1. Chapitre II. Base de données objet-relationnelles

Introduction

Si le modèle logique relationnel a prouvé sa puissance et sa fiabilité au cours des 20 dernières années, les nouveaux besoins de l'informatique industrielle ont vu l'émergence de structures de données complexes mal adaptées à une gestion relationnelle.

La naissance du courant **orienté objet** et des langages associées (Java et C++ par exemple) ont donc également investi le champ des SGBD afin de proposer des solutions pour étendre les concepts du relationnel et ainsi mieux répondre aux nouveaux besoins de modélisation.

Les SGBDR

Le modèle relationnel a été définit par E. Codd en1970, il est basé sur la théorie des ensembles et assure l'indépendance entre les programmes d'applications et la représentation interne des données et il fournit une base solide pour traiter les problèmes d'incohérence et de redondance. Le schéma relationnel définit la structure de la relation où les n-uplets représentent les différents éléments. Les relations respectent des propriétés définies à partir des dépendances fonctionnelles (en général, troisième forme normale (3NF)).

Les avantages du modèle relationnel

Les atouts du modèle relationnel peuvent se résumer comme suit :

1. C'est un modèle fondé sur une théorie rigoureuse et des principes simples. Il est aussi fiable et performant car il assure l'indépendance entre les programmes et les données
2. Les SGBDR sont les plus utilisés, connus et maîtrisés.
3. L'utilisation du SQL pour une implémentation standard du modèle relationnel, avec des adaptations pour la plupart des langages de programmation.
4. Les SGBDR incluent des outils performants de gestion de requêtes, de générateurs d'applications, d'administration, d'optimisation, etc.
5. C'est un modèle bien implanté dans le monde professionnel, car il permet une organisation structurée des données et un accès concurrent par des utilisateurs.

Les faiblesses du modèle relationnel :

Parmi les faiblesses de ce modèle, on peut citer:

1. le non support de domaines composés (on ne peut pas avoir pas exemple un attribut qui correspond à une adresse avec le numéro de la rue, le nom de la rue, la ville, … cela est impossible à cause de la première forme normale, qui impose l'atomicité des attributs).

2. La non intégration des opérations (la partie dynamique) car la notion de méthode ne peut être intégrée au modèle logique, elle doit être gérée au niveau de l'implémentation physique.
3. Le mapping MCD vers MLD entraîne une perte de sémantique car la structure de donnée en tables est pauvre d'un point de vue de la modélisation logique,
4. La manipulation de structures relationnelles par des langages objets entraîne une un décalage entre les structures de données pour le stockage et les structures de données pour le traitement.
5. La 1FN est inappropriée à la modélisation d'objets complexes car la normalisation entraîne la genèse de structures de données complexes et très fragmentées, qui peuvent notamment poser des problèmes de performance ou d'évolutivité,

Les SGBD Orienté Objet (SGBDOO)

Un SGBDOO (Orienté Objet) doit d'abord supporter les fonctionnalités d'un SGBD comme la persistance des objets, la concurrence d'accès, la fiabilité des objets et la facilité d'interrogation. Les SGBDOO ont été créés pour gérer des structures de données complexes, en profitant de la puissance de modélisation des modèles objets et de la puissance de stockage des BDs classiques. Ce type de SGBD a pour objectifs:

- Offrir aux langages de programmation orientés objets des modalités de stockage permanent et de partage entre plusieurs utilisateurs,
- Offrir aux BD des types de données complexes et extensibles,
- Permettre la représentation de structures complexes et/ou à taille variable,
- Fournir un ensemble de fonctionnalités OO (Support d'objets atomiques et complexes, Identité d'objets, Héritage simple et polymorphisme)

Les avantages des SGBDOO

Les avantages des SGBDOO peuvent être résumés comme suit:

1) Le schéma d'une BD orienté objet est plus facile à appréhender que celui d'une BD relationnelle (il contient plus de sémantique, il est plus proche des entités réelles),
2) L'héritage permet de mieux structurer le schéma et de factoriser certains éléments de modélisation,
3) La création de ses propres types et l'intégration de méthodes permettent une représentation plus directe du domaine,
4) L'identification des objets permet de supprimer les clés artificielles souvent introduites pour atteindre la 3FN et donc de simplifier le schéma,
5) Les principes d'encapsulation et d'abstraction du modèle orienté objet permettent de mieux séparer les BD de leurs applications (notion d'interface).

Les inconvénients des SGBDOO

Les SGBDOO présentent certains inconvénients comme:

1. La gestion de la persistance et de la coexistence des objets en mémoire (pour leur manipulation applicative) et sur disque (pour leur persistance) complexe,
2. La gestion de la concurrence (transactions) plus difficile à mettre en œuvre,
3. L'interdépendance forte des objets entre eux,
4. La gestion des pannes,
5. La complexité des systèmes (problème de fiabilité),
6. Le problème de compatibilité avec les SGBDR classiques,

Le passage entre SGBDR/SGBDRO/SGBDOO

Pourquoi ne pas passer directement du SGBDR vers SGBD OO ?

Le relationnel a ses avantages, en particulier sa grande facilité et efficacité pour effectuer des recherches complexes dans des grandes bases de données. En outre, ce modèle facilite la spécification des contraintes d'intégrité sans programmation car il est basé sur une théorie solide et des normes reconnues. Pour cette raison, il existe de très nombreuses bases relationnelles en fonctionnement. D'autre part, les SGBDO manquent de normalisation pour et s'appuient sur beaucoup de solutions propriétaires. En outre, les SGBDOO sont moins souple que le relationnel et peu d'informaticiens sont formés dans ce domaine. Pour toutes ces raisons, le modèle objet relationnel (OR) peut permettre un passage en douceur.

SGBDOO ou SGBDRO

On peut constater que les SGBDOO sont plus "propres" du point de vue objet et les mieux adaptés pour traiter les objets mais ils sont complètement absents du monde professionnel. D'autre part, les SGBDRO sont basés sur des SGBD robustes et éprouvés répandus dans le monde professionnel mais qui ne sont pas prévus pour gérer l'objet. Pour cette raison, les concepteurs et les utilisateurs de bases de données ne sont pas prêts à remettre en cause leurs savoirs et à redévelopper toutes leurs applications sur de nouveaux systèmes et donc ils vont privilégier les SGBDRO.

Discussions

Il existe deux manières d'utiliser l'objet dans les SGBD :

(1) soit on part des langages objet dans lesquels on intègre les notions des SGBD (persistance des données, aspect multi-utilisateurs, partage de données,…), ce sont les SGBD orientés objet comme o2 (basé sur c++) ou,

(2) on part des SGBD relationnels dans lesquels on insère des notions objet, ce sont les SGBD relationnels objet : oracle 8 (SQL 3).

Dans ce qui suit, on va s'intéresser de près à la deuxième solution qui s'appuie sur l'intégration des notions objets dans des SGBDR existants pour donner naissance à des SGBDRO.

Les SGBDRO

Les SGBDRO sont nés du double constat de la puissance nouvelle promise par les SGBDOO et de l'insuffisance de leur réalité pour répondre aux exigences de l'industrie des BD classiques. Leur approche est plutôt d'introduire dans les SGBDR, les concepts apportés par les SGBDOO plutôt que de concevoir de nouveaux systèmes. Cette approche intègre des éléments de structure complexe dans une relation de type NF2 (NFNF Non First Normal Form) et pourra contenir un attribut composé d'une liste de valeurs ou de plusieurs attributs.

Dans ce contexte, les SGBDRO peuvent :

(1) Gérer des données complexes (temps, géo-référencement, multimédia, types utilisateurs, etc.),
(2) Rapprocher le modèle logique du modèle conceptuel et
(3) Réduire les pertes de performance liées à la normalisation et aux jointures.

Par rapport au modèle relationnel standard : on a moins besoin d'aplatir les données, ici on a une seule table avec des tables imbriquées au lieu de 3 tables, on a moins besoin de faire des jointures pour récupérer les informations sur les accidents du contrat 1111, on accède simplement à l'attribut accidents.

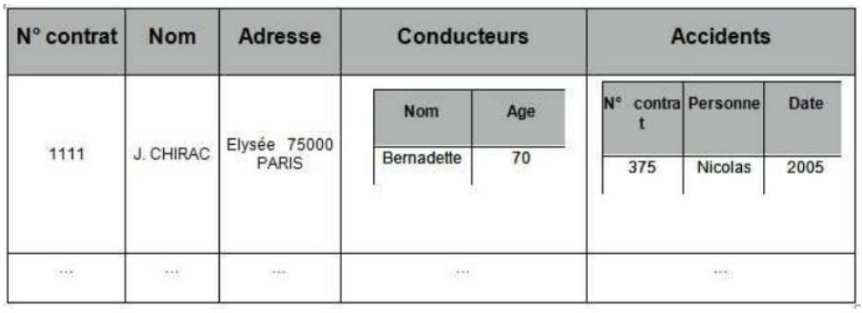

Figure 1: exemple d'une structure issue du modèle objet relationnel

Figure 1: Exemple objet relationnel

Modèles NF2

L'introduction de modèles NF2 (ou non first NF) permet de se libérer de la 1ère Forme Normale afin d'avoir la possibilité de définir des attributs comme des listes de valeurs.

D'autre part, l'extension du LDD pour définir des types structurés engendre la modification des opérateurs de base comme la restriction, la projection et la jointure.

Pour atteindre cet objectif, l'extension des systèmes relationnels nécessitent de modifier le noyau de base du SGBD pour supporter l'intégration des objets complexes et leurs manipulations. Par conséquent, le langage de définition de données (LDD) doit évoluer pour créer des relations définies à partir de ces nouveaux types

D'autre part, le modèle objet-relationnel (OR) ou le modèle relationnel-objet (RO) est un modèle relationnel étendu avec des principes objet pour en augmenter les potentialités (Informix, Oracle, Sybase, IBM/DB2, CA-OpenIngres, PostGres, ...). Il offre de nouvelles possibilités comme :

2. La définition de nouveaux types complexes avec des fonctions pour les manipuler (une colonne peut contenir une collection (ensemble, liste) et une ligne considérée comme un objet avec un objet)
3. Utilisation de références aux objets ou des Identificateur (*Object Identifier* OID)
4. Extensions du langage SQL (SQL3) pour la recherche et la modification des données

Cependant, l'OR ne s'appuie pas sur une théorie solide comme le modèle relationnel et il manque d'implémentation standard de SGBD. Le concept central du RO est celui de type utilisateur, correspondant à la classe en POO, qui permet d'injecter la notion d'objet dans le modèle relationnel. Les apports principaux des types utilisateurs sont :

- La gestion de l'imbrication (données structurées et collections)
- Les méthodes et l'encapsulation
- L'héritage et la réutilisation de définition de types
- L'identité d'objet et les références physiques

Les types personnalisés

Postgre nous permet de créer nos propres types pour se rapprocher des données qui sont manipulé par les langages de programmation orienté objet. La commande CREATE TYPE enregistre un nouveau type de données utilisable dans la base courante. L'utilisateur qui définit un type en devient le propriétaire.

Si un nom de schéma est précisé, le type est créé dans ce schéma. Sinon, il est créé dans le schéma courant. Le nom du type doit être distinct du nom de tout type ou domaine existant dans le même schéma. Les tables possèdent des types de données associés. Il est donc nécessaire que le nom du type soit également distinct du nom de toute table existant dans le même schéma.

Il existe cinq formes de CREATE TYPE, elles créent respectivement :

1. **U**n type composite,
2. **U**n type enum**,**
3. **U**n type range (intervalle),
4. **U**n type de base ou
5. **U**n type shell.

Types composites

La première forme de CREATE TYPE crée un type composite. Le type composite est défini par une liste de noms d'attributs et de types de données. Un collationnement d'attribute peut aussi être spécifié si son type de données est collationnable. Un type composite est essentiellement le même que le type ligne (NDT : row type en anglais) d'une table, mais l'utilisation de CREATE TYPE permet d'éviter la création d'une table réelle quand seule la définition d'un type est voulue.

Pour pouvoir créer un type composite, vous devez avoir le droit `USAGE` sur les types de tous les attributs.

Un *type composite* représente la structure d'une ligne ou d'un enregistrement ; il est en essence une simple liste de noms de champs et de leurs types de données. PostgreSQL™ autorise l'utilisation de types composite identiques de plusieurs façons à l'utilisation des types simples. Par exemple, une colonne d'une table peut être déclarée comme étant de type composite.

Déclaration de types composite

Voici deux exemples simples de définition de types composite :

```
CREATE TYPE complexe AS (
    r       double precision,
    i       double precision
);

CREATE TYPE element_inventaire AS (
    nom             text,
    id_fournisseur  integer,
    prix            numeric
);
```

La syntaxe est comparable à **CREATE TABLE** sauf que seuls les noms de champs et leur types peuvent être spécifiés ; aucune contrainte (telle que `NOT NULL`) ne peut être inclus actuellement. Notez que le mot clé `AS` est essentiel ; sans lui, le système penserait à un autre genre de commande **CREATE TYPE** et vous obtiendriez d'étranges erreurs de syntaxe.

Après avoir défini les types, nous pouvons les utiliser pour créer des tables :

```
CREATE TABLE disponible (
    element   element_inventaire,
    nombre    integer
);

INSERT INTO disponible VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

ou des fonctions :

```
CREATE FUNCTION prix_extension(element_inventaire, integer) RETURNS
numeric
AS 'SELECT $1.prix * $2' LANGUAGE SQL;

SELECT prix_extension(element, 10) FROM disponible;
```

Quand vous créez une table, un type composite est automatiquement créé, avec le même nom que la table, pour représenter le type de ligne de la table. Par exemple, si nous avions dit :

```
CREATE TABLE element_inventaire (
    nom             text,
    id_fournisseur  integer REFERENCES fournisseur,
    prix            numeric CHECK (prix > 0)
);
```

Alors le même type composite `element_inventaire` montré ci-dessus aurait été créé et pourrait être utilisé comme ci-dessus. Néanmoins, notez une restriction importante de l'implémentation actuelle : comme aucune contrainte n'est associée avec un type composite, les contraintes indiquées dans la définition de la table *ne sont pas appliquées* aux valeurs du type composite en dehors de la table. (Un contournement partiel est d'utiliser les types de domaine comme membres de types composites.)

Entrée d'une valeur composite

Pour écrire une valeur composite comme une constante littérale, englobez les valeurs du champ dans des parenthèses et séparez-les par des virgules. Vous pouvez placer des guillemets doubles autour de chaque valeur de champ et vous devez le faire si elle contient des virgules ou des parenthèses (plus de détails ci-dessous). Donc, le format général d'une constante composite est le suivant :

```
'( val1 , val2 , ... )'
```

Voici un exemple :

```
'("fuzzy dice",42,1.99)'
```

qui serait une valeur valide du type `element_inventaire` défini ci-dessus. Pour rendre un champ NULL, n'écrivez aucun caractère dans sa position dans la liste. Par exemple, cette constante spécifie un troisième champ NULL :

```
'("fuzzy dice",42,)'
```

Si vous voulez un champ vide au lieu d'une valeur NULL, saisissez deux guillemets :

```
'("",42,)'
```

Ici, le premier champ est une chaîne vide non NULL alors que le troisième est NULL.

La syntaxe d'expression `ROW` pourrait aussi être utilisée pour construire des valeurs composites. Dans la plupart des cas, ceci est considérablement plus simple à utiliser que la syntaxe de chaîne littérale car vous n'avez pas à vous inquiéter des multiples couches de guillemets. Nous avons déjà utilisé cette méthode ci-dessus :

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Le mot clé ROW est optionnel si vous avez plus d'un champ dans l'expression, donc ceci peut être simplifié avec :

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

Accéder aux types composite

Pour accéder à un champ d'une colonne composite, vous pouvez écrire un point et le nom du champ, un peu comme la sélection d'un champ à partir d'un nom de table. En fait, c'est tellement similaire que vous pouvez souvent utiliser des parenthèses pour éviter une confusion de l'analyseur. Par exemple, vous pouvez essayer de sélectionner des sous-champs à partir de notre exemple de table, `disponible`, avec quelque chose comme :

```
SELECT element.nom FROM disponible WHERE element.prix > 9.99;
```

Ceci ne fonctionnera pas car le nom `element` est pris pour le nom d'une table, et non pas d'une colonne de `disponible`, suivant les règles de la syntaxe SQL. Vous devez l'écrire ainsi :

```
SELECT (element).nom FROM disponible WHERE (element).prix > 9.99;
```

ou si vous avez aussi besoin d'utiliser le nom de la table (par exemple dans une requête multi-table), de cette façon :

```
SELECT (disponible.element).nom FROM disponible WHERE
(disponible.element).prix > 9.99;
```

Types énumération

Les types énumérés (enum) sont des types de données qui comprennent un ensemble statique, prédéfini de valeurs dans un ordre spécifique. Ils sont équivalents aux types enum dans de nombreux langages de programmation. Les jours de la semaine ou un ensemble de valeurs de statut pour un type de données sont de bons exemples de type enum.

Déclaration de types énumérés

Les types enum sont créés en utilisant la commande `CREATE TYPE`. Par exemple :

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Une fois créé, le type enum peut être utilisé dans des définitions de table et de fonction, comme tous les autres types :

```
CREATE TYPE humeur AS ENUM ('triste', 'ok', 'heureux');
CREATE TABLE personne (
```

```
    nom text,
    humeur_actuelle humeur
);
INSERT INTO personne VALUES ('Moe', 'heureux');
SELECT * FROM personne WHERE humeur_actuelle = 'heureux';
 name | humeur_actuelle
------+-----------------
 Moe  | heureux
(1 row)
```

Tri

L'ordre des valeurs dans un type enum correspond à l'ordre dans lequel les valeurs sont créées lors de la déclaration du type. Tous les opérateurs de comparaison et les fonctions d'agrégats relatives peuvent être utilisés avec des types enum. Par exemple :

```
INSERT INTO personne VALUES ('Larry', 'triste');
INSERT INTO personne VALUES ('Curly', 'ok');
SELECT * FROM personne WHERE humeur_actuelle > 'triste';
 nom   | humeur_actuelle
-------+-----------------
 Moe   | heureux
 Curly | ok
(2 rows)

SELECT * FROM personne WHERE humeur_actuelle > 'triste' ORDER BY
humeur_actuelle;
 nom   | humeur_actuelle
-------+--------------
 Curly | ok
 Moe   | heureux
(2 rows)

SELECT nom
FROM personne
WHERE humeur_actuelle = (SELECT MIN(humeur_actuelle) FROM personne);
 nom
-------
 Larry
(1 row)
```

Surêté du type

Chaque type de données énuméré est séparé et ne peut pas être comparé aux autres types énumérés. Par exemple :

```
CREATE TYPE niveau_de_joie AS ENUM ('heureux', 'très heureux',
'eustatique');
CREATE TABLE vacances (
    nombre_de_semaines integer,
    niveau_de_joie niveau_de_joie
);
```

```
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (4,
'heureux');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (6, 'très
heureux');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (8,
'ecstatique');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (2,
'triste');
ERROR:  invalid input value for enum niveau_de_joie: "triste"

SELECT personne.nom, vacances.nombre_de_semaines FROM personne, vacances
  WHERE personne.humeur_actuelle = vacances.niveau_de_joie;
ERROR:  operator does not exist: humeur = niveau_de_joie
```

Si vous avez vraiment besoin de ce type de conversion, vous pouvez soit écrire un opérateur personnalisé soit ajouter des conversions explicites dans votre requête :

```
SELECT personne.nom, vacances.nombre_de_semaines FROM personne,
vacances
  WHERE personne.humeur_actuelle::text = vacances.niveau_de_joie::text;
 nom  | nombre_de_semaines
------+--------------------
 Moe  |          4
(1 row)
```

Détails d'implémentation

Une valeur `enum` occupe quatre octets sur disque. La longueur du label texte d'une valeur enum est limité au paramètre NAMEDATALEN codé en dur dans **PostgreSQL™** ; dans les constructions standards, cela signifie un maximum de 63 octets.

Les labels enum sont sensibles à la casse, donc 'heureux' n'est pas identique à 'HEUREUX'. Les espaces blancs sont significatifs dans les labels.

Les traductions des valeurs enum internes vers des labels texte sont gardées dans le catalogue système pg_enum. Interroger ce catalogue directement peut s'avérer utile.

Exemple

```
postgres=# \c TP2
Vous êtes maintenant connecté à la base de données « TP2 » en
tant qu'utilisateur « postgres ».
TP2=#
```

Il est possible de créer une énumération comme dans les langages de programmation :

```
CREATE TYPE msg A ENUM ('info','alerte','fatal') ;
```

```
CREATE TABLE log(
Id serial not null primary key,
Lib_log text,
Msg_log msg
) ;
Insert into log (Lib_log, msg_log) values ('Erreur de
connexion','fatal')
Insert into log (Lib_log, msg_log) values ('User already
present','alerte')
Insert into log (Lib_log, msg_log) values ('BB maj','info')
```

Ici on voit qu'il y a un ordre dans les énumérations :

```
TP2=# select * from log where msg_log>'info';
 id |       lib_log         | msg_log
----+-----------------------+---------
  2 | User already present | alerte
  3 | Erreur_de_connexion  | fatal
(2 lignes)
```

Remarque: il n'est pas possible de comparer les valeurs de deux énumérations même si le texte est le même :
Exemple :
TP2=# create type type_comment as enum('approuve','alerte','rejete');
CREATE TYPE

```
create table log(
TP2(# id serial not null primary key,
TP2(# lib_log text,
TP2(# msg_log msg,
TP2(# com_log type_comment
TP2(# );

CREATE TABLE
TP2=#    Insert    into    log    (Lib_log,    msg_log,com_log)    values
('Erreur_de_connexion','alerte','alerte');
INSERT 0 1
TP2=# select * from log where msg_log=com_log;
ERREUR:  l'opérateur n'existe pas : msg = type_comment
LIGNE 1 : select * from log where msg_log=com_log;
                  ^
```

ASTUCE : Aucun opérateur ne correspond au nom donné et aux types d'arguments.
Vous devez ajouter des conversions explicites de type.
On ne peut pas comparer deux éléments qui sont différents :
Caster la variable en texte :

```
TP2=# select * from log where msg_log::text = com_log::text;
 id |       lib_log       | msg_log | com_log
----+---------------------+---------+---------
  1 | Erreur_de_connexion | alerte  | alerte
   (1 ligne)
```

Héritage

L'héritage est un concept issu des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes en conception de bases de données.

Soit deux tables : une table `villes` et une table `capitales`. Les capitales étant également des villes, il est intéressant d'avoir la possibilité d'afficher implicitement les capitales lorsque les villes sont listées.

Une meilleure solution peut être :

```
CREATE TABLE villes (
population real,
elevation int -- (en pied)
);

CREATE TABLE capitales (
etat char(2) UNIQUE NOT NULL
) INHERITS (villes);
```

Dans ce cas, une ligne de **capitales** hérite de toutes les colonnes (nom, population et elevation) de son parent, villes. Le type de la colonne nom est `text`, un type natif de PostgreSQL pour les chaînes de caractères à longueur variable. La table `capitales` a une colonne supplémentaire, etat, qui affiche l'abréviation de cet état.

Sous PostgreSQL, une table peut hériter de zéro à plusieurs autres tables. La requête qui suit fournit un exemple d'extraction des noms de toutes les villes, en incluant les capitales des états, situées à une elevation de plus de 500 pieds :

```
SELECT nom, elevation FROM villes
WHERE elevation > 500;
```

ce qui renvoie :

```
nom         | elevation
------------+-----------
Las Vegas   | 2174
Mariposa    | 1953
Madison     | 845
(3 rows)
```

À l'inverse, la requête qui suit récupère toutes les villes qui ne sont pas des capitales et qui sont situées à une élévation d'au moins 500 pieds :

```
SELECT nom, elevation
FROM ONLY villes
WHERE elevation > 500;
nom | elevation
-----------+-----------
Las Vegas | 2174
Mariposa | 1953
(2 rows)
```

Ici, ONLY avant villes indique que la requête ne doit être exécutée que sur la table villes, et non pas sur les tables en dessous de villes dans la hiérarchie des héritages.