Bases de données relationnelles objet dans Oracle 9i

Si le modèle relationnel a fait ses preuves et montré ses avantages, il ne permet pas dans de nombreux cas de représenter les données complexes auxquelles nous faisons face. Typiquement, il est courant d'avoir affaire à des modèles de données qui ne sont pas en première forme normale (Non first Normal Form) pour représenter des listes de valeurs. Les objets complexes sont représentés sur plusieurs tables, ce qui nécessite de nombreuses opérations de jointure (très lentes). De plus, il n'est pas possible d'intégrer des opérations sur les données (définition de méthodes, encapsulation impossibles). Des modèles orientés objet (O2 par exemple) ont été proposés, mais ils ne sont pas compatibles avec les modèles relationnels qui sont utilisés presque toujours.

Les modèles relationnels-objets ont donc été proposés. Ils permettent de tirer partie à la fois du modèle relationnel et des concepts objet.

La norme SQL2 a donc été étendue dans SQL3 pour intégrer les concepts objets :

- objet complexe
- encapsulation
- identifiant d'objet (OID - Object Identifier)
- classe ou type
- attributs complexes et multivalués (collections)
- surcharge et lien dynamique
- complétude et extensibilité
- héritage

N.B. l'objectif est de faire cohabiter une base plate (en 3NF) avec une base objet (non en 1NF) donc la relation reste un concept fondamental. Il n'y a pas de notion de classe. Il faut créer un type abstrait de données (TAD). L'héritage est possible. L'encapsulation est simulée en créant un package PL/SQL.

Le modèle relationnel-objet repose principalement sur la définition de types abstraits de données (utilisant des collections) puis sur l'utilisation de tables objet ou de vues objet.

1 Types abstraits de données (TAD) et tables objets

Il est possible de définir ses propres types (*types abstraits de données* ou TAD) qui seront ensuite réutilisés dans la définition des relations. Ce sont des *types d'objets* car il est possible de leur associer des méthodes, qui sont des procédures ou des fonctions. L'objectif est de considérer un groupe de colonnes comme un tout, comme une seule entité.
N.B. aucune instance n'est associée au TAD, il faut l'associer une relation.

1.1 Composition de types

1.1.1 Déclaration

```
create or replace type TNom as object
(
element1,
element2
)
/
```

où un élément est

- un attribut défini avec son nom et son type :

  ```
  nom_attribut type_attribut
  ```

- ou une méthode :

  ```
  mamethode member ... ,
      static ... ,
  ```

  member s'applique sur l'objet tandis que *static* correspond aux méthodes appelées par la classe (par exemple le constructeur).

Après la création d'un TAD, le '/' remplace le ';' pour terminer l'ordre SQL.

Exemple

```
create type Tadresse as object
(
rue varchar2(50),
ville varchar2(25),
code_postal number
);
/
```

Il est ensuite possible d'utiliser ce TAD :

1. pour le ré-utiliser dans une relation :

   ```
   create table Personne(
   id_pers number,
   nom varchar2(25),
   adresse Tadresse
   );
   ```

   N.B. Personne n'est pas une table d'objets. Il n'y a pas de mécanisme d'OID.

2. pour créer une table d'objets

   ```
   create table Adresse of Tadresse;
   ```

   Adresse est alors une table d'objets avec gestion des OID associés.

3. pour définir un autre TAD complexe :

```
create type Tpersonne as object
(
nom varchar2(25),
adresse Tadresse
);
/

create table Opersonne of Tpersonne ;
```

Opersonne est alors une table d'objets.

Pour connaître la description du TAD, il faudra utiliser les tables systèmes : *user_types (type_name, attributes, methodes, ...)* et *user_type_attrs* pour connaître les attributs d'un TAD.

Si l'on n'est pas propriétaire d'un TAD, il est nécessaire pour l'utiliser d'avoir les droits :

```
grant execute on nom_tad to utilisateur ;
```

Ceci donne également accès aux différentes méthodes du TAD. De la même façon que pour les classes, il est associé au TAD un constructeur de même nom.

N.B. il est obligatoire de valuer tous les attributs (même à Null).

Dans le cas de **TAD inter-dépendants** (par exemple $TVoiture$ qui utilise $TPersonne$ et $TPersonne$ qui utilise $TVoiture$), il est impossible de créer un TAD complet tant que l'autre n'est pas déjà déclaré. Oracle permet alors de définir un TAD incomplet sans attribut ni méthode qui est complété ensuite.

1.1.2 Insertion

1. dans une table incluant un TAD

   On utilise alors un constructeur pour le sous-type construit.

   ```
   insert into personne values (10,
   'Dupont',
   Tadresse('10 rue des champs','Montpellier',34000)
   );
   ```

2. dans une table associée à un TAD

   L'utilisation du constructeur n'est pas obligatoire :

   ```
   insert into adresse values ('18 rue du puits','Montpellier',34000));
   insert into Opersonne values (
   Tpersonne('Dupont',Tadresse('20 cours Gambetta','Montpellier',34000)));
   ```

   ou

   ```
   insert into Opersonne values
   ('Durand',
   Tadresse('30 cours Gambetta','Montpellier',34000));
   ```

   N.B. Dans le dernier exemple ci-dessus, si le constructeur $Tpersonne$ n'est pas obligatoire, **le constructeur** $Tadresse$ **est obligatoire**.

1.1.3 Sélection

```
select * from Opersonne;
```

N.B. la syntaxe suivante est **fausse.**

```
select nom, ville from Opersonne;
```

Pour accéder à la valeur de la ville, il faut indiquer les attributs des TAD avec leur chemin complet avec notation pointée. On utilise alors une *variable de corrélation* pour accéder aux attributs des TAD :

```
select nom, p.adresse.ville from Opersonne p;
select nom from Opersonne p where p.adresse.ville like 'Montpellier' ;
```

1.1.4 Update-delete

```
update Opersonne p
set p.adresse.ville = 'Paris'
where p.adresse.code_postal like '75%';
```

La même syntaxe est valable pour les suppressions

Si une table d'objets utilise un TAD non imbriqué, les ordres update et delete respectent la même syntaxe que pour des tables relationnelles.

N.B. si le TAD n'appartient pas à l'utilisateur, il est nécessaire de spécifier le nom du propriétaire.

```
nom_utilisateur.constructeur
```

1.1.5 Index

Il est possible de créer des index sur les attributs appartenant à un TAD. Il faut toujours spécifier le chemin complet. Il n'y a pas besoin de variable de corrélation.

```
create index ind_ville_Opersonne
on Opersonne(adress.ville);
```

L'index ne concerne que la table spécifiée et non le TAD.

1.2 Collections

Il existe deux moyens de définir des collections : les tableaux Varray (limités en nombre de valeurs), et les tableaux imbriqués (non limités en nombre de valeurs).

1.2.1 tableaux Varray

Un tableau Varray est un tableau de taille limitée à longueur variable. Il est représenté par une seule valeur de type n-uplet dans la table.

```
create or replace type Ttabvar as varray(nb_max_elts) of type_elt;
/
create or replace type Tliste_prenoms as varray(4) of varchar2(15);
/
```

On peut utiliser ces collections pour construire des types complexes.

```
create or replace type Tpersonne as object
(
nom varchar2(15),
prenoms Tliste_prenoms,
date_naissance date,
adresse Tadresse
)
;
/

create table Opersonne of Tpersonne;
```

N.B. Un type ne peut être supprimé ou recréé que si aucune table ne l'utilise. Sinon, il faut utiliser la clause **alter type**. La clause **cascade** permet alors de faire le changement sur tous les types et objets dépendants.

Un tableau de type Varray est stocké comme une unique valeur.

```
insert into Opersonne values
('Durand', Tliste_prenoms('Pierre','Paul','Jacques'),
'18-03-1978',
Tadresse('10 av Palavas','Montpellier',34000))
);
```

Pour accéder au contenu du Varray, on peut :

- utiliser la requête suivante : select * from Opersonne
- utiliser un programme PL/SQL en utilisant une fonction count et la notation tab(i) pour accéder au ième élément du tableau
- utiliser la clause table :

  ```
  select nom, q.*
  from Opersonne p, table (p.prenoms) q;
  ```

1.2.2 tables imbriquées

Une table imbriquée est du type collection et est représentée sous la forme d'une colonne dans la table d'origine.
N.B. une table imbriquée n'est pas une table d'objets. Un seul niveau d'imbrication est possible.

Pour définir une table imbriquée :

1. créer le TAD représentant la structure d'un n-uplet de la table imbriquée,
2. créer un TAD spécifiant la table imbriquée,
3. définir la table utilisant un attribut du type de la table imbriquée.

Exemple

```
create type Tadresse as object
(
rue varchar2(50),
ville varchar2(25),
code_postal number
)
/

create type NT_adresse as table of Tadresse;
/

create table personne
(
nom varchar2(25),
adresses NT_adresse
)
nested table adresses store as NT_adresse_tab ;
```

La clause *nested* crée une table avec une colonne adresses qui est une table imbriquée. NT_adresse_tab a soit été créée avant, soit est créée automatiquement. Les valeurs de cette colonne seront stockées dans la table indiquée dans l'ordre *nested*. Oracle maintient les pointeurs adéquats.

Il est possible d'insérer des n-uplets dans une table imbriquée. Il faut alors utiliser les constructeurs :

```
insert into Personne
values
('Durand',
NT_adresse(
Tadresse('18 rue du puits','Montpellier',34000),
Tadresse('20 av des Champs Elysees','Paris',75000)));
```

Il est également possible d'insérer de nouvelles valeurs dans la table imbriquée grâce au mot-clé **the** :

```
insert into the (select adresses from personne where nom='Durand')
values
Tadresse('place du capitole','Toulouse',31000));
```

Pour effectuer des insertions groupées (par exemple deux personnes ayant les mêmes adresses), on utilise les mots clés **cast** qui indique qu'il s'agit d'une table imbriquée et **multiset** qui traite le cas de plusieurs valeurs.

```
insert into personne value
('Dupont2',
cast(
multiset(select * from the (select adresses from personne where nom='Dupont'))
));
```

On peut également utiliser **update the** pour modifier les n-uplets de la table imbriquée et **delete the**.

```
update the
(select adresses from personne where nom = 'Dupont') nttab
set nttab.ville='Lyon'
where nttab.ville='Toulouse';
```

La suppression d'une collection complète est possible :

```
update personne set adresses = null where nom = 'Dupont';
```

Il est possible de créer des collections **multi-niveau** (*nestedtable* de *nestedtable*, *nestedtable* de *varray*, *varray* de *nestedtable*, *varray* de *varray*, *nestedtable* ou *varray* dont le type est un TAD possédant un attribut qui est un *varray* ou une *nestedtable*).

L'interrogation d'une table imbriquée est possible :

- en utilisant le mot-clé **the** : il faut alors accéder à la colonne correspondant à la sous-table, utiliser le connecteur *the* avec alias, et exprimer la requête sur la sous-table.

  ```
  select nt.ville
  from the (select adresses from personne where nom = 'Durand') nt
  where nt.rue like '%Champs Elysees';
  ```

- en utilisant le mot-clé **table** :

  ```
  select nt.ville
  from personne p, table (p.adresses) nt
  where nom='Durand' and nt.rue like '%Champs Elysees';
  ```

Le tableau ci-dessous résume les différences principales entre nested table et varray.

	Varray	Nested Table
éléments ordonnés	oui	non
stockage	1 valeur	1 table
requêtes	table	the - table
mise à jour	difficile utilisation de PL/SQL	update the

2 Méthodes

Les méthodes sont déclarées lors de la création des TAD.

```
create [or replace] type nom_type as object
(
att1 type1,
...
attn typen,
MEMBER function|procedure signature_methode_1,
STATIC function|procedure signature_methode_2
);
/
```

MEMBER renvoie aux méthodes invoquées sur les instances du TAD.
STATIC renvoie aux méthodes invoquées sur le TAD.
signature_methode suit la même syntaxe que les procédures ou fonctions stockées.
Exemple :

```
create or replace type Tpersonne as object
(
nom varchar2(25),
prenoms Tliste_prenoms,
date_naiss date,
MEMBER function age(date_naiss date) return numeric
);
/
```

Le corps des méthodes est défini par :

```
create or replace type body nom_type
as
Member nom_methode is
begin
...
end ;
...
end;
```

Il est possible de surcharger des noms de méthode. On déclare alors deux méthodes avec le même nom et des paramètres différents et Oracle choisit la méthode à exécuter.

Il est possible d'utiliser le paramètre **self** dans le corps de la méthode.

L'**appel des méthodes** s'effectue de la même manière que pour les fonctions et procédures stockées en indiquant le chemin avec la notation pointée comme pour les attributs :

```
select p.nom, p.age(p.date_naiss)
from Opersonne p;
```

Les méthodes peuvent également être utilisées dans un programme PL/SQL, java ...

Il est parfois nécessaire d'indiquer la portée des méthodes pour éviter les mécanismes de verrouillage.

```
pragma restrict_references(nom_methode,
WNDS //write no database state
ou    RNDS //read no database state);
```

3 Identification d'objet OID

Dans une table d'objet, chaque n-uplet possède un identifiant d'objet (OID ou Object Identifier).
N.B. L'OID est affecté par le système, il permet :

- d'identifier de manière unique ce n-uplet,
- de référencer cet objet de manière explicite.

Une table d'objet est créée comme étant associée à un TAD.

```
create table Opersonne of Tpersonne;
create table Oadresse of Tadresse;
```

3.1 la fonction ref

La fonction **ref** permet d'accéder à l'OID des objets concernés.

```
select ref(p)
from Opersonne p
where p.nom='Dupont';
```

N.B. La fonction ref ne s'applique qu'aux objets et non aux colonnes associées à un TAD.

3.2 référencement explicite

On spécifie alors pour le type de l'attribut le fait qu'il pointe vers un objet dont la structure est le TAD spécifié. dans une table d'objets (figure 1).

```
nom_att REF type_objet
```

Exemple

```
create type Tvoiture as object
(
modele varchar2(15),
immat varchar2(10)
)
/

create table Ovoiture of Tvoiture;

create or replace type Tpersonne as object
(
nom varchar2(15),
prenoms Tliste_prenoms,
voiture REF Tvoiture
);
/

create table Opersonne of Tpersonne;
```

La table système *user_dependencies* stocke les dépendances entre types via les REF.
L'insertion s'effectue en utilisant la fonction REF qui retourne l'OID d'un n-uplet spécifié :

```
insert into Opersonne values
(
'Dupont', Tliste_prenoms('Pierre','Paul'),
(select ref(v) from Ovoitures V where V.immat='4444 XX 34' )
);
```

Le problème est qu'il faut connaître une valeur pour accéder à l'objet. L'attribut de type ref est stocké comme un pointeur dans la table.

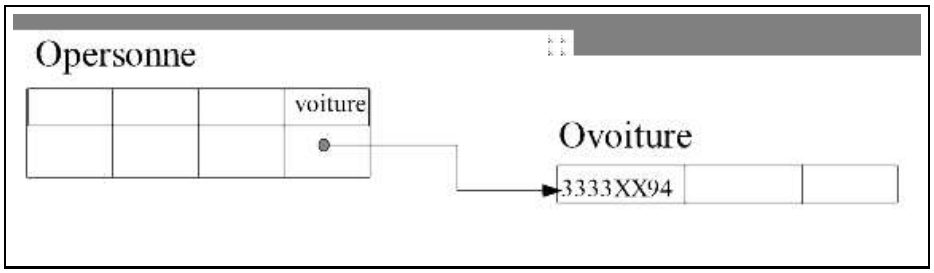

Figure 1: Exemple d'utilisation de REF

3.3 la fonction deref

La fonction *deref* est l'inverse de *ref* et sert, à partir d'un OID, à accéder à l'objet associé.

Exemple :

```
select deref(p.voiture)
from Opersonne p
where p.voiture.modele like 'megane%' ;
```

3.4 références invalides

Lorsque l'objet référencé est supprimé, la référence devient invalide. On parle alors de *dangling ref*. Le prédicat *IS DANGLING* permet de savoir si une *ref* est valide ou non.

Exemple :

```
update Opersonne set voiture=Null where voiture is dangling ;
```

3.5 la fonction value

La fonction *value* s'utilise sur une table d'objets.

4 Vues objet

L'objectif des vues objets est d'implémenter une structure orientée objet sur des tables relationnelles *plates*. Ce principe est très intéressant puisqu'il permet de faire cohabiter une ancienne base relationnelle avec une nouvelle base objet, ou bien de conserver les acquis du relationnel tout en montrant une base objet à l'utilisateur.

Par exemple, on considère la vue objet VOPersonne permettant de recueillir les informations sur une personne :

VOPersonne	Num_Secu	Nom	Prenom	Adresses
	1700347234283	Dupont	Jean	'10 av Foch' '20 av Palavas'

Cette vue objet s'appuiera sur une base relationnelle classique en 3NF :

Personne	Num_Secu	Nom	Prenom
	1700347234283	Dupont	Jean

Habite	Num_Secu	Adresse
	1700347234283	'10 av Foch' '20 av Palavas'

Le mécanisme d'OID se base sur la clé primaire.

Pour créer une telle vue objet, on procédera selon les étapes suivantes :

1. Crération des tables relationnelles.

```
create table Personne
(num number constraint cp_personne primary key,
nom varchar2(25),
prenom varchar2(25)) ;

create table Habite (num ...);
```

2. Crération des types de données abstraits.

```
create type Tadresse as object
(
rue varchar2(50),
ville varchar2(25),
code_postal number
)
/

create type TListe_adresses as Varray(10) of Tadresse;
/

create type TPersonne as object(
num number,
nom varchar2(25),
prenom varchar2(25),
adresses Tliste_adresses
)
/
```

3. Crération de la vue objet.

```
create view VOPersonne of TPersonne
with object oid(num)
as
select num, nom, prenom, cast(
multiset(
select Tadresse(rue, ville, code_postal)
from habite h
where p.num = h.num
)
as TListe_adresses)
from personne p ;
```

   La clause *objectoid* indique à Oracle sur quelle colonne s'appuyer pour construire les OID.

On transforme les clés étrangères en objets de type *REF* en utilisant la fonction **make_ref**.

```
create view VAdresse as
select make_ref(VOPersonne,num) num,rue ... from Habite ;
```

VAdresse est une vue contenant un attribut de type *REF*.
La sélection s'effectue en utilisant la fonction **DEREF**.

```
select deref(VA.num)
from VAdresse VA
where rue like '10%' ;
```

On accède à l'objet de VOPersonne qui est pointé.

Les insertions, suppressions et mises à jour se font via :

- les tables relationnelles,
- ou les vues objet en utilisant les constructeurs.

Il est possible de définir des **triggers sur les vues objet**. Ceci permet de mettre à jour les tables concernées à travers la vue grâce aux triggers de type **instead of**.

```
create trigger nom_trigger
instead of type_operation on nom_vue_objet
[for each row]
begin
...
end;
```

Dans le corps du trigger, on peut bien sûr utiliser : *new* et : *old*. Le type d'opération sur la vue est substitué par l'action déclarée dans le corps du trigger.

Exemple :

```
create trigger ins_vopers_trigger
instead of insert on VOPersonne
for each row
declare i integer;
begin
insert into Personne values(:new.num, :new.nom, :new.prenom);
if :new.adresses is not null and :new.adresses.count>0
then
for i in :new.adresses.first .. :new.adresses.last
loop
insert into habite values
(:new.num,:new.adresses(i).rue,
 :new.adresses(i).ville,
 :new.adresses(i).code_postal);
end loop;
end if;
end;
```

Ce trigger sera appelé après les requêtes du type :

```
insert into VOPersonne
values (123,'Dupont','Paul',
adresses(Tadresse('place E. Bataillon','Montpellier',34000),
Tadresse('30 rue Vaugirard','Paris',75015));
```

Il existe un ensemble de **méthodes prédéfinies** valables sur les types collection (tables imbriquées et Varray).
N.B. Ces fonctions sont à utiliser dans des programmes PL/SQL et non dans des requêtes.

```
exist(i) vrai si le ieme element existe
count nombre d elements de la collection
limit nombre maxi d elements d un Varray
first, last indice du 1er et dernier element de la collection
prior(i), next(i) indice de l'element avant le ieme ou apres
trim(i) supprime le ieme element a partir de la fin
extend[(i)] ajoute un (ou i) element(s) NULL
extend(i,j) ajoute i copies du jeme element
delete supprime toute la collection
delete(i,[j]) supprime le ieme element ou du ieme au jeme
```

5 Héritage

L'héritage n'était pas supporté avant la version 9 d'Oracle. Il permet de spécialiser un type via la clause **NOT FINAL** :

```
create type nom_type as object
(...)
not final
/
```

ou

```
alter type nom_type not final ;
```

Il est alors possible de créer des sous-types via la clause **UNDER** :

```
create type sous_type UNDER super_type
( attribut_specialise_1 type_att1,
...
attribut_specialise_n type_attn,
)
/
```

Il y a héritage des attributs et des méthodes. La surcharge des méthodes est possible. Dans le cas de la généralisation, il est possible de déclarer un type comme **NOT INSTANTIABLE**.

```
create type super_type as object(...) no instantiable not final
/
```

La table créée au niveau du super-type stocke des sous-types et des super-types. Par exemple, si on considère le type Personne et le sous-type Etudiant, on peut insérer des étudiants dans Personne.

Pour accéder à certains n-uplets d'un type spécifié, on utilise **TREAT** :

```
select TREAT(value(p) as TEtudiant).num from OPersonne p ;
```

L'héritage est également possible sur les vues objet grâce à la clause **UNDER nom_vue**.