Bases de Données Avancées

S. Cerrito

Année 2007-2008, Evry

Plan du Cours

1. Introduction et rappels du modèle relationnel
2. Modèles objet et objet-relationnel
3. XML et le Modèle "Données Semi-structurées"
4. A définir

INTRODUCTION

Historique

- Avant 1970 : BD=fichiers d'enregistrements, "modèles" *réseaux* et *hiérarchique*; pas de vraie indépendance logique/physique.
- En 1970 : modèle *relationnel* (Codd) : vraie indépendance logique/physique.
- Années 80 et 90 : nouveaux modèles :
  modèle à objets et object-relationnel
  modèle à base de règles (Datalog)
- Fin années 90 : données dites *semi-structurées* (XML).

Centre de ce cours : modèle à objets et object-relationnel, puis modèle semi-structuré.

1 Notions essentielles des BD relationnelles

Mots clés :

- Univers U, Attributs $A_1, \cdots, A_n$
- Domaine $Dom(A)$ d'un attribut A
- Schéma d'une relation dont le nom est R.
- n-uplet sur un ensemble E d'attributs
- Relation (ou "table") sur un schéma de relation
- Schéma d'une BD
- Base de données B sur un schéma de base

Un *univers* U est un ensemble fini et non-vide de noms, dits *attributs*.
Le *domaine* d'un attribut A ($Dom(A)$) est l'ensemble des valeurs possibles associé 'a A.

Exemple :
$U = \{NomFilm, Realisateur, Acteur, Producteur, NomCinema, Horaire\}$

$Dom(NomFilm) = Dom(Realisateur) = Dom(Acteur) = Dom(Producteur) =$
$Dom(NomCinema)$= chanes de caractères.

$Dom(Horaire) = \{h.m \;\mid\; h \in [1, \cdots, 24],\; m \in [0, \cdots, 60]\}$

Un *schéma* d'une relation dont le nom est R est un sous-ensemble non-vide de l'univers U.

Suite de l'exemple :

- Schéma de la relation $Film$=$\{NomFilm, Realisateur, Acteur, Producteur\}$
- Schéma de la relation $Projection$=$\{NomFilm, NomCinema, Horaire\}$

Intuition : Format de deux tables.

$Film$:

NomFilm	Realisateur	Acteur	Producteur
⋮	⋮	⋮	⋮

$Projection$:

NomFilm	NomCinema	Horaire
⋮	⋮	⋮

Soit $E = \{A_1, \cdots, A_n\}$ le schéma d'une relation. Un *n-uplet* n sur E est une fonction $E \rightarrow Dom(A_1) \cup \cdots \cup Dom(A_n)$ telle que, pour tout $A_i \in E$, $n(A_i) \in Dom(A_i)$.

Si $E' \subset E$, la restriction de n à E se note $n(E')$.

Exemple.

Un n-uplet possible sur le schéma de $Projection$:
$\langle ``Jugez - moi\ coupable'', ``Gaumont\ Alesia'', 13.35\rangle$.

Sa restriction à $\{NomCinema, NomFilm\}$:
⟨ "Jugez-moi coupable", "Gaumont Alesia"⟩.

Pourquoi définir un n-uplet comme une *fonction* plutôt que comme un élément de $Dom(A_1) \times \cdots \times Dom(A_n)$?

Une *relation* (table) r sur un schéma de relation S est un ensemble d'n-uplets sur S. On dit aussi : S est le schéma de r.

Exemple.

Film :

NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

$Projection$:

NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

Un schéma $\mathcal{S}$ *d'une base* sur un univers U est un ensemble non-vide d'expressions de la forme $N(S)$ o S est un schéma de relation et N un nom de relation.

Exemple(on omet les {}).

$U =$
$\{NomFilm, Realizateur, Acteur, Producteur, NomCinema, Horaire, Spectateur\}$

$$\mathcal{S} =$$

$\{$
$Film(NomFilm, Realizateur, Acteur, Producteur),$
$Projection(NomFilm, NomCinema, Horaire), Aime(Spectateur, NomFilm)$
$\}$

Schéma de la base = Format des données de la base.

Quel est le format de la base de l'exemple ?

- Une *base de données* B sur un schéma de base $\mathcal{S}$ (avec univers U) est un ensemble de relations finies $r_1, \cdots r_n$ o chaque r_i est associée à un nom de relation N_i et est telle que si $N_i(S) \in \mathcal{S}$, alors r_i a S comme schéma.
- On peut aussi imposer des *contraintes* sur les données. Par exemple : les *dépendances fonctionnelles* (‘a voir), qui fixent, entre autres, les *clés* des relations (‘a voir).
- Ces contraintes, dites d’*intégrité*, font aussi partie de la spécification du format des données de la base.

Exemple d'une base.

Film			
NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

Projection		
NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

Aime	
NomFilm	Spectateur
nf1	s1
nf1	s2
nf2	s1
nf3	s3

2 Fondements des Langages de Requête (qque soit le modèle)

- Informellement : *Requête sur une base* = question que l'on pose à la base.
- *Langage de requête*= langage permettant d'écrire des requêtes
- Importance d'un langage de requête formel et rigoureux :
  1. Conception de langages commerciaux
  2. Evaluation de la puissance d'expression de chaque langage commercial
  3. Possibilité de déterminer ce qu'un langage commercial ne pourra pas exprimer
  4. Notion d'équivalence entre deux expressions de requête ⇒ Optimisation "logique" de l'évaluation d'une requête

Un langage formel pour le modèle relationnel : *algèbre relationnelle*

Question : Et pour d'autres modèles de données ?

2.1 Les opérateurs de l'algèbre relationnelle

- Opérateurs ensemblistes : union ($\cup$), intersection ($\cap$), différence ($\setminus$), produit cartésien ($\times$)
- projection sur un ensemble d'attributs E (π_E), sélection d'un ensemble de n-uplets selon une condition C (σ_C), jointure "naturelle" ($\bowtie$), division ($\div$), renommage (ρ).

3 Limites du modèle relationnel

1. On ne peut pas imbriquer les informations
2. La structure du schéma est très rigide
3. On ne peut pas exprimer la clôture transitive d'une relation (par ex. vol(départ, arrivée) par rapport à vol_direct(départ, arrivée)

Commençons par (1).

En relationnel ("première forme normale") :

OPERAS :

Auteur	*Titre*	*Langue*
Mozart	La Flûte Enchantée	Allemand
Mozart	Don Juan	Italien
Mozart	Les noces de Figaro	Italien
Bizet	Carmen	Français
Bizet	Djamileh	Français

Redondance.

Si on imbrique :

$OPERAS$:

<table>
<tr><th>Auteur</th><th colspan="2">Opéra</th></tr>
<tr><td rowspan="4">Mozart</td><td>Titre</td><td>Langue</td></tr>
<tr><td>La Flûte Enchantée</td><td>Allemand</td></tr>
<tr><td>Don Juan</td><td>Italien</td></tr>
<tr><td>Les noces de Figaro</td><td>Italien</td></tr>
<tr><td rowspan="3">Bizet</td><td>Titre</td><td>Langue</td></tr>
<tr><td>Carmen</td><td>Français</td></tr>
<tr><td>Djamileh</td><td>Français</td></tr>
</table>

4 Modèle à Objets

- Extension de concepts de langages comme C^{++} or Java au cas des BD, où la *persistance* des données est primordiale.
- Concepts clés :
  - types,
  - classes et objets,
  - identité des objets,
  - héritage.

Ici, on choisi *ODL* (Object Definition Language) comme langage de spécification de la structure d'une BD à objets → Ecriture du **schéma**.

Types

Types atomiques. Integer, float, char, string, boolean et les *énumérations* (listes de noms, par ex. (couleur, noir&blanc), chaque type enumeré ayant son propre nom, par ex. couleurs).

Constructeurs de Types Collection :

- Set<Type>. *Ex.* : Set<integer>. *NB* : ensembles finis.
- Bag<Type>. Si T est un type, Bag<T> est un type T' dont les valeurs sont des *multi-ensembles finis* d'éléments de type T. *Ex.* : $\{1, 2, 1\}$ est de type Bag<integer>.
- List<Type>. *Ex.* : List<char> (les chaînes de caractères), List<integer>. *NB* : listes finies.
- Array<Type, entier> : type des tableaux de n (entier) éléments de type Type. *Ex.* : Array<char, 10>.
- Dictionary<Type1, Type2> : un type dont les valeurs sont des ensembles finis de couples $\langle$ clé, val $\rangle$, où clé est de type Type1 et val est de type Type2.
- Struct Nom { Type1 Nom1,...,TypeN NomN }.
  Ex. : Struct Adresse { string rue, string ville}. (Type record !)

Possibilité d'imbriquer les constructeurs de types (la déf. des types est récursive !)

Classes et Objets

- Une *classe* est un *type abstrait* : on défini les propriétés d'un objet et ce que l'objet peur faire (*méthodes*).
- Un objet o est une instance d'une classe C et a un et un seul idéntificateur (*OID*).
- Déclaration d'une classe en ODL :
  `class` Nom = { liste de propriétés et méthodes }

- La sorte la + simple de propriété : un *attribut*.
- Dćlaration d'un attribut : on indique son type et sa valeur.
- *Exemple* :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
};
```

  Attribut `SorteFilm` : de type énumération; le nom de ce type est `couleurs`, le nom de l'attribut est `SorteFilm`.

- Un objet de cette classe : ("Autant en emporte le vent", 1939, 231, couleur)

Un autre exemple de classe :

```
class Star {
attribute string nom;
attribute Struct Adr { string rue, string ville  } adresse
};
```

Ici l'attribut adresse est de type “structure” (record), ce type s'appelle “Adr” et le type des 2 champs est string.

Autre sorte de propriétés : Associations entre objets

- Mot clé dans la déclaration d'une classe : `relationship`
- Définition + riche de la classe `Film` :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Star> acteurs;
};
```

Possibilité d'indiquer les acteurs d'un film donné (objet de la classe `Film`).

Associations INVERSES entre objets

- Mot clé : `inverse`
- Définition de classes encore + riches :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Star> acteurs
inverse Star::JoueDans };

class Star {
attribute string nom;
attribute Struct Adr = { string rue, string ville } adresse;
relationship Set<Film> JoueDans
inverse Film::acteurs ;
};
```

Méthodes

- *Méthode* = code exécutable qui peut être appliqué à un objet.
- En ODL, on n'écrit pas le code d'une méthode : on indique juste son nom et sa *signature* : les types des entrées/sorties.
- Comme dans les langages de programmation orientés objet, l'objet est un argument "caché" d'une méthode m.
- Une méthode m peut soulever des *exceptions*. Mot clé : `raises`.
- Les paramètres d'une méthode m sont spécifiés par :
  (a) `in` (entrée)
  (b) `out` (sortie).
  Une méthode peut aussi (c) renvoyer une valeur.
  Différence entre (b) et (c) : selon (b) la méthode m se comporte comme une procédure d'un langage impératif (C, par ex.), selon (c) m est comme une fonction. Si (b), alors et le type du résultat est `void` (vide), car on n'a pas calculé une fonction, mais effectué une action.

Exemple

Définition encore + riche de la classe `Film`, avec 3 méthodes :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Stars> acteurs
inverse Star::JoueDans;
float longeurHeures() raises(ErreurHoraire);
void NomsActeurs(out Set<String>);
void AutresFilms(in Star, out Set<Film>) raises(PasValable);;
};
```

Associations liant + que 2 classes

En ODL on peut spécifier seulement des relations binaires. Comment faire si l'on veut modéliser une relation R entre n classes, où $n > 2$? On introduit une nouvelle classe C qui joue le rôle de R et on définit n relations binaires entre C et chaque C_i.

Exemple On veut modéliser une relation *contrats* qui lie une star, un film et un studio.

On crée une nouvelle classe `Contrat` :

```
class Contrat {
attribute integer salary;
relationship Film leFilm
inverse ...;
relationship Studio leStudio
inverse ...;
};
```

Ici, `leFilm` est le nom de la relation qui lie un objet de la classe `Contrat` au seul objet de la classe `Film` auquel **ce** contrat fait référence et Film est le “type d'arrivée” de cette relation (un contrat est lié à un objet de la classe `Film`).
Pour pouvoir remplir la partie ... qui suit `inverse` après la déclaration de la relation `leFilm` de la classe `Contrat`, en y écrivant, par ex. :

```
Film::ContratsPour
```

il faut modifier la définition de la classe `Film`, en y ajoutant :

```
relationship Set<Contrat> ContratsPour inverse Contrat::leFilm ;
```

4.1 Sous-classes et Héritage

Avec ODL on peut déclarer qu'une classe C est une *sous-classe* d'une autre classe D.

Mot-clé : `extends`

Par ex. :

```
class Cartoon extends Film {
relationship Set<Star> voix;
}
```

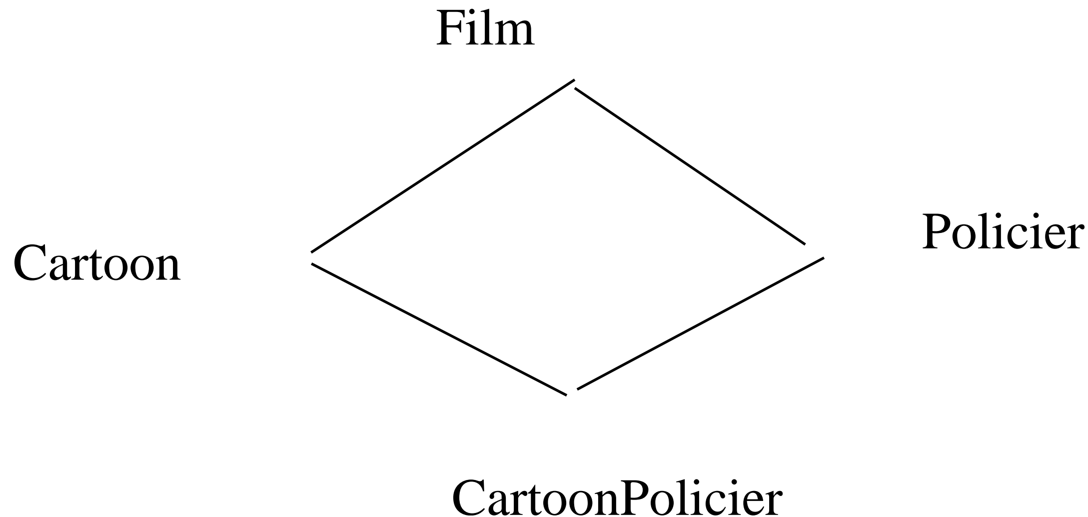
Ici, les objets de `Cartoon` ont, par rapport à `Film`, la relation nouvelle `voix` avec l'ensemble de stars qui donnent leur voix aux personnages.

De plus, une sous-classe C de D *hérite* toutes les propriétés de D. Dans l'exemple : tout objet de `Cartoon` a les attributs `titre, année, longueur, SorteFilm` et les relations `JoueDans`, etc.

Héritage Multiple ?

Supposons que nous avons défini aussi une classe `Policier`, qui est une autre sous-classe de `Film`. Le film *Roger Rabbit* appartient au même temps à `Cartoon` et `Policier`.

Comment faire en ODL ? Déclarer une nouvelle classe : `CartoonPolicier`



`extends` va être suivi par plusieurs noms de classes, séparées par ":"

```
class CartoonPolicier extends Policier :  Cartoon;
```

Héritage multiple ? Problème des conflits de noms

Exemple. Une sous-classe de `Film` qui s'appelle `FilmAmour` a un attribut `fin`, de type énuméré { `happy, triste` }. Une autre sous-classe de `Film` qui s'appelle `FilmTribunal` a aussi un attribut `fin`, de type énuméré { `coupable, innocent` }.
Ambiguïté pour la classe `FilmTribunalEtAmour`, sous-classe des 2.

Le standard ODL ne dicte pas quoi faire.

Possibilités :

- Interdire l'héritage multiple :-(
- Préciser la classe dont il faut hériter la signification de l'attribut `fin`
- Renommer un attribut. Par ex., attribut `fin` de `FilmTribunal` $\rightsquigarrow$ `verdict`

Un objet d'une classe C qui est une sous-classe de la classe D hérite aussi les méthodes de D.

C'est exactement comme dans le langages de programmation orientés objet.

Ensemble des définitions ODL décrivant les propriétés des classes et leur relations hiérachiques

$$=$$

Définition d'un **schéma** de base de donnés **à objets**

Extension d'une classe

Une définition d'une classe C en ODL spécifie le format (schéma) de la classe.
Pour faire référence à l'ensemble des instances de la classe (l'"extension" de C))
et pouvoir interroger la base : mot clé `extent`.

```
class Film (extent Films)
attribute string titre;
⋮
```

N.B. : Expression `Film` $\neq$ Expression `Films`.
(On aurait pu utiliser `Totos`, à la place de `Films`, certes, mais pas `Film`).

Même si chaque objet a un'identité unique on PEUT vouloir traiter plusieurs objets comme "non-distinguables" par rapport aux propriétés observables, même si chacun garde sa propre identité.

C'est alors sensé de déclarer une *clé* : mot clé = `key`

```
class Film
(extent Films key (titre,année))
{
attribute string titre;
⋮
}
```

4.2 Un langage de Requête pour les BD Objet : OQL. Quelques Notions

- *Object Query Language*
- Notation à la SQL.
- Utilisé comme extension d'un langage de programmation hôte, comme C^{++} ou Java.

Format Général d'une requête OQL

```
SELECT liste d'expressions
FROM liste d'une ou plusieurs d\eclarations de variables.
WHERE condition C de selection
```

Une variable est déclarée en indiquant :

1. Une expression dont la valeur a un type collection, par ex. `set` ou `bag`.
2. Le nom de la variable.

L'expression (1) indique l'extension d'une classe, par exemple `Films`. Analogie avec une relation dans une requête SQL.

Notation "."

Soit o un objet d'une classe C.

- Si p est un attribut, $o.p$ est la valeur de p pou o.
- Si p est une relation, $o.p$ est l'objet ou la collection d'objets reliés à o par p.
- Si p est une méthode (éventuellement avec paramètres $a_1, \cdots a_k$), $o.p(....)$ est le résultat de l'application de p à a.

  N.B : Puisque la déclaration de la méthode `NomsActeurs` de la classe `Film` est :
  `void NomsActeurs(out Set<String>);`
  si `MonFilm` est un objet de la classe `Film`, l'expression `MonFilm.NomsActeurs(mesStars)` ne renvoi pas de valeur, mais, comme effet de bord, pose la valeur de la variable output `mesStars` de la méthode comme étant un ensemble de noms d'acteurs.

Exemple d'une BD à objet pur illustrer OQL

```
class Film
(extent Films key (titre, année))
{
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Star> acteurs inverse Star::JoueDans;
relationship Studio appartient-à inverse Studio::possède;
float longeurHeures() raises(pasLongueurTrouvée);
void NomsActeurs(out Set<String>);
void autresFilms (in Star, out Set<Film>) raises(PasActeur);
};

class Star
(extent Stars key nom)
{
attribute string nom;
attribute Struct Adr
{string rue, string ville } adresse;
relationship Set<Film> JoueDans inverse Film::acteurs;
};

class Studio
(extent Studios key nom)
{
attribute string nom;
attribute string adresse;
relationship Set<Film> :possède inverse Film::appartient-à;
};
```

Exemples de Requêtes OQL

```
SELECT m.année
FROM Films m
WHERE m.titre = ``Autant en emporte le vent''
```

Ici, `Films m` est une déclaration d'une variable `m` qui a sa valeur dans l'*extension* `Films` de la classe dont le nom est `Film`, i.e. : valeur(`m`) $\in$ `Films`.

Exemples de Requêtes OQL, suite

```
SELECT s.nom
FROM Films m, m.acteurs s
WHERE m.titre = ‘‘Casablanca’’
```

Dans l’évaluation, tous les couples (m,s) tels que m est un film et s est un acteur de m (selon la relation `acteurs` de `Film`) sont considérés :

FOR chaque m dans FILMS DO
    FOR chaque s dans m.acteurs DO
        IF m.titre = “Casablanca” THEN
            ajouter s.nom au multi-ensemble résultat.

La condition après le `WHERE` limite l’espace de recherche pour `m` aux films dont le titre est “Casablanca”.

Exemples de Requêtes OQL, suite

```
SELECT DISTINCT s.nom
FROM Films m, m.acteurs s
WHERE m.apppartient-à.nom = ‘‘Disney’’
```

Comme en SQL, le mot clé `DISTINCT` élimine le répétitions dans le multi-ensemble (bag) résultat.

Exemples de Requêtes OQL, suite

Liste des noms des films du studio Disney, ordonnés par leur longueur; si deux films ont la même longueur, trier selon l'ordre alphabétique des titres.

```
SELECT m
FROM Films m
WHERE m.appartient-à.nom = ‘‘Disney’’
ORDER BY m.longueur, m.titre
```

Exemples de Requêtes OQL, suite

On veut l'ensemble des couples de stars qui vivent à la même adresse :

```
SELECT DISTINCT Struct(star1:s1, star2:  s2)
FROM Stars s1, Stars s2
WHERE s1.adresse = s2.adresse AND s1.nom < s2.Nom
```

Pour chaque couple qui passe le test on produit un record, avec 2 champs, nommés `star1` et `star2`. Le type de chaque champ est la classe `Star`.
Le type du résultat final de cette requête est :

```
Set<Struct{star1:  Star, star2:  Star}>.
```

Exemples de Requêtes OQL, suite

Utilisation de "sous-requêtes"

Une autre façon de chercher les noms des acteurs des films du studio Disney :

```
SELECT DISTINT s.nom
FROM (SELECT m
      FROM Films m
      WHERE m.appartient-à.nom =''Disney'') f,
      f.acteurs s
```

La variable `m` prend valeur dans la collections des films du studio Disney, qui est le résultat de la sous-requête :

```
SELECT m
      FROM Films m
      WHERE m.appartient-à.nom =''Disney''
```

La variable `s` prend valeur dans la collections des acteurs du film de Disney `f`. **NB** : un autre `WHERE` inutile ici.

5 Modèle Relationnel-Objet (à vol d'oiseau)

- Années 90 : un certain succès du modèle des BD à objets "pur"'.
- Puis, abandon, mais intégration de certaines idées objet dans les SGBD relationnels :
  - Types structurés pour les attributs, avec utilisation de constructeurs comme `struct, set, bag`....
    NB : a `set` de `struct` est essentiellement une relation ! $\Rightarrow$ Une valeur d'une composante d'une tuple d'une relation R peut être une autre relation S.
  - Idéntificateurs : même 2 tuples qui ont partout les mêmes valeurs peuvent avoir des idéntificateurs $\neq$.
  - Références à des tuples.
  - Méthodes.

Types structurés pour les attributs

Définitions mutuellement récursives des notions : *type d'un attribut* et *type (schéma) d'une relation* :

- *Base.*
  Un type *atomique* (entier, réel, string, etc.) peut être le type d'un *attribut.*
  Toute structure de la forme `nom_relation(At1,..., AtN)` où les attributs ont des types atomique peut être le type d'une *relation.*

- *Cas de Récurrence.*
  Un type construit à partir d'autres types en utilisant les constructeurs de type de ODL (`struct, set, bag` etc.) peut être le type d'un *attribut.* Un schéma de relation peut être le type d'une *attribut.*
  Toute structure de la forme `nom_relation(At1,..., AtN)` où les attributs sont typés peut être le type d'une *relation.*

Relations Imbriquées !

Exemple : *Stars(nom, adresse(rue,ville), dateNaissance, films(titre,année,longueur))*

<table>
<tr><th>nom</th><th>adresse</th><th>dateN</th><th>films</th></tr>
<tr><td>Fisher</td><td><table><tr><th>rue</th><th>ville</th></tr><tr><td>Maple</td><td>Hollywood</td></tr><tr><td>Locust</td><td>Malibu</td></tr></table></td><td>9/9/60</td><td><table><tr><th>titre</th><th>année</th><th>longueur</th></tr><tr><td>Star Wars</td><td>1977</td><td>124</td></tr><tr><td>Empire</td><td>1980</td><td>127</td></tr></table></td></tr>
<tr><td>Smith</td><td><table><tr><th>rue</th><th>ville</th></tr><tr><td>Oak</td><td>New York</td></tr><tr><td>Locust</td><td>Malibu</td></tr></table></td><td>5/3/56</td><td><table><tr><th>titre</th><th>année</th><th>longueur</th></tr><tr><td>Star Wars</td><td>1977</td><td>124</td></tr><tr><td>Empire</td><td>1980</td><td>127</td></tr></table></td></tr>
</table>

Références

Dans l'exemple précédent, redondance : le même film apparaît 2 fois.

A la place d'incorporer un tuple s dans un tuple t, on permet au tuple t de pointer vers s.

On ajoute 2 autres possibilités de typage pour un attribut A :

1. Type *référence* à un seul tuple dont le schéma de relation est R. On écrira : $A(*R)$.

2. Type ensemble de références à des tuples dont le schéma de relation est R. On écrira : $A(\{*R\})$.

Exemple : DEUX relations :
Films(titre, année, longueur)
*Stars(nom, adresse(rue, ville), dateNaissance, leurFilms({ *Films}))*

Stars :

<table>
<tr><th>nom</th><th>adresse</th><th>dateN</th><th>films</th></tr>
<tr><td>Fisher</td><td><table><tr><th>rue</th><th>ville</th></tr><tr><td>Maple</td><td>Hollywood</td></tr><tr><td>Locust</td><td>Malibu</td></tr></table></td><td>9/9/60</td><td><table><tr><td>→ 1</td></tr><tr><td>→ 2</td></tr></table></td></tr>
<tr><td>Smith</td><td><table><tr><th>rue</th><th>ville</th></tr><tr><td>Oak</td><td>New York</td></tr><tr><td>Locust</td><td>Malibu</td></tr></table></td><td>5/3/56</td><td><table><tr><td>→ 1</td></tr><tr><td>→ 2</td></tr></table></td></tr>
</table>

Films :

	titre	année	longueur
1	Star Wars	1977	124
2	Empire	1980	127

Interroger des BD relationnelles-objet

Par ex : le langage SQL:1999 supporte des aspects objets.

Pour en savoir plus :

`http://www.service-architecture.com/database/articles/sql1999.html`

ou :

`http://sqlpro.developpez.com/SQL_AZ_991.html`

6 XML et les Données Semi-structurées

- L'apparition de XML (*eXtensible Markup Language*) (plus "évolué" que HTML) a mené au nouveau concept de *données semi-structurées*.
- XML : standard W3C d'échange de données sur le Web. Permet un échange sur un format standard, independemment des formats de stockage de ces données.
- Grande flexibilité.
- Multitude de standards associés:
  - Formats de Schémas : DTD et XML-schéma
  - Langages de Requête : XPATH, XQUERY (extension de XPATH),...
  - XSLT : notation pour transformer un document XML d'un format à un autre.

  etc.
- Lien avec le cours *Documents Structurés* ?

Parmi les applications BD :

- Intégration de Données (“mediateur”).
- Bases de Données en Biologie.

6.1 HTML

HTML (*Hyper Text Markup Language*) : un standard d'écriture de documents pour le Web.

HTML est un langage à **balises** ("étiquettes"). Ces balises sont **fixes**, à fonctions prédéfinies.

Les balises de HTML permettent de :

- **Mettre en forme un texte**
  Ex. `<B> </B>, <I> </I>, <CENTER> </CENTER>`,....
- **Créer des liens** (balises "amarres").
  Ex :

```
<A HREF="http://www.univ-evry.fr/">Universit\'e d'Evry Val d'Essonne </A>
```

Le rôle des balises autres que les “amarres” est celui de *présenter visuellement* du *texte* en un certain format.

Par exemple :

- `<B> bla </B>` sert à écrire **bla** à la place de bla
- `<I> bla </I>` sert à écrire *bla* à la place de bla
- `<CENTER> bla </CENTER>` sert à écrire

  bla

  à la place de :
  bla
- `<H1> bla </H1>`, `<H2> blabla </H2>` `<H3> blablabla </H3>` servent à introduire des titres (des “sections”), par ordre d’importance décroissant.

Exemple de document HTML : un morceau de ma page personnelle, écrite en HTML :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
   <TITLE> Serenella  Cerrito </TITLE>
   <META NAME="GENERATOR" CONTENT="Mozilla/3.01Gold (X11; I;
SunOS 5.5 sun4m) [Netscape]">
</HEAD>
<BODY TEXT="#004D0F" BGCOLOR="#EEFFFF">

<H1 ALIGN=CENTER>
<IMG SRC="klee.tunisian-gardens.jpg" HEIGHT=370 WIDTH=370>
</H1>
</BR>

<H1 ALIGN=CENTER> <FONT COLOR="#FF7F50">
Serenella Cerrito </H1></FONT>

<H2><IMG SRC="blue-bullet.gif" HEIGHT=20 WIDTH=14> Current Position
<IMG SRC="home03.gif"></H2>

<H3>
Professor :</H3>
<UL>
<LI><A HREF="http://www.univ-evry.fr">Universit&eacute; d'Evry Val d'Essonne </A>
<LI> <A HREF = "http://www.ibisc.fr"> Member of I.B.I.S.C (previously : L.a.M.I.)</A>
<LI> <A HREF =  http://www.ibisc.univ-evry.fr/Equipes/RMF>  RMF Team </A>
</UL>
```

```
<H2><IMG SRC="blue-bullet.gif" HEIGHT=20 WIDTH=14 >Research
<IMG SRC="idea.gif"></H2>
<UL>
<LI><H3><A HREF="nuovocv.ps">Curriculum Vitae, postcript file (may 2001 version). </A></H3><BR>
</UL>
<UL>
<LI>
<H3>   Research Interests </H3>
<UL>
<LI><I>Logic and Automated Deduction</I></LI>
<LI><I>Logic and  Databases </I></LI>
<LI><I>Logic and  Logic Programming </I></LI>
<LI><I>Logic and Functional Programming.</I></LI>
</LI>
</UL>
```

6.2 Limites de HTML

HTML n’est pas adapté à l’interrogation des données. Il permet de mettre en **FORME** un texte; **il ne permet pas de STRUCTURER “logiquement”** un contenu.

Example 1 *Une organisation publies des données financiers stockées dans une BD relationnelle et des pages web sont crées après une requête SQL. Une autre organisation veut une analyse financière de ces données, mais elle a accès seulement aux pages HTML. Pour cela, elle ne peut qu'écrire du logiciel qui transforme du texte HTML en une structure de données adaptée à l'analyse.*

- *Une petite modification du format d'un élément d'une page web peut casser ce logiciel !*
- *Même si on a besoin seulement de la valeur moyenne d'une colonne d'une table, on peut avoir besoin de charger une base entière via plusieurs requêtes de pages HTML.*

7 XML

XML : nouveau standard adopté par le World Wide Web Consortium (W3C) comme complément de HTML permettant un échange aisé de données de sur le web.

- Le but principal de XML n'est pas de décrire un format de texte, mais de **structurer** logiquement un contenu.
- Les balises ont le rle de **classer des données** selon une hiérarchie **définie par l'auteur** du document XML.

- Avec XML, la mise en forme textuelle est effectué dans une *feuille de style*, un document separé qui associe des formes de présentations (texte en gras, en italique, centré, etc.) aux balises. Des feuilles différentes permettent des formattages différents du même document.
- Des outils permettent de convertir un document XML en HTML, afin de pouvoir afficher une page web.

Example 2 Un petit document XML

```
<?xml version=''1.0 encoding=''iso-8859-1''?>

<communication prior=''important''>
<pour> Virginie </pour>
<sujet> Rappel </sujet>
<message> N'oublie pas de lire l'article
<lire> Lutz et al. 2002. </lire>
Il faut bien comprendre
<reflechir> la preuve de terminaison.  </reflechir>
Rendez-vous <date> mercredi </date> <lieu> dans mon bureau </lieu>
</message>
<signature> Serena </signature>
</communication>
```

Suite de l'exemple

Résultat de la mise en forme grce à une feuille de style :

Priorité : important
Pour : Virginie
Sujet : Rappel

N'oublie pas de lire l'article *Lutz et al. 2002.* Il faut bien comprendre **la preuve de terminaison.** Rendez-vous **mercredi** *dans mon bureau*

Serena

Suite de l'exemple

Résultat de la mise en forme avec une **autre** feuille de style :

Priorité : IMPORTANT
Pour : VIRGINIE

Sujet : RAPPEL

N'oublie pas de lire l'article *Lutz et al. 2002.*

Il faut bien comprendre *la preuve de terminaison.*

Rendez-vous **mercredi dans mon bureau**

Serena

XML modèlise des informations :

- En organisant les données en un *graphe* d'*objets complexes*
- En les structurant de façon plus *flexible* par rapport au au modèle relationnel ou objet : les données sont dites *semistructurées*

graphe, objet complexe, flexible, semistructuré = ????

$\Rightarrow$ Voir la suite...

7.1 Syntaxe de base de XML

La composante essentielle est l'*élément*, un morceau de document delimité par une balise d'ouverture (ex. `<toto>`) et une de fermeture (ex. `</toto>`.

Un élement peut contenir du texte, des autres élements (→ "objet complexe"), ou un mélange des deux.

- Les balises (leur noms) **sont définies par les utilisateurs**.
- Elles n'ont pas de signification prédéfinie : elles indiquent seulement **comment structurer le document sous forme de arbre** (ou, plus généralement, de graphe) !!

Example 3

```
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
```

Ici on a :

- *Un élément complexe de "sorte" ("type")* `personne`*, qui consiste d'un triplet d'éléments ayant les "sortes"* `nom`,`age`,`email`*.*
- *Un élément* `Alan` *de "sorte"* `nom`
- *Un élément* `42` *de "sorte"* `age`
- *Un élément* `agb@abc.com` *de "sorte"* `email`

Suite de l'exemple

Le contenu de ce document peut être représenté :

- Soit par un arbre où les *noeuds internes* sont etiquéttés par les baliseses.
- Soit par un arbre où les *arcs* sont etiquéttés par les balises.

FIGURES AU TABLEAU

Example 4

```
<gens>
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
<personne>
<nom> Patricia  </nom>
<age> 36 </age>
<email> ptn@abc.com </email>
</personne>
</gens>
```

Remarque : on peut utiliser plusieurs éléments ayant la même balise pour représenter une collection.

Dans l'exemple 4, une entité de "sorte" **`gens`** est une collection de personnes...

A nouveau, on peut représenter ces informations sous forme d'un arbre, avec 2 possibilités.

Example 5

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
</livre>
<livre>
.
.
.
</livre>
</biblio>
```

7.2 Pourquoi “objet complex” ?

Comparer avec les BD relationnelles (en première forme normale), où le domaine de tout attribut contient seulement des valeurs atomiques, et toute “entité est plate” :

nom	titre	nom-ed	rue-ed	ville-ed	etat-ed	pays-ed
Abit	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Bune	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Suciu	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
⋮						

Dans le document XML de l’exemple 5, un livre est un objet **complex**, composé d’une séquence d’auteurs, d’un titre et d’une adresse (comme dans le BD à objet). La première et la troisième composantes sont elles mêmes des objets complexes.

7.3 Pourquoi “semistructuré” ?

Un premier élément de réponse :
un objet complexe peut avoir des composantes optionnelles, le “schéma” de la base n’est pas rigide.

Différence par rapport au modèle relationnel, où le nombre d’attributs du schéma d’une relation est fixé en avance.

Example 6 *Un livre peut* **éventuellement** *être stocké avec son prix; le champ* `pays-ed` *est aussi optionnel :*

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
<prix-en-euros> 44 <prix-en-euros>
</livre>
<livre>
<auteurs>
<nom> Gardarin </nom>
</auteurs>
<titre> Internet/Intranet et Bases de Donn\'ees </titre>
<edition>
<nom-ed> Eyrolles </nom-ed>
<adresse-edition>
<rue-ed> 61, Bld Saint Germain </rue-ed>
<ville-ed> Paris </ville-ed>
```

```
<etat-ed> France </etat-ed>
</adresse-edition>
</edition>
</livre>
</biblio>
```

Dans les exemples vus jusqu'à ici, les données sont organisées en arbres. Les réferences produisent des **graphes**.

On peut faire référence à un sommet déjà existant dans le graphe car on peut associer à un *idéntificateur* à chaque élément.

Pour "pointer" vers un élément ayant identificateur, disons `cle` (nom choisi par l'auteur), on exploite l'existence des *attributs* XML.

En général, un attribut XML sert à dèfinir une **proprieté des données**; sa valeur est une chaîne de caractéres. (Dans l'exemple 2, `prior` était un attribut).

La syntaxe <u>générale</u> de la déclaration d'attributs est :

```
<balise attribut1=valeur1 ...  attributN = valeurN> ...  </balise>
```

Par ex. :

```
<nom langue=français> Abiteboul </nom>

<nom langue=anglais> Bunemann </nom>
```

Pour identifier un élément il suffit d'utiliser un attribut dont le type déclaré (où ?) est ID :

`<balise attribut=valeur> </balise>` marche, à condition que `attribut` soit de type ID.
La syntaxe abrégée `<balise attribut=valeur/>`
est aussi possible.

Par ex. :
`<Livre ISBN="isbn-95456255"/>`
C'est dans le schéma du document XML (voir après) que l'on on déclare l'attribut ISBN comme ayant le type ID.

Pour faire **référence** à un élément on utilise un attribut de type IDREF :

`<balise attributRef= identificateur> </balise>`

La syntaxe abrégée

`<balise attributRef= identificateur /balise>`

est aussi possible.

Par ex :

`<Edition Editeur="LinuxFrench Edition1" EditeurRef= "LFNET" />` Ici, on déclarera l'attribut `EditeurRef` comme ayant le type IDREF.

Un élément de la forme :

`<balise attributRef= identificateur /balise>`

où `attributRef` est de type IDREF n'a pas de contenu. Il est dit dit *élément vide*.

Example 7

```
<geographie-USA>
</etats>
<etat>
<etat cle = ''e1''>
<code-etat> IDA </code-etat>
<nom-etat> Idaho </nom-etat>
<capitale ref-cap = ''v1' />
<villes-dans  ref-a-villes = ''v1'' />
<villes-dans  ref-a-villes = ''v3'' />
...
</etat>
<etat>
...
</etat>
</etats>
<villes>
<ville>
<ville iden = ''v1''>
<code-ville> BOI </code-ville>
<nom-ville> Boise </nom-ville>
<etat-de-la-ville ref-a-etat = ''e1'' />
</ville>
<ville>
<ville iden = ''v2''>
<code-ville> CCN </code-ville>
<nom-ville> Carson City </nom-ville>
<etat-de-la-ville ref-a-etat = ''e2'' />
</ville>
<ville>
<ville iden = ''v3''>
<code-ville> MO </code-ville>
```

```
<nom-ville> Moscow </nom-ville>
<etat-de-la-ville ref = ''e1'' />
</ville>
...
</villes>
</geographie-USA>
```

La possibilité de faire des références fait passer de la structure de *arbre* à celle plus générale de *graphe* orienté avec une racine.

Figure au tableau.

XML permet de mélanger des données textuelles et des sous-éléments au sein d'un élément :

Example 8
```
<personne>
Voici mon meilleur ami
<nom> Alan  </nom>
<age> 42 </age>
Je ne suis pas sure de l'adresse e-mail suivante~:
<email> agb@abc.com </email>
</personne>
```

Pas naturel du point de vue BD, mais du à l'origine de XML comme langage de documents hyper-texte.

7.4 Schémas pour des documents XML

Deux formats :

1. Un *DTD* (**D***ocument* **T***ype* **D***efinition*)
2. Un *XML-schema*, qui a une structure de typage plus riche.

7.4.1 Les DTD

Un DTD peut être vu comme une sorte de schéma pour les données XML. Il est **optionnel** $\Rightarrow$ données **semistructurées**.

Une document XML qui, en outre d'être syntaxiquement correct, a un DTD, et le respecte, et dit *valide*.

Syntaxe des DTD

Un document XML est composé de deux lignes initiales optionnelles suivie par une suite des éléments. La première des deux lignes initiales indique la version de XML utilisée, la deuxième contient le DTD.

```
<? xml version=''1.0''?>
<?DOCTYPE nom [Declarations-de-Type]>
<nom> ... </nom>
```

La balise `nom` est la balise racine. `Declarations-de-Type` est une suite $Déclaration_1$,...,$Déclaration_n$
où toute déclaration introduit le nom d'un élément et sa "sorte", c.à.d une déscription de la "forme de son contenu".

Syntaxe d'une Déclaration du DTD

Pour le moment, ignorons les déclarations des types des attributs.

Chaque déclaration du DTD est constituée du symbole <, puis de la chaîne de caractères !ELEMENT, puis d'une balise, puis d'un modèle de contenu, et, enfin, le délimiteur de fin > :

<!ELEMENT balise modèle_contenu>

Il y a cinq sortes différents de modèles de contenu.

Modèles de contenu dans une déclaration d'un DTD

1. Contenu vide : `<!ELEMENT balise EMPTY >`
2. Pas de contraintes sur le contenu : `<!ELEMENT` balise `ANY>`.
   (NB : données "semistructurées" !)
3. Elément ne contenant que des données textuelles :
   `<!ELEMENT balise #PCDATA>`
4. Elément ne contenant que d'autre éléments : `<!ELEMENT` balise motif `>`
   où `motif` est une *expression XML-regulière* sur l'alphabet des noms des balises.
5. Eléments de contenu "mixte": mélange à la fois d'éléments et de données textuelles.

Les opérateurs des expression XML-regulières

- Le symbole , indique la concaténation.

  Exemple : `chat`,`chien` signifie qu'un chien doit suivre un chat. L'ordre compte dans les documents XML (arbres ordonnés). (**pourquoi ??**)

- Le symbole | est le XOR logique.

  Exemple : `chat` | `tortue` | `chien` signifie que soit un chat soit une tortue soit un chien est acceptable (mais un seul de ces animaux).

- Le symbole ? rend l'expression immédiatement précédente optionnelle.

  Exemple : (`chat`,`chien`) ? signifie que une suite d'un chat puis d'un chien peut être placée à cet endroit, ou omise.

Les opérateurs des expression XML-regulières, suite

- Le symbole + signifie qu'une suite non vide d'éléments conformes à l'expression immédiatement précédente est réquise.

  Exemple : (`chat` | `chien`)$^+$ signifie qu'il doit y avoir un nombre non nul de chats et de chiens.

- Le symbole $*$ signifie qu'une suite éventuellement vide d'éléments conformes à l'expression immédiatement précédente est réquise.

  Exemple : (`chat`,`chien`)$*$ signifie que, à cet endroit, ou bien il n'y a rien du tout, ou alors il y a une suite de chats et chiens telle que tout chat est immédiatement suivi par un chien et tout chien est immédiatement précédé par un chat.

Example 9

```
<!ELEMENT article
(titre, sous-titre?, auteur*, (paragraphe|table|figures)+, bibliographie?)>
```

Cette déclaration décrit le contenu d'un article comme étant composé d'un titre suivi éventuellement d'un sous-titre, puis de 0 ou plusieurs auteurs, puis d'une combinaison (non-vide) de paragraphes, tables et figures, puis, éventuellement, d'une bibliographie.

Example 10 *Un exemple simple de DDT.*

```
<!DOCTYPE gens [
<!ELEMENT gens (personne)*>
<!ELEMENT personne (nom, age, e-mail)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
]>
```

Le document de l'exemple 4 est valide par rapport à ce DTD.

En résumant, il y a plusieurs raisons pour lesquelles on peut qualifier des données représentées dans un document XML comme étant *semi-structurées* :

1. Le document n'a pas de schéma (DTD ou XML-schéma). Dans ce cas, on a juste du texte, que l'on ne sait pas comme interroger ! (Sauf par recherche de mot clé, comme pour les documents HTML)
2. Le document a un schéma. Mais :
   (a) Une déclaration de la forme `<!ELEMENT` balise `ANY>` ne donne aucune information sur la structure.
   (b) Une déclaration comportant ? prévoit l'optionalité d'une balise B dans le motif associé à une balise A. (Mais : penser aux valeurs nulles dans les SGBD relationnels...)

A la place d'inclure le DTD dans le document, on peut aussi le sauver dans un fichier separé, qui peut être placé à une URL différente. Ceci permet à différents sites web de partager un unique schéma.

Déclaration d'attributs dans un DTD

Un DTD permet aussi de déclarer le type des attributs. Par exemple `ID` est le type des attributs permettant de donner des idéntificateurs aux éléments (voir l'exemple 7 : `cle` est de type `ID`). Si un attribut A est déclaré comme ayant le type IDREF, ceci indique que la valeur de A est un idéntificateur d'un élément (l'élément pointé").

Le mot-clé ATTLIST est utilisé pour déclarer le type d'une liste d'attributs.

En outre de déclarer le type des attributs, on décrit leur comportement; le mots-clés `#REQUIRED, #IMPLIED` indiquent, respectivement, si un attribut est obligatoire ou optionnel.

Syntaxe d'une déclaration de types pour une liste d'attributs :

<!ATTLIST nom_{att1} $type_{att1}$ $descr_{att1}, ..., nom_{attN}$ $type_{attN}$ $descr_{attN}$>

Example 11 *Un DDT pour les données de l'exemple 7 :*

```
<!DOCTYPE  geographie-USA [
<!ELEMENT geographie-USA (etats|villes)*>
<!ELEMENT etats (etat)*>
<!ELEMENT etat (code-etat,nom-etat,capitale,villes-dans*)>
<!ATTLIST etat cle ID #REQUIRED>
<!ELEMENT code-etat (#PCDATA)>
<!ELEMENT nom-etat (#PCDATA)>
<!ELEMENT capitale EMPTY>
<!ATTLIST capitale ref-cap IDREF #REQUIRED>
<!ELEMENT villes-dans EMPTY>
<!ATTLIST villes-dans ref-a-villes IDREFS #REQUIRED>
<!ELEMENT villes (ville)*>
<!ELEMENT ville (code-ville,nom-ville,etat-de-la-ville)>
<!ATTLIST ville iden ID #REQUIRED>
<!ELEMENT code-ville (#PCDATA)>
<!ELEMENT nom-ville (#PCDATA)>
<!ELEMENT etat-de-la-ville EMPTY>
<!ATTLIST etat-de-la-ville ref-a-etat IDREF #REQUIRED>
]>
```

Un défaut des DTD : on ne peut pas déclarer que les sortes des valeurs de l'attribut `ref-cap`*, par exemple, sont des villes. Les DTD n'offrent pas un typage adéquat des données semistructurées.*