Bases de données avancées

Hala Skaf-Molli
Maître de Conférence
Nancy-Université skaf@loria.fr

14 mai 2007

1 Introduction

Dans ce cours, nous allons étudier *les bases de données Orienté objets*, *bases de données Objet-relationnel* et *la correspondance Objet-relationnel*.

1.1 Bases de données orienté objets

Les bases de données OO permet de rendre persistant un objet (un type complexe). Rendre persistant signifie sauvegarder les objets sur un support non volatile de telle sorte qu'un objet identique à cet objet pourra être recréé lors d'une session ultérieure.

Le modèle de données ODL permet d'assurer la persistance des objets.

1. **Modèle de données ODL** (Object Definition Language) : c'est un standard pour les bases de données OO développé par OMG. Il s'agit d'une extension de IDL (Interface definition Language). IDL est un composant de CORBA (Common Object Request Broker Architecture) dont l'objectif est de faire la programmation OO distribuée.
2. **Langage de requêtes OQL** (Object Query Language) : c'est un langage de requêtes standard pour les bases de données orientés objets. Il est SQL-Like Syntaxe. L'idée est de ramener le meilleur de SQL dans le monde Orienté Objets.

1.2 Les bases de données Objet-Relationnel

L'objectif est de faire une extension objets au langage SQL. Cette extension est nommée SQL-99 (SQL3)

1. le modèle de données est le modèle relationnel avec de des possibilités de définir de types complexes.
2. le langage de requête est SQL99 (SQL-like Syntaxe).

1.3 Correspondance Objet-Relationnel (mapping OR)

L'objectif est d'utiliser le modèle relationnel pour faire persister le modèle objets. Le principe est de faire correspondre le modèle de données Orienté Objets (de type JAVA) au modèle de données relationnel. Ceci sans changer ces deux modèles de données . C'est l'approche utilisé par Hibernate `www.hibernate.org` et par TopLink (un produit Open Source développé par Oracle). TopLink permet de faire la correspondance de l'objet vers relationnel et il permet aussi de faire la correspondance de l'objet vers Objet-Relationnel). Hibernate et toplink implémentent le standard JPA (JAVA Persistence API).

2 Bases de données Orienté objets

Pourquoi ? afin de modéliser des objets complexes, et profiter de toutes les avantages des langages OO (la réutilisation, héritage, encapsulation, polymorphisme,...)

Comment ? Définir un nouveau modèle de données. Ce modèle s'appelle ODL.

2.1 Types

Les types dans ODL sont :

1. types atomiques : integer, string, float,
2. le nom d'une classe
3. constructeur de type : Set, List, Bag(Sac), Array, Dictionnary
4. Si T est un type alors Set$< T >$, Bag$< T >$, List$< T >$, Array$< T, i >$, Dictionnay$< T, S >$ sont de types.

Set, list, bag, array, dictionnary sont de type collection.

Exemple 2.1. *Bag<integer> est un type, {1,2,1} est un bag. Attention, un bag contient une duplication (ce n'est pas un ensemble).*

Le type Array$< chat, 10 >$ est un String de longueur 10.

ODL est basé sur la notion de classes. Une classe possède trois propriétés : attributs, associations et méthodes.

2.2 Les classes

2.2.1 Les attributs

Exemple 2.2. *Une classe avec des attributs des types simples*

L'exemple suivant montre la déclaration de la classe Film. Cette classe a des attributs de types simples (string, integer, ...)

```
Class Film {
Attribute string titre ;
attribute integer annee;
attribute integer longueur;
attribute enum Movie {couleur, BlancetNoir} filmType ;
}
```

La classe Film a 4 attributs (titre, annee, longueur et filmType) de types simples.

Un objet (une instance) de la classe Film est défini par : "Gone with Wind",1939,231,color).

Chaque objet a un OID (identificateur). L'OID est géré par le système. Comme dans les langages de programmation orienté objets, l'utilisateur ne peut pas accéder à l'OID..

Exemple 2.3. *Une classe avec des attributs de types complexes*

```
Class Acteur {
  attribute String nom;
attribute Struct addr {String rue, string ville} adresse;
};
```

L'attribut adresse est de type complexe, il s'agit d'une 'structure'.

2.2.2 Les associations

Dans cette section, nous montrons les différents types d'associations possibles entre les classes ODL.

Exemple 2.4. *Les associations binaires dans ODL*

Question : comment je peux attacher à chaque film ses acteurs ? Dans Film, j'ajoute :

```
Class Film {

Attribute String Titre ;
attribute integer annee;
attribute integer longueur;
attribute enum Movie {couleur, BlancetNoir} filmType ;
relationship Set <Acteur> Stars;
}
```

Chaque objet de la classe Film référence un ensemble d'objets de la classe Acteur.

Pour chaque association, il est obligatoire de définir l'association inverse.

Exemple 2.5. *L'association inverse dans la classe Acteur :*

```
Class Acteur {
attribute String nom;
attribute Struct addr {String rue, string ville} Adresse;
relationship set<Film> StarredIn inverse Film::Stars;
};
```

De manière générale, pour chaque association R entre deux classe C et D, R(C,D), on a :

C	D
x1	y1
x2 y2	

et la relation inverse :

D	C
y1	x1
y2	x2

Pour les associations de type 3-ways, vous devez regarder vos notes de cours.

2.2.3 Cardinalité des associations

Comme dans le modèle E/A, les associations ont quatre cardinalités possibles :

1. many-many :
2. many-one
3. one-many
4. one-one

(Exemple, voir vos notes de cours)

2.2.4 Les méthodes

Chaque méthode a une définition et une implantation. Elles sont déclarées lors de la définition de la classe.

Exemple 2.6. *Définition d'une méthode :*

```
Class Film {
attribute atring titre ;
attribute integer annee;
attribute integer longueur;
attribute enum Movie {couleur, BlancetNoir} filmType ;
relationship Set <Acteur> Stars;
float starNames(out Set<String>);
float lengthInHours() raises(noLengthFound);
}
```

Dans la définition de la classe Film, on a ajouté la signature des méthodes (starNames et lengthInHours).

2.3 Héritage

Le mot clé *extends* permet de définir les sous-classes.

Exemple 2.7. *Héritage :*

```
class Animation extends Film{
relationship set <Acteur> voix;
}
```

La classe Animation est une sous-classe de la classe Film.

2.4 Extent

Le mot clé **extent** permet de :

1. faire la distinction entre une classe et ses instances (l'ensemble d'objets de cette classe). Comme dans le modèle relationnel, on fait la différence entre une relation et ses instances (ses tuples).
2. désigner les objets persistants (stockés dans la base) de la classe. En fait, lorsque on fait la création d'un objet, cet objet n'est pas forcement persistant, il devient persistant s'il est attaché à l'extent de sa classe.

Exemple 2.8. *Persistance par l'extent :*

```
Class Film (extent Films) {
attribute ..
}
```

Films désigne tous les objets persistants de la classe Film. Films est utilisé par la suite dans le langage de requêtes OQL pour accéder aux objets de la base.

2.5 Le langage de requêtes Object Query Language OQL

OQL est un langage de requêtes standard pour les bases de données OO. Le modèle de données utilisé est ODL. OQL a les mêmes notations que SQL (SQL-Like requêtes). Les types en OQL sont ceux de l'ODL.

Dans OQL, on peut utiliser les type complexes *Set(Struct), Bag(Struct)* dans la partie *from* d'une requête. Ceci signifie que ces types peuvent jouer le rôle d'une relation (l'extent d'une classe).

Il est possible de mixer de requêtes OQL avec un langage hôte (c++, smalltalk, java) sans avoir les problème de conversion de type et les problèmes de paradigmes différents comme c'est le cas avec le JDBC.

2.5.1 Expressions de chemins

On utilise '.' pour accéder à une propriété d'une classe. Soit x un objet de la classe C :

1. si a est un attribut de C, alors x.a est la valeur de cet attribut.
2. si r est une association de C alors, x.r est la valeur de l'association :
   (a) soit c'est un objet ou un ensemble d'objets selon le type de r.
3. si m est une méthode de C, alors x.m(..) est le résultat de l'appel de la méthode m à x.

Exemple 2.9. *Dans les exemples suivants, nous allons utiliser la base de données Orienté Objets ci-dessous.*

```
Class Film
 (extent Films key (titre, année))
{
Attribute String titre ;
attribute integer année;
attribute integer longueur;
attribute enum Movie {couleur, BlancetNoir} filmType ;

relationship Set <Acteur> stars inverse Acteur::filmographie;
relationship Studio producteur inverse Studio::produit;

float starNames(out Set<String>);
float lengthInHours() raises(noLengthFound);
}

Class Acteur
 (extent Acteurs key nom)
{
attribute String nom;
attribute Struct addr {String rue, string ville} adresse;

relationship set<Film> filmographie inverse Film::stars;
};

Class Studio
  (extent Studios key nom)
{
attribute string nom;
attribute string Adresse;

relationship set<Film> produit inverse Film::producteur;
}
```

L'expression de chemins :

Soit monFilm un objet de la classe Film :

1. monFilm.titre est le titre l'objet monFilm.
2. monFilm.producteur.nom est le nom du studio qui a produit monFilm.
3. monFilm.stars.nom : pas légal , stars est un ensemble. C'est possible si stars est un objet.

2.5.2 OQL Select-From-Where

La syntaxe d'une requête OQL est :

```
Select  <liste de valeurs>   /* le résultat de la requête */

from <liste de collections et les noms de membres>
/* l'extent d'une classe (ex. Films) où bien une  expression dont
l'évaluation donne une collection, (ex. monFilm.stars) */

where <condition>
```

Nous allons montrer les différents types de requêtes OQL à travers des exemples.

Exemple 2.10. *Afficher l'année de sortie du film 'Gone with the Wind'.*

```
select f.annee /* le résultat de la requête */

from Films f /* Une collection, c'est l'extent de la classe Film, f c'est un alias */

where f.titre="Gone With the Wind"; /* la condition, mêmes conditions
que SQL */
```

Attention, OQL utilise le double-quotes.
Pour évaluer cette requête, on fait une itération sur la collection Films :

```
pour chaque f dans Films faire
 if f.titre="Gone With the Wind" alors
   ajouter f.année au résultat
```

Exemple 2.11. *Autre syntaxe :*

```
select f.annee
from f in Films
where f.titre="Gone With the Wind";
```

Le type de résultat

Par défaut, le type du résultat d'une requête est un bag (ensemble avec duplication).

Le type de résultat de la requête précédente est : bag<int> ;

Exemple 2.12. *Afficher les noms d'acteurs ayant joués dans le film "Gone With the Wind".*

```
select a.nom
from f in Films , a in f.stars
where f.titre="Gone With the Wind";

Type du résultat : bag<String> ;
```

Exemple 2.13. *Autre syntaxe :*

```
select a.nom
from Films  f,   f.stars  a
where f.titre="Gone With the Wind";
```

Exemple 2.14. *Trouver les noms d'acteurs ayant joués dans un film produit par "Disney" ?*

```
select a.nom
from Films  f,   f.stars  a
where f.producter.nom="Disney";

type de résultat : bag<String> ;
\end{verabtim}

\subsubsection*{Renommer le résultat}

Pour renommer le résultat d'une requête OQL, il suffit de mettre le
nouveau nom suivi par ':' avant le champ du résultat:

\begin{Exemple}

Renommage:
\end{Exemple}

\begin{verbatim}
select  Star: a.nom
from Films  f,   f.stars  a
where f.producter.nom="Disney";

type du résultat : bag<String> ,
nom du résultat est : Star.
```

Changement de type de résultat

En ajoutant les mot clés DISTINCT ou UNIQUE, on peut changer le type du résultat. Le type devient un ensemble.

Exemple 2.15. *Le type est un ensemble :*

```
select  DISTINCT a.nom
from Films  f,   f.stars  a
where f.producter.nom="Disney";

Type de résultat: set<String>
```

En ajoutant ORDER BY, on obtient une liste.

Exemple 2.16. *Le type est une liste :*

```
select  f
from Films  f
where f.producter.nom="Disney";
order by f.longueur, f.titre ;

Type de résultat: list<Film>
```

Requêtes avec un résultat de type complexe

Exemple 2.17. *Trouver les couples d'acteurs qui habitent à la même adresse.*

```
select DISTINCT Struct(acteur1 : a1, acrteur2 : a2)
from Acteurs a1, Acteurs a2
where a1.adresse =a2.adresse and a1.nom < a2.nom

Type de résultat : set<Struct{acteur1 : Acteur, acteur2 : Acteur}>
```

Sous Requêtes

Une expression select-from-where peut être utilisée comme une sous requêtes :

1. dans from (comme une collection)
2. dans les expressions exists et FOR ALL.

Exemple 2.18. *Trouver les noms d'acteurs ayant joués dans un film produit par "Disney" ?*

```
select  DISTINCT a.nom
from (select f
      from  Films  f
      where f.producter.nom="Disney") d  , d.stars a
```

Quantificateurs

Il existe deux expressions booléennes utilisable dans la partie *where* :

1. FOR ALL x in <collection> : <condition>. Cette expression est évaluée à vraie si tous les membre de la collection vérifient la condition.
2. Exists x IN <collection> : <condition>. Cette expression est évaluée à vraie s'il existe un membre de la collection qui vérifie la condition.

Exemple 2.19. *Trouver les noms d'acteurs ayant joués dans un film produit par "Disney" ?*

```
select a
from  Acteurs a
where exists f in a.filmographie: f.producteur.nom="Disney"
```

Exemple 2.20. *Trouver les acteur qui ayant joués seulement dans les films produit par "Disney ".*

```
select a
from  Acteurs a
where for all f in a.filmographie : f.producteur.nom="Disney"
```

Fonctions d'agrégations

Les fonctions d'agrégations AVG, SUM, MIN, MAX et count sont appliquées aux collections.

Exemple 2.21. *La fonction AVG*

```
AVG(select f.longueur
   from Film f)
```

Exemple 2.22. *Donner le nom d'acteurs et le nombre de films joués par ces acteurs.*

```
select  struct (name: a.nom, filmcount :count(a.filmographie))
where Acteurs a
order by name

Type de résultat : list<struct{name :String, filmcount :integer}>
```

La fonction Element

Si le bag contient un seul élément, alors on peut utiliser l'opérateur ELEMENT.

```
ELEMENT(bag(x))= x
```

2.6 Conclusion

Voir vos notes de cours.

Les bases de données orienté objets sont toujours en vie (still alive). Pour plus d'information vous pouvez consulter l'adresse :

`http://www.dzone.com/rsslinks/seaside_gemstone_as_new_web_platform_glass.html`

3 Modèle Objet-relationnel

Le modèle Objet-Relationnel (OR) reprend le modèle relationnel en ajoutant quelques notions qui comblent les plus grosses lacunes du modèle relationnel. Le modèle de données est les relations et on ajoute des possibilités de définir de type complexe avec de méthodes.

Le principe de base est de permettre à l'utilisateur de définir ses propres types UDT (User-Defined-types). On peut dire que les classes des bases de données OO sont transformées à de types définis par l'utilisateur..

La norme SQL-99 (SQL3) reprend beaucoup d'idées du modèle OR. Dans ce cours, on prend comme exemple l'extension objets dans Oracle.

3.1 Pourquoi étendre le modèle relationnel ?

1. L'impossibilité de créer de nouveaux types implique un manque de souplesse et une interface difficile avec les applications orientées objet
2. L'OR permet de définir de nouveaux types utilisateur simples ou complexes (UDT) avec des fonctions ou procédures associées comme dans les classes des langages objet
3. L'OR supporte l'héritage de type pour profiter du polymorphisme et faciliter la réutilisation

En bref, ajouter des possibilités objets au modèle relationnel.

3.2 Les nouvelles possibilités de l'OR

1. Le modèle OR permet de définir de nouveaux types complexes avec des fonctions pour les manipuler. Ainsi, la colonne d'une table peut contenir une collection (ensemble, sac, liste).
2. Le ligne d'une table peut être considérée comme un objet, avec un identificateur (Object Identifier OID). L'utilisateur peut manipuler l'OID (pas comme dans les bases de données OO).
3. L'utilisation de références aux objets permet d'accéder rapidement aux objets référencés.
4. L'extension du langage SQL (SQL3 ou SQL99) pour la recherche et la modification des données.

3.3 Définition de types

1. Nouveaux types prédéfinis : l'OR ajoute des types prédéfinis à la norme SQL : référence et collection (voir plus loin).
2. Type défini par l'utilisateur (UDT) : on distingue deux types possibles types distincts et types structurés.

Types distincts

Ces types permettent de différencier les domaines de colonnes. Ils sont formés à partir de types de base.

Exemple 3.1. *Types distincts*

```
create type poids as integer ;
create type age  as integer;

create table Personne (
personneID varchar(20) primary key,
personneAge  age,
personnePoids poids);
```

Ces types s'utilisent exactement avec les mêmes instructions que le type de base sous-jacent.

Je n'ai pu tester dans Oracle 10g.

Types structurés

Ces types correspondent aux classes du langage ODL.
Chaque UDT a :

1. un constructeur de même nom que le type,
2. attributs (variables d'instances),
3. fonctions et procédures (méthodes). Elles peuvent être écrites en SQL ou en un autre langage

3.4 Création d'un type de données

Dans SQL-99 :
Create type T as ¡attributes and methode declarations¿ ;

Exemple 3.2. *Définition d'un UDT personneT :*

```
create type personneT as (
            nom varchar(20),
    rue varchar(30),
            ville varchar(20)
) ;
```

La syntaxe ressemble à l'ODL sauf que :

1. la clé est définie dans la table et pas dans le type (voir section 3.8). Ceci implique que plusieurs tables peuvent avoir les mêmes types avec de clés différentes.
2. les associations ne sont pas de propriétés de l'UDT. Les associations sont présentées par de tables séparées.

Un UDT peut-être le type :

1. d'un attribut dans la déclaration un autre UDT,
2. d'une table i.e. le type de ses tuples,
3. d'un attribut (colonne) dans une table.

Dans Oracle :

```
CREATE OR REPLACE TYPE <type_name>
AS OBJECT (
<attribute> <attribute data_type>,
<subprogram spec>)
<FINAL | NOT FINAL> <INSTANTIABLE | NOT INSTANTIABLE>;
/
```

Exemple 3.3. *Définition de UDT dans Oracle :*

```
create type departementT as Object (
            numID integer
    nomD  varchar(30),
            lieu  varchar(20)) ;
```

Remarques :

1. Dans SQL-99 pas OBJECT après le AS.
2. Un type ne contient pas de contrainte d'intégrité
3. Je peux utiliser la commande "create or replace".

3.5 Les méthodes dans l'UDT

Les UDT sont plus qu'une structure de données, ils ont des méthodes.

1. on déclare les méthodes dans create type.
2. on définit les méthodes in create type body :
   - on utilise la syntaxe de PL/SQL.
   - la variable $self$ fait référence à l'objet auquel j'applique la méthode.

Déclaration On déclare les méthodes dans la définition de UDT :

```
create type departementT as Object (
            numID integer
nomD  varchar(30),
            lieu  varchar(20)) ;
Member Function getLieu return varchar) ;
```

La définition de méthode (le style d'Oracle)

```
CREATE TYPE BODY <type name> AS
<method definitions = PL/SQL
procedure definitions, using
"MEMBER FUNCTION" in place of
"PROCEDURE">
END;
/
create type body personneType as
 member function getLieu return varchar is
begin
return lieu;
end;
end;
```

3.5.1 Utilisation de méthodes

Il est possible d'utiliser les méthodes dans les requêtes SQL.

```
select  a.salaire,  a.salaireInEuros(1.44)
from  Acteurs a
where a.name='Jackson' ;
```

Dans l'exemple, ci-dessus la méthode *salaireInEuros* est définie dans la classe Acteurs.

3.6 Héritage (création de sous type)

Les types supportent l'héritage multiple avec le mot-clé UNDER :

```
CREATE OR REPLACE TYPE <type_name>
UNDER <supertype_name> (
<attribute> <data_type>,
<inheritance clause> <subprogram spec>, <pragma clause>)
<FINAL | NOT FINAL> <INSTANTIABLE | NOT INSTANTIABLE>;
/

create  type  personT as Object (
        nom varchar(20),
```

```
rue varchar(30),
       ville varchar(20),
Member Function getNom return varchar)
NOT FINAL;

create  type  etudiantType under personT (
        note float);
```

personT est le type mère et *etudiantType* est un sous-type.

Le type est final par défaut. Le type mère dans une héritage doit être NOT FINAL. On peut rendre un type NOT FINAL, en utilisant :

```
alter type tye_name NOT FINAL;
```

3.6.1 Ajout d'un attribut dans un type

```
alter type etudiantType
add attribute date_naissance date
cascade ;  // propager aux tables déjà construites à partir du type.

SQL> desc etudiantType
 etudiantType extends SKAF.PERSONT
 etudiantType is NOT FINAL
           Name                           Null?    Type
           ------------------------------ -------- ---------------------

    1      NOM                                     VARCHAR2(20)
    2      RUE                                     VARCHAR2(30)
    3      VILLE                                   VARCHAR2(20)
    4      NOTE                                    FLOAT(126)
    5      DATE_NAISSANCE                          DATE

METHOD
------
 MEMBER FUNCTION GETNOM RETURNS VARCHAR2
```

3.6.2 Ajout d'une méthode/fonction à un type

```
alter type etudiantType
add member function age return integer
cascade;
```

3.6.3 Type de ligne

SQL-99 possède aussi la notion de type de ligne qui correspond aux structures du C : c'est un ensemble non encapsulé d'attributs. Le type peut-être nommé ou non

Type de ligne non nommé :

create table EMP (nomE varchar(35), adresse ROW(numero integer, rue varchar(30)))

ne marche pas dans Oracle

Type de ligne nommé :

```
CREATE ROW TYPE adresse_t
(numero integer,
rue varchar(30))
```

On peut ensuite utiliser ce type pour une déclaration d'attribut ou même pour créer une table à partir de ce type (comme pour les autres types)

3.7 Vues du dictionnaire des données

1. USER_TYPES pour les types
2. USER_coll_TYPES, pour les collections
3. USER_OBJECT_TABLES pour les tables objet relationnelles
4. USER_VARRAYS : type collection

```
SQL> desc user_types
SQL>       Name                          Null?    Type
           ------------------------------ -------- ----------------------

    1      TYPE_NAME                     NOT NULL VARCHAR2(30)
    2      TYPE_OID                      NOT NULL RAW(16)
    3      TYPECODE                               VARCHAR2(30)
    4      ATTRIBUTES                             NUMBER
    5      METHODS                                NUMBER
    6      PREDEFINED                             VARCHAR2(3)
    7      INCOMPLETE                             VARCHAR2(3)
    8      FINAL                                  VARCHAR2(3)
    9      INSTANTIABLE                           VARCHAR2(3)
   10      SUPERTYPE_OWNER                        VARCHAR2(30)
   11      SUPERTYPE_NAME                         VARCHAR2(30)
   12      LOCAL_ATTRIBUTES                       NUMBER
   13      LOCAL_METHODS                          NUMBER
   14      TYPEID                                 RAW(16)
```

3.8 Tables

Les données d'un type ne sont persistantes que si elles sont rangées dans une table. Il est possible de créer une table à partir d'un type.

Syntaxe : create table ¡table name¿ of ¡type name¿ ;

Exemple 3.4. *Création d'une table OR :*

```
create table etudiantTable of  etudiantType;
```

La table etudiantTable correspond à (extent) d'une classe dans un schéma ODL.

On peut ajouter des contraintes d'intégrité :

```
create table etudiantTable of  etudiantType
(constraint pkctr primary key(nom));
```

3.8.1 Héritage de tables

Une table peut hériter d'une ou plusieurs tables. Pas supporté par Oracle 10g.

3.8.2 Caractéristiques d'une table Objet-Relationnel

Une table est une table Objet-Relationnel si elle a été construite à partir d'un type (create table .. OF). Les lignes de ces tables sont considérées comme des objets avec un identifiant (OID). On peut utiliser des références pour désigner les lignes de ces tables (pas possible pour les autres tables)

3.8.3 Vues du dictionnaire des données

```
USER_OBJECT_TABLES pour les tables objet relationnelles
```

3.8.4 Insertion de données

Exemple 3.5. *Insertion simple :*

```
insert into etudiantTable values('Dupond', 'laxou','nancy',10,'22-dec-02');
```

Exemple 3.6. *Insertion avec constructeur :*

On peut aussi utiliser le "constructeur du type" avec lequel la table a été construite :

```
insert into etudiantTable values(etudiantType('Dupond', 'laxou',
'nancy',10,'22-APR-02'));
```

Si le type est utilisé par un autre type, l'utilisation du constructeur du type est obligatoire

3.8.5 Modifications

On peut utiliser la notation pointée comme en SQL92 :

```
update etudiantTable
set etudiantTable.note= 12
where eudiantTable.nom = 'titi';
\begin{verbatim}

SQL-99 fournit aussi la notation " .. " pour désigner un attribut d'une
colonne de type structuré :

\begin{verbatim}
update etudiantTable set etudiantTable.adresse..numero = 12
 where etudiantTable.nom = 'titi';
\begin{verbatim}

\subsubsection{Appel de procédure ou fonction}

\begin{verbatim}
select e.nom, age(e)          // Le " this " est passé en paramètre
from etudiantTable e
where age(e) < 40

Sous Oracle :

select e.nom, e.age()
from etudiantTable e
where e.age() < 40
```

3.8.6 Références

On peut indiquer dans la définition d'un type qu'un attribut contient des références (et non des valeurs) à des données d'un autre type ; la syntaxe est "REF nom-du-type" : (C'est un pointeur vers un objet de type nom-du-type)

```
create type employeType as Object (
            nom varchar(20),
    rue varchar(30),
            ville varchar(20),
    dept REF depType);
```

```
\begin{verbatim}

\begin{itemize}

\item Si T est un type alors REF T est le type d'une référence à T.
C'est un pointeur vers l'identié d'objet (OID) de type T.

\item Dans les langages OO l'OID n'est pas visible à l'utilisateur,
tandis que REF est visible, mais c'est de ''charabia''.
\end{itemize}


Par la suite, on utilise la table employe:

\begin{verbatim}
create table employe of employeType;
```

3.8.7 Exemple de select avec référence

La notation pointée permet de récupérer les attributs d'un type dont on a un pointeur (le point permet de suivre les références dans Oracle i.e. la suivie de REF est implicite). C'est une bonne habitude d'utiliser un alias quand on utilise les fonctionnalités OR.

Exemple 3.7. *Lieu de travail des employés (avec Oracle).*

```
select nom, e.dept.lieu
from employe e

Cette requête est correcte. Par contre:

select nom, dept.lieu
from employe

Pas correcte.
```

Exemple 3.8. *Le même exemple en SQL-99 :*

```
select e.nom(), e.dept.lieu() from employe e
```

Attention, l'alias e est indispensable
Dans SQL-99, chaque attribut d'un UDT a deux méthodes :

1. Un générateur (get) et un mutator (set). Ces methods ont les mêmes noms que les attributs.
2. Le générateur n'a pas d'arguments. Le mutator prend en argument la nouvelle valeur.

3.8.8 Insertions avec référence

Exemple 3.9. *La référence est un pointeur vers NULL*

```
insert into employe values (1230, 'Durand', NULL);
```

Exemple 3.10. *Référece vers le dept numéro 10*

```
insert into employe(matricule, nom, dept)
select 1240, 'Dupond', REF(d)
from dept d
where dept.numDept = 10;
```

3.8.9 L'opérateur DEREF d'Oracle-Motivation

Exemple 3.11. *Trouver le département d'un employé ?*

```
select e.dept
from  employe e
where e.marticule=44;
```

C'est légal mais e.dept c'est une référence donc c'est de 'Charabia'. L'opérateur DEREF permet de voir la valeur de l'objet de type departementType.

```
select DEREF(e.dept)
from  employe e
where e.marticule=44;
```

3.9 Collections

Il existe deux types :

1. Varray (voir vos notes de cours)
2. tables imbriquées (nested tables)

3.9.1 Tables imbriquées

```
CREATE OR REPLACE TYPE CourseList AS TABLE OF VARCHAR2(64);

desc courselist

SELECT type, text
FROM user_source
WHERE name = 'COURSELIST';

CREATE TABLE department (
```

```
name     VARCHAR2(20),
director VARCHAR2(20),
office   VARCHAR2(20),
courses  CourseList)
NESTED TABLE courses STORE AS courses_tab;
```

Plus d'information sur : `\http://www.psoug.org/reference/nested_tab.html`

3.10 Remarques

Le modèle Objet-Relationnel assure la compatibilité ascendante : les anciennes applications relationnelles fonctionnent dans le monde OR

Le produit open source d'Oracle *TopLink* permet de faire la correspondance entre objet et Objet-Relationnel.

4 Correspondance Objet-relationnel (Mapping OR)

Les bases de données OO n'ont pas tenues leur promises (problème de coût, de legacy,....) (voir vos notes de cours).

Les bases de données Objet-Relationnel ne sont pas très populaires (voir vos notes de cours).

La solution est de faire la correspondance entre l'objet et le relationnel. Ce cours explique les problèmes de base qui se posent quand on veut faire correspondre les données contenues dans un modèle objet avec les données contenues dans une base de données relationnelle

4.1 Les Problèmes

1. les classes n'ont pas de clé : solution transformer un attribut en une clé.
2. les objets ont d'identités. Ceci peut être différente de la clé. (voir vos notes)
3. les associations.
4. l'héritage.
5. Navigation entre les objets.
6. et les méthodes ? ? ? ici on ne s'occupe pas de méthodes

4.2 de classes vers relations

La solution le plus simple est :

1. transformer chaque classe en une relation (table).
2. les attributs de la classe devient un attribut (une colonne) de la relation.

Bien sûr, cette solution marche si les attributs de la classe sont de type atomiques (pas de collections).

Exemple 4.1. *La classe Film en java*

```
Class Film  {
titre String;
anne integer;
longueur integer;
}
```

Devient la realtion :

```
Film(titre,annee,longueur) ;
```

On choisit *titre et année* comme clé.

Cas des attributs non atomiques

Un cas simple lorsqu'un attribut de type classe avec de champs dont les valeurs sont atomiques.

La solution est de présenter chaque champ par un attribut dans la relation de la classe.

```
Class Acteur  {
  nom String ;
  adresse  Adresse;
}

En relationnel:

Acteur(nom, rue, ville);
```

La classe Adresse n'a pas de correspondance sous forme d'une table séparée dans le modèle relationnel. Les attributs de la classe Adresse sont insérées (embedded) dans la table qui représente la classe Acteur. Les objets de la classe Adresse n'ont pas d'identification liée à la base de données (ils n'ont pas une clé primaire).

Cas des attributs de type collection

Ces attributs sont considères comme des associations.

4.3 Les associations

Dans le code objet une association peut être représentée par

– une variable d'instance représentant "l'autre" objet avec lequel se fait l'association (associations 1 :1 ou N :1) (one-one, one-many, many-one).

– une variable d'instance de type collection ou map représentant tous les autres objets avec lesquels se fait l'association (associations 1 :N ou M :N) (one-many, many-many).
– une classe "association" (M :N) (many-many).

Dans le monde relationnel, une association peut être représentée par
– une ou plusieurs clés étrangères (associations 1 :1, N :1 ou 1 :N)
– une table association (associations M :N)

Exemple 4.2. *Voir vos notes de cours.*

Objet dépendant

Un objet dont le cycle de vie dépend du cycle de vie d'un autre objet auquel il est associé, appelé objet propriétaire. Aucun autre objet que le propriétaire ne doit avoir de référence directe vers un objet dépendant. En ce cas, la suppression de l'objet propriétaire doit déclencher la suppression des objets qui en dépendent (déclenchement « en cascade ») (C'est l'association de composition).

4.4 Traduction de l'héritage

voir le tp

4.5 Conclusion

La correspondance OR est l'approche utilisé par (Hibernate `www.hibernate.org`) et par TopLink (un produit Open Source développé par Oracle. TopLink permet à la fois de faire la correspondance de l'objet vers relationnel et de l'objet vers Objet-Relationnel). Hibernate et toplink implémentent le standard JPA (JAVA Persistence API).

5 Synthèse

Les avantages et les inconvénients de bases de données OO, Objet-relationnel et le mapping..

Voir vos notes de cours.