Troisième partie

Développement : validation de nos contributions et réalisation industrielle

La troisième et dernière partie de ce mémoire est consacrée au travail de développement. Ce travail s'inscrit dans deux contextes différents : industriel d'une part et scientifique d'autre part.

Tout d'abord, nous avons été amenés à réaliser pour le compte de LCL une plateforme logicielle permettant la gestion du processus des demandes de marketing local. Ainsi, dans le chapitre 8, nous donnons les différents éléments concernant la conception et le développement de cette plateforme que nous avons baptisée MARKLOC.

À partir de ce travail d'ingénierie, nous avons conçu un entrepôt de données test nommé LCL-DW sur lequel se sont posés les problèmes scientifiques que nous avons traités dans ce mémoire en apportant des contributions. Pour valider ces dernières, nous avons donc développé la plateforme WEDriK que nous présentons dans le chapitre 7.

La plateforme WEDriK

Résumé

Ce chapitre a pour but de présenter les développements informatiques que nous avons réalisés dans le but de valider nos contributions en terme de personnalisation des analyses. Ainsi, nous avons implémenté un applicatif, baptisé WEDriK pour data Warehouse Evolution Driven by Knowledge. Dans ce chapitre, nous nous attachons à décrire cet applicatif, comment il fonctionne, quelles sont ses fonctionnalités, etc.

Sommaire

Chapitre 7

La plateforme WEDriK

7.1 Introduction

Pour valider notre approche de personnalisation des analyses, nous avons développé un prototype nommé WEDriK[1] (data Warehouse Evolution Driven by Knowledge) selon une configuration client/serveur. Comme nous l'avons évoqué dans [FBB06e], ce prototype a pour but de mettre en œuvre nos contributions sur la personnalisation.

Le prototype WEDriK est implémenté dans un environnement relationnel. Ainsi, il correspond à une couche applicative qui permet la personnalisation des analyses dans un entrepôt de données stocké dans le SGBD relationnel Oracle 10g[2]. Pour mettre en œuvre la personnalisation, l'utilisateur interagit avec le système via une interface web. Des scripts PHP[3] sont utilisés. L'interaction entre l'interface et le SGBD se fait grâce à des fonctions OCI (Oracle Call Interface) qui sont amenées, parfois, à lancer des scripts PL/SQL.

L'objectif de WEDriK est de permettre l'évolution de schéma de l'entrepôt de données selon les besoins exprimés par l'utilisateur et ce, quel que soit l'entrepôt de données considéré. Pour ce faire, nous avons implémenté le méta-modèle d'entrepôt qui permet de décrire différents schémas d'entrepôt dans une base de données dans le SGBDR Oracle 10g. Nous présentons ici les fonctionnalités de la plateforme WEDriK elle-même. Par ailleurs, nous avons intégré dans WEDriK un module assurant l'évolution incrémentale de la charge.

[1] http ://eric.univ-lyon2.fr/~cfavre/wedrik
[2] http ://www.oracle.com
[3] http ://www.phpfrance.com

7.2 Fonctionnalités

WEDriK offre différentes fonctionnalités qui permettent la réalisation de notre approche de personnalisation selon son architecture globale. La figure 7.1 montre les fonctionnalités de la plateforme WEDriK pour l'utilisateur : visualisation du schéma courant de l'entrepôt, expression des connaissances sous forme de règles, construction des requêtes d'analyse.

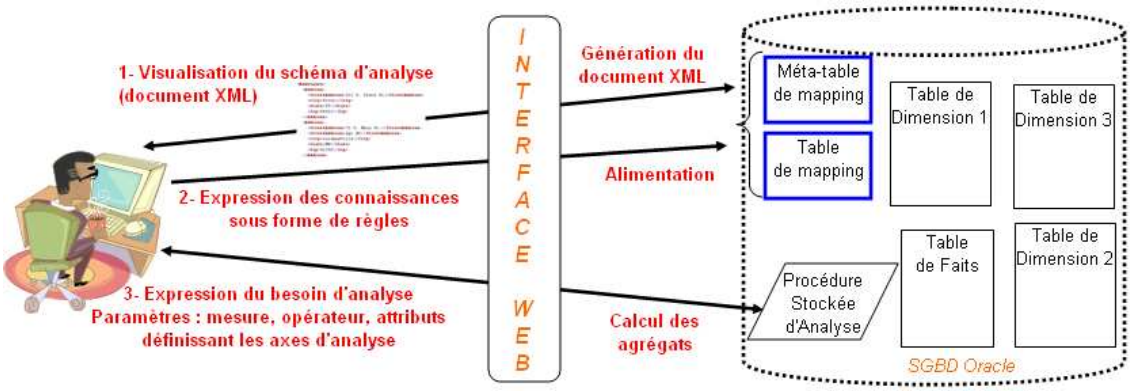

FIG. 7.1 – Fonctionnement de WEDriK

7.2.1 Schéma de l'entrepôt de données

Avant d'envisager d'étendre les possibilités d'analyse de l'entrepôt, il faut que l'utilisateur soit en mesure de connaître les possibilités d'analyse actuelles. En effet, étant donné que les utilisateurs font évoluer l'entrepôt de façon incrémentale, il est nécessaire de pouvoir visualiser le schéma que l'on peut qualifier de courant (photographie du schéma au moment où on l'observe).

Pour ce faire, un document XML (eXtensible Markup Language : langage de balisage extensible) est généré à partir de l'interrogation du méta-modèle. L'avantage de XML est de permettre aux utilisateurs de naviguer à travers les hiérarchies du modèle et de décrire correctement les possibilités d'analyse. Ainsi, ce document va permettre aux utilisateurs de les aider dans leur choix d'analyse. Cette fonctionnalité exploite donc le standard XML, évitant le recours à un outil de visualisation utilisant un format propriétaire. Ce document XML est généré grâce à l'interrogation de la base implémentant le méta-modèle. En effet, grâce à des requêtes sur cette base, il est possible de déterminer les faits, les dimensions, les hiérarchies, et l'ensemble des attributs qui les décrivent.

7.2.2 Saisie des règles d'agrégation

Notre objectif pour permettre la saisie des règles était de faire en sorte que la démarche soit conviviale et de rendre transparents les concepts théoriques sous-jacents de notre méthode.

La saisie des règles est rendue facile pour les utilisateurs parce qu'elle est guidée. En effet, l'exploitation du méta-modèle permet de fonctionner avec des listes de valeurs prédéfinies comme nous l'indiquons dans ce qui suit.

Les différentes étapes de la saisie sont les suivantes :

1) Choix du niveau à partir duquel est créé le nouveau niveau : ce niveau est sélectionné dans une liste ;
2) Choix des attributs qui seront utilisés dans les conditions des règles : les attributs du niveau choisi figurent dans une liste ;
3) Saisie des conditions (figure 7.3) : les attributs sélectionnés apparaissent, l'utilisateur choisit l'opérateur qui est utilisé dans la règle (inférieur, supérieur, égal, dans la liste, etc.) et saisit la ou les valeurs par rapport à cet opérateur, il demande la saisie d'une nouvelle condition pour passer à une autre règle ;
4) Définition de la structure du nouveau niveau : l'utilisateur donne un nom au nouveau niveau et précise le nombre d'attributs qu'il contiendra (la clé est gérée automatiquement, elle n'est pas explicitée par l'utilisateur) ;
5) Saisie des attributs du nouveau niveau (figure 7.2) : l'utilisateur saisit un nom pour chacun des attributs et lui affecte un type de données (chaîne de caractères, entier, etc.) ;
6) Association des conditions aux attributs du nouveau niveau : l'utilisateur considère les conditions des différentes règles et, pour chacune, détermine la valeur des attributs du nouveau niveau.

Ainsi les étapes 1 et 2, permettent de définir la clause «si» de la méta-règle, les étapes 4 et 5 définissent la clause «alors» de cette dernière. L'étape 3 permet d'instancier la clause «si» de la méta-règle en définissant la clause «si» des différentes règles d'agrégation. L'étape 6 permet d'instancier la clause «alors» de la méta-règle en définissant la clause «alors» des différentes règles d'agrégation.

FIG. 7.2 – Saisie de la méta-règle d'agrégation

FIG. 7.3 – Saisie des règles d'agrégation

Après validation, la table de mapping est générée automatiquement. La vérification des contraintes liées au concept de partition a lieu selon l'algorithme que nous

avons présenté dans la section 5.3. Dans le cas où les règles ne sont pas valides, elles doivent être ressaisies par l'utilisateur.

7.2.3 Analyse

Concernant l'analyse, compte tenu du mode de stockage relationnel utilisé, cette fonctionnalité permet la création conviviale de requêtes SQL. Il s'agit en l'occurrence de choisir sur l'interface la mesure, l'opérateur à appliquer, ainsi que les attributs représentant les axes d'analyse.

La création de la requête est facilitée dans la mesure où, encore une fois, l'exploitation du méta-modèle permet de guider le choix de l'utilisateur sur l'interface (figure 7.4).

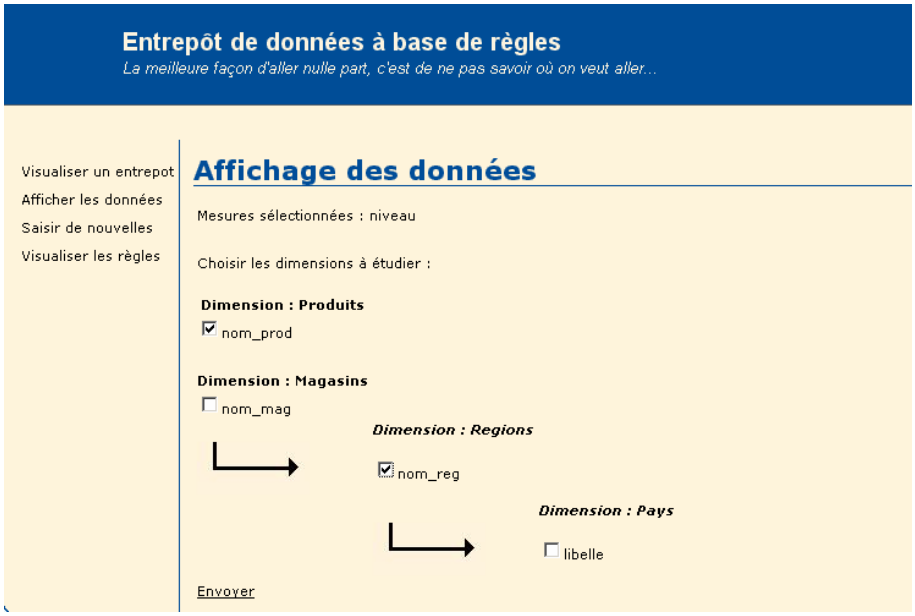

FIG. 7.4 – Choix des paramètres de l'analyse

7.3 Module d'évolution de charge

Pour mettre en œuvre le module dédié à l'évolution de la charge, nous sommes restés dans une configuration client/serveur. Le processus d'évolution se fait au moyen de scripts PHP lancés à partir d'un client. Le fonctionnement général de ce module est décrit dans la figure 7.5.

La charge est stockée, comme l'entrepôt de données, dans le SGBD Oracle 10g.

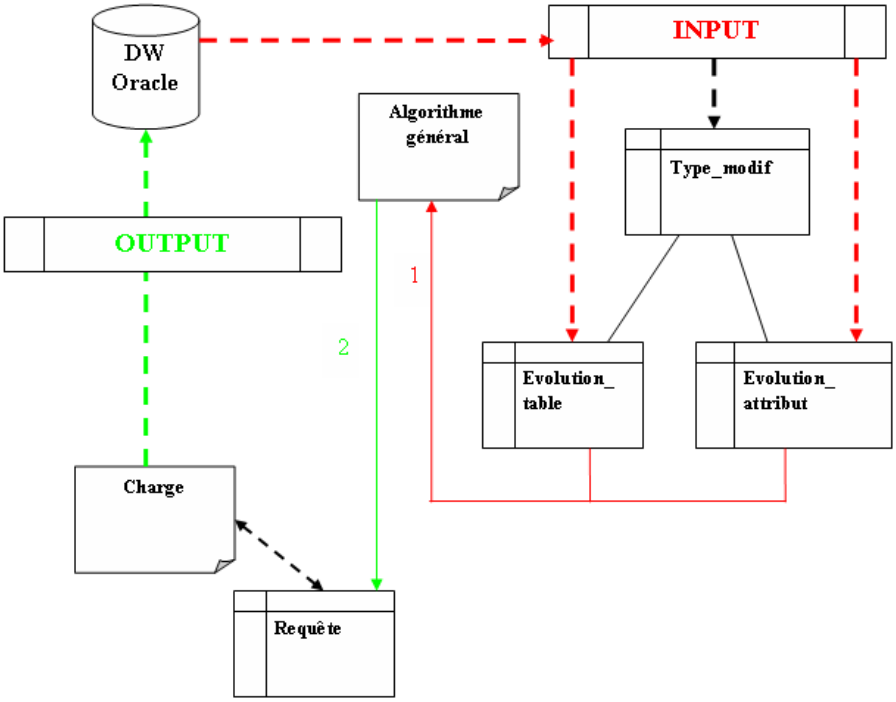

FIG. 7.5 – Fonctionnement du module d'évolution de charge

L'hypothèse de base que nous adoptons est qu'un outil nous fournit l'ensemble des modifications subies par le modèle, que ces modifications soient faites par l'administrateur de l'entrepôt ou qu'elles soient engendrées par WEDriK dans le cadre de la personnalisation des analyses. En effet, ce module a pour but de supporter l'évaluation de performances de modèle évolutif, sans se restreindre au seul cas de l'évolution induite par la personnalisation.

Dans le contexte relationnel, nous supposons que ces changements sont stockés dans deux tables relationnelles : une pour les changements effectués sur les tables, une pour les changements effectués sur les attributs. En effet, en entrée du processus, nous disposons des tables «evolution_table» et «evolution_attribut». Ces tables présentent différentes informations utiles telles que le type de changement opéré, le nom de l'élément qui a subi le changement, etc. Nous émettons l'hypothèse qu'elles sont mises à jour dès lors qu'une modification sur l'entrepôt se produit. En sortie du processus, nous obtenons le fichier SQL «charge».

En effet, les requêtes de la charge sont préparées pour être transformées en une représentation relationnelle dans une table nommée «requete» et ce, afin de faciliter leur traitement. Cette table contient les attributs stockant l'identifiant de la requête, la clause SELECT, la clause FROM, la clause WHERE et la clause GROUP BY.

L'algorithme est ensuite appliqué et, une fois les modifications réalisées, le fichier «charge» est reconstruit à partir de cette table.

Lors de l'exécution de l'algorithme, les tables evolution_table et evolution_attribut sont parcourues, chaque ligne correspondant à une modification du schéma. Pour chaque enregistrement, la valeur de l'attribut nommé «id_type_modif» va permettre d'indiquer le type de la modification. Suivant la valeur de cet attribut, le traitement adéquat est réalisé selon l'algorithme présenté précédemment.

Notons que pour ce traitement, nous exploitons également le méta-modèle de l'entrepôt présenté dans la section précédente.

L'interrogation de celui -ci est nécessaire pour déterminer, par exemple, quelles sont les hiérarchies de dimension, les liens entre les tables, etc. afin d'écrire les nouvelles requêtes.

7.4 Étude de cas : le LCL

7.4.1 Construction de l'entrepôt de données test LCL-DW

Afin d'appliquer nos propositions, nous avons conçu et développé un entrepôt de données test que nous avons appelé LCL-DW. Cet entrepôt porte sur l'analyse du NBI qui, rappelons-le, correspond à ce que rapporte un client à l'établissement bancaire.

Pour construire cet entrepôt, nous avons d'une part recensé les sources de données exploitables et avons recueilli d'autre part les besoins d'analyse.

Nous avons retenu les sources de données suivantes :

1) une source sous un format propriétaire appelée *SIAM* contenant les informations marketing sur les clients dont leur NBI annuel
2) une base de données sous Access appelée *Structures* qui fournit la structure organisationnelle de la direction d'exploitation Rhône-Alpes Auvergne pour laquelle nous disposons de données.

Ainsi, le schéma de LCL-DW que nous avons conçu est présenté dans la figure 7.6. Le NBI est analysé en fonction des dimensions `AGENCY` (agence), `YEAR` (année) et `CUSTOMER` (client). Deux de ces dimensions sont hiérarchisées : `AGENCY` et `CUSTOMER`. La dimension `AGENCY` présente une hiérarchie avec les niveaux `COMMERCIAL_UNIT` (unité commerciale) et `DIRECTION` (direction). La dimension `CUSTOMER` présente deux hiérarchies : la première contient le niveau `POTENTIAL_SCORING` (score potentiel) qui correspond au potentiel du client, la seconde contient le niveau `REAL_SCORING` (score

réel) qui correspond en quelque sorte à la «valeur» réelle du client.

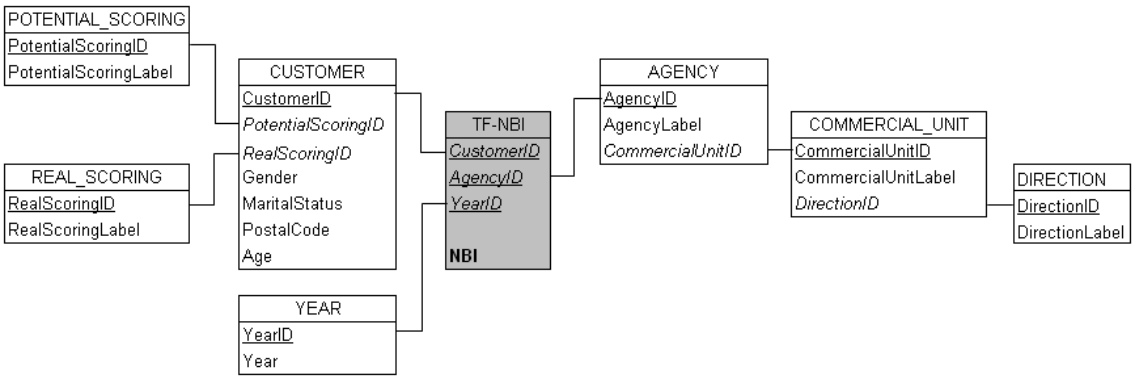

FIG. 7.6 – Schéma de LCL-DW

Nous avons créé la structure de LCL-DW sous Oracle 10g. Nous avons ensuite procédé à la phase ETL en développant nos propres scripts PHP.

Notons que concernant la source SIAM, les données sont uniquement accessibles via un requêteur. Ainsi, nous avons bâti une requête afin d'extraire les données qui nous intéressaient et les avons récupérées dans un fichier texte.

7.4.2 Application de la personnalisation

Nous nous basons sur l'entrepôt LCL-DW.

Supposons qu'un utilisateur veuille analyser le NBI selon le type d'agence ; il sait qu'il en existe trois : type «étudiant» pour les agences ne comportant que des étudiants, type «non résident» lorsque les clients ne résident pas en France et le type «classique» pour les agences ne présentant pas de particularité. Ces informations n'étant pas dans l'entrepôt, il est impossible pour lui d'obtenir cette analyse.

Nous proposons alors à l'utilisateur d'intégrer sa propre connaissance sur les types d'agence afin de mettre à jour la hiérarchie de la dimension agence en ajoutant le niveau `AGENCY_TYPE` au-dessus du niveau `AGENCY`. Afin de réaliser une analyse du NBI en fonction du type d'agence, notre approche est utilisée.

Considérons les extraits de la table de dimension `AGENCY` et de la table de faits `TF-NBI` de la figure 7.7.

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

TF-NBI			
AgencyID	*CustomerID*	*YearID*	*NBI (€)*
01000	1	2006	2000
01903	2	2006	1000
01000	3	2006	4000
03730	4	2006	2000
01929	5	2006	5000
01000	6	2006	2000
01029	7	2006	1000
02600	8	2006	1000
01905	9	2006	2000
01929	10	2006	3000

FIG. 7.7 – Extraits des tables pour l'exemple du NBI

7.4.2.1 Phase d'acquisition.

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 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` ∈ {'01903','01905','02256'} *then* `AgencyTypeLabel`='student'
(R2) *if* `AgencyID` = '01929' *then* `AgencyTypeLabel`='foreigner'
(R3) *if* `AgencyID` ∉ {'01903','01905','02256','01929'} *then* `AgencyTypeLabel`='classical'

En effet, les trois types d'agences sont «student», «foreigner» et «classical». Il exprime les différents types en fonction des identifiants des agences «AgencyID».

7.4.2.2 Phase d'intégration.

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 et les informations

concernant cette table sont insérées dans la méta-table $MAPPING\_META\_TABLE$ (figure 7.8). Ainsi la table de mapping contient les attributs `AgencyID`, `AgencyTypeLabel` respectivement, que l'on retrouve dans la méta-table $MAPPING\_META\_TABLE$ et dont les rôles sont respectivement «*conditioned*» et «*generated descriptor*». 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. 7.8 – 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. 7.9 – Méta-table de mapping pour le niveau `AGENCY_TYPE`

7.4.2.3 Phase d'évolution.

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 7.10).

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

7.4.2.4 Phase d'analyse.

Finalement, la phase d'*analyse* permet d'exploiter le schéma enrichi d'un nouvel axe d'analyse. Classiquement, dans un environnement d'analyse en ligne dans un contexte relationnel, les requêtes décisionnelles consistent en la création d'agrégats en fonction des niveaux de granularité dans les dimensions. En effet, étant donné un schéma, le processus d'analyse permet de résumer les données en exploitant (1) des opérateurs d'agrégation tels que SUM et (2) des clauses de regroupement telles que GROUP BY. Dans notre cas, l'utilisateur souhaitait une analyse sur la somme du NBI en fonction des types d'agence qu'il avait définis. La requête correspondante et le résultat de cette requête sont présentés dans la figure 7.11.

```
SELECT AgencyTypeLabel, SUM(NBI)
FROM TF-NBI, AGENCY, AGENCY_TYPE
WHERE TF-NBI.AgencyID=AGENCY.AgencyID
AND AGENCY.AgencyTypeID=AGENCY_TYPE.AgencyTypeID
GROUP BY AgencyTypeLabel ;
```

AgenyTypeLabel	NBI (€)
student	3000
foreigner	8000
classical	12000

FIG. 7.11 – Analyse du NBI en fonction du type d'agence

7.5 Conclusion

Le développement du prototype WEDriK a permis de mettre en œuvre notre approche de personnalisation des analyses dans les entrepôts de données. En effet, il permet aux utilisateurs de définir de nouveaux niveaux de granularité, avant de les exploiter pour obtenir des analyses personnalisées.

Grâce à la mise en place d'un méta-modèle définissant la structure de l'entrepôt de données évolutif, notre prototype permet non seulement de faciliter la saisie des utilisateurs, mais également de s'adapter à n'importe quel entrepôt. En d'autres termes, WEDriK est une plateforme générique indépendante de l'entrepôt de données qu'elle permet de faire évoluer.

Notons que notre objectif était de valider notre approche d'évolution de schéma par l'exploitation de règles définies par l'utilisateur. L'interface conviviale permet alors une saisie facile des règles, sans avoir connaissance de l'approche théorique qui est derrière (notion de méta-règle, de partie évolutive, etc.). Ceci permet une personnalisation dans la phase d'analyse en ligne. Par ailleurs, le module dédié à l'évolution de charge permet de tenir compte à la fois des évolutions du schéma induites par WEDriK et des évolutions réalisées par l'administrateur.