Chapitre 2

Modèle objet - relationnel SQL3

Pourquoi étendre le modèle relationnel ?

- Le modèle relationnel présente des points forts indiscutables, mais il a aussi des points faibles
  - ✓ L'objet répond à ces faiblesses
- Inertie de l'existant : de très nombreuses bases de données en fonctionnement sont basées sur le modèle relationnel
- Manque de normalisation pour les SGBDO
  - ✓ Trop de solutions propriétaires

Pourquoi étendre le modèle relationnel ? (2)

- D'où
  - ✓ La nécessité de conserver la compatibilité avec l'existant
  - ✓ L'intérêt d'une intégration douce

- Le modèle Objet-Relationnel (OR) permet un passage en douceur

Pourquoi étendre le modèle relationnel ? (3)

- En modèle relationnel, la reconstitution d'objets complexes éclatés en tables relationnelles est très coûteuse car elle nécessite de nombreuses jointures

- Pour remédier à ce problème (éclatements-jointures), le modèle OR emploie:
  - ✓ Les références qui permettent de réaliser des structures complexes
  - ✓ Les attributs multivaluées (tableaux, ensembles et listes)
  - ✓ Les références facilitent aussi l'utilisation et le partage des données volumineuses (ex. multimédia) d'une manière simple et à moindre coût sans jointure

Pourquoi étendre le modèle relationnel ? (4)

- L'impossibilité de créer de nouveaux types dans le modèle relationnel implique
  - ✓ Un manque de souplesse
  - ✓ Une interface difficile avec les applications orientées objet
- Le modèle OR permet de
  - ✓ Définir de nouveaux types utilisateur (*User data type*) simples ou complexes
  - ✓ Avec des opérations pour les manipuler
- L'OR supporte l'héritage de type
  - ✓ Cela permet de profiter du polymorphisme et de faciliter la réutilisation

Le modèle Objet-Relationnel OR

Le modèle objet-relationnel

- Idée de base
  - ✓ Étendre le modèle relationnel en ajoutant des concepts essentiels de l'objet

- Le cœur du modèle objet-relationnel reste conforme au relationnel
- On ajoute les concepts clés de l'objet pour faciliter l'intégration des deux modèles

- Cela permet de combler les plus grosses lacunes du modèle relationnel

Le modèle objet-relationnel (2)

- Abstraction fondamentale: La relation
  - ✓ et pas la classe comme pour les SBBDO
- Extension du modèle relationnel
  - ✓ Attributs structurés et multivalués
  - ✓ Héritage pour les relations et les types
  - ✓ ADTs (Types abstraits de données) pour les domaines
  - ✓ Identification d'objet pour les tuples
  - ✓ Surcharge d'opérations

- Extension de SQL pour supporter les concepts OO

Compatibilité relationnel - OR

- Une compatibilité ascendante
  - ✓ Les anciennes applications relationnelles fonctionnent dans le monde OR (Objet-Relationnel)

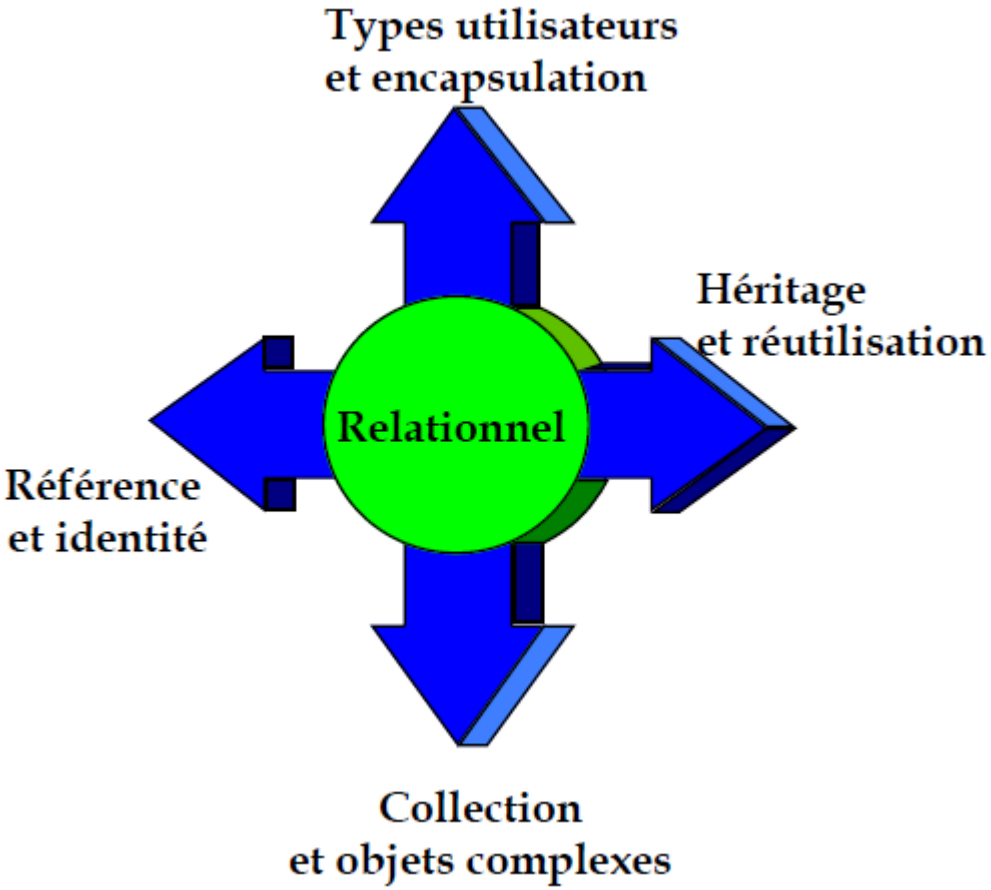
Tables et Objets: Exemple

- Un objet complexe dans une table

Police	Nom	Adresse	Conducteurs		Accidents		
			Conducteur	Âge	Accident	Rapport	Photo
24	Paul	Paris	Paul	45	134		
			Robert	17	219		
					037		

Objet Police

Les extensions apportées au relationnel



Les concepts additionnels essentiels

Type de données utilisateur

- *User data type*
- Le modèle OR offre la possibilité de définir de nouveaux types complexes avec des opérations pour les manipuler
- Le système de type du SGBD devient extensible
  - ✓ Il n'est plus limité aux types alphanumériques de base comme avec le relationnel pur et SQL2
  - ✓ Exemple: on peut définir des types: texte, image, point, ligne, etc

Collections

- Permettant de supporter des attributs multivalués
- Les SGBD objet-relationnels offrent différents types de collections, tels que:
  - ✓ Table
  - ✓ Tableau dynamique
  - ✓ Liste
  - ✓ Ensemble
  - ✓ etc

- Exemple
  - ✓ LIGNE LISTE (POINT): Permet de définir des lignes sous forme de listes de points

Référence d'objet - *Object Reference*

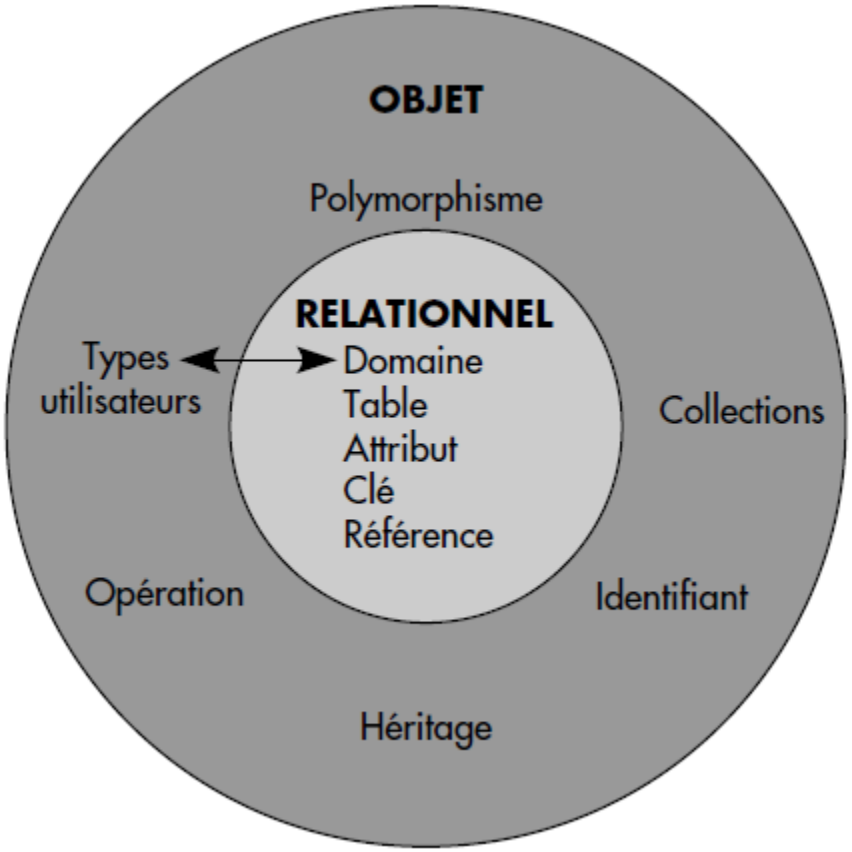
- Le modèle OR permet d'utiliser des références aux objets
- Une référence est l'identifiant d'objet (OID)
- Les références permettant chaîner directement les objets entre eux, sans passer par des valeurs nécessitant des jointures
- Un référence
  - ✓ Immuable
  - ✓ Unique
  - ✓ Non réutilisable après destruction de l 'objet
  - ✓ Généré par le système (performance en navigation)
- ✓ Contrairement au SGBDO, les références ont des valeurs affichables

Héritage

- Le modèle OR permet l'héritage de type et l'héritage de table
- Héritage de type (*Type inheritance*):
  - ✓ La possibilité de définir un sous-type d'un type SQL ou d'un type utilisateur
  - ✓ Le sous-type hérite de la structure et des opérations du type de base

- Héritage de table (*Table inheritance*):
  - ✓ La possibilité de définir une sous-table d'une table existante
  - ✓ La sous-table hérite de la structure (et des opérations) de la table de base

Résumé



N.B. Le modèle OR conserve les notions de base du modèle relationnel: domaine, table, attribut, clé et contrainte référentielle

Produits

- PostGres
- IBM Db2
- Illustra
- UniSQL/X
- Informix
- Oracle
- Sybase

- Attention, peu respectent SQL3 !!!!

Extensions du langage de requêtes – SQL3

Extensions du langage de requêtes

- Extensions du langage SQL (SQL3 ou SQL-99)
- SQL3 : Comporte un langage de requêtes étendu
  - ✓ Appels d'opérations
  - ✓ Support automatique de l'héritage
  - ✓ Parcours de références
  - ✓ Constitution de tables imbriquées en résultat
  - ✓ Etc

- Compatible SQL3-SQL2
  - ✓ Les BD et applications existantes en SQL2 marchent avec SQL3
  - ✓ Ce qui permet un passage facile du relationnel au OR

SQL3

- Nous allons présenter les commandes essentielles :
  - ✓ définition de types abstraits
  - ✓ support d'objets complexes
  - ✓ héritage de type et de table

Types

Types

- L'utilisateur (le programmeur) peut créer ses propres types de données
  - ✓ DISTINCT
  - ✓ Types structurés

- En utilisant la commande CREATE TYPE
- Une instance d'un type peut être un objet ou une valeur

Types DISTINCT

- Le mot clé DISTINCT est utilisé pour renommer un type de base existant déjà
- Permet de différencier les domaines des colonnes
- Les types DISTINCT
  - ✓ Sont définis à partir des types de base et s'utilisent avec les mêmes instructions que ces types de base

- Exemple

```
CREATE DISTINCT TYPE matricule AS INTEGER
```

Types structurés-OBJECT

- Correspondent aux classes des langages objets
- Peuvent contenir des constructeurs, attributs (variables d'instances), fonctions et procédures (méthodes)
- Les **membres** peuvent être *public*, *protected* ou *private*
- Les fonctions et procédures peuvent être écrites en SQL ou en un autre langage

Types OBJECT

- Syntaxe

  CREATE TYPE *nom-type* **AS OBJECT**

  (nom_1 *nom-type*$_1$, nom_2 *nom-type*$_2$, …

  <*définition de méthodes*> **)**

- *nom-type*$_i$ peut être :
  - ✓ Un type usuel de SQL (CHAR, VARCHAR, NUMBER…)
  - ✓ Un type défini par l'utilisateur

Types OBJECT-Exemple

```
CREATE TYPE departementType AS OBJECT
(  numDept INTEGER, nomDept VARCHAR(30),
   lieu  VARCHAR(30)  )
```

```
CREATE TYPE Tadresse AS OBJECT
(  rue VARCHAR(20), numéro VARCHAR(4), ville
   VARCHAR(20)  )
```

Méthodes

- Un type OBJECT peut avoir des méthodes
- Syntaxe

  CREATE TYPE nom-type **AS OBJECT**

  (*<déclaration des attributs>* ,

  <déclaration des signatures des méthodes> **)**

- Exemple

```
CREATE TYPE departementType AS OBJECT
  (  numDept integer, nomDept VARCHAR(30),
     lieu  varchar(30),
     MEMBER FUNCTION getLieu RETURN VARCHAR  )
```

Méthodes - Corps

- Le corps contient le code de méthodes
- Il peut contenir
  - ✓ Des instructions SQL ou d'un langage de programmation (JAVA)
  - ✓ Appels de méthodes

- Exemple

```
CREATE TYPE BODY departementType AS
   MEMBER FUNCTION getLieu RETURN varchar IS
      begin
            return lieu;
            end;
      end;
```

Héritage

- Les types supportent l'héritage
- Le mot clé UNDER permet de créer des sous-types
- Héritage des attributs et méthodes
- Possibilité de redéfinir le code des méthodes dans les sous-types

Héritage (2)

- Exemple

```
CREATE TYPE TPersonne AS OBJECT
   ( NSS INTEGER , nom VARCHAR(20) , prénom
     VARCHAR(20), conjoint REF TPersonne  )
     NOT FINAL

CREATE TYPE TEtudiant UNDER TPersonne
   (  faculté VARCHAR(18), cycle VARCHAR(18)  )
```

- Un type est ***final*** par défaut
- Le mot clé NOT FINAL est obligatoire si le type a des sous-types

Tables

Table

- La création de type ne crée pas d ’objets
- Les objets d’un type sont persistants que lors ils sont insérés dans des tables déclarées

- Il est possible de créer trois types de tables :
  - ✓ Tables du relationnel classique
  - ✓ Tables de valeurs structurées
  - ✓ Tables à partir d’un type de données

Tables de valeurs structurées

- Exemple

```
CREATE TABLE personne
   ( NSS INTEGER , nom VARCHAR(20) , prénoms
     TPrénoms, adr Tadresse )
```

```
CREATE TYPE TPrénoms AS VARRAY(4) OF
      VARCHAR(20)
```

```
CREATE TYPE Tadresse AS OBJECT
(   rue VARCHAR(20), numéro VARCHAR(4), ville
    VARCHAR(20)   )
```

Des tables à partir des types

- Exemple

```
CREATE TYPE TEmploye AS OBJECT
   (  matricule INTEGER , nom VARCHAR(20) ,
      prénoms TPrénoms, adr Tadresse,
      dept INTEGER, salaire INTEGER     )

CREATE TABLE employe OF TEmploye
```

- On peut aussi indiquer des contraintes d'intégrité

```
CREATE TABLE employe OF TEmploye
      (  primary key (matricule) )
```

Contraintes d'intégrité

- On peut associer aux tables les contraintes usuelles de SQL :
  - ✓ PRIMARY KEY (nom-col)
  - ✓ UNIQUE (nom-col)
  - ✓ FOREIGN KEY (nom-col) REFERENCES nom-table [(nom-col)]…
  - ✓ CHECK (condition)

Caractéristiques d'une table objet-relationnelle

- Une table peut hériter d'une autre table
- Les lignes des tables objet-relationnelles sont considérées comme des objets avec des identifiants (OID)
- On peut utiliser des références pour désigner les lignes de ces tables

Requêtes

Insertion de données

- Se fait comme avec une table ordinaire

```
NSERT INTO employe (matricule , nom,… ) VALUES
(5423, 'Ben Ali', …)
```

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

```
NSERT INTO employe values (  TEmplye (5423,
'Ben Ali', …) )
```

Modification

- On utilise la notation pointée **(.)** pour accéder aux attributs

```
UPDATE employe SET employe.salaire = 12000
WHERE employe.nom = 'Ben ali'
```

- On utilise la double notation pointée **(.)** pour accéder aux attributs d'une colonne de type **structuré**

```
UPDATE employe
SET employe.adresse.numero = 12
WHERE employe.nom = 'Ben ali'
```

Appel de fonctions

```
SELECT e.NOM, e.AGE()
FROM employe e
WHERE e.AGE() < 35;
```

Références

Références

- SQL3 permet de parcourir les associations représentées par des attributs de type références
- Dans la définition d'un type, on peut indiquer qu'un attribut contient des références (et non des valeurs) à des données d'un autre type
- Si l 'objet est de type T, sa référence est de type REF(T)
- Syntaxe: REF nom-du-type
- Exemple

```
CREATE TYPE TVoiture ( numero CHAR(9), couleur
VARCHAR (10), proprietaire REF TPersonne  )
```

✓ Création de la table voiture

```
CREATE TABLE voiture OF TVoiture
```

Requêtes avec références - Sélection

```
SELECT v.proprietaire.nom
FROM voiture v
WHERE v.couleur = 'ROUGE'
AND v.proprietaire.adr.ville = 'Mila'
```

- **N.B.** L'alias **v** est indispensable
- N.B. La notation . permet d'accéder aux attributs de l'objet référencé
- **Question:** Que fait la requête précédente?
- **Réponse:**

Requêtes avec références – Sélection (2)

- **Question:** Donner la requête permettant de rechercher les numéros des voitures dont le propriétaire habite la rue des "martyres" à Alger et a pour nom "Brahimi"?

Requêtes avec références – Sélection (2)

- **Question:** Donner la requête permettant de rechercher les numéros des voitures dont le propriétaire habite la rue des "martyres" à Alger et a pour nom "Brahimi"?
- **Réponse**

```
SELECT v.numero
FROM voiture v
WHERE v.proprietaire.adr.ville = 'Alger'
AND v.proprietaire.adr.rue = 'martyres'
AND v.proprietaire.nom = 'Brahimi'
```

Requêtes avec références - Insertion

- Requête avec pointeur null

```
INSERT INTO voiture VALUES (9123, 'Gris', NULL)
```

- Requête avec une référence vers le propriétaire (personne) ayant le NSS = 1548442

```
INSERT INTO voiture (numero, couleur,
proprietaire)
VALUES (  9123, 'Gris',
          SELECT REF(p) FROM proprietaire p
          WHERE proprietaire.NSS = 1548442 )
```

Requêtes avec références - Modification

```
UPDATE voiture
SET proprietaire = ( SELECT REF(p)
                     FROM proprietaire p
                     WHERE NSS = 1548442  )
WHERE numero = 9123
```

- **Question:** Que fait la requête précédente?

Collections

Collections

- Permettent de représenter des colonnes multivaluées
- Types de collections:
  - ✓ Ensemble (sans doublons)
  - ✓ Sac (avec des doublons)
  - ✓ Liste (ordonnée et indexée par un entier)
  - ✓ D'autres types de collections peuvent être ajoutées par les SGBD
- Les constructeurs de collections peuvent être appliqués sur tout type déjà défini

Collections - Exemple

```
CREATE TYPE TEmploye AS OBJECT
      (  matricule INTEGER, nom VARCHAR(30),
         prenoms LIST ( VARCHAR (15)),
         enfants SET  ( personne ),   …)

CREATE TABLE employe OF TEmploye
```

Collections - Utilisation

- Les collections peuvent être utilisées comme des tables en utilisant le mot-clé TABLE
- Exemple

```
SELECT nom FROM employe e
WHERE nom IN (  SELECT *
                FROM TABLE ( e.prenoms ) )
```

Collections – Utilisation (2)

- Ajout d'une colonne à une table
  - ✓ Nous allons ajouter la colonne passetemps (ensemble de chaînes de caractères) à la table personne

```
ALTER TABLE personne ADD COLUMN passetemps
SET (VARCHAR)
```

- Donner la requête permettant de retrouver les références des personnes ayant pour passe-temps le vélo

Collections – Utilisation (2)

- Ajout d'une colonne à une table
  - ✓ Nous allons ajouter la colonne passetemps (ensemble de chaînes de caractères) à la table personne

```
ALTER TABLE personne ADD COLUMN passetemps
SET (VARCHAR)
```

- Donner la requête permettant de retrouver les références des personnes ayant pour passe-temps le vélo
- Réponse:

```
SELECT REF(p) FROM personne p
WHERE 'VELO' IN
                SELECT * FROM TABLE (p.passetemps)
```

Tableau dynamique - *varray*

- Une collection **ordonnée** de taille (en nombre d'éléments) limitée
- Contient des éléments d'un même type
- Syntaxe
  - ✓ **CREATE TYPE** *nom-type* **AS VARRAY** (*nbmax*) OF *nom-type2*
  - ✓ nom-type2 peut être:
    - type usuel de SQL (CHAR, VARCHAR, NUMBER, …)
    - type défini par l'utilisateur

- varray: valeur multivaluée de type vecteur

Tableau dynamique – *varray* (2)

- Exemple

```
CREATE TYPE TPrénoms AS VARRAY(4) OF
            VARCHAR(20)
```
Exemple de valeur : `('Fatima', 'Amina')`

```
CREATE TABLE personne
  ( NSS INTEGER , nom VARCHAR(20) , prénoms
    TPrénoms )
```

```
INSERT INTO personne (NSS, nom, prénoms)
VALUES ( 1548401, 'Brahimi',
         TPrénoms ('Fatima', 'Amina') )
```

Tables imbriquées (*nested Table*)

- Une table relationnelle peut contenir d'autres tables imbriquées
- Exemple

```
CREATE TYPE TTelephone AS TABLE OF CHAR(10)
```

```
CREATE TABLE personne
   ( NSS INTEGER , nom VARCHAR(20) ,
     tel TTtelephone  )
     NESTED TABLE tel STORE AS TableTel
```

- L'utilisateur doit donner un nom à la table qui contiendra les téléphones de toutes les personnes

Tables imbriquées (2)

- valeur multivaluée de type table
  - ✓ Pas de nombre maximum de valeurs
  - ✓ Pas d'ordre
  - ✓ Indexable

Exemple table imbriqués

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

create type NT_adresse as table of Tadresse;
/
```

Exemple table imbriqués

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

Insertion table imbriqués

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)));
```

Le mot clé the

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));
```

Mise a jour

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';
```

Consultation

- 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';
```

Differences entre Varray et table imbriqués

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

Conclusion

- Les SGBD Relationnels traditionnels (SQL2) sont mal adaptés:
  - ✓ Aux nouvelles applications : aide à la décision, conception, géographie, bureautique, SIG, multimédia, …
  - ✓ Aux nouvelles techniques : IHM, programmation orientée objet, programmation en logique, architectures réparties, ...
- Les SGBD Orientés Objet (SGBDO) :
  - ✓ Ne tirent pas suffisamment profit de l'existant et des investissements considérables réalisés dans le relationnel
  - ✓ Manque de normalisation
  - ✓ Trop de solutions propriétaires

Conclusion (2)

- Les SGBD Objet-Relationnel (OR) :
  - ✓ s'appuient sur l'existant
  - ✓ disposent d'un langage normalisé SQL3
  - ✓ Adoptés par de nombreux grands opérateurs (IBM, Oracle, …)