Par exemple, considérons le schéma de dimension de la figure 5.2(a). Lors de la suppression du niveau AGENCIES GROUP, la propagation consiste à définir le lien entre AGENCY et COMMERCIAL UNIT tel que c'est illustré dans la figure 5.2(b).

Cette propagation, qui correspond à la définition de nouveaux liens entre niveaux, est réalisée de façon automatique, en déterminant ce lien à partir des liens existants. La détermination du lien se base sur les données. Ainsi, une fois que la structure prend en compte le lien, ce dernier est établi au niveau des instances.

Dans le cas de la suppression, le lien entre le niveau inférieur et le niveau supérieur du niveau à supprimer est établi avant que la suppression n'ait lieu. Dans le cas de la création, le lien entre le niveau créé et le niveau supérieur est déterminé après la création du nouveau niveau en se basant sur le lien existant entre le niveau inférieur et le niveau supérieur du niveau créé.

5.2.4 Algorithmes

Afin de clarifier la terminologie utilisée pour les niveaux de granularité dans les algorithmes, nous avons représenté les termes employés dans la figure 5.3. Le niveau courant correspond à celui qui subit la mise à jour, i.e. celui qui est créé ou supprimé.

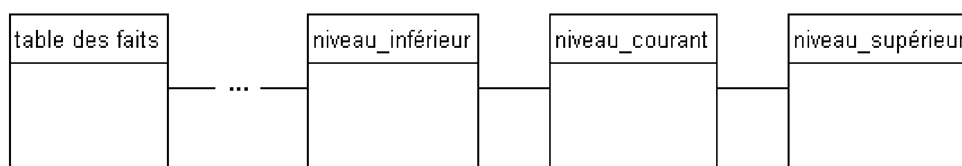

FIG. 5.3 – Terminologie des niveaux de granularité

L'algorithme 1 montre les étapes de suppression d'un niveau. Il s'agit d'abord de supprimer le lien entre le niveau inférieur et le niveau supprimé (ligne 1), avant de pouvoir supprimer le niveau lui-même (ligne 2). Ensuite, l'appel à la propagation est lancé si le niveau supprimé est placé au milieu de la hiérarchie (lignes 3 à 5).

Algorithme 1 Suppression d'un niveau de hiérarchie

Entrée: niveau_courant le niveau qui doit être supprimé, place_fin un booléen (=vrai si niveau_courant est en fin de hiérarchie)

1: Suppression dans niveau_inférieur de *id* (l'identifiant de niveau_courant dans niveau_inférieur)
2: Suppression de niveau_courant
3: **si** place_fin=faux **alors**
4: Propagation des mises à jour dans la hiérarchie (niveau_courant,suppression)
5: **fin si**

L'algorithme 2 montre les étapes de création d'un niveau. Il s'agit de créer la structure du nouveau niveau et d'y insérer les données (lignes 1 et 2). Puis le lien entre le niveau inférieur et le niveau créé doit être établi, au niveau de la structure (ligne 3), puis des données (lignes 4 à 6). Ensuite, l'appel à la propagation est lancé si le niveau n'est pas créé en fin de hiérarchie (lignes 7 à 9).

Algorithme 2 Création d'un niveau de hiérarchie

Entrée: place_fin un booléen (=vrai si niveau_courant est en fin de hiérarchie)

1: Création de la structure de niveau_courant : identifiant *id* et autres descripteurs
2: Insertion des données dans niveau_courant
3: Mise à jour de la structure de niveau_inférieur en ajoutant *id*
4: **pour tout** instance de niveau_inférieur **faire**
5: Mise à jour de la valeur de *id* dans niveau_inférieur
6: **fin pour**
7: **si** place_fin=faux **alors**
8: Propagation des mises à jour dans la hiérarchie (niveau_courant,création)
9: **fin si**

L'algorithme 3 permet de réaliser la propagation, que ce soit lors d'une création ou d'une suppression de niveau de granularité. Lors d'une création, il s'agit d'établir le lien entre le niveau créé et le niveau supérieur du point de vue de la structure d'une part (lignes 2 et 3), du point de vue des données d'autre part (lignes 4 à 7). Quand il s'agit d'une suppression, il faut créer le lien entre le niveau inférieur et le niveau supérieur du niveau supprimé sur le plan structurel d'une part (lignes 10 et 11) et sur le plan des données d'autre part (lignes 12 à 15).

Algorithme 3 Propagation des mises à jour dans la hiérarchie

Entrée: niveau_courant le niveau qui est créé ou supprimé, type_opération le type de l'opération subie par niveau_courant

{Création d'un niveau}
1: **si** type_opération=création **alors**
2: Détermination de *id* l'élément identifiant de niveau_supérieur
3: Création dans niveau_courant de *id*
4: **pour tout** instance de niveau_courant **faire**
5: Récupération de la valeur de *id* dans niveau_inférieur
6: Mise à jour de la valeur de *id* dans niveau_courant
7: **fin pour**
8: **fin si**
{Suppression d'un niveau}
9: **si** type_opération=suppression **alors**
10: Détermination dans niveau_courant de *id* l'élément identifiant niveau_supérieur
11: Création dans niveau_inférieur de *id*
12: **pour tout** instance de niveau_inférieur **faire**
13: Récupération de la valeur de *id* dans niveau_courant
14: Mise à jour de la valeur de *id* dans niveau_inférieur
15: **fin pour**
16: **fin si**

5.3 Déploiement de notre démarche dans un contexte relationnel

5.3.1 Principe général

Nous présentons ici la mise en œuvre de notre approche dans un contexte relationnel (ROLAP), en se focalisant sur la création de niveaux de granularité puisque c'est elle qui nécessite l'expression de règles utilisateurs pour répondre à leurs besoins d'analyse, contrairement à la suppression de niveau.

Afin de déployer notre approche dans un contexte ROLAP, nous présentons le modèle d'exécution que nous avons proposé dans [FBB07a]. Ce modèle d'exécution permet de mettre en œuvre les différents modules de notre architecture globale grâce à un enchaînement d'algorithmes (figure 5.4). Nous nous intéressons ici aux différentes étapes qui mènent jusqu'à la mise à jour des hiérarchies.

L'entrepôt de données évolutif étant stocké dans un SGBD relationnel (SGBDR), nous exploitons alors les structures relationnelles de ce SGBDR pour déployer notre approche. L'idée clé consiste alors à transformer les règles qui définissent un nouveau niveau de granularité en une table relationnelle, que nous appelons table de mapping.

Le point de départ de la séquence d'algorithmes est bien évidemment l'ensemble des règles exprimées par l'utilisateur grâce au module d'acquisition. Le module d'intégration permet alors de transformer les règles en une table de mapping. Cette table de mapping permet de stocker les règles sous une forme relationnelle et de vérifier la validité des règles vis-à-vis des contraintes que nous avons posées dans le modèle *R-DW*. Si les règles ne sont pas valides, au sens de ces contraintes, l'utilisateur doit les modifier. Lorsque celles-ci le sont, le module d'évolution permet de créer le niveau de granularité en fin de hiérarchie ou entre deux niveaux existants, nous parlons respectivement d'ajout ou d'insertion de niveau (i.e. création sans ou avec propagation).

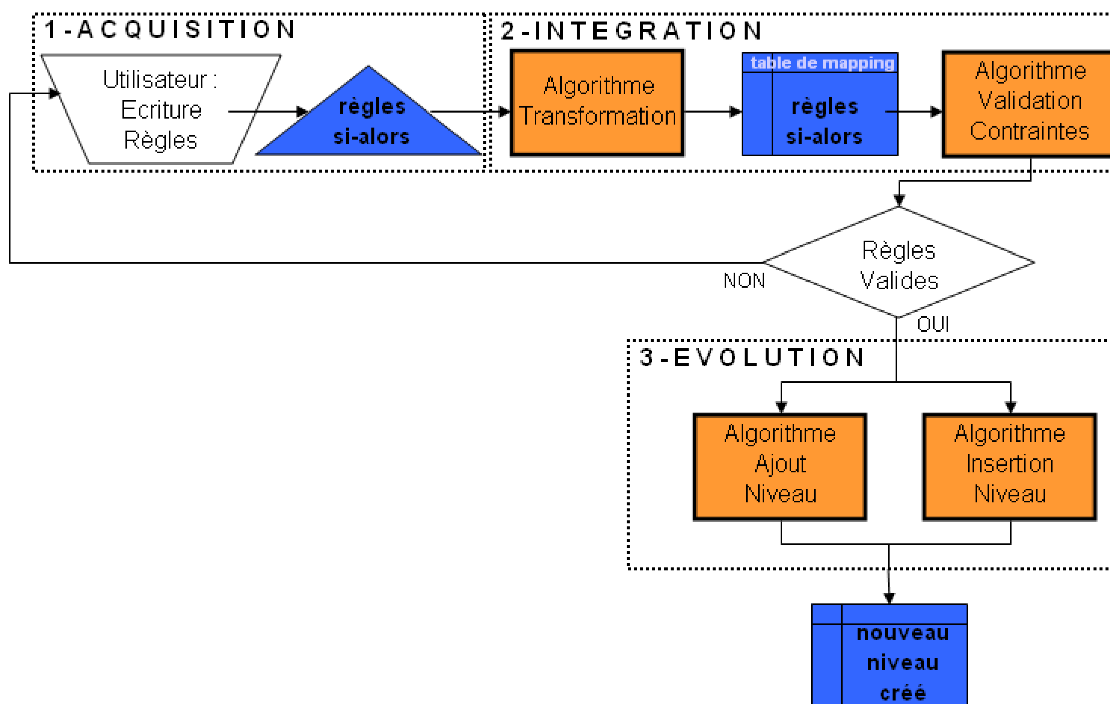

FIG. 5.4 – Modèle d'exécution pour l'évolution des hiérarchies de dimension

5.3.2 Règles et méta-règles

1. Principe

L'acquisition des règles se déroule en deux étapes :

1) *Définition d'une méta-règle d'agrégation qui précise la structure du lien d'agrégation entre deux niveaux.* En effet, la méta-règle permet à l'utilisateur de définir, le niveau de granularité qu'il veut créer, les attributs caractérisant ce nouveau niveau, le niveau de granularité existant à partir duquel il crée le nouveau niveau et les attributs du niveau existant utilisés pour créer le lien d'agrégation.

2) *Définition des règles d'agrégation qui déterminent le lien d'agrégation au niveau des instances.* Ces règles correspondent aux règles d'agrégation utilisées dans le modèle *R-DW*. Pour les définir, l'utilisateur instancie la méta-règle en exprimant une règle par instance du niveau généré afin d'y associer les instances du niveau existant.

2. Définitions

1) *Définition d'une méta-règle*

Soit EL le niveau existant.
Soit $\{EA_i,\ i = 1..m\}$ le sous-ensemble des m attributs pris parmi les m' attributs existants de EL, sur lesquels vont porter les conditions.
Soit GL le niveau généré et $\{GA_j,\ j = 1..n\}$ l'ensemble des n attributs générés de GL.
La méta-règle MR est définie de la façon suivante :

$$MR : \mathit{if\,SelectionOn}(EL, \{EA_i,\ i = 1..m\})\ \mathit{then\,Generate}(GL, \{GA_j,\ j = 1..n\})$$

2) *Définition de l'instanciation de la méta-règle*
Dans la formalisation du modèle *R-DW*, nous avons défini ce qu'est une règle d'agrégation. Une règle d'agrégation est basée sur un ensemble $\mathcal{T}$ de z termes de règles, notés RT_x, tel que :

$$\mathcal{T} = \{RT_x, 1 \leq x \leq z\} = \{EA_x\ op_x\ \{ens|val\}_x\}$$

où EA_x est un attribut du niveau inférieur, op_x est un opérateur soit relationnel ($=$, $<$, $>$, $\leq$, $\geq$, $\neq$, ...), soit ensembliste ($\in$, $\notin$, ...) et $\{ens|val\}_x$ correspond respectivement soit à un ensemble de valeurs, soit à une valeur finie. Ces valeurs appartiennent au domaine de définition de EA_x.

Nous notons ici $c_x = op_x\ \{ens|val\}_x$ la condition portée sur l'attribut EA_x incluant l'opérateur et la valeur ou l'ensemble de valeurs.

Soit q le nombre d'instances du niveau généré GL, un ensemble $R = \{r_d, d = 1..q\}$ de q règles d'agrégation doit être défini.

Ainsi, la condition se rapportant à l'attribut EA_i dans la règle r_d est notée c_{di}. Par ailleurs, nous notons v_{dj} la valeur attribuée à l'attribut généré GA_j dans la règle r_d.

Une règle r_d qui instancie la méta-règle MR est notée :

$$r_d : if\ RT_1\ AND\ ...\ AND\ RT_x\ AND\ ...\ AND\ RT_z$$

$$then\ GA_1 = v_{d1}\ AND\ ...\ AND\ GA_j = v_{dj}\ AND\ ...\ AND\ GA_n = v_{dn}$$

Nous avons donc, en utilisant le concept de condition :

$$r_d : if\ EA_1\ c_{d1}\ AND\ ...\ AND\ EA_i\ c_{di}\ AND\ ...\ AND\ EA_m\ c_{dm}$$

$$then\ GA_1 = v_{d1}\ AND\ ...\ AND\ GA_j = v_{dj}\ AND\ ...\ AND\ GA_n = v_{dn}$$

Cette définition est illustrée dans la figure 5.5.

MR : *if* SelectionOn(EL,{EA_i, i=1..m})	*then* Generate(GL,{GA_j, j=1..n})

r_1 :	*if* EA_1 c_{11} AND ... EA_i c_{1i} ... AND EA_m c_{1m}	*then* GA_1=v_{11} AND ... GA_j=v_{1j} ... AND GA_n=v_{1n}
	...	...
r_d :	*if* EA_1 c_{d1} AND ... EAi c_{di} ... AND EA_m c_{dm}	*then* GA_1=v_{d1} AND ... GA_j=v_{dj} ... AND GA_n=v_{dn}
	...	...
r_q :	*if* EA_1 c_{q1} AND ... EAi c_{qi} ... AND EA_m c_{qm}	*then* GA_1=v_{q1} AND ... GA_j=v_{qj} ... AND GA_n=v_{qn}

FIG. 5.5 – Illustration de la méta-règle et des règles d'agrégation

3. Exemple

Dans l'exemple de LCL, lors de la phase d'*acquisition*, l'utilisateur définit la méta-règle d'agrégation MR pour spécifier la structure du lien d'agrégation pour le type d'agence. Elle exprime donc le fait que le niveau `AGENCY_TYPE` va être caractérisé par l'attribut `AgencyTypeLabel` et qu'il sera créé au-dessus du niveau `AGENCY` ; les regroupements des instances de `AGENCY` se baseront sur des conditions exprimées sur l'attribut `AgencyID`.

MR : *if* SelectionOn(`AGENCY`,{`AgencyID`})
then Generate(`AGENCY_TYPE`{`AgencyTypeLabel`})

Puis l'utilisateur définit les règles d'agrégation qui instancient la méta-règle pour créer les différents types d'agence. Ainsi, il définit une règle pour chaque type d'agence, en exprimant à chaque fois la condition sur l'attribut `AgencyID` et en affectant à l'attribut généré `AgencyTypeLabel` sa valeur correspondante :

(R1) *if* `AgencyID` $\in$ {'01903','01905','02256'} *then* `AgencyTypeLabel`='student'
(R2) *if* `AgencyID` = '01929' *then* `AgencyTypeLabel`='foreigner'
(R3) *if* `AgencyID` $\notin$ {'01903','01905','02256','01929'} *then* `AgencyTypeLabel`='classical'

5.3.3 Table de mapping et méta-table de mapping

1. Principe

L'intégration des règles dans l'entrepôt se décompose en deux étapes :

1) **Transformation des règles en une table de mapping.** Il s'agit de transformer les règles de type «si-alors» en une table de mapping et de la stocker dans le SGBDR au moyen d'une structure relationnelle. Pour un ensemble donné de règles qui définit un nouveau niveau de hiérarchie, nous y associons une table de mapping. Pour réaliser cette transformation, nous exploitons dans un premier temps la méta-règle d'agrégation pour définir la structure de la table de mapping. Dans un deuxième temps, nous utilisons les règles d'agrégation elles-mêmes pour insérer les enregistrements correspondants dans la table de mapping. Cette table de mapping reprend les attributs sur lesquels portent des conditions, ainsi que les attributs générés caractérisant le niveau créé.
2) **Stockage des informations sur la table de mapping dans une méta-table de mapping.** Nous définissons une méta-table de mapping pour rassembler les informations sur les différentes tables de mapping. En effet, leur structure peut différer en fonction du nombre d'attributs supportant des conditions et du nombre d'attributs générés caractérisant le nouveau niveau. Elle contient également le rôle des attributs dans le processus.

2. Définitions

1) *Définition d'une table de mapping*

Soit ET (resp. GT) la table existante (resp. générée), correspondant à EL (resp. GL).
Soient $\{EA_i,\ i = 1..m\}$ l'ensemble des m attributs de ET et $\{GA_j,\ j = 1..n\}$ l'ensemble des n attributs générés de GT.
Soit GA_{key} l'attribut ajouté automatiquement qui sera la clé primaire de GT.

La table de mapping MT_GT construite à partir de la méta-règle MR correspond à la relation suivante :

$$MT_GT(EA_1,\ ...,\ EA_i,\ ...,\ EA_m,\ GA_1,\ ...,\ GA_j,\ ...,\ GA_n,\ GA_{key})$$

Les règles d'agrégation permettent d'insérer dans cette table de mapping les conditions sur les attributs $\{EA_i,\ i = 1..m\}$ et les valeurs des attributs générés correspondants $GA_j,\ j = 1..n$. Pour l'attribut GA_{key}, la valeur est ajoutée de

façon incrémentale. Ainsi, pour chaque enregistrement d de la table de mapping, on a les conditions appliquées aux attributs $(c_{di},\ i = 1..m)$ définissant quelles instances de la table ET sont concernées, à quelle instance elles correspondent dans GT. Cette dernière est caractérisée par les valeurs des attributs de la table GT $(v_{dj},\ j = 1..n)$.
Cette définition est illustrée dans la figure 5.6.

MT_GT										
EA_1	...	EA_i	...	EA_m	GA_1	...	GA_j	...	GA_n	GA_{key}
c_{11}	...	c_{1i}	...	c_{1m}	v_{11}	...	v_{1j}	...	v_{1n}	key_1
...	...	...	...	...	...	...	...	...	...	...
c_{d1}	...	c_{di}	...	c_{dm}	v_{d1}	...	v_{dj}	...	v_{dn}	key_d
...	...	...	...	...	...	...	...	...	...	...
c_{q1}	...	c_{qi}	...	c_{qm}	v_{q1}	...	v_{qj}	...	v_{qn}	key_q

FIG. 5.6 – Illustration de la table de mapping

2) *Définition d'une méta-table de mapping*

La structure de la méta-table de mapping est représentée par la relation suivante :

$$MAPPING_META_TABLE(Mapping_Table_ID,$$
$$Mapping_Table_Name, Table_Name, Attribute_Name, Attribute_Type)$$

où *Mapping_Table_ID* et *Mapping_Table_Name* correspondent respectivement à l'identifiant et au nom de la table de mapping ; *Attribute_Name* et *Table_Name* caractérisent l'attribut impliqué dans le processus d'évolution et sa table d'appartenance ; *Attribute_Type* fournit le rôle de cet attribut.

Plus précisément, ce dernier attribut a trois modalités : «*conditioned*» s'il apparaît dans la clause «si» et «*generated descriptor*» ou «*generated key*» s'il est dans la clause «alors» selon qu'il s'agit d'un attribut clé ou non. Notons que même si un attribut a le rôle de «*generated descriptor*» ou «*generated key*» dans une table de mapping, il peut avoir le rôle «*conditioned*» dans une autre table de mapping pour servir à la construction d'un autre niveau. Ainsi, il est possible de construire deux niveaux successifs dans une hiérarchie.
Cette définition est illustrée dans la figure 5.7.

MAPPING_META_TABLE				
Mapping_Table_ID	**Mapping_Table_Name**	**Table_Name**	**Attribute_Name**	**Attribute_Type**
...	...	...	...	...
y	MT_GT	ET	EA_i	conditioned
y	MT_GT	...	...	...
y	MT_GT	GT	GA_j	generated descriptor
y	MT_GT	...	...	...
y	MT_GT	GT	GA_{key}	generated key
...	...	...	...	...

FIG. 5.7 – Illustration de la méta-table de mapping

3. Exemple

Dans l'exemple LCL, la phase d'*intégration* exploite la méta-règle et les règles d'agrégation R1, R2 et R3 pour générer la table de mapping MT_AGENCY_TYPE (figure 5.8) et les informations concernant cette table sont insérées dans la méta-table *MAPPING_META_TABLE* (figure 5.9). Ainsi la table de mapping contient les attributs `AgencyTypeLabel` et `AgencyID` respectivement, que l'on retrouve dans la méta-table *MAPPING_META_TABLE* et dont les rôles sont respectivement «*generated descriptor*» et «*conditioned*». La clé `AgencyTypeID` est rajoutée automatiquement et figure donc dans la méta-table avec le rôle «*generated key*».

MT_AGENCY_TYPE		
AgencyID	***AgencyTypeLabel***	***AgencyTypeID***
IN ('01903', '01905', '02256')	student	1
= '01929'	foreigner	2
NOT IN ('01903', '01905', '02256', '01929')	classical	3

FIG. 5.8 – Table de mapping pour le niveau `AGENCY_TYPE`

MAPPING_META_TABLE				
Mapping_Table_ID	***Mapping_Table_Name***	***Attribute_Table***	***Attribute_Name***	***Attribute_Type***
1	MT_AGENCY_TYPE	AGENCY	AgencyID	conditioned
1	MT_AGENCY_TYPE	AGENCY_TYPE	AgencyTypeLabel	generated descriptor
1	MT_AGENCY_TYPE	AGENCY_TYPE	AgencyTypeID	generated key

FIG. 5.9 – Méta-table de mapping prenant en compte le niveau `AGENCY_TYPE`

5.3.4 Création du niveau

3. Principe

Le processus d'évolution est réalisé à partir de la table de mapping dédiée à la création du niveau et des instances qui lui sont associées dans la méta-table. Quel que soit le type d'évolution appliqué (ajout ou insertion), la structure de la table GT est donc créée selon les attributs de la méta-table $MAPPING_META_TABLE$ qui comportent la mention «*generated key*», i.e. GA_{key}, pour la clé de la table et «*generated descriptor*» pour les autres attributs $\{GA_j,\ j = 1..n\}$.

Les instances de cette table sont insérées selon le contenu de la table de mapping MT_GT, en l'occurrence les valeurs de la clé $\{key_d,\ d = 1..q\}$ et les valeurs des autres attributs $\{v_{dj},\ d = 1..q,\ j = 1..n\}$.

La structure de la table ET est mise à jour avec l'ajout de l'attribut GA_{key} qui constitue une clé étrangère référençant la table GT (coloration en gris clair dans la figure 5.10). La valeur de cet attribut est mise à jour pour chacune des instances de la table en fonction des conditions exprimées sur les attributs de ET dans la table de mapping.

Dans le cas où GT est inséré entre ET et $ET2$, la table GT comporte également un attribut L qui constituera la clé étrangère référençant l'attribut L de $ET2$ pour établir le lien (liaison en pointillé sur la figure 5.10). Cet attribut est alimenté grâce à la valeur qu'il a dans la table ET (coloration en gris foncé dans la figure 5.10). Ainsi, dans le cas d'une insertion de niveau, le lien entre le niveau créé et le niveau supérieur est défini de façon automatique.

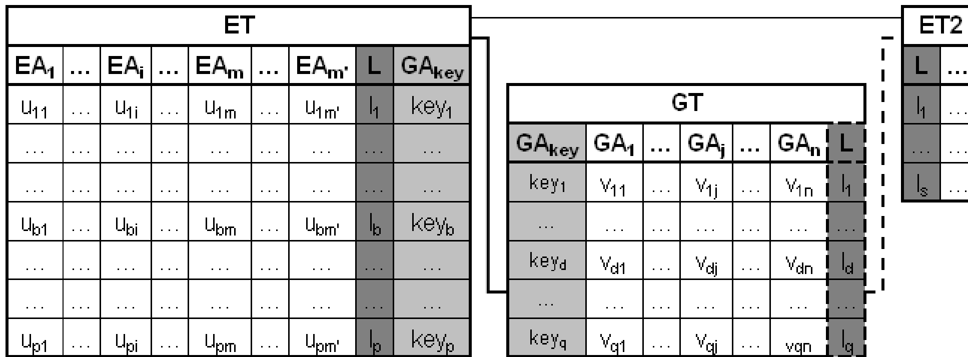

Fig. 5.10 – Évolution de la hiérarchie de dimension

2. Exemple

Dans notre exemple, l'évolution choisie par l'utilisateur correspond à un ajout.

En effet, il n'y a pas de lien possible d'agrégation entre le type d'agence et l'unité commerciale qui correspond à un regroupement géographique des agences. La phase d'*évolution* qui suit permet donc d'une part de créer et d'alimenter la table `AGENCY_TYPE` ; et d'autre part de mettre à jour la table `AGENCY` pour la relier à la table `AGENCY_TYPE`, avec l'ajout de l'attribut `AgencyTypeID` et la mise à jour de ses valeurs (figure 5.11).

AGENCY			
AgencyID	AgencyLabel	...	AgencyTypeID
01000	LYON REPUBLIQUE	...	3
01029	LYON GERLAND	...	3
01903	LYON III UNIVERSITE	...	1
01905	LYON LA DOUA	...	1
01929	AGENCE INTERNATIONALE	...	2
02256	CLERMONT LAFAYETTE (Etud)	...	1
02600	GRENOBLE	...	3
03730	ANNONAY	...	3

AGENCY_TYPE	
AgencyTypeID	AgencyTypeLabel
1	student
2	foreigner
3	classical

FIG. 5.11 – La table `AGENCY_TYPE` créée et la table `AGENCY` mise à jour

5.3.5 Algorithmes

Le modèle d'exécution exploite différents algorithmes afin de mettre en œuvre les modules de notre architecture. Ces algorithmes sont basés sur les concepts et les définitions que nous venons de présenter.

Concernant le module d'intégration, les règles sont donc transformées en une table de mapping (algorithme 4). Puis la validité de ces règles est vérifiée en testant le contenu de la table de mapping (algorithme 5). Le module d'évolution permet la création du niveau : soit le niveau de granularité est ajouté à la fin d'une hiérarchie, comme niveau le plus grossier de la hiérarchie (algorithme 6), soit il est inséré entre deux niveaux existants (algorithme 7).

5.3.5.1 Transformation des règles en une table de mapping

L'algorithme 4 permet de transformer les règles en une table de mapping. Plus précisément, pour une méta-règle MR et l'ensemble des règles d'agrégation qui lui sont associées R, nous construisons une table de mapping MT_GT (ligne 1). La structure de la table de mapping est définie à l'aide de la méta-règle. En effet, comme nous l'avons vu précédemment, la table de mapping contient à la fois les attributs du niveau inférieur sur lesquels porteront les conditions définissant le regroupement des instances et les attributs créés pour le nouveau niveau. La clé primaire de ce

nouveau niveau est également ajoutée de façon automatique. Ensuite, le contenu (les instances) de la table de mapping est déterminé par rapport à l'ensemble des règles d'agrégation (lignes 2 à 4). Chacune des règles d'agrégation correspond à une instance dans la table de mapping : les conditions portant sur les attributs du niveau inférieur et les valeurs des attributs du nouveau niveau sont donc reportées dans la table de mapping.

En outre, diverses informations sur la table de mapping sont insérées dans la méta-table $MAPPING_META_TABLE$ (lignes 5 à 10). Cela permet de connaître, à terme, les différentes tables de mapping et leur structure. Rappelons en effet que chaque table de mapping a sa propre structure puisque le nombre d'attributs sur lesquels portent des conditions et le nombre d'attributs dans le niveau créé sont variables.

Algorithme 4 Pseudo-code pour la transformation des règles en une table de mapping

Entrée: (1) méta-règle MR : *if* SelectionOn(EL,$\{EA_i\}$) *then* Generate(GL, $\{GA_j\}$) où EL est le niveau existant, $\{EA_i,\ i = 1..m\}$ est l'ensemble des m attributs de EL, GL est le niveau à créer et $\{GA_j,\ j = 1..n\}$ est l'ensemble des n attributs générés de GL ; (2) l'ensemble des règles d'agrégation R ; (3) la méta-table de mapping $MAPPING_META_TABLE$

Sortie: la table de mapping MT_GT

{Création de la structure de MT_GT}
1: CREATE TABLE MT_GT($\{EA_i\}$,$\{GA_j\}$)
{Insertion des valeurs dans MT_GT}
2: **pour tout** règle $\in R$ **faire**
3: INSERT INTO MT_GT VALUES (SelectionOn(EL,$\{EA_i\}$) , Generate(GL, $\{GA_j\}$))
4: **fin pour**
{Insertion des valeurs dans $MAPPING_META_TABLE$}
5: **pour tout** $EA_i \in \{EA_i\}$ **faire**
6: INSERT INTO $MAPPING_META_TABLE$ VALUES(MT_GT, EL, EA_i, 'conditioned')
7: **fin pour**
8: **pour tout** $GA_j \in \{GA_j\}$ **faire**
9: INSERT INTO $MAPPING_META_TABLE$ VALUES(MT_GT, GL, GA_i, 'generated')
10: **fin pour**
11: **return** MT_GT

5.3.5.2 Vérification des contraintes

L'algorithme 5 vise à vérifier que les règles satisfont les contraintes liées à la partition et à la bijection. Cette vérification se fait par rapport aux données contenues dans l'entrepôt suivant cette méthode :

1) Pour chaque enregistrement de MT_GT (ce qui correspond à une règle), nous formulons la requête correspondante afin de construire la vue qui contient l'ensemble des instances concernées de ET (lignes 2 à 4).
2) Vérification de la contrainte liée au concept de partition

- Vérifier qu'aucune des vues n'est vide (lignes 5 à 10)
- Vérifier que l'intersection des vues prises deux à deux est vide (lignes 11 à 16)
- Vérifier que l'union des instances de l'ensemble des vues correspond à l'ensemble des instances de la table ET (lignes 17 à 23).

3) Vérification de la contrainte liée au concept de bijection. Cette contrainte est vérifiée si les contraintes liées au concept de partition sont satisfaites. Compte tenu de la forme de la table de mapping et de son contenu, il suffit alors de vérifier que le nombre de valeurs distinctes (non nulles) correspondant au nombre d'instances du niveau créé est égal au nombre de ligne de la table MT_GT, cela représente l'idée qu'une instance du niveau créé n'est pas mise en correspondance avec plus d'un sous-ensemble d'instances du niveau inférieur et que chaque sous-ensemble a son instance correspondante (lignes 24 à 30). Notons que nous ne pouvons nous baser sur la valeur de la clé GA_{key} puisque celle-ci est générée automatiquement lors de la création de la table de mapping.

Algorithme 5 Pseudo-code pour vérifier les contraintes

Entrée: table existante ET, table de mapping MT_GT
Sortie: NonEmpty_checked, Intersection_checked, Union_checked et Bijection_checked quatre booléens
{Initialisation des booléens}
1: NonEmpty_checked=true ; Intersection_checked=true ; Union_checked=true ; Bijection_checked=true
{Création des vues}
2: **pour tout** tuple $t \in MT_GT$ **faire**
3: CREATE VIEW v_t AS SELECT * FROM ET WHERE Conjunction($t(\{EA\})$)
4: **fin pour**
{Vérification de la contrainte de partition}
{Vérification qu'aucune des vues n'est vide}
5: **pour** $x = 1$àt_{max} **faire**
6: nb=SELECT COUNT(*) FROM v_x
7: **si** $nb \neq 0$ **alors**
8: NonEmpty_checked=false
9: **fin si**
10: **fin pour**
{Vérification que l'intersection des vues prises deux à deux est vide}
11: **pour** $x = 1$ à $(t_{max} - 1)$ **faire**
12: I=SELECT * FROM v_x INTERSECT SELECT * FROM $v_x + 1$
13: **si** $I \neq \emptyset$ **alors**
14: Intersection_checked=false
15: **fin si**
16: **fin pour**
{Vérification que l'union de toutes les vues correspond à la table ET}
17: U=SELECT * FROM v_1
18: **pour** $x = 2$ à t_{max} **faire**
19: U=U UNION SELECT * FROM v_x
20: **fin pour**
21: **si** U $\neq$ SELECT * FROM ET **alors**
22: Union_checked=false
23: **fin si**
{Vérification de la bijection}
24: **si** (NonEmpty_checked **et** Intersection_checked **et** Union_checked) **alors**
25: nb'=SELECT COUNT(DISTINCT GA_j) FROM MT_GT
26: **si** $nb' \neq$ SELECT COUNT(*) FROM MT_GT **alors**
27: Bijection_checked=false
28: **fin si**
29: **return** Bijection_checked
30: **fin si**
31: **return** NonEmpty_checked, Intersection_checked, Union_checked

5.3.5.3 Algorithmes de création d'un niveau de hiérarchie

L'algorithme 6 permet d'ajouter (fin de hiérarchie) une nouvelle table GT, à partir d'une table existante ET, en exploitant une table de mapping MT. L'opération s'effectue en trois étapes : (1) créer la table GT (ligne 1) ; (2) modifier la structure de la table ET pour la lier à GT (ligne 2) ; (3) insérer dans GT les valeurs nécessaires en fonction de la table de mapping (lignes 3 à 6) .

Algorithme 6 Pseudo-code pour ajouter un nouveau niveau de hiérarchie

Entrée: (1) table existante ET, (2) table de mapping MT, (3) $\{EA_i,\ i = 1..m\}$ l'ensemble des m attributs de MT qui contiennent les conditions sur les attributs, (4) $\{GA_j,\ j = 1..n\}$ l'ensemble des n attributs générés de MT qui contiennent les valeurs des attributs générés, (5) GA_{key} l'attribut clé primaire de GT

Sortie: table générée GT

{Création de la table GT}

1: CREATE TABLE GT (GA_{key}, $\{GA_j\}$) ;

{Mise à jour de la structure de ET pour la lier à GT}

2: ALTER TABLE ET ADD (GA_{key}) ;

{Alimentation de GT, liaison entre ET et GT}

3: **pour tout** (tuple t of MT) **faire**

4: INSERT INTO GT VALUES (GA_{key},$\{GA_j\}$) ;

5: UPDATE ET SET $ET.GA_{key} = GT.GA_{key}$ WHERE $\{EA_i\}$;

6: **fin pour**

7: **return** GT

L'algorithme 7 permet d'insérer une nouvelle table GT entre deux tables existantes (ET et $ET2$), en prenant en compte une table de mapping MT. Cette table de mapping ne contient que le lien entre la table inférieure ET et la table générée GT. Les étapes sont les mêmes que pour l'ajout d'un niveau. Ensuite, le lien d'agrégation est déterminé de façon automatique entre GT et $ET2$, en le déterminant à partir du lien existant entre ET et $ET2$ (ligne 6).

Algorithme 7 Pseudo-code pour insérer un nouveau niveau de hiérarchie

Entrée: table existante inférieure ET, table existante supérieure $ET2$, table de mapping MT, $\{EA_i,\ i = 1..m\}$ l'ensemble des m attributs de MT qui contiennent les conditions sur les attributs, $\{GA_j,\ j = 1..n\}$ l'ensemble des n attributs générés de MT qui contiennent les valeurs des attributs générés, GA_{key} l'attribut clé primaire de GT, L l'attribut liant ET à $ET2$

Sortie: table générée GT

{Création de la table GT}

1: CREATE TABLE GT (GA_{key}, $\{GA_j\}$, L) ;

{Mise à jour de la structure de ET pour la lier à GT}

2: ALTER TABLE ET ADD (GA_{key}) ;

{Alimentation de GT, liaison entre ET et GT, liaison entre GT et $ET2$}

3: **pour tout** (tuple of MT) **faire**

4: INSERT INTO GT VALUES (GA_{key},$\{GA_j\}$) ;

5: UPDATE ET SET $ET.GA_{key} = GT.GA_{key}$ WHERE $\{EA_i\}$;

6: UPDATE GT SET L=(SELECT DISTINCT L FROM ET WHERE $ET.GA_{key}$=$GT.GA_{key}$ AND $\{EA_i\}$) ;

7: **fin pour**

8: **return** GT

5.3.5.4 Complexité des algorithmes de mises à jour

Pour étudier la complexité des algorithmes de création de niveau (ajout, insertion), nous nous focalisons sur les opérations d'écriture et de lecture. Nous supposons ici que la création de la structure d'une table ou que la modification d'une structure de table existante par l'ajout d'un attribut sont négligeables. La complexité est me-

surée en terme de nombre d'opérations de lecture ou d'écriture d'un attribut pour un enregistrement donné.

Pour l'ajout d'un niveau de hiérarchie, il y a deux types d'opérations d'écriture : les insertions dans la nouvelle table GT et les mises à jour dans la table existante ET pour créer le lien entre ces deux tables. La complexité de ces opérations est de :

$$\omega((p+n+1)\times q+p)$$

En effet, dans la table GT, il y a $n+1$ attributs et q enregistrements. Donc il y a $(n+1)\times q$ opérations d'écriture (insertion des valeurs). Pour faire le lien entre ET et GT, la valeur de l'attribut représentant la clé étrangère est mise à jour pour chaque enregistrement de la table ET, soit concernant le nombre d'opérations d'écriture : p. Ainsi, le total des opérations d'écriture est de : $p+(n+1)\times q$. Pour les opérations de lecture, afin de mettre à jour la valeur de la clé étrangère, pour chacun des q enregistrements de la méta-table, les p enregistrements de la table ET sont parcourus, ainsi on obtient pour les opérations de lecture : $p\times q$.

L'insertion d'un niveau de hiérarchie entre deux niveaux existants comprend, en plus des opérations décrites précédemment dans le cas de l'ajout, une opération de mise à jour pour lier la table générée GT avec la table existante de niveau supérieur $ET2$. La complexité totale de ces opérations est de :

$$\omega((p+n+2)\times q+p)$$

En effet, pour mettre à jour l'attribut représentant la clé étrangère dans le niveau créé GT, il y a q opérations. L'opération de lecture qui y est associée est assimilable à celle qui est réalisée lors de l'ajout et n'augmente donc pas la complexité.

Ainsi notre approche est plus pertinente lorsque q (nombre d'enregistrements dans la table générée GT) et p (nombre d'enregistrements dans la table existante ET) sont petits du point de vue de la complexité, ce qui est le cas pour des niveaux de granularité élevé. C'est également le cas d'un point de vue pratique, puisque l'utilisateur doit définir une règle d'agrégation par instance du niveau créé. Ainsi, si q est élevé, l'utilisateur doit définir un grand nombre de règles, ce qui rend l'approche très fastidieuse. Notre approche trouve donc son utilité pour créer de nouveaux agrégats lorsque les niveaux d'agrégation créés comportent peu d'instances de regroupement.

5.4 Discussion

5.4.1 Extension des possibilités d'analyse

Différents logiciels décisionnels de restitution de données tels que Business Object[1] fournissent aux utilisateurs une interface qui leur permet de manipuler les données comme ils le souhaitent. Une des fonctionnalités qu'ils offrent est la création de variables, avec la possibilité de regrouper des instances de dimension. Il s'agit donc également de personnaliser les analyses. Néanmoins cette fonctionnalité diffère réellement de notre approche sur les points suivants.

Tout d'abord, au niveau de l'expressivité du regroupement, mentionnons que les instances doivent être pointées, il n'y a pas de «règles» exprimables de regroupement. Cela rend la tâche d'autant plus fastidieuse lorsqu'il y a un certain nombre d'instances à regrouper et cela peut être source d'erreurs.

Ensuite, cette création de variable se fait au niveau des rapports (état d'analyse) directement. La création de ces regroupements n'est donc pas persistante, ni vis-à-vis d'autres états d'analyse, ni par rapport aux autres utilisateurs ; autrement dit la nouvelle possibilité d'analyse générée n'est pas partageable avec d'autres utilisateurs.

Ainsi notre approche permet cette création de variable en allant bien au-delà. En effet, il s'agit de créer un nouveau niveau avec la possibilité de créer plusieurs descripteurs pour ce niveau. En outre, grâce à l'usage de règles, nous permettons une plus grande expressivité dans les conditions de regroupement, avec une source d'erreurs moindre. Enfin, en proposant une évolution de modèle, nous rendons persistantes les nouvelles possibilités d'analyse créées individuellement, que ce soit pour le créateur lui-même, ou pour d'autres utilisateurs.

5.4.2 Évolution des règles

Jusque là, nous avons traité d'évolution de schéma grâce à l'utilisation de règles. Mais une fois ces règles exprimées, il est possible qu'elles-mêmes évoluent et ce, pour deux raisons. Le premier cas se produit si l'utilisateur souhaite les modifier, autrement dit, changer la définition du niveau créé. Le second cas se produit lorsque le rafraîchissement des données dans l'entrepôt rendent les règles obsolètes. Il est possible par exemple que les conditions de partition ne soient plus satisfaites suite à l'ajout d'instances dans un niveau inférieur. Notons d'ailleurs que, dans le cas de mises à jour sur les données (ajout, suppression, modification), les contraintes

[1] http ://www.france.businessobjects.com/

doivent être re-vérifiées. Selon les fonctionnalités du SGBDR, on peut envisager le recours à des déclencheurs (triggers) paramétrés pour réeffectuer les vérifications de façon automatique.

Si les règles définissant un niveau sont modifiées, la table de mapping doit subir également les modifications, ces dernières devront être répercutées sur le niveau de hiérarchie impacté. Ainsi, le problème de l'évolution des règles s'apparente finalement à celui de l'évolution de modèle telle que nous l'avons définie dans l'état de l'art. On peut alors réaliser soit une mise à jour des règles, soit un versionnement de celles-ci, qui conduiront respectivement à une mise à jour ou à un versionnement des hiérarchies de dimension.

Se posent alors différentes questions. Doit-on adopter une solution de façon systématique ? En effet, n'est-t-il pas préférable d'adopter des solutions différentes en fonction de cas différents, comme l'évoquait Kimball à travers ses trois types de «dimensions changeantes à évolution lente» [Kim96]. S'il y a une distinction à faire selon les cas, qui doit la faire ? Est-ce le rôle de l'utilisateur qui définit le niveau de granularité en question ? Est-ce que sa tâche ne s'en trouve pas trop complexifiée ? Autant de questions qui méritent d'être soulevées et qui doivent être approfondies.

5.5 Conclusion

Dans ce chapitre, nous avons présenté la mise à jour des hiérarchies de dimension. Cette mise à jour est basée sur la création et la suppression de niveaux de granularité. La création permet quant à elle la réponse à des besoins d'analyse émergents et nécessitent l'expression des règles d'agrégation. Ces actions de création et de suppression de niveaux de granularité peuvent nécessiter une propagation dans la hiérarchie de dimension lorsque les niveaux en question ne se situent pas en fin de hiérarchie.

Notre modèle d'exécution, proposé dans un contexte relationnel, gère l'ensemble des processus liés aux modules de notre architecture globale. Il se base ainsi sur des structures relationnelles en gérant la transformation des règles en une table de mapping qui est exploitée durant toute la démarche. Ainsi, ce modèle d'exécution permet de mettre en œuvre l'évolution du schéma de l'entrepôt, qui est rendue possible grâce au modèle d'entrepôt évolutif que nous avons proposé.

Nous abordons alors dans le chapitre suivant la problématique de l'évaluation de ce modèle évolutif.

Vers une évaluation des modèles d'entrepôt de données évolutifs

Résumé

Afin d'évaluer les performances de notre approche, nous nous sommes intéressés à l'évolution incrémentale de la charge (ensemble de requêtes) en fonction des changements subis par le schéma de l'entrepôt. Dans ce chapitre, nous présentons notre approche d'évolution de charge qui a pour but de permettre l'évaluation de modèles d'entrepôt évolutifs. L'objectif est double : maintenir la cohérence des requêtes existantes de la charge vis-à-vis de l'évolution du schéma, générer de nouvelles requêtes pour traduire les besoins d'analyse potentiels lorsque l'évolution du schéma de l'entrepôt engendre un enrichissement des possibilités d'analyse.

Sommaire

6.1 Introduction 119
6.2 État de l'art 120
6.3 Évolution de schéma : conséquences sur la charge 124
6.4 Évolution de charge 125
6.5 Exemple 128
6.6 Discussion 131
6.7 Conclusion 133