Cours de BDA

Bases de Données Avancées

M1 Informatique et MIAGE Initiale, 2016-2017

Serenella Cerrito et Philippe Declerc

Laboratoire Ibisc, Université d'Evry Val d'Essonne, France

Plan du Cours

1. Introduction : rappels du modèle relationnel, certaines limites du relationnel (S. Cerrito)
2. Modèle dit "Données Semi-structurées" (Xml, Xquery) (S.Cerrito)
3. NoSQL (Ph. Declerc)

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)
- NoSQL

Ce cours : modèle semi-structuré (S.Cerrito), et NoSQL (Ph. Declerc)

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

Rappel des BD relationnelles

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 à A.

Exemple : $U =$
$\{NomFilm, Realisateur, Acteur, Producteur, NomCinema, Horaire\}$
$Dom(NomFilm) = Dom(Realisateur) = Dom(Acteur) =$
$Dom(Producteur) = Dom(NomCinema)=$ chaînes de caractères.
$Dom(Horaire) = \{h.m \mid h \in [0, \cdots, 24],\ m \in [0, \cdots, 59]\}$

Rappel des BD relationnelles

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
⋮	⋮	⋮

Rappel des BD relationnelles

Soit $E = \{A_1, \cdots, A_n\}$ le schéma d'une relation.

Un *n-uplet* n sur E est un ensemble $\{A_1 : v_1, ..., A_n : v_n\}$ où $v_i \in Dom(A_i)$.

Un n-uplet possible sur le schéma de *Projection* :
$\{NomFilm : \text{“}Jugez - moi\ coupable''$, $NomCinema :$
$\text{“}Gaumont\ Alesia''$, $Horaire : 13.35\}$,
ce qui est la même chose que
$\{NomFilm : \text{“}Jugez - moi\ coupable''$, $Horaire :$
$13.35\rangle$, $NomCinema : \text{“}Gaumont\ Alesia''\}$.
Toutefois, le plus souvent on note :
$\langle \text{“}Jugez - moi\ coupable''$, $\text{“}Gaumont\ Alesia''$, $13.35\rangle$.

Pourquoi la définition précise de n-uplet demande d'indiquer l'attribut correspondant à chaque valeur ?

Rappel des BD relationnelles

Si t est un n-uplet sur E, et $E' \subset E$, la restriction de t à E se note $t(E')$.

La restriction de
$\langle$“*Jugez – moi coupable*”, “*Gaumont Alesia*”, 13.35$\rangle$ à
$\{NomCinema, NomFilm\}$ est :
$\langle$“*Jugez – moi coupable*”, “*GaumontAlesia*”$\rangle$.

Rappel des BD relationnelles

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

Rappel des BD relationnelles

Un schéma $\mathcal{S}$ *d'une base* sur un univers U est un ensemble non-vide d'expressions de la forme $N(S)$ où 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* (relationnelle) B sur un schéma de base $\mathcal{S}$ (avec univers U) est un ensemble de relations finies $r_1, \cdots r_n$ où 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*, qui fixent, entre autres, les *clés* des relations (cours SGBD L3).
- Ces contraintes, dites d'*intégrité*, font aussi partie de la spécification du format des données de la base.

Rappel des BD relationnelles

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

- 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 $\Rightarrow$ Optimisation "logique" de l'évaluation d'une requête

Deux langages formels pour le modèle relationnel : *algèbre relationnelle* et *calcul relationnel* (cours SGBD L3).

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).

Quelques limites du modèle relationnel

1. On ne peut pas imbriquer les informations
2. La structure du schéma est très rigide

Imbriquer

On ne peut pas imbriquer = ?
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 sur l’auteur.

Imbriquer

Si on imbrique :

OPERAS :

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

On est sorti de la norme “Première Forme Normale”, primordiale pour le modèle relationnel.

Rigidité

La structure du schéma est très rigide = ??

Selon le modèle relationnel une base a un nombre fixé de tables, une table a un nombre fixe d'attributs etc.

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.

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/">Université 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.

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.

Exemple

- Une organisation publies des données stockées dans une BD relationnelle. Des pages web sont crées. Une autre organisation veut une analyse de ces données; son logiciel a acces seulement aux pages HTML.
- 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.

XML

XML : 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 rôle de classer des données selon une hiérarchie définie par l'auteur du document XML.

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.

Exemple

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 <
</message>
<signature> Serena </signature>
</communication>
```

Suite de l'exemple

Résultat de la mise en forme grâce 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

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...

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 ($\rightarrow$ "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).

Exemple

```
<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

Exemple

```
<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>
```

Suite Exemple

On peut utiliser plusieurs éléments ayant la même balise pour représenter une collection.

Dans l'exemple, 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.

Exemple Bibliographie

```
<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>
```

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 de bibliographie, 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.

Pourquoi “semistructuré” ?

Un premier élément de réponse :
un objet complex 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.

Champs optionnels (exemple : prix_en_euros)

```
<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 Données </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>
```

Arbres ou Graphes ?

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 du message, `prior` était un attribut).

Syntaxe des attributs

La syntaxe de la déclaration d'attributs est :

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

Par ex. :

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

Possibilité d'une syntaxe abrégée : `<balise attribut=valeur/>`

Attributs de type ID

Pour identifier un élément il suffit d'utiliser un attribut dont le type déclaré est ID.

Par ex., avec la syntaxe abrégée : :

```
<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.

Types Référence

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

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

Possibilité de la syntaxe abrégée

`<balise attributRef= identificateur />`

Par ex :

`<Livre NOM=''Tout sur Linux'' EditeurRef= "LFE" />`

(on pointe à l'élément identifié par `LFE` et décrivant l'éditeur *Linux French Edition*).

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 contient juste un **pointeur** (attributRef) vers un autre noeud. Il est dit dit *élément vide*.

USA

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

Références

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.

Exemple

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

```
<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.

Dans la suite, on aura pas de mélange, et les données (texte) seront toujours aux feuilles de l'arbre qu'on a si on ignore les réf.

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.

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

Structure d'un DTD :

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

La première ligne, optionnelle, indique la version de XML utilisée, la deuxième contient le DTD proprement dit.
La balise `nom` est la balise racine. `Declarations-de-Type` est une suite
$\mathit{Déclaration}_1,\ldots,\mathit{Déclaration}_n$
où toute déclaration introduit le nom d'un élément et sa "sorte", c.à.d une déscription de la "forme de son contenu", ou bien introduit les attributs d'un élément donné, et leur types.

Pour le moment, ignorons les déclarations pour les attributs.
Chaque déclaration d'élément 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.

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 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 expressions regulières utilisés dans un DTD

- 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 regulières utilisés dans un DTD, suite

- Le symbole $+$ signifie qu'une suite non vide d'éléments conformes à l'expression immédiatement précédente est réquise.
  Exemple : $(\texttt{chat} \mid \texttt{chien})^{+}$ 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 : $(\texttt{chat,chien})*$ 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.

En effet, un DTD est une grammaire, qui spécifie un arbre.

C'est une grammaire *context free* élargie en permettant des expressions regulières dans la droite des règles de production.

Les opérateurs reguliéres sont ceux habituels, car :

ϵ = EMPTY
ab = a,b
a+b = a | b
a*=a*
aa*=a+
ϵ + a = a?

Un document D valide par rapport à un DTD S est arbre qui appartient au langage d'arbres accepté par l''automate d'arbres associés à la grammaire d'arbres S.

Exemple

```
<!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.

Exemple

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 des gens est valide par rapport à ce DTD.

Données semistructurées

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 :
   - 2.1 Une déclaration de la forme `<!ELEMENT balise ANY>` ne donne aucune information sur la structure.
   - 2.2 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 la 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.

Une DTD permet aussi de déclarer des attributs, et leur type.

`ID` est le type des attributs permettant de donner des idéntificateurs aux éléments.

`IDREF` (ou IDREFS) est le type des attributs de référénce.
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").

Exemple du document sur la géographie des USA (p.41) :
dans la DTD correspondante : `cle` est de type `ID`, `ref-cap` est de type `IDREF`, et `ref-à-villes` est de type `IDREFS`).

Déclaration d'attributs dans un DTD, suite

Le mot-clé ATTLIST est utilisé pour déclarer une liste d'attributs (pouvant contenir un seul attribut).
En outre de déclarer le type des attributs, on décrit aussi leur "comportement" : le mots-clés `#REQUIRED`, `#IMPLIED` indiquent, respectivement, si un attribut est obligatoire ou optionnel.

Syntaxe d'une déclaration d'une liste d'attributs de l'élément ayant balise *B* :
<!ATTLIST B nom_at1 type_at1 descr_at1, ..., nom_atN type_atN descr_atN>

Exemple de DTD

```
<!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 (et pas des états, par exemple). Les DTD n'offrent pas un typage adéquat des données semistructurées.

Pourquoi ce défaut ?

XML Schema : C'est quoi ?

Ce sujet est traité dans un autre cours de la MIAGE, et ici on se limite à le mentionner.

Un fichier XML-Schema est lui-même un document XML ! Mais ce document fixe le format d'autres documents XML.

XML Schema : Types

XML a un système de types riches, et ces types sont ceux utilisés par XQuery aussi.

Types Simples de XML Schema : Il en a beaucoup. Par ex., STRING, BOOl, INTEGER, POSITIVEINTEGER, CDATA, DATE, ID, IDREF, IDREFS, NMTOKEN (une lettre, une chiffre, un point, un tiré, une virgule..)

Types Complexes : On peut construire des types complexe à l'aide de constructeurs. Par exemple, grâce au constructeur `<xsd:sequence>`, on peut déclarer qu'un élément `personne` a un type complexe qui est défini comme étant une *séquence* d'un `nom`, de type `string`, d'un `prénom`, de type `string`, d'une `date_de_naissance`, de type `date`, d'une `adresse`, de type `string`, et, enfin, d'un `e-mail`, de type string.

Un langage de requête pour les données XML : XQUERY

XQUERY : langage d'interrogation de documents (bases de données) XML.
Ces diapositives sont fortement inspirées par le document :
Katz, Xquery : A guided Tour
disponible en ligne avec les supports de cours.

Historique

- 1998 : W3C organise un workshop sur XML Query
- 1999 : W3C lance le *XML Query Working Group* (39 membres, 25 companies)
- 2000 : publication des objectifs, des cases d'utilisation et du modèle de données .
- 2001 : draft de la spécification du langage
- 2002 : mises à jour périodiques de cette draft
- 2003 : redaction complete des objectifs et des cases d'utilisation
- Version finale de XQuery Version 1.

Généralités sur XQUERY

- XQUERY est déclaratif : toute expression XQUERY produit une *valeur*; c'est un langage de style *fonctionnel* (comme CAML).
- les expressions le plus banales utilisent des *constantes* (et des opérateurs simples) :
  3+4
  est un programme XQUERY qui s'évalue à l'entier 7.
- On a aussi des *variables* : $x, $y, $z. On verra comment utiliser et lier les variables.

XPATH

XQUERY utilise une partie du langage XPATH.
XPATH permet de naviguer sur un arbre XML.
La base de la syntaxe XPATH est semblable à celle de l'adressage du systéme de fichiers en Unix ou Linux. On peut descendre d'un niveau, etc.
Ses expressions sont dites expressions de chemin (*path expressions*).

Expressions de chémin XPATH

Si l'expression de chemin commence par '/', alors elle représente un chemin absolu vers l'element requis.
Elément (sous-arbre) interrogé :

```
<AAA>
   <BBB>
      <CCC>
          <FFF>
          </FFF>
          <DDD>
               <EEE>
               </EEE>
          </DDD>
      </CCC>
   </BBB>
</AAA>
```

Requête `/AAA`. **Reponse** : L'élément (arbre) interrogé lui-même.

Expressions de chémin XPATH

L'operateur // permet de sélectionner tous les descendants indiqués.
Elément (arbre) interrogé :

```
<AAA>
   <BBB>
       <CCC>
            <BBB>
            </BBB>
       </CCC>
   </BBB>
</AAA>
```

Requête : //BB. Réponse : deux éléments (sous-arbres) :

```
   <BBB>
       <CCC>
            <BBB>
            </BBB>
       </CCC>
   </BBB>
```

et

```
            <BBB>
            </BBB>
```

Exemple de fichier interrogé

On va illustrer XQUERY en étudiant des requêtes sur des données bibliographiques, contenues dans le document books.xml :

Document : books.xml

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1992">
        <title>Advanced Programming in the UNIX Environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
        </book>
    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>65.95</price>
    </book>
    <book year="1999">
        <title>The Economics of Technology and Content
               for Digital TV</title>
        <editor>
           <last>Gerbarg</last>
           <first>Darcy</first>
           <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
```

DTD validée par books.xml

Ce document valide le DTD :

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, (author+ | editor+ ), publisher, pric
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first )>
<!ELEMENT editor (last, first, affiliation )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT affiliation (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

Modèle de données sous-jacent

- Un document est un arbre (ordonné).
- Sept sortes de noeuds :
  1. *Document Node* (le tout premier noeud, représentant le document même)
  2. *Element Node*
  3. *Attribute Node*[1]
  4. *Text Node*
  5. *Comment Node*
  6. *ProcessingInstruction node*
  7. *Namespace Node*
- Même si deux noeuds de l'arbre contiennent les mêmes informations, ils sont vus comme 2 individus distincts (ils ont des positions différentes dans l'arbre global !)

[1] bien que nous avons représenté un attribut autrement. Pourquoi ?

Les valeurs des expressions

- Valeur Atomique = une valeur de type atomique : un booléen, un entier, un flottant,.. (Les types atomiques de XQuery sont les mêmes que pour XML-schema).
- Un *item* est ou bien un noeud ou bien une valeur atomique.
- Une valeur d'une expression (requête) XQUERY est une *séquence ordonnée* de zero ou plusieur items.
  - Il n'y a pas de distinction entre un item et une séquence de longueur 1 : (2) = 2.
  - Il n'y a pas de séquence imbriquée : (1, (2,3),(4)) = (1,2,3,4)
  - Une séquence peut être vide (notée ())
  - Une séquence peut conténir des *données hétéreogènes*

La fonction d'entrée `doc()` renvoie un document (le *document node*).
Exemple : `doc(“books.xml”)`
Elle sert à indiquer le document XML interrogé.

En XQUERY des expressions de chemins (XPATH) sont utilisées pour localiser des noeuds :
`doc(“books.xml”)/bib/book`
renvoie la séquence des noeuds qui sont des livres
dans l'ordre du document. Pour le document de l'exemple, elle est équivalente à :
`doc(“books.xml”)//book`
Pourquoi ?

Filtrage par prédicats

On peut filtrer des noeuds, calculés par une expression de chemin, en utilisant un *prédicat*, c.à.d. une expression booléenne entre crochets :

```
doc("books.xml")/bib/book/author[last="Stevens"]
```

est évaluée comme la sequence des noeuds (éléments) *n* tels que :
- *n* est un `author`
- le fils `last` de *n* vaut "Stevens" (c.à.d. l'expression booléenne `last="Stevens"` s'évalue à Vrai).

Cette expression calcule donc les auteurs dont le nom de famille est "Stevens".

Prédicat numérique

Cas particulier : on a une valeur numérique N entre crochets. C'est interpreté comme le prédicat : "position = N" :

- L'expression suivante renvoie le premier auteur de chaque livre :
  `doc("books.xml")/bib/book/author[1]`

  N.B. La sous-expression `author[1]` est évaluée pour chaque livre.
- Si, par contre, on veut le premier auteur dans le document, il faut écrire :
  `(doc("books.xml")/bib/book/author)[1]`
  Remarquer l'utilisation des parenthéses ici.

Attention

:

- Un chaîne de caractéres juste après l'opérateur de descente d'un niveau, est interpretée comme le nom d'un élément fils du noeud courant :
  `doc("books.xml")/bib/book/author`
  calcule les auteurs des livres.

  Remarquer : un noeud `author` est un fils d'un noeud `book`.
- Mais si ce mot commence par le caractère spécial @, c'est compris comme indiquant un attribut du noeud courant :

  `doc("books.xml")/bib/book/@year`
  calcule les valeurs de l'attribut `year` des livres.
  Les noeuds attributs ne sont pas VRAIMENT des fils de l'élément dont ils parlent !

Requêtes créant des documents

Une requêteXQUERY peut être utilisée pour créer un nouveau document XML, ou une partie de document.

```
<mon_example>
   <mon_text> J'utilise le document : </mon_text>
   <ma_ref>{doc("books.xml")//book[1]/title}</ma_ref>
</mon_example>
```

s'évalue à l'élement XML (la réponse):

```
<mon_example>
   <mon_text> J'utilise le document : </mon_text>
   <ma_ref> <title>TCP/IP Illustrated</title></ma_ref>
</mon_example>
```

car :

1) `<mot> ... </mot>` est un constructor qui crée un élément XML dont la balise est "mot"

2) une expression entre { } est une expression qui va être évaluée.

Requêtes créant des documents, suite

Requête qui crée un document :

```
 document {
  <book year="1977">
    <title>Harold and the Purple Crayon</title>
    <author><last>Johnson</last><first>Crockett</first></aut
    <publisher>HarperCollins Juvenile Books</publisher>
    <price>14.95</price>
  </book>
}
```

Le constructeur `document` crée un noeud document.

Restructuration de valeurs

Une requête peut aussi restructurer des valeurs existants. La requête suivante liste les titres des livres du document interrogé, et les compte.

```
<LesTitres nombre="{ count(doc('books.xml')//title) }">
 {
  doc("books.xml")//title
 }
</LesTitres>
```

Réponse :

```
<LesTitres nombre = "4">
   <title>TCP/IP Illustrated</title>
   <title>Advanced Programming in the Unix Environment</tit
   <title>Data on the Web</title>
   <title>The Economics of Technology and Content for
   Digital TV</title>
</LesTitres>
```

Restructuration de valeurs, suite

NB. Dans la requête précédente :
On construit l'attribut `nombre` de l'élément `LesTitres`.

On **pose** la valeur de cet attribut comme étant égale à l'évaluation de l'instruction `count` (pre-définie), qui compte.

Constructeurs d'élémént et d'attributs

Un élément de balise bla et contenu strumpf peut être crée en écrivant la requête : `<bla> Strumpf </bla>`, comme on a vu. Mais il existe aussi une syntaxe alternative.

La requête suivant crée un élément (qui a un attribut) grâce aux constructeurs : `element` et `attribute` (mots predéfinis !) :

```
element book
{
    attribute year { 1977 },
    element author
    {
        element first { "Crockett" },
        element last { "Johnson" }
    },
        element publisher {"HarperCollins Juvenile Books"},
        element price { 14.95 }
}
```

Expressions FLWOR

Une expression FLWOR :
1) Lie des variables à des valeurs qui sont dans l'espace de valeurs indiqué par le `for` et le `let`.
2) Utilise ces liasons pour céer des valeurs nouveaux.
Une combinaison de liasons de variables crée par le `let` et le `for` est appellée *tuple*.

F : For, **L** : Let, **O** : Order by, **W** : Where, **R** : Return.

On commencera par étudier l'exemple de requête :

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

qui calcule les titres des livres publiés en 2000.

Exemples de requêtes

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

Cette requête lie la variable `$b` à chaque book, un à la fois, pour céer une séquence de tuples. Chaque tuple contient une liaison dans laquelle `$b` est lié à un seul livre.
Le `where` teste chaque tuple, pour voir si `$b/@year` est égal à "2000".
Avec le `return` on renvoie les valeurs de `$b/title` tels que la valeur de `$b` a satisfait la condition du `where`.
Réponse :

```
<title>Data on the Web</title>
```

car on a un seul livre qui passe le test, dans notre base.
NB. Pas de `let`, dans cette requête. On n'est pas obligé à avoir les 2, `for` et `let`. Ici pas de `order by` non plus.

FLOWR, suite

En géneral :

- `for` $x `in` *E*
  on associe une variable $x à une expression *E*, et on crée des *tuples* où chaque tuple lie la variable $x à un élément de la séquence d'objets qui est la valeur de *E*
  Ex : `for $i in (1, 2)` associe la variable $i à l'expression (1,2), qui est une séquence d'entiers, et crée : une liaison de $i à 1 et une liaison de $i à 2 (2 tuples).
- `let` $y := *E*
  lie une variable $y au résultat d'une expression *E*, et ajoute ces liaisons aux tuples générés par le `for`.
  Ex : `for $i in (1, 2) let $j:=(i, i+1)`. On lie $j à (1,2) (pour le tuple qui dit que $i est 1), puis on lie $j à (2,3) (pour le tuple qui dit que $i est 2).
- `where` : filtrage des tuples selon une condition;
- `order by` : tri des tuples selon un ordre (croissant, par défaut);
- `return` : construction du résultat.

FLOWR, suite

Une expression FLOWR :

commmence par des `for` et/ou des `let`,

après on a un `where optionnel`,

puis un `order by` optionnel,

puis un `return` obligatoire.

Exemples de requêtes

Cet exemple de requête crée des éléments qui s'appellent `tuple` (ma balise, pas predéfinie !)

```
for $i in (1, 2, 3)
return <tuple><valeur_de_i>{ $i }</valeur_de_i></tuple>
```

On associe la variable $i à l'expression E=(1,2,3), et on crée 3 liasons : de $i à 1, de $i à 2 et de $i à 3 (autant de liaisons que d'éléments dans la valeur de E). La réponse est donc :

```
<tuple><valeur_de_i>1</valeur_de_i></tuple>
<tuple><valeur_de_i>2</valeur_de_i></tuple>
<tuple><valeur_de_i>3</valeur_de_i></tuple>
```

Remarquer la différence avec la requête :

```
let $i := (1, 2, 3)
return <tuple><valeur_de_i>{ $i }</valeur_de_i></tuple>
```

dont la réponse contient 1 seul tuple (Pourquoi ?) :

```
<tuple><valeur_de_i>1 2 3</valeur_de_i></tuple>
```

Exemples de requêtes

Quand le `for` et le `let` sont combinés, les liaisons générées par le `let` sont ajoutés aux tuples générés par le `for` :

```
for $i in (1, 2, 3)  let $j := (1, 2, 3)
return
   <tuple> <valeur_de_i>{ $i }</valeur_de_i>  <valeur_de_j>{ $j }</valeur_de_j></tuple>
```

dont la réponse est :

```
<tuple>
  <valeur_de_i>1</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
<tuple>
  <valeur_de_i>2</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
<tuple>
  <valeur_de_i>3</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
```

Exemples de requêtes

On veut lister les titres des livres :

```
for $b in doc("books.xml")//book
let $c := $b/title
return <book> {$c} </book>
```

qui s'évalue :

```
<book> <title>TCP/IP Illustrated</title> </book>
<book>
       <title>Advanced Programming in the UNIX Environment</titl
</book>
<book> <title>Data on the Web</title> </book>
<book>
       <title>The Economics of Technology and Content for Digita
</book>
```

Exemples de requêtes

On veut lister les titres des livres et leur nombre d'auteurs.

```
for $b in doc("books.xml")//book
let $c := $b/author
return <book> {$b/title, <nombre> { count($c) }</nombre>}
</book>
```

Réponse

```
<book>
  <title>TCP/IP Illustrated</title>
  <nombre>1</nombre> </book>
<book>
  <title>Advanced Programming in the UNIX Environment</title>
  <nombre>1</nombre>
 </book>
<book>
 <title>Data on the Web</title>
 <nombre>3</nombre> </book>
<book>
  <title>The Economics of Technology and Content for Digital TV<
  <nombre>0</nombre>
</book>
```

Exemples de requêtes

On peut liér plus qu'une variable avec le `for` :

```
 for $i in (1, 2, 3),
     $j in (4, 5, 6)
return
   <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

réponse: le produit cartésien :

```
<tuple><i>1</i><j>4</j></tuple>
<tuple><i>1</i><j>5</j></tuple>
<tuple><i>1</i><j>6</j></tuple>
<tuple><i>2</i><j>4</j></tuple>
<tuple><i>2</i><j>5</j></tuple>
<tuple><i>2</i><j>6</j></tuple>
<tuple><i>3</i><j>4</j></tuple>
<tuple><i>3</i><j>5</j></tuple>
<tuple><i>3</i><j>6</j></tuple>
```

Exemples de requêtes

On veut seulement les titres des livres qui coûtent moins que 50 dollars :

```
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

réponse : Il n'y en a pas. La réponse est la séquence vide.

Exemples de requêtes

On veut seulement les titres des livres qui ont plus que 2 auteurs :

```
for $b in doc("books.xml")//book
let $c := $b//author
where count($c) > 2
return $b/title
```

réponse :

```
<title>Data on the Web</title>
```

N.B. : let `let` a lié $c à la séquence des auteurs, pour chaque livre valeur de $b, et cette séquence a plus que 2 éléments, pour le livre "Data on the Web"

Exemples de requêtes

Lest titres des livres, triés en ordre croissant :

```
for $t in doc("books.xml")//title
order by $t
return $t
```

Le `for` génére une sequence de tuples, avec un titre par tuple. Le `order by` reorganize cette séquence, en la triant selon la valeur des titres, et le `return` renvoie les titres ainsi triés :

```
<title>Advanced Programming in the Unix Environment</title>
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>
The Economics of Technology and Content for Digital TV
</title>
```

Exemples de requêtes

Toutes les informations sur les auteurs (nom de famille et prénom), mais triées en ordre decroissant, selon le noms de famille, d'abord, puis le prénom :

```
for $a in doc("books.xml")//author
order by $a/last descending, $a/first descending
return $a
```

Réponse

```
 author>
    <last>Suciu</last><first>Dan</first>
</author>
<author>
    <last>Stevens</last><first>W.</first>
</author>
<author>
    <last>Stevens</last><first>W.</first>
</author>
<author>
    <last>Buneman</last><first>Peter</first>
</author>
<author>
    <last>Abiteboul</last><first>Serge</first>
</author>
```

Pourquoi on obtient 2 fois W. Stevens ?

Exemples de requêtes

Une expression complexe peut apparaître dans un `return`.
Afficher des éléments `info-livre`, contenant le titre et le prix.

```
for $b in doc("books.xml")//book
return
    <info-livre>{ $b/title, $b/price }</info-livre>
```

réponse :

```
<info-livre>
   <title>TCP/IP Illustrated</title>
   <price>65.95</price> </info-livre>
<info-livre>
   <title>Advanced Programming in the UNIX Environment</title>
   <price>65.95</price>
</info-livre>
<info-livre>
   <title>Data on the Web</title>
   <price>39.95</price> </info-livre>
<info-livre>
  <title>The Economics of Technology and Content for Digital TV</title>
  <price>129.95</price>
</info-livre>
```

Exemples de requêtes

Des constructeurs d'éléments peuvent être utilisés pour présenter les données autrement.

```
for $a in doc("books.xml")//author
return
  <author>{ string($a/first), " ", string($a/last) }</author>
```

Réponse :

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

N.B. L'instruction `string`(argument) transforme son argument (ici : un élément de balise `first`, puis un élément de balise `last`), en chaîne de caractéres. Les balises `first` et `last` ne sont pas dans la réponse, qui contient juste les chaînes de caractéres correspondant aux VALEURS des 2 éléments.

Variable de POSITION : le at

Dans le `for $x in` E, on peut utiliser une variable de position qui donne la position d'un individu dans la sequence valeur de E; il faut utiliser le mot-clé `at`.
Les titres des livres, avec un attribut `pos` qui en donne la position dans la séquence des livres *selon le document* :

```
for $t at $i in doc("books.xml")//title
return <title pos="{$i}">{string($t)}</title>
```

```
<title pos="1">TCP/IP Illustrated</title>
<title pos="2">
Advanced Programming in the Unix Environment</title>
<title pos="3">Data on the Web
</title>
<title pos="4">
The Economics of Technology and Content for Digital
TV
</title>
```

Exemples de requêtes

La requête `doc("books.xml")//author/last)` produit :

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

On veut éliminer les répetitions. Si on écrit :
`distinct-values(doc("books.xml")//author/last)`
on obtient : `Stevens Abiteboul Buneman Suciu`.
Mais la balise `last` a disparu, aussi. Pourquoi ?

Exemples de requêtes

La requête `doc("books.xml")//author/last)` produit :

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

On veut éliminer les répetitions. Si on écrit :
`distinct-values(doc("books.xml")//author/last)`
on obtient : `Stevens Abiteboul Buneman Suciu`.
Mais la balise `last` a disparu, aussi. Pourquoi ?
La fonction `distinct-values()` extrait les valeurs d'une séquence de noeuds et crée une séquence de valeurs sans répétitions.

Exemples de requêtes

Comparer la requête précédente avec :

```
for $l in distinct-values(doc("books.xml")//author/last)
return <last>{ $l }</last>
```

qui donne la réponse :

```
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

Exemples de requêtes

Une requêtepeut lier plusieurs variables avec le `for` afin de combiner des informations provenant d'expressions différentes, voir de fichiers différents.

Exemple. En plus de books.xml, on va utiliser le fichier suivant, reviews.xml, qui donne des critiques de livres :

```
<reviews>
   <entry>
    <title>TCP/IP Illustrated</title>
    <rating>5</rating>
    <remarks>
    Excellent technical content. Not much plot.
    </remarks>
   </entry>
</reviews>
```

Suite de l'exemple ⇒

Suite de l'exemple

Requête : Pour chaque livre in books.xml, en donner les remarques qu'on trouve dans review.xml :

```
for $t in doc("books.xml")//title,
      $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

Réponse :

```
<review>
      <title>TCP/IP Illustrated</title>
      <remarks>
      Excellent technical content. Not much plot.
      </remarks>
</review>
```

C'est une sorte de jointure !

Usage du for

Les deux requêtes :

```
for $t in doc("books.xml")//title,
     $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

et

```
for $t in doc("books.xml")//title,
for $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

calculent la même chose : les titres des livres et les remarques, en donnant exclusivement ces livres pour les quels une critique existe.

Exemples de requêtes

Par contre, la requête :

```
for $t in doc("books.xml")//title
return
  <review>
   { $t }
   {
     for $e in doc("reviews.xml")//entry
     where $e/title = $t
     return $e/remarks
   }
  </review>
```

donne tout livre, accompagné de ses critiques quand elles existent.
D'où vient la différence entre cette requêteet les deux précédentes ?

Exemples de requêtes

Lister les livres publiés par chaque maison d'édition (*publisher*), en donnant d'abord la maison d'édition, puis tous les livres qu'elle publie.

```
<listings>
  {for $p in distinct-values(doc("books.xml")//publisher)
    order by $p
    return
        <result>
            { $p }
            {
                for $b in doc("books.xml")/bib/book
                where $b/publisher = $p
                order by $b/title
                return $b/title
            }
        </result>}
</listings>
```

Exemples de requêtes

On obtient :

```
<listings>
 <result>
     <publisher>Addison-Wesley</publisher>
   <title>Advanced Programming in the Unix Environment</title>
   <title>TCP/IP Illustrated</title>
 </result>
 <result>
   <publisher>Kluwer Academic Publishers</publisher>
   <title>The Economics of Technology and Content for
   Digital TV</title>
  </result>
  <result>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <title>Data on the Web</title>
  </result>
</listings>
```

Exemples de requêtes

Quantificateurs : Quantificateur Existentiel.
L'expression `some $x in` *E* `satisfies` C($x) permet de tester si au moins un individu dans la séquence valeur de *E* satisfait la condition C.
Quels livres ont au moins un auteur qui s'appelle W. Stevens ?

```
for $b in doc("books.xml")//book
where some $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

On obtient :

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
```

Exemples de requêtes

Quantificateurs : Quantificateur Universel.

L'expression `every $x in` *E* `satisfies` C($x) teste si tout individu dans la séquence valeur de *E* satisfait la condition C.

Quels livres sont tels que tous leur auteurs s'appellent W. Stevens ?

```
for $b in doc("books.xml")//book
where every $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
<title>The Economics of Technology and Content for Digital TV</t
```

Pourquoi on obtient aussi le dernier titre ?

Exemples de requêtes

Quantificateurs : Quantificateur Universel.

L'expression `every $x in` *E* `satisfies C($x)` teste si tout individu dans la séquence valeur de *E* satisfait la condition C.

Quels livres sont tels que tous leur auteurs s'appellent W. Stevens ?

```
for $b in doc("books.xml")//book
where every $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
<title>The Economics of Technology and Content for Digital TV</t
```

Pourquoi on obtient aussi le dernier titre ?

Ce livre n'a pas d'auteurs.

Donc, pour ce livre, `b/author` s'évalue à une séquence **vide**. Tout élément de la séquence vide satisfait la condition C, c'est une question de logique !

Exemples de requêtes

Exemple d'utilisation des quantificateurs pour reorganizer les données .

Lister les livres par auteur (on remarquera aussi les `return` imbriqués).

```
<author-list>
  { let $a := doc("books.xml")//author
    for $l in distinct-values($a/last),
        $f in distinct-values($a[last=$l]/first)
    order by $l, $f
    return
        <author>
          <name>{ $l, ", ", $f }</name>
           { for $b in doc("books.xml")/bib/book
             where some $ba in $b/author satisfies
                   ($ba/last=$l and $ba/first=$f)
             order by $b/title
             return $b/title }
        </author> }
</author-list>
```

Réponse

On obtient :

```
 author-list>
    <author>
        <name>Stevens, W.</name>
        <title>Advanced Programming in the Unix Environment</title>
        <title>TCP/IP Illustrated</title>
    </author>
    <author>
        <name>Abiteboul, Serge</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Buneman, Peter</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Suciu, Dan</name>
        <title>Data on the Web</title>
    </author>
</author-list>
```

Expressions Conditionnelles

Pour chaque livre, donner ses 2 premiers auteurs, et “et. al” s’il en a d’autres.

```
for $b in doc("books.xml")//book
return
  <book>
    { $b/title }
    {
      for $a at $i in $b/author
      where $i <= 2
      return <author>{string($a/last), ", ",
                      string($a/first)}</author>
    }
    { if (count($b/author) > 2)
      then <author>et al.</author>
      else () }
   </book>
```

NB : Le `if...then ...else` va après le `retun`

Expressions Conditionnelles: Réponse

```
 <book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Advanced Programming in the Unix Environment</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>et al.</author>
</book>
<book>
    <title>The Economics of Technology and Content for
    Digital TV</title>
</book>
```

Expressions Conditionnelles: Suite Réponse

N.B : Pour la liaison de $b au dernier livre, qui n'a pas d'auteur, l'évaluation de

```
{
      for $a at $i in $b/author
      where $i <= 2
      return <author>{string($a/last), ", ",
                      string($a/first)}</author>
}
```

a donne le résultat vide, pas un message d'erreur.

Operateurs

En XQUERY:

Operateurs Arithmetiques
Operateurs de Comparaison (de plusieurs sortes, voir après)
Operateurs sur les séquences de noeuds.

Operateurs Arithmetiques

+, -, *, `div` (division pour des types numériques quelconques), `idiv` (division sur les entiers), `mod` (modulo).

NB : 2 + (), 2 * (),...etc s'évaluent à la séquence vide (), qui se comporte comme le `NULL` de SQL.

Operateurs de Comparaison

Les ***value comparaison operators*** ne sont pas la même chose que les ***general comparison operators***, même s'il y a une correspondance entre eux :

Value Comparison Operators	General Comparison Operators
`eq`	`=`
`ne`	`!=`
`lt`	`<`
`le`	`<=`
`gt`	`>`
`ge`	`>=`

Operateurs de Comparaison, Suite

Quelles sont les differences entre le premier groupe (VCO) et le second (GCO) ?

1. Les VCO sont stricts par rapport au types : on peut comparer des strings avec des strings, des nombre decimales avec des decimales, etc.
   Les GCO sont plus souples.
2. Si un VCO *op* est appliqué à un argument qui est une séquence S de longueur supérieure à 1, on a un message d'erreur.
   Ce n'est pas le cas pour un GCO : une expression $S\ op\ a$ est lue : $\exists s \in S\ : s\ op\ a$?

Operateurs de Comparaison. Suite

Illustration de la première différence.

- Avec le VCO `gt` (*greater than*) :

```
for $b in doc("books.xml")//book
where $b/price gt 100.00
return $b/title
```

Le DTD n'a pas défini le type de `price` comme étant un décimal (avec XML-schéma on aurait pu le faire), mais un PCDATA. La comparaison `gt` avec un décimal donne une erreur.

- Avec le GCO `>` :

```
for $b in doc("books.xml")//book
where $b/price > 100.00
return $b/title
```

Puisque 100.00 est un décimal, le `>` converti le prix à un décimal.

Operateurs de Comparaison. Suite

Illustration de la seconde différence.

- Avec le VCO `eq` (*equal to*) :

```
for $b in doc("books.xml")//book
where $b/author/last eq "Stevens"
return $b/title
```

Puisque un livre peut avoir plusieurs auteurs, l'expression `$b/author/last` peut s'évaluee à une séquence d'auteurs qui contient plus qu'un élément : erreur !

- Avec le GCO = :

```
for $b in doc("books.xml")//book
where $b/author/last = "Stevens"
return $b/title
```

Pas d'erreur !
En fait, `$b/author/last eq "Stevens"` est lue comme : Existe-t-il un auteur du livre `$b` tel que son nom de famille (`last`) vaut "Stevens" ?

Operateurs de Comparaison. Suite

Mais, attention, le comportement du GCO = vis à vis d'une séquence peut reserver des surprises !

```
for $b in doc("books.xml")//book
where $b/author/first = "Serge"
  and $b/author/last = "Suciu"
return $b
```

At-on démandé quels sont livres dont un auteur s'appelle "Serge Suciu" ? (Remarquer qu'il n'y en a pas).

Operateurs de Comparaison. Suite

Non ! On obtient la réponse :

```
<book year = "2000">
  <title>Data on the Web</title>
  <author>
    <last>Abiteboul</last> <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last> <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last> <first>Dan</first>
  </author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>
```

Pourquoi ? Voir la diapositive suivante.

Operateurs de Comparaison. Suite

La condition C après le `where` :

```
      $b/author/first = "Serge"
  and $b/author/last = "Suciu"
```

a été lue :
$\exists$ un auteur $a1$ du livre $b tel que le `first` de $a1$ vaut ‘‘Serge’’ et
$\exists$ un auteur $a2$ du livre $b tel que le `last` de $a2$ vaut ‘‘Suciu’’.
Le livre Data on the Web satisfait C !

Si on veut demander quels sont livres dont un auteur s'appelle ‘‘Serge Suciu’, il faut écrire :

```
for $b in doc("books.xml")//book,
    $a in $b/author
where $a/first="Serge"
  and $a/last="Suciu"
return $b
```

Ici, on raisonne sur un auteur donné, valeur de $a.

Operateurs de Comparaison de Noeuds

Rappel : chaque noeud est identifié de façon unique par sa position dans l'arbre, son contenu ne suffit pas !

n `is` *m* teste si *n* et *m* sont le même noeud, et *n* `is not` *m* teste s'ils ne le sont pas.

Est-il vrai que le livre les plus cher coincide avec le livre qui a le nombre le plus grand d'auteurs et éditeurs ?

```
let $b1 := for $b in doc("books.xml")//book
           order by count($b/author) + count($b/editor)
           return $b
let $b2 := for $b in doc("books.xml")//book
           order by $b/price
           return $b
return $b1[last()] is $b2[last()]
```

où la fonction pre-définie `last()` calcule le dernier élément d'une séquence.

NB: Valeurs de $b1 et $b2 : les réponses de 2 FLOWR

Operateurs de Comparaison de Noeuds, suite

Observer :

$n = m$, en révanche, teste si la valeur (contenu) des deux noeuds est le même.

$n = m$ et n `is not` m peuvent être vrais au même temps !

Operateurs de Comparaison de Noeuds

Un autre operateur de comparaison de nodes est « qui teste si un noeud *n* précéde un noeud *m* dans l'ordre du document.

Quels sont les livre dont Abiteboul est bien un auteur, mais pas le premier auteur ?

```
for $b in doc("books.xml")//book
let $a := ($b/author)[1],
    $sa := ($b/author)[last="Abiteboul"]
where $a << $sa
return $b
```

réponse : il n'y en a pas.

Opérateurs sur les séquences de noeuds

`union` (ou |), `intersect` et `except` combinent 2 séquences de noeuds, et la séquence résultat suit l'ordre du document.

`union` (ou |) : union de séquences

`intersect` : intersection de séquences

`except` : différence de séquences

Opérateurs sur les séquences de noeuds

Donner la liste triée des noms de famille des auteurs et des editeurs :

```
let $l :=
distinct-values(doc("books.xml")//(author | editor)/last)
order by $l
return <last>{ $l }</last>
```

Réponse :

```
<last>Abiteboul</last>
<last>Buneman</last>
<last>Gerbarg</last>
<last>Stevens</last>
<last>Suciu</last>
```

Opérateurs sur les séquences de noeuds

Donner toutes les informations du livre TCP/IP Illustrated, sauf son prix.

```
for $b in doc("books.xml")//book
where $b/title = "TCP/IP Illustrated"
return
   <book>
       { $b/@* } { $b/* except $b/price }
   </book>
```

Réponse :

```
<book year = "1994">
<title>TCP/IP Illustrated</title>
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
</book>
```

Comment utiliser Xquery sur votre machine

Apprendre à utiliser la plateforme exists en allant sur la page :

`http://exist-db.org/exist/apps/homepage/index.html`

Il suffit de lire et suivre les instructions pour avoir le moteur de Xquery.

En TD-TP, vous allez interroger le fichier mondial.xml, après avoir lu, bien sûr, sa DTD. Vous les trouvez ici :

`http://www.dbis.informatik.uni-goettingen.de/Mondial/`

(il faut chercher les bons onglets dans cette page, qui contient plein de choses).

Fonctions Pre-définies

En XQUERY on a des fonctions `min`, `max`, `count`, `sum` et `average` analogues à celles de SQL. On a déjà vu des exemples avec `count`. Quels livres sont plus chers que la moyenne ?

```
let $b := doc("books.xml")//book
let $avg := average( $b//price )
return $b[price > $avg]
```

On obtient :

$\Rightarrow$

Fonctions Pre-définies, average

```
<book year = "1999">
    <title>The Economics of Technology and Content for
    Digital TV</title>
    <editor>
       <last>Gerbarg</last>
       <first>Darcy</first>
       <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
</book>
```

N.B : `price` est le nom d'un élément, pas un type tel que on puisse calculer le maximum de $< valeur_1(price), ..., valeur_n(price) >$, mais `average` extrait la séquence des valeurs de `price`, et fait la conversion réquise.

Fonctions Pre-définies, Autres

On a aussi des fonctions numériques comme `round`, `floor` et `ceiling`.

On a des fonctions des chaînes de caractères comme :
`concat`,
`string-length`,
`starts-with`,
`end-with`,
`substring`,
`upper-case`,
`lower-case`.

Puis : `distinct-values` et `doc` (déjà vues).

Fonctions Pre-définies, not

Et encore : `not`.

Quels sont les livres dont aucun auteur s'appelle Stevens ?

```
for $b in doc("books.xml")//book
where not(some $a in $b/author satisfies $a/last="Stevens")
return $b
```

Fonctions Pre-définies, empty

La fonction `empty` teste si une séquence est vide.
Quels sont les livres qui ont au moins un auteur ?

```
for $b in doc("books.xml")//book
where not(empty($b/author))
return $b
```

On aurait pu aussi écrire :

```
for $b in doc("books.xml")//book
where exists($b/author)
return $b
```

Fonctions Pre-définies, string et data

La fonction `string`, appliquée à un noeud, renvoie la réprésentation sous forme de chaîne de caractères du texte trouvé dans le noeud . Par exemple, la requête :

```
string((doc("books.xml")//author)[1])
```

calcule la chaîne “`Stevens W.`” (le nom du premier auteur dans le document).

La fonction `data`, appliquée à un noeud, renvoie la valeur typée d'un noeud; exemples d'utilisation de cette fonction dans le TD.

Fonctions définies par l'utilisateur

On peut définir une fonction qui calcule une requête donnée, et la réutiliser. Par exemple :

```
define function books-by-author($last, $first)
  as element()*
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
          ($ba/last=$last and $ba/first=$first)
  order by $b/title
  return $b/title
}
```

définie une fonction qui qui liste les livres par auteur, et la nomme `books-by-author`.
(Pas utilisée en TD).

Une Etude de Cas

On va modéliser un même ensemble d'informations à l'aide des 2 modèles: relationnel et semi-structuré.

But : illustrer analogies et différences entre les deux approches, limites respectives, etc.

La BD des BD

On veut modéliser les informations suivantes, pour en faire une base de données :

Une bande dessinnée a un ou plusieurs auteurs, qui sont scénaristes ou/et dessinateurs et/ou coloristes de l'ouvrage en question. Elle a un unique éditeur, un unique titre, un unique ISBN, un unique format et un unique prix.
Les mangas sont des bandes dessinnées particulières, qui n'ont pas de coloriste mais ont un attribut supplémentaire, sens_de_lecture, qui vaut gauche_droite ou droite_gauche.
Un auteur a un nom, **parfois** un prénom, une ou plusieurs adresses, **parfois** une thématique, et **parfois** il est sous contrat d'exclusivité avec un éditeur.
Un éditeur a un nom, une adresse et **parfois** a des auteurs sous contrats.
Les adresses sont constituées d'un numéro, d'un nom de rue, d'une ville et d'un pays.

Une Modélisation possible Relationnelle

Ce schéma relationnel a 11 tables : BD, BD_pas_Mangas, Mangas, Adresses, Infos_Auteur, Auteur_Adresses, Scénaristes, Dessinateurs, Coloristes, Editeurs, Sous_Contrat.

Ce n'est pas le seul schéma possible, mais il a une bonne propriété : Chaque schéma de table est en BCNF (*Forme Normale de Boyce Codd*)=tout attribut d'une table dépend exclusivement de la clé de la table. C'est une *forme normale* qui évite au max les rédondances de données.

Résondances $\Rightarrow$ perte d'éspace, source d'erreurs de mise à jour, etc. A éviter, dans la mésure du possible ! (cours classique de SGBD relationnelles).

NB : Pour chaque schéma de table les attributs appartenant à sa clé primaire seront soulignés.

Une Modélisation possible Relationnelle

BD :

ISBN	format	larg	prix	Ident_edit	Manga

Domaine de Manga: $\{\text{“}Oui\text{”}, \text{“}Non\text{”}\}$.

Dépendances Fonctionnelles : toutes celles de la forme ISBN $\rightarrow A$, où A est tout autre attribut que ISBN.
Ce schéma de table est en BCNF : les seules dépendances fonctionnelles significatives sont celles à partir de la clé.

Une Modélisation possible Relationnelle

MANGAS :

ISBN	Sens_Lecture

Domaine de Sens_Lecture : { DG, GD } (de droite à gauche, ou l'inverse).

Evidemment, cette table est en BCNF.

Une Modélisation possible Relationnelle

PAS_MANGAS :

ISBN	Code_de_pas_Manga

Ici, ISBN, la clé de *BD*, est une clé étrangère.

Code_de_pas_Manga est un idéntifiant propre à ces bandes déssinnées qui ne sont pas des Mangas (donc qui admettent un coloriste).

Evidemment, cette table est en BCNF.

Une Modélisation possible Relationnelle

Infos_Auteur :

Id_Aut	nom	prenom	thematique

Ce schéma de table est en BCNF.

Une Modélisation possible Relationnelle

Adresses :

Code_Postal	Num_Civique	rue	ville	pays

Pour simplifier, on supposera ici que le Code Postal permette de retrouver **tous** les attributs, donc que c'est la clé primaire.

Et nous avons alors la BCNF, car il n'y a pas de dépendance fonctionnelle (significative) $X \rightarrow Att$ où X ne soit pas la clé.

Une Modélisation possible Relationnelle

Auteur_Adresses :

Id_Aut	Code_Postal

NB : Le fait que les deux attributs soulignés appartiennent à la clé (primaire), qui est donc { Id_Autn Code_Postal } est du à l'hypothèse qu'un auteur peut avoir plusieurs adresses (voir le texte) mais aussi à l'hypothèse que deux auteurs peuvent vivre à la même adresse. Cette dernière hypothèse n'était pas explicite dans le texte décrivant la situation à modéliser. C'est un **choix** de l'auteur de la modèlisation (S. Cerrito).

Ici, la BCNF est banale, puisque tout attribut de la table fait partie de la clé.

Une Modélisation possible Relationnelle

Scenaristes :

Id_Aut	ISBN

NB : Le fait qu'il faut que les deux attributs appartiennent à la clé (primaire) est du à l'hypothèse qu'un auteur peut être scénariste de plusieurs BD, et une BD donnée peut avoir plusieurs auteurs, donc, en particulier, plusieurs scénaristes.

A nouveau, la BCNF est banale, ici..

Une Modélisation possible Relationnelle

Dessinateurs :

Id_Aut	ISBN

NB : Le fait qu'il faut que les deux attributs appartiennent à la clé (primaire) est du à l'hypothèse qu'un auteur peut dessiner plusieurs BD, et une BD donnée peut avoir plusieurs auteurs, donc, en particulier, plusieurs dessinateurs.

A nouveau, la BCNF est évidente.

Une Modélisation possible Relationnelle

Coloristes :

Id_Aut	Code_de_pas_Manga

Comme pour *Scenaristes* et *Dessinateurs* mais, attention, ici Code_de_pas_Manga remplace ISBN. Cela, car seulement une BD qui n'est pas une MANGA a un coloriste.

Une Modélisation possible Relationnelle

Infos_Editeur :

Id_Ed	nom	prenom	Code_Postal

On peut prendre Id_Ed comme clé car un éditeur a **une et une seule** adresses (à la différence d'un auteur).

Toujours la BCNF.

Une Modélisation possible Relationnelle

Sous_Contrat :

Id_Aut	nom	prenom	Id_Ed

On peut prendre Id_Aut comme clé car on a dit que si un auteur a un contrat avec un éditeur, alors c'est un contrat d'exclusivité.

Toujours la BCNF.

Une Modélisation possible Relationnelle : Pour et Contre

Observer qu on peut repérer pour quelle bd un auteur donné est seulement scénariste et pour quelle autre est aussi dessinateur, par exemple.

Et on a exprimé aussi la contrainte que les mangas n'ont pas de coloriste.

Une Modélisation possible Relationnelle : Pour et Contre

On n'exprime pas les "parfois" :

- On n'a pas exprimé qu'un auteur a parfois une thématique (donc il peut aussi ne pas en avoir);
- On n'a pas exprime qu'un auteur a parfois un prénom.

Pour pallier ces problèmes, il faudrait admettre des valeurs nuls. Remarquer que le modèle relationnel "propre" ne veut pas de valeurs nuls !

Alternative : à la place de *Infos_Auteur*(Id_Aut,nom, prenom,thematique),
3 tables : *Infos_Base_Auteur*(Id_Aut,nom), *Infos_Pren_Auteur*(Id_Aut, prenom) et *Infos_thematiques*(Id_Aut, thematique). Dans la deuxième table, seulement les auteurs dont on connaît le prénom, dans la troisième, seulement les auteurs ayant une thématique.
Ceci dit, Le fait de ne savoir pas exprimer : *l'attribut A a 0 ou 1 valeurs* est intrinsèque au modèle relationnel !

Une Modélisation possible Relationnelle : Pour et Contre

On n'a pas exprimé, non plus, que *tout* auteur a *au moins une adresse.*

C'est à nouveau, un cas de *0 ou 1* valeurs pour une propriété ! L'expression de ce type de contraintes est en général problèmatique, en relationnel.

Une modélisation semi-structurée (XML)

Une DTD que l'on peut écrire : $\Rightarrow$

Une DTD possible (=schema)

```
<!ELEMENT BD (bande_dessinnée+, auteur+,editeur+, adresse+)>
<!ELEMENT bande_dessinnée (manga|pas-manga)>
<!ELEMENT pas_manga  (titre,format,prix)>
<!ATTLIST pas_manga code_pas_manga ID #REQUIRED
ISBN PCDATA #REQUIRED  a_scénariste IDREFS #REQUIRED
a_dessinateur IDREFS #REQUIRED  a_coloriste IDREFS #IMPLIED
a_editeur IDREF #REQUIRED>
<!ELEMENT manga  (titre,format,prix)>
<!ATTLIST  ISBN ID #REQUIRED  a_scénariste IDREFS #REQUIRED
a_dessinateur IDREFS #REQUIRED a_editeur IDREF #REQUIRED
sens_lecture (GD|DG)  #REQUIRED>
<!ELEMENT titre PCDATA>
<!ELEMENT format (long,larg)>
<!ATTLIST format unité_mésure CDATA #REQUIRED>
<!ELEMENT prix PCDATA>
<!ATTLIST prix dévise CDATA #REQUIRED>
<!ELEMENT long PCDATA>
<!ELEMENT larg PCDATA>
```

Une DTD possible, suite

```
<!ELEMENT auteur (nom,prenom?,thématique?)>
<!ATTLIST auteur  Id_Aut ID #REQUIRED
scenariste_de  IDREFS #IMPLIED
dessinateur_de  IDREFS #IMPLIED
coloriste_de  IDREFS #IMPLIED
contrat_avec_ed IDREF #IMPLIED
a_adresse IDREFS  #REQUIRED>
<!ELEMENT nom PCDATA>
<!ELEMENT prenom PCDATA>
<!ELEMENT thématique PCDATA>
<!ELEMENT adresse (numéro, rue, ville, pays)>
<!ATTLIST adresse code_postal ID #REQUIRED
adresse_de_auteur IDREFS #IMPLIED
adresse_de_editeur IDREFS #IMPLIED>
<!ELEMENT numéro PCDATA>
<!ELEMENT rue PCDATA>
<!ELEMENT ville PCDATA>
<!ELEMENT pays PCDATA>
<!ELEMENT editeur (nom,adresse)>
```

Une DTD possible, suite

```
<!ATTLIST editeur  Id_Ed ID #REQUIRED édite  IDREFS #REQUIRED
contrat_avec_aut IDREFS #IMPLIED a_adresse IDREF #REQUIRED >
```

Une modélisation semi-structurée : POUR ET CONTRE

On exprime :

- Les mangas sont des BD;
- La contrainte qu'un auteur a forcement au moins une adresse . Ici, grâce au REQUIRED de l'attribut a_adresse. Et si on avait choisi de poser adresse comme *fils* de auteur, alors il aurait suffi d'écrire adresse$^+$.
- La contrainte qu'une Manga n'a jamais de coloriste.
- Pour une bd un auteur est seulement scénariste et pour une autre est aussi dessinateur, par exemple.

Une modélisation semi-structurée (XML) : POUR ET CONTRE

On exprime les "parfois" :

- Un auteur a parfois une thématique (fils `thématique?` de `auteur`);
- Un auteur a parfois un prénom (`prénom?`).
- Un auteur a parfois un contrat (L'attribut `contrat_avec_ed` de `auteur` est optionnel)

N.B. La possibilité de déclarer `balise?` dans le DTD permet d'exprimer qu'une relation lie un individu x à 0 ou 1 individus, chose que le relationnel ne permet pas de faire.

Une modélisation semi-structurée : limites intrinséques

A-t-on exprimé la contrainte que la cible d'un lien `a_scénariste` doit forcement être un noeud `auteur` ? (même question pour `a_dessinateur` et `a_coloriste`).

Non : on peut écrire un document xml où, pour une bd, `a_scénariste` pointe à une autre bd, par exemple, et le document sera validé !

En général, dans un DTD on ne peut pas spécifier la balise attendue pour la cible d'un attribut de type IDREF ou IDREFS !

Les DTD sont des grammaires d'ARBRES !

Valider un document XML : Les outils

Plusieurs outils de validation d'un document xml par rapport à une DTD.

Par exemple, une recherche google avec les mots clés :

```
validate xml file using dtd
```

m'a donné ces réponses (et autres encore)

https://www.xmlvalidation.com

https://www.oxygenxml.com/doc/versions/18.0/ug-editor/topics/validating-XML-documents-against-schema.html

https://support.microsoft.com/en-us/kb/315533

https://www.youtube.com/watch?v=rMcPQpAJGJI

Valider un document XML : Les principes

Base des algorithmes sous-jacents les plusieurs otils de validation d'un document XML par rapport à une DTD :

1. Un document XML est un arbre (on ignore les références !)
2. Une DTD est une grammaire d'arbres, specifiant un langage d'arbres (finis), c'est à dire un ensemble d'arbres. Plus exactement, elle est une *Regular Tree Grammar* (RTG) ;
3. Pour reconnaitre si un arbre donné fait partie du langage des arbres généré par une DTD donnée, on peut utiliser des automates d'arbres (finis)

Préiminaires

Rappel : si E est un ensemble de symboles, l'ensemble des termes qu'on peut construire à partir de E est defini recursivement par : tout symbole de E qui a zero arguments est un terme, et si $t_1, ..., t_n$ son des termes et f est un sumbole de E à n arguments, alors $f(t_1, ..., t_n)$ est un terme.

Par exemple, si $E = \{0,s\}$ avec 0 qui a zero argument, et s qui a un argument, alors l'ensemble des termes qu'on peut construire à partir de E est : $\{0, s(0), s(s(0)), s(s(s(0)))....\}$.

RTG

Une RTG est un quadruplet $\langle S, N, F, R \rangle$ où:

- S est un symbole de départ (start) ;
- N est un ensemble de symboles non-terminaux, et $S \in N$;
- F est un ensemble de symboles terminaux, disjoint de N ;
- F est un ensemble de règles de production, de la forme : $n \to T(N \cup F)$ où $n \in N$ et $T(N \cup F)$ est l'ensemble des termes qu'on peut construire à partir des symboles de F et N.

NB : On suppose que tout symbole $\in N \cup F$ a un nombre d'arguments *fixé*, et que tout symbole $\in N$ a zero arguments (càd est une constante).

RTG

Exemple simple de RTG

La RTG $\langle S, N, F, R \rangle$ où $S =$ List, N contient List et Bool, F contient *true* *false* et *nil* à zero arguments, cons à deux arguments et F composé des règles :

$List \rightarrow nil$
$List \rightarrow cons(Bool, List)$
$Bool \rightarrow false$
$Bool \rightarrow true$

génére l'ensemble de toutes les listes (finies) de booléens.

NB : chaque liste peut être vue comme un arbre. Pourquoi ?

RTG

Exemple simple de RTG

La RTG $\langle S, N, F, R \rangle$ où $S =$ List, N contient List et Bool, F contient *true* *false* et *nil* à zero arguments, cons à deux arguments et F composé des règles :

$List \rightarrow nil$
$List \rightarrow cons(Bool, List)$
$Bool \rightarrow false$
$Bool \rightarrow true$

génére l'ensemble de toutes les listes (finies) de booléens.

NB : chaque liste peut être vue comme un arbre. Pourquoi ?

Exemple au tableau de génération de la liste [*true*, *false*, *false*]

Variation de RTG

Une DTD peut être vu comme une grammaire d'arbres qui est une variante des RTG. Ici, une grammaire reste un 4-plet $\langle S, N, F, R \rangle$ mais :

- L'ensemble des symboles terminaux F est $\Sigma \cup D$ où Σ est un ensemble fini de types d'éléments (les balises !) et D est un ensemble fini de types de données. Un symbole de Σ peut avoir un nombre variable d'arguments.
- Chaque règle de production de F a la forme : $n \to a(r)$ où :
  - $n \in N$
  - $a \in \Sigma$
  - r est un élément de D ou bien une expression régulière construite à partir de N

Rappel. Les opérators réguliers sont : , (la concatenation), |, +, *, ?.

La différence essentielle par rapport à une RTG est que l'espression $a(r)$ dans une regle $n \to a(r)$ peut contenir des symboles ayant un nombre d'arguments variable. Voir l'exemple suivant.

RTG

Exemple de DTD vu comme RTG

```
<! ELEMENT body (paper*)>
<! ELEMENT paper(title,athor*,journal?)>
<! ELEMENT title (#PCDATA)>
<! ELEMENT author (#PCDATA)>
<! ELEMENT journal (#PCDATA | EMPTY)>
```

La RTG correspondante :
$F = \{$body, paper,title,author,journal$\}$, $D = \{$ # PCDATA, $\epsilon\}$,
$N = \{$nb,np,nt,na, nj$\}$, S=nb et R contient les règles :

nb $\rightarrow$ body(np*)
np $\rightarrow$ paper(nt,na*,nj?)
nt $\rightarrow$ title(# PCDATA)
na $\rightarrow$ author(# PCDATA)
nj $\rightarrow$ journal(# PCDATA)
nj $\rightarrow$ journal(ϵ)

NB : Dans la règle nb $\rightarrow$ body(np*) le symbole body a un nombre d'arguments **variable**.

Des grammaires d'arbre aux automates d'arbre

- Une grammaire d'arbres peut être convertie en automate d'arbres finis, qui sera reconnaitre si un arbre donné est généré par la grammaire.
- Il y a plusieurs sortes d'arbres finis qu'on peut vouloir reconnaître :
  - L'arbre a reconnaître est *ranked* : tout noeud avec une etiquette donnée a un nombre fixe de fils.
  - L'arbre a reconnaître est *unranked* : un noeud avec une étiquette donnée peut avoir un nombre variable de fils.
    NB: Les arbres XML sont *unranked* ! Par exemple : si dans la DTD d'un document XML on a déclaré le motif (ville)* pour l'élément villes, un noeud avec balise villes peut avoir une nombre variables de fils.
  - Deux sortes d'automates pour les arbres finis *ranked* : *bottom-up* et *top-down*

Des grammaires d'arbres aux automates d'arbre

- Les automates d'arbres *bottom-up* lisent un arbre à partir des feuilles et en remontant vers la racine ;
- Les automates d'arbres *topdown* partent de la racine et descendent vers les feuilles.
- Les documents xml sont des arbres *unranked*. La définition d'automates d'arbres appropriés à des arbres *unranked* est assez technique.
- Choix de ce cours : On présentera seulement de automates pour les arbres finis *ranked*. Et on se limitera aux automates *bottom-up*.

Raisons du choix fait

- La théorie des automates adaptés aux arbres *unranked* a été developpée à partir de celle des automates pour les arbres *ranked* : donc ceux-ci sont un bon point de départ.
- Le sujet des automates pour des arbres *unranked* est assez technique.
- On peut toujours transformer un arbre *unranked* en un arbre *ranked* : plus sur cela après.
- Pour les arbres *ranked*, on ne per pas de généralité si on se limite à considerer les automates *bottom-up* : plus sur cela après.

Automates bottom up pour les arbres finis (*ranked*)

Un automate (non-deterministe) *bottom up* est un 4-ple $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ où :

- Σ est un ensemble fini de symboles de fonction, chacun avec son nombre n d'arguments où $n \geq 0$ (si $n = 0$ alors on dit que le symbole est une constante).
- Q est un ensemble fini d'états et $F \subseteq Q$ est un ensemble d'états acceptants ;
- Δ est un ensemble de règles de transition de la forme : $f(q_1, ..., q_n) \to q$, où $f \in \Sigma$, $q_1, ..., q_n, q \in Q$.

Idée : On lit un arbre T à partir des feuilles. Quand on lit un noeud n :

- Si n est une feuille et son étiquette est a, une règle $r : a \to q$ de Δ associera à n un état q ;
- Si n est un noeud interne et et son étiquette est f, une règle $r : f(q_1, ..., q_n) \to q$ de Δ, où $n \geq 1$, associera à n un état q si on a déjà associé les états $q_1, ..., q_n$ à ses n fils.

Automates bottom up pour les arbres finis (*ranked*)

L'automate accepte un arbre T quand à la fin de la lecture la racine est associée à un état acceptant

Automates bottom up : un exemple

L'automate $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ où

- Σ contient les costantes : *false*, *true*, *nil* et la fonction à deux arguments *cons* ;
- $Q = \{Bool, List\}$ et $F = \{List\}$
- Δ contient les règles : $false \to Bool$, $true \to Bool$, $nil \to List$, $cons(Bool, List) \to List$

accepte la liste $[true, false, false]$ (vue comme arbre) mais refuse $[5, false, false]$.

Automates bottom up : un exemple

L'automate $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ où

- Σ contient les costantes : *false*, *true*, *nil* et la fonction à deux arguments *cons* ;
- $Q = \{Bool, List\}$ et $F = \{List\}$
- Δ contient les règles : $false \to Bool$, $true \to Bool$, $nil \to List$, $cons(Bool, List) \to List$

accepte la liste $[true, false, false]$ (vue comme arbre) mais refuse $[5, false, false]$. Déroulement au tableau

Automates bottom up : un exemple

L'automate $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ où

- Σ contient les costantes : *false*, *true*, *nil* et la fonction à deux arguments *cons* ;
- $Q = \{Bool, List\}$ et $F = \{List\}$
- Δ contient les règles : $false \to Bool$, $true \to Bool$, $nil \to List$, $cons(Bool, List) \to List$

accepte la liste [*true*, *false*, *false*] (vue comme arbre) mais refuse [5, *false*, *false*]. Déroulement au tableau

On remarquera l'analogie avec la RTG qui génére les listes de booleans.
Les non-terminaux de la grammaire sont dévenus les états, le symbole de start est dévenu l'état acceptant, les terminaux les éléments de Σ, et on a renversé les flêches !

Propriétés

- **Théorème**. Soit T n'importe quel arbre fini *unranked*. T peut être génére par une RTG si et seulement si il peut être accepté par un automate d'arbre finis *unranked bottom-up*.
- Même si un arbre XML est *unranked*, il peut être codé par un arbre *ranked*, qui aura plus de noeuds que l'arbre de départ. L'idée est que dans l'arbre *ranked* chaque noeud interne aura deux pointeurs : un qui va à son premier fils, l'autre qui va au premier frère.

Ces propriétés justifient le choix du cours de présenter seulement les automates d'arbres finis *unranked*, de style t *bottom-up*

Sources pour la partie *Validation de documents* du cours

- Le livre *Tree Automata Techniques and Applications*. Distribution libre :

  `http://tata.gforge.inria.fr`

- Des diapositives de cours de Master d'un des auteurs du livre ci-dessus :

  `http://www.lsv.ens-cachan.fr/~ciobaca/TATA-1-terms.pdf`

- Juste quelques exemples à partir de l'article de Boris Chidlovskii : *Using Regular Tree Automata as XML schemas* :

  `https://www.researchgate.net/publication/2948963_Using_Regular_Tree_Automata_as_XML_schemas`

- Juste quelques remarques des diapositives du cours de Master :

  `http://www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/MasterBlois-Automata.pdf`