République Algérienne Démocratique et Populaire
Ministre de l'enseignement supérieur et de la recherche scientifique

UNIVERSITE MENTOURI DE CONSTANTINE
FACULTE DES SCIENCES DE L'INGENIEUR
DEPARTEMENT D'INFORMATIQUE

N° d'ordre :…..
Série :…..

MEMOIRE

Présenté pour l'obtention du diplôme de
Magister en Informatique

Validation des requetes de mise à jour dans les bases de données XML natives

---

Présenté par :

Telghamti Samira

Dirigé par :

Pr. Boufaida .M

Soutenu le: .. /.. /2007

)

Sommaire

INTRODUCTION GENERALE **5**

1. CONTEXTE 5
2. PROBLEMATIQUE ET OBJECTIFS DU TRAVAIL 6
3. ORGANISATION DU MEMOIRE 7

CHAPITRE I : XML ET LES BASES DE DONNEES **8**

1. INTRODUCTION 8
2. STRUCTURE DU DOCUMENT XML 8
   2.1 Contenu orienté données *10*
   2.2 Contenu orienté document *11*
3. STOCKAGE DES DOCUMENTS XML 11
4. STOCKAGE DES CONTENUS ORIENTES DONNEES 12
   4.1 Correspondance basée sur des tables *14*
   4.2 Correspondance basée sur un modèle objet /relationnel *15*
   4.3 Génération de schéma XML à partir de schéma relationnel et vice-versa *15*
   4.3.1 Génération de schéma relationnel à partir de schéma XML *15*
   4.3.2 Génération de schéma XML à partir de schéma relationnel *18*
5. STOCKAGE DES CONTENUS ORIENTES DOCUMENTS 19
   5.1 Stockage des documents sur un système de fichiers *19*
   5.2 Stockages des documents dans des BLOB(s) *19*
   5.3 Stockage des documents dans des bases de données XML natives *20*
6. CONCEPTION D'UNE BASE DE DONNEES XML 21
7. DIFFERENTS TYPES DE LANGAGES DE REQUETES 23
   7.1 Langages de requêtes basés sur des modèles *23*
   7.2 Langages de requêtes basés sur SQL *24*
   7.3 Langages de requêtes XML *25*
      7.3.1 XML Path Language (XPath) 25
      7.3.2 XML Query Language (XQuery) 25
      7.3.3 Optimisation des requêtes XML 26
8. EXEMPLES DES BASES DE DONNEES XML 26
9. CONCLUSION 28

CHAPITRE II : CARACTERISTIQUES DES BASES DE DONNEES XML NATIVES **29**

1. INTRODUCTION 29
2. DEFINITION D'UNE BASE DE DONNEES XML NATIVE 29
3. PROPRIETES D'UNE BASE DE DONNEES XML NATIVE 30
   3.1 Collections de documents *30*
   3.2 Langages de requêtes *30*
   3.3 Mises à jour et suppression *30*
   3.4 Transaction et verrous *30*
   3.5 Application programming interface *31*
   3.6 Round-tripping *31*
   3.7 Mise à jour des données distantes *31*
   3.8 Références externes *31*
   3.9 Index *32*
   3.10 Normalisation *32*
   3.11 Intégrité référentielle *32*
4. CLASSIFICATION DES BASES DE DONNEES XML NATIVES 33
   4.1 Bases de données XML natives basées sur le texte *33*
   4.2 Bases de données XML natives basées sur le modèle *33*
5. DOMAINES D'UTILISATION DES BASES DE DONNEES XML NATIVES 34
   5.1 Intégration des données *34*
      5.1.1 Architectures de requêtes 35
      5.1.2 Maintien des schémas hétérogènes 35
   5.2 Evolution de schémas *36*
6. AVANTAGES ET INCONVENIENTS DES BASES DE DONNEES XML NATIVES 36
7. SCHEMAS DES BASES DE DONNEES XML NATIVES 38

7.1 DTD *39*
7.1.1 Déclarations d'éléments 39
7.1.2 Déclarations d'attributs 40
7.2 XML Schema *41*
7.2.1 Structure de base 42
7.2.2 Déclarations d'éléments 42
7.2.3 Déclarations d'attributs 45
7.2.4 Espaces de nom 46
7.3 Choix du schéma à utiliser *46*
8. MISE A JOUR ET VALIDATION DES DOCUMENTS XML 47
8.1 Validation statique des documents XML *48*
8.1.1 DOM 49
8.1.2 SAX 50
8.1.3 Comparaison entre DOM et SAX 51
8.2 Validation incrémentale des requêtes de mise à jour *51*
9. CONCLUSION 54
CHAPITRE III : UNE APPROCHE DE VALIDATION INCREMENTALE DES DOCUMENTS XML 55
1. INTRODUCTION 55
2. APERÇU GENERAL DE L'APPROCHE PROPOSEE 56
2.1 Modèle arbre du document XML *56*
2.2 Contraintes d'intégrité référentielles *57*
2.3 Contraintes structurelles *58*
3. SPECIFICATION DES CONTRAINTES 60
3.1 SPECIFICATION DES CONTRAINTES STRUCTURELLES 60
3.2 Spécification des contraintes d'intégrité référentielles *62*
4 VALIDATION DE L'OPERATION DE SUPPRESSION : SUPPRIMER(X) 64
4.1 Vérification des contraintes structurelles *64*
4.2 Vérification des contraintes d'intégrité référentielle *65*
4.3 Algorithmes pour la validation de l'opération de suppression *66*
5. VALIDATION DE L'OPERATION D'INSERTION 67
5.1 Validation de l'opération d'insertion: inserer(x,y) *67*
5.1.1 Vérification des contraintes structurelles 68
5.1.2 Vérification des contraintes d'intégrité référentielles 68
5.2 Validation de l'opération d'insertion: insérer_après(x,y) *69*
5.3 Algorithmes pour la validation de l'opération d'insertion *70*
6. ETUDE DE COMPLEXITE DES ALGORITHMES 72
7. DISCUSSION 73
8. CONCLUSION 74
CHAPITRE IV : ETUDE DE CAS : GESTION DES DOCUMENTS UNIVERSITAIRES 75
1. INTRODUCTION 75
2. ENONCE 75
3. EXEMPLES DE REQUETES DE MISE A JOUR 79
3.1 Requêtes de suppression *79*
3.2 Requêtes d'insertion *80*
4. CONCLUSION 83
CONCLUSION GENERALE 84
1. BILAN DU TRAVAIL 84
2. PERSPECTIVES 85
BIBLIOGRAPHIE 86

Liste des figures

Fig.1.1 Document XML 8
Fig.1.2 Décomposition et recomposition des documents XML 13
Fig.1.3 Arbre représentant un document XML.. 16
Fig.1.4 DTD et le sous graphe correspondant. 18
Fig.1.5 Stockage des documents XML dans des BLOBs.. 20
Fig.1.6 Document XML et sa structure d’arborescence.. 21
Fig.2.1 Processus de la validation statique [4].. 48
Fig.2.2 Arbre DOM du document XML 49
Fig.2.3 Exécution de l’automate sur l’arbre XML.. 53
Fig.3.1 Exemple d’un document XML 57
Fig.3.2 Arbre XML marqué.. 64
Fig.4.1 Exemple d’un document XML 75
Fig.4.2 Document XML Schema 76
Fig.4.3 Fragment d’un document XML Schema 77
Fig.4.4 Arbre représentant un document XML marqué 78
Fig.4.5 Validation de la requête supprimer(étudiant) 79
Fig.4.6 Requête insérer(université, enseignant).. 80
Fig.4.7 Processus de validation de la requête insérer(université,enseignant).. 80
Fig.4.8 Requête insérer (département, centre_recherche) 81
Fig.4.9 Requête insérer_après(adresse, encadre) 82
Fig.4.10 Validation de la requête insérer_après(adresse, encadre).. 82

Liste des tableaux

Tableau 1.1 Table des relations 17
Tableau 1.2 Table des valeurs.. 17
Tableau 1.3 Bases de données XML du marché.. 27
Tableau 2.1 Déclaration d’éléments dans une DTD 40
Tableau 2.2 Liste des indicateurs d’occurrences.. 45
Tableau 3.1 Table d'éléments.. 61
Tableau 3.2 Table d'attributs.. 62
Tableau 3.3 Table des clés.. 63
Tableau 3.4 Table des clés étrangères 63
Tableau 4.1 Table d'éléments.. 77
Tableau 4.2 Table des clés.. 78
Tableau 4.3 Table des clés étrangères 78

Introduction générale

1. Contexte

C'est au début de l'année 1999 que le XML *eXtended Markup Language* [16], langage de description et de structuration de données, a commencé à apparaître au sein des notices techniques de certaines solutions d'entreprise. A l'époque, les premiers éditeurs à s'y intéresser se recrutent principalement dans le domaine de la gestion de contenu et dans celui des standards d'échanges B2B (*Business to Business)*. Dans les années qui ont suivi, cette sphère d'application a évolué vers le segment de l'intégration : d'un simple langage de balisage de données, XML se voit enrichi de briques (les schémas XML), pour travailler plus intimement avec les applications.

Les documents XML doivent, comme n'importe quel type d'informations au format électronique être stockés dans un référentiel pour être répertoriés. Plus communément appelée "base de données", cette catégorie de solutions a pour but de classer les contenus en vue d'en faciliter la gestion depuis des applications tierces.

Il existe deux type de documents XML : les contenus orientés données (*data centric document*) et les contenus orientés documents (*document centric document*). Les contenus orientés données utilisent XML en tant qu'un vecteur de données, ils sont conçus pour être exploités par les machines, tel que : les prévisions de vols, tandis que les contenus orientés documents sont conçus pour être utilisés par des humains, tel que : les messages électroniques.

En règle générale (mais qui n'est pas absolue), les données sont stockées dans une base de données traditionnelle (relationnelle ou orienté objet). Cela peut être réalisé à l'aide d'un logiciel intermédiaire (middleware) ou par la base elle-même, dans ce cas la base de données est qualifiée de compatible XML ''XML–Enabled''. Les documents sont alors stockés dans une base de données XML native [1,12, 11, 15, 17].

Une base de données XML native, définit un modèle logique pour le document XML, et en fonction de ce modèle, elle stocke et retrouve le document. L'unité fondamentale de stockage est le document XML. L'unité de stockage physique n'est pas spécifique, elle peut être bâtie sur les bases des données traditionnelles (relationnelle ou orienté objet) ou bien utilisée d'autre techniques de stockage, telle que les fichiers indexés ou compressés [1, 18,32]. L'avantage de la solution native, est que l'utilisateur n'effectue pas une transformation de son document vers une autre structure, il va juste insérer des données XML, et les retrouver en

format XML, spécialement lorsque la structure du document XML est complexe, ce qui rend sa transformation vers des bases de donnés structurées difficile ou impossible.

2. Problématique et objectifs du travail

La base de données XML native souffre de certaines limitations fonctionnelles, ce qui renforce les arguments en faveur de l'utilisation des bases de données relationnelles. Ces inconvénients peuvent se résumer dans les points suivants :

- Les langages de mise à jour proposés traitent des commandes simples et ne réalisent pas la jointure entre les documents dans les requêtes de mise à jour [2].
- L'absence d'un support pour la validation des opérations de mise à jour et de préservation des contraintes [2, 7,8].
- Les bases de données XML natives ne permettent pas aux utilisateurs d'interroger simultanément des documents XML, et d'autres données stockées dans une base de données relationnelle [44].

Les travaux de recherche à l'heure actuelle, se focalisent sur la proposition des solutions aux problèmes liés à la gestion des documents XML stockés sous forme native, telles que la proposition des langages de mise à jour [3, 6,10], et des techniques pour la validation des contraintes lors des opérations de mise à jour [5, 8,7, 49,4].

Les contraintes qui doivent être respectées par un document XML, peuvent être décrites à l'aide du langage XML Schema ou la DTD (*Document Type Definition*). Un document XML est un document valide s'il respecte toutes les contraintes imposées par le schéma associé. Lors d'une mise à jour d'un document XML valide, la vérification de la validité du document résultant est nécessaire. Pour cela, la plupart des applications XML utilisent la validation dites *from scrach*, qui consiste à parcourir tout le document XML en visitant tous les nœuds de l'arbre. Cette technique et non efficace, car elle induit un coût élevé de l'opération en terme du temps de validation, en particulier lorsque les mises à jour sont fréquentes, ou bien la taille du document est importante. Du fait de l'inefficacité de cette approche, les travaux de recherche à l'heure actuelle se focalisent à proposer des approches de validation incrémentale [5, 8, 4,49], où seules les parties concernées par la mise à jour sont re-vérifiées. Le processus de la mise à jour des documents XML nécessite un mécanisme de validation incrémentale, afin de maintenir l'état cohérent des documents XML modifiés. Ceci est assuré par le rejet de toute requête de mise à jour menant à une violation des contraintes imposées par le schéma XML, et l'acceptation de celles qui les maintiennent.

Nous envisageons à travers ce travail d'atteindre l'ensemble des objectifs suivants :

- Etudier les différents modèles de stockage des documents XML.
- Etudier les caractéristiques des bases de données XML natives, les technologies qui lui sont associées, ainsi que leurs performances et leurs limitations.
- Se focaliser sur l'étude des techniques de validation des requêtes de mise à jour, en présentant une étude détaillée de l'approche de validation statique, et l'approche de validation incrémentale avec quelques travaux qui fondent sur son principe.
- Proposer une approche de validation incrémentale des documents XML, pour la préservation des contraintes spécifiées en XML Schema, lors des opérations d'insertion et de suppression des sous arbres.

3. Organisation du mémoire

Ce mémoire est organisé en 4 chapitres :

Dans le premier chapitre, nous présentons tout d'abord, les différents modèles de stockage de données XML. Nous décrivons par la suite les critères à prendre en considération dans le choix de la base de données XML à utiliser. Ainsi nous présentons les architectures des langages de requêtes XML. Nous terminons le chapitre par quelques exemples de base de données XML.

Le deuxième chapitre présente une étude détaillée des bases de données XML natives, par exposition des caractéristiques, des domaines d'utilisations et des types d'architectures de ces bases de données, ainsi que leurs avantages et leurs inconvénients. Ce chapitre présente, également les schémas des bases de données XML natives, (DTD, XML Schema) ainsi que les approches de validation des requêtes de mise à jour : l'approche de validation statique et l'approche de validation incrémentale.

Dans le troisième chapitre nous nous focalisons sur le problème de la validation des requêtes de mise à jour XML. L'approche que nous proposons permet la validation des contraintes structurelles et les contraintes d'intégrité référentielles, lors d'une opération de mise à jour. Pour cela nous utilisons une table d'éléments et une autre table d'attributs permettant de représenter le modèle d'arbre des documents XML. Nous utilisons aussi un processus de marquage qui permet d'étiqueter les clés et les clés étrangères dans l'arbre XML.

Le quatrième chapitre fournit une étude de cas sur des collections des documents XML représentant une base de données de l'université, associées à un document XML Schema. L'ensemble des requêtes émises sur le document XML, est validé par application des algorithmes proposés.

Chapitre I : XML et les bases de données

1. Introduction

Pour exploiter les documents XML, une solution de stockage devient indispensable. Les entreprises vont devoir choisir entre deux catégories de produits, les bases natives et les relationnelles compatibles XML, en se basant sur leurs performances et les coûts afin d´opter pour la meilleure solution.

Dans ce chapitre nous allons discuter d'une part les différentes stratégies de stockage des documents XML, selon leur types, à savoir un contenu orienté données (*data centric document*) et un contenu orienté document (*document centric document*). D'autre part nous allons présenter les différents types de langages de requêtes permettant de rechercher et de retrouver les données XML.

2. Structure du document XML

Un document XML est constitué d'un ensemble d'éléments et d'attributs : Nous considérons l'exemple d'un document XML simple, qui est illustré dans la figure (Fig.1.1) :

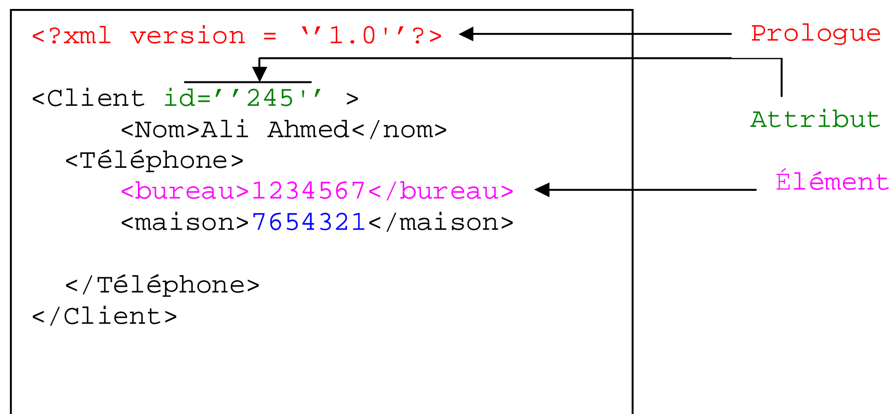

Fig.1.1 Document XML

Dans un document XML, après le prologue, se trouve une série d'éléments. Un élément est l'unité de base d'un document XML, composé de données textuelles et de balises [39]. Les frontières d'un élément sont définies par une balise de début et une balise de fin. Un

élément peut contenir des attributs additionnels. Par exemple, `<nom>Ali Ahmed</nom>` est un élément et `<Client id=''245'' > <Nom>Ali Ahmed</nom></Client>` est un élément contenant un attribut.

Notons que XML permet la définition d'éléments vides : ils n'ont pas de contenu et pas de balise fermante. Ces éléments sont formés en ajoutant une barre oblique''/'' à la fin de la balise ouvrante.

Un élément ne peut contenir que du texte ou une combinaison de textes d'autres éléments par exemple : <étudiant>Sana Bilel<age>18</age></étudiant>.

Un document XML bien formé doit contenir au moins un, et un seul élément non vide, appelé élément racine. Celui-ci peut contenir d'autres éléments définissant ainsi un arbre.

Les éléments peuvent, contenir des attributs qui fournissent des informations supplémentaires. Ces attributs sont des couples (nom, valeur) de la forme nom="valeur" par exemple : id=''245''. Ils doivent se trouver dans la balise ouvrante après le nom de l'élément. Notons que les noms d'attributs doivent être uniques au sein d'un même élément et que les valeurs des attributs se trouvent obligatoirement entre guillemets. La décision d'utiliser un attribut plutôt qu'un élément n'est pas claire car leurs fonctionnalités sont similaires. Les deux formes suivantes sont correctes :

```
<personne sexe="m">Ali Ahmed</personne>
<personne><nom>Ali Ahmed</nom><sexe>m</sexe></personne>
```

Ceci signifie que la déclaration de l'entité ''sexe'' comme étant un attribut ou un élément permet d'obtenir le même effet.

Des raisons valides pour utiliser des attributs au lieu d'éléments sont l'affectation d'un identificateur à un élément: <étudiant id="3876">Sana Bilel</étudiant>, Ou bien la description des caractéristiques de l'élément lui-même.

Notons qu'un document XML est un document bien formé s'il respecte la recommandation XML [22]. Un document XML est un document valide si en plus d'être bien formé, il respecte les contraintes définies par le schéma associé une DTD(D*ocument Type Definition*) ou un document XML Schema. A l'aide d'une DTD ou d'un XML Schema et du document XML correspondant, un analyseur, appelé aussi parser[42], tel que SAX (*Simple API for XML*) ou DOM(*Document Object Model*) peut confirmer que le document est conforme à la structure et aux contraintes imposées.

Nous reviendrons en détail sur ces concepts dans le deuxième chapitre.

Le stockage des données XML nécessite de distinguer entre un contenu orienté données et un contenu orienté document (présentation), pour pouvoir décider sur le type de bases de données XML à utiliser [1, 39].

2.1 Contenu orienté données

Les contenus orientés données utilisent XML en tant que vecteur de données [1, 39]. Ils sont conçus pour être analysés par l'ordinateur. Il n'est pas important pour une base de données ou l'application de garder ces données stockées sous forme d'un document XML. Les caractéristiques des contenus orientés données sont :

- Une structure assez régulière.
- Une granularité fine.
- Peu ou pas du tout de contenu mixte.
- L'ordre des balises n'a pas d'importance.

Ces données peuvent être originaire d'une base de données et XML est utilisé pour sa publication, par exemple : les données stockées dans une bases de données relationnelle, ou bien elles peuvent être situées en dehors de la base et elles sont converties en format XML pour pouvoir les stockés dans la base de données, par exemple : les données scientifiques collectées par un système de mesure et converties en format XML.

L'exemple suivant présente un document XML orientés données :

```
<Vols>
      <compagnie> Air Algérie </compagnie>
      <départ> Constantine </départ>
   <Vol>
      <destination> Alger</destination>
      <heure depart>7.30</heure depart>
      <heure arrive>8.30</heure arrive>
   </Vol>
   <Vol>

      <destination> Alger</destination>
      <heure depart>14.00</heure depart>
      <heure arrive>15.00</heure arrive>
   </Vol>
</Vols>
```

Ce document XML qui décrit les vols, est un document orienté données car sa structure est régulière, l'ordre des éléments `Vol` sous l'élément `Vols` n'est pas important ainsi que l'information significative se trouve au niveau d'un élément de type PCDATA[1] , par

[1] PCDATA désigne les types d'éléments textuels devrant être traités par l'analyseur.

exemple : l'élément `<destination> Alger</destination>` signifie que `Alger` est la ville de destination du vol.

2.2 Contenu orienté document

Les contenus orientés document sont élaborés pour la lecture humaines [1,39]. Citons à titre d'exemple : les messages électroniques, les documents issus du Word.
Ils se caractérisent par :

- Structure irrégulière.
- Granularité forte.
- Contenu mixte.
- L'ordre d'apparition d'éléments est important.

Les contenus orientés document ne sont pas originaires de la base. Ils sont écrits en XML ou dans un autre format tel que PDF ou SGML puis ils sont convertis vers XML.
A titre d'exemple le document XML suivant est orienté présentation :

```
<Document >
<Titre>
DBXML
</Titre>
<Sous titre>
</Sous titre>
<Introduction> une base de données et une collection d'informations regroupées de telle manière que l'accès et la manipulation de ces données soient souples et rapides
</Introduction> <body> actuellement les différents types de base de données
</body>
<Conclusion></conclusion>
</Document>
```

Ce document XML qui décrit un document Word est orienté présentation, car l'ordre des éléments est significatif, par exemple : l'élément `conclusion` ne peut pas apparaître avant l'élément `titre`, ainsi que l'information significative se trouve au niveau du document lui-même.

Cette classification est la plus adaptée dans la littérature. Cependant, [15] ajoute un troisième type de documents XML qui sont les flux de données services Web : des séquences de petits fragments structurés et indépendants.

3. Stockage des documents XML

En pratique, la distinction entre les deux types de documents n'est pas claire : Les contenus orientés données peuvent contenir des données de granularité forte et de structure irrégulière [1], par exemple dans un document XML qui décrit une facture, il peut inclure des

descriptions. Les documents orientés présentation peuvent eux aussi inclure des données de granularité fine et régulièrement structurées [1], tel que les documents juridiques ou médicaux, ils sont écrits sous forme de prose, mais ils contiennent des parties distinctes, tel que les dates, les noms et les procédures, et ils doivent souvent être stockés dans leur intégralité pour des raisons légales.

En règle générale (mais qui n'est pas absolue) [1,12, 11, 15, 17] : les données sont stockées dans une base de données traditionnelle (relationnelle, orienté objet ou hiérarchique) [33]. Cela peut être réalisé à l'aide d'un logiciel intermédiaire (middleware), ou bien par la base elle-même qui dispose plus de fonctionnalité pour supporter les données XML. Dans ce dernier cas la base de données est qualifiée de compatible-XML [« Enabled –XML »]. Les documents sont alors stockés dans une base de données XML native (une base de données conçue spécialement pour stocker du XML).

4. Stockage des contenus orientés données

Dans le but de transférer des données entre les documents XML et une base de données relationnelle, il est nécessaire de faire correspondre le schéma du document XML (cas de la DTD, XML schema) avec le schéma de la base de données. Le logiciel de transfert de données est alors construit au dessus de cette correspondance [1,37]. Cette correspondance est effectuée sur les types d'éléments, les attributs et les textes. Elles omettent la plupart du temps la structure physique (les sections CDATA[2], les entités et les informations concernant l'encodage) et certaines structures logiques (telles que les instructions de traitements, les commentaires, ainsi que l'ordre dans lequel les éléments et les PCDATA apparaissent dans une filiation) [1,37].

L'avantage principal de cette approche est de pouvoir bénéficier de l'utilisation du langage SQL au niveau relationnel pour l'interrogation et la mise à jour des données XML. La recherche du document XML décomposé exige ultérieurement la recomposition du document, ce processus est appelé la publication du XML ‘‘ XML publishing’’ [17]. La figure (fig2.1) illustre la décomposition et la recomposition des documents XML.

[2] Les sections CDATA désignent les données textuelles qui ne devront pas être traitées par l'analyseur

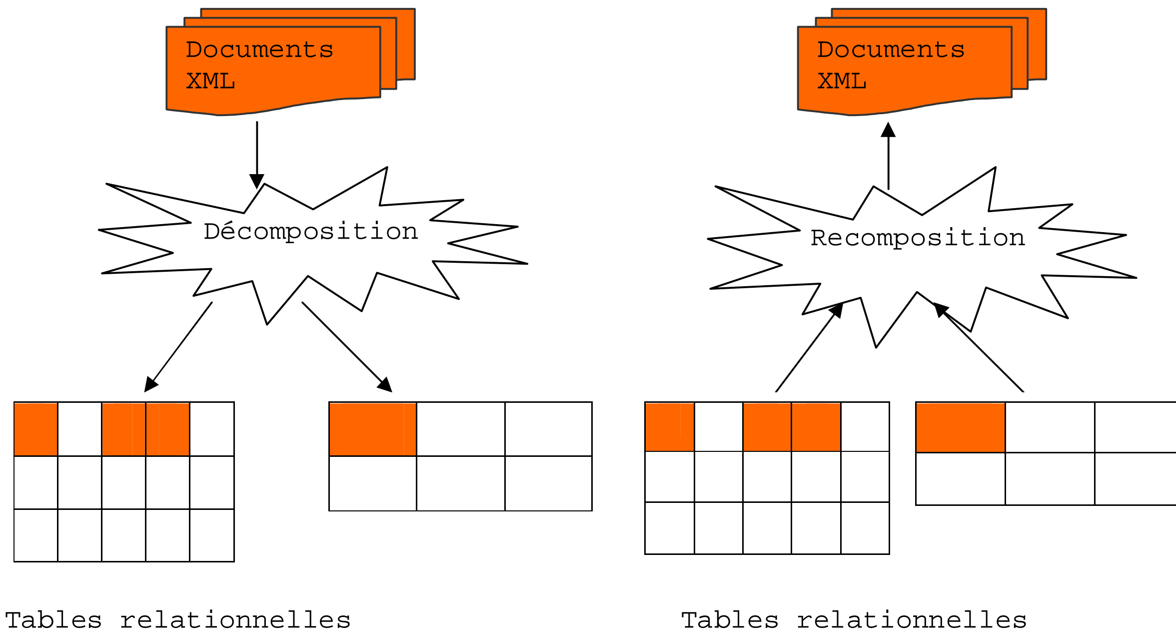

Fig.1.2 Décomposition et recomposition des documents XML

Une conséquence de tout cela, est que l'aller et le retour d'un document « le round–tripping » c-à-d le stockage des données depuis un document dans la base de données et la reconstruction ultérieure du document à partir des données de la base, conduit souvent vers un autre document.

Le caractère acceptable ou non de ce fait dépend des besoins et peut influencer le choix du logiciel. L'utilisation de cette technique est appropriée pour les applications satisfaisant les propriétés suivantes [17] :

- XML est seulement utilisé comme un format de transfert des données. La structure de document n'est plus importante, une fois les données sont stockées dans la base de données. En particulier, l'ordre dans le document n'est pas important.
- Les données XML à stocker doivent être intégrées avec les données relationnelles existantes.
- La structure des documents XML est suffisamment simple pour permettre un mapping efficace vers le schéma relationnel cible.
- Le schéma de XML est stable. Les changements de schéma ne se produisent pas ou sont très rares.
- Les applications relationnelles existantes et les outils d'enregistrement doivent accéder aux données dans le format tabulaire.

- Le besoin de construire des documents XML qui sont différents de ceux qui ont été insérés dans la base de données.
- Les requêtes et les mises à jour peuvent facilement être indiquées dans SQL. Le mapping du langage XML au SQL est simple ou non requis.
- Les mises à jour sont fréquentes sur les différentes valeurs d'éléments ou d'attributs, et l'exécution des mises à jour est critique.

Deux correspondances sont couramment utilisées pour faire coïncider le schéma du document XML avec le schéma de la base de données [1] : la correspondance basée sur des tables et la correspondance basée sur un modèle objet/relationnel.

4.1 Correspondance basée sur des tables

La correspondance basée sur des tables[1], est utilisée par de nombreux logiciels intermédiaires qui transfèrent les données entre un document XML et une base de données relationnelle, elle modélise les documents sous forme d’un ensemble de tables. Cela signifie que la structure d’un document XML doit être comme suit :

```
<Database>
 <Table>
 <row>
 <Column1> </Column1>
</Column2 </Column2>
</row>
<row>
….
</row>
</Table>
<Table>
</Table>
</Database>
```

Le terme table est souvent interprété de manière vague, autrement dit, le transfert des données depuis une base vers un document, une table peut être constituée par n’importe quel ensemble de résultats, et le transfert des données depuis un documents XML vers la base, une table peut être une véritable table ou une vue.

La correspondance basée sur des tables est utile pour sérialiser des données relationnelles, comme par exemple pour transférer des données entre deux bases de données relationnelles. Son inconvénient est qu’elle ne peut pas être utilisée pour un document qui n’est pas conforme au schéma exposé ci-dessus.

4.2 Correspondance basée sur un modèle objet /relationnel

La correspondance basée sur un modèle objet / relationnel [1], est utilisée par toutes les bases de données relationnelles compatibles XML [Enabled-XML] et quelques produits intermédiaires. Les données du document sont alors modélisées comme un arbre d'objets spécifiques aux données tel que, les types d'éléments possédant des attributs, les contenus d'éléments ainsi que les contenus mixtes sont généralement modélisés comme des classes. Les types d'éléments contenant seulement des PCDATA, les attributs et les PCDATA elles même sont modélisés comme des propriétés scalaires. Le modèle est alors mis en correspondance avec la base relationnelle tel que les classes correspondent à des tables et les propriétés scalaires à des colonnes.

4.3 Génération de schéma XML à partir de schéma relationnel et vice-versa

La génération de schéma XML à partir de schéma relationnel et vice-versa est une opération intervenant lors de la conception[1], parce que la plupart des applications orientées données travaillent avec un corpus bien établi de schémas XML et de schémas de base de données. Elles ne nécessitent pas de générer des schémas lors de l'exécution.

4.3.1 Génération de schéma relationnel à partir de schéma XML

Pour générer un schéma relationnel à partir d'un schéma XML il convient de [1] :

- Créer une table et une colonne clé primaire pour tout type d'élément complexe.
- Pour chaque type d'élément possédant un contenu mixte créer une table séparée dans laquelle est stockée les PCDATA ; cette table est liée à la table parente grâce à la clé primaire de celle-ci.
- Pour chaque attribut de ce type d'élément qui possède une valeur unique, et pour chaque élément fils simple présentant une seule occurrence, créer une colonne dans cette table. Si le schéma XML contient des informations concernant le type de données, affecter le type de données de la colonne au type qui lui correspond. Dans le cas contraire, affecter lui un type prédéterminer comme ''Character Large Objects'' CLOB ou ''une variable de type caractère longue'' VARCHAR. Si l'occurrence de l'élément fils ou de l'attribut est optionnel, attribuer à la colonne la possibilité d'y affecter des valeurs nulles.
- Pour chaque attribut possédant plusieurs valeurs et pour chaque élément fils simple présentant plusieurs occurrences, créer une table séparée pour stocker ces valeurs. Cette table est liée à la table parente grâce à la clé primaire de celle-ci.

- Pour chaque élément complexe, lier la table de l'élément parent à la table de l'élément fils à l'aide de la clé primaire de la table parente.

Dans ce qui suit nous présentons deux exemples de schémas de stockage : Edge [15], et Inlining [15], qui permettent le stockage d'un arbre XML dans des tables relationnelles :

A) Le schéma de stockage Edge

Soit l'arbre XML :

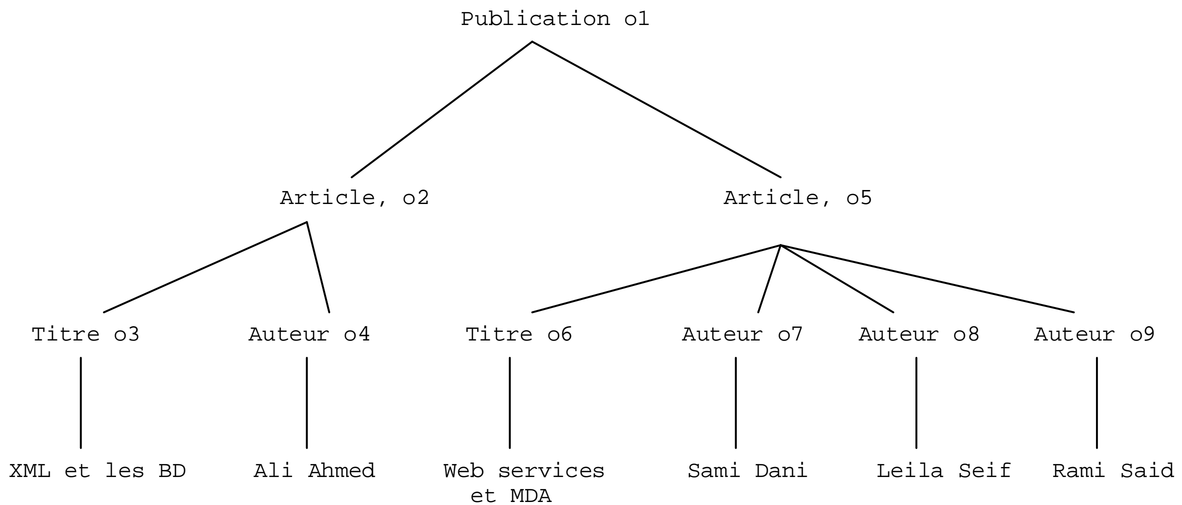

Fig.1.3 Arbre représentant un document XML.

Le schéma Edge consiste à créer une table pour stocker la position (ordre), le nom, le type (élément ou attribut) et le parent de chaque noeud sous forme d'un n-uplet – voir tableau 1.1–, et une autre table pour stocker les valeurs des attributs et des éléments simples –voir tableau 1.2– :

Part	Pos	Lab	Type	id
O0	1	Publication	Ref	O1
O1	1	Article	Ref	O2
O1	2	Article	Ref	O5
O2	1	Titre	cdata	O3
O2	2	Auteur	cdata	O4
O5	1	Titre	cdata	O6
O5	2	Auteur	cdata	O7
O5	3	Auteur	cdata	O8
O5	4	Auteur	cdata	O9

Tableau 1.1 Table des relations

Nœud	Val
O3	XML et les BD
O4	Ali Ahmed
O6	Web service et MDA
O7	Sami Dani
O8	Leila Seif
O9	Rami Said

Tableau 1.2 Table des valeurs

Le schéma Edge permet un format de stockage générique et un espace d'utilisation faible, par ailleurs, il nécessite une lecture sur la table entière et beaucoup de jointure.

B) Le schéma de stockage Inlining

Il existe une autre approche permettant le stockage des arbres XML dans le schéma relationnel qui est l'approche de ''mapping guidée par la DTD'', dans cette approche le schéma relationnel prend en compte la DTD du document. Inlining [15] est une stratégie qui utilise la DTD pour créer un graphe structurel, il permet de diminuer la fragmentation à partir des informations dans le graphe, et décider quand un élément est mis dans la table de son parent ("Inlining"), et quand il faut créer une table séparée. Il existe trois approches : Basic,

Shared et Hybride. Nous considérons dans ce qui suit un exemple qui décrit l'approche Basic Inlining, dont l'idée consiste à:

- Créer pour chaque type d'élément une ou plusieurs relations pour stocker les fils et les attributs (similaire à la traduction E/A → relationnel) :
- Extraire le graphe DTD qui contient tous les noeuds qu'on peut directement atteindre à partir du type d'élément.
- Créer récursivement les tables à partir de ce sous-graphe :
  - Une table pour la racine.
  - Une table pour chaque noeud cible d'un arc*
  - Les autres noeuds sont transformés en attributs

Exemple

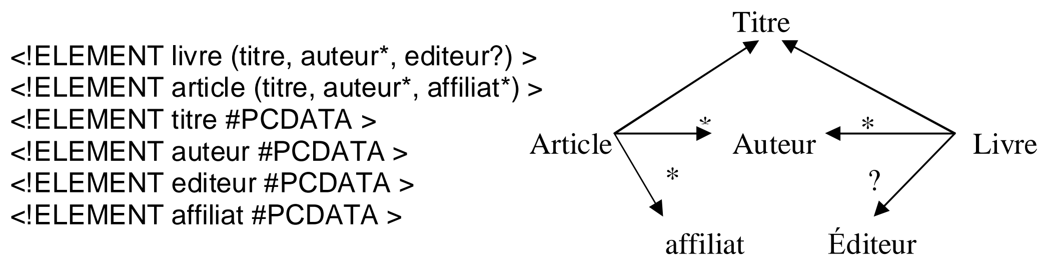

Fig.1.4 DTD et le sous graphe correspondant.

Les tables crées à partir du sous graphe sont :

```
livre(id, parId, titre, editeur)
livre_auteur(id, parId, auteur)
article(id, parId, titre)
article_auteur(id, parId, auteur)
article_affiliat(id, parId, affiliat)
titre(id,titre)
editeur(id, parId, editeur)
auteur(id, parId, auteur)
```

4.3.2 Génération de schéma XML à partir de schéma relationnel

Pour générer un schéma XML à partir d'un schéma relationnel [1], il convient de :

- Créer un type d'élément par table.
- Pour chaque colonne de cette table qui ne soit pas une clé et pour la (les) colonne(s) correspondante(s) à la clé primaire ajouter un attribut au type d'éléments ou ajouter un élément fils de ce type PCDATA seul à son modèle de contenu.

- Pour chaque clé étrangère, ajouter un élément fils au contenu du modèle et traiter récursivement la table de la clé étrangère.

Dans cette section nous avons présenté les techniques pour la génération des schémas XML à partir du schéma relationnel et vice vers ça, des procédures similaires pour la génération des schémas des documents XML (DTD et XML Schema) à partir d'un modèle objet sont présentées avec des exemples dans [31,35].

5. Stockage des contenus orientés documents

Il existe principalement deux méthodes de base pour stocker des documents XML [1] : les enregistrer sur le système de fichiers ou sous forme de BLOB[3] ''Binary Large Object'' dans une base de données relationnelle, et accepter des fonctionnalités XML limitées, ou les stocker dans une base de données XML native .

5.1 Stockage des documents sur un système de fichiers

Le système de fichier constitue la meilleure méthode pour le stockage d'un ensemble élémentaire de documents [1] :

La recherche des données se fait à l'aide de l'outil «GREP» et leur modification se fait à l'aide de l'outil « SED », "Stream Editor". Les recherches pleins textes sur des documents XML sont inexactes car elles ne peuvent pas facilement distinguer les balises du texte et ne peuvent pas interpréter l'usage des entités. Cependant dans un petit système, de telles inexactitudes peuvent être acceptables.

5.2 Stockages des documents dans des BLOB(s)

Le stockage des documents comme des BLOB(s) dans une base relationnelle apporte certains avantages propres aux bases de données[1], tel que le contrôle de transaction, sécurité et l'accès multi utilisateur. En outre, de nombreuses bases de données relationnelles possèdent des outils de recherches compatibles avec XML, ce qui élimine les problèmes liés à la recherche des documents XML en tant qu'un simple texte.

Le stockage des documents XML sous forme de BLOBs permet au développeur d'implémenter facilement son propre indexation, même si la base de données ne sait pas indexer du XML, l'une des méthodes de réaliser cela, consiste à créer deux tables, une table d'index et une table des documents. La table des documents contient une clé primaire et une colonne BLOB où le document est stocké. La table d'index contient une colonne contenant la

[3] *Les BLOBs sont des objets binaires. Ils permettent le stockage des données, de type arbitraire.*

valeur à indexer et une clé étrangère pointant sur la clé primaire de la table des documents. Voir la figure (fig.1.5):

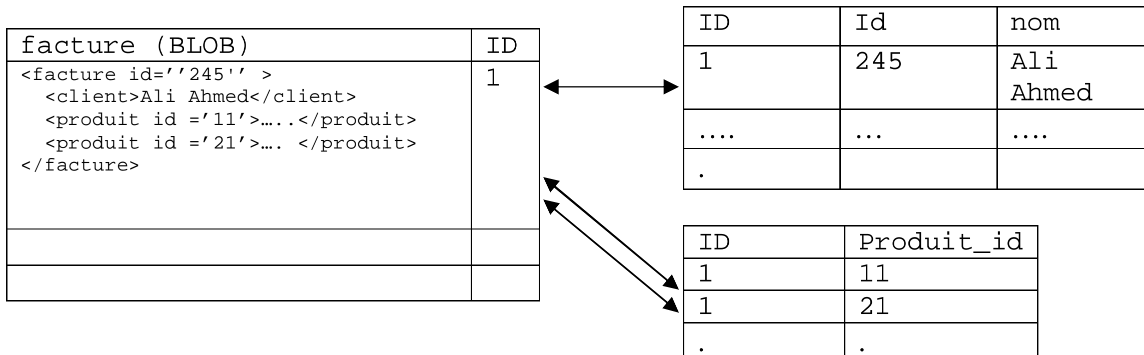

Fig.1.5 Stockage des documents XML dans des BLOBs

5.3 Stockage des documents dans des bases de données XML natives

Une base de donnée XML native, définit un modèle logique pour le document XML, et en fonction de ce modèle, elle stocke et retrouve le document. L'unité fondamentale de stockage est le document XML. L'unité de stockage physique n'est pas spécifique, elle peut être bâtie sur les bases des données traditionnelles (relationnelle ou orienté objet) ou bien utilisée d'autres techniques de stockage, telles que les fichiers indexés ou compressés [1, 17,36].

Les bases de données XML natives sont des bases conçues spécialement pour stocker des documents XML comme toutes les autres bases, elles possèdent des fonctionnalités telles que les transactions, la sécurité, les accès multi utilisateurs, un ensemble d'API(s), des langages de requêtes … . La seule différence par rapport aux autres bases c'est qu'elles sont basées sur XML [1]. La figure (fig 1.6) montre un document XML et sa représentation hiérarchique, un véritable système de base de données XML native, utilise la structure d'arbre comme étant le modèle de base pour le stockage et le traitement [17].

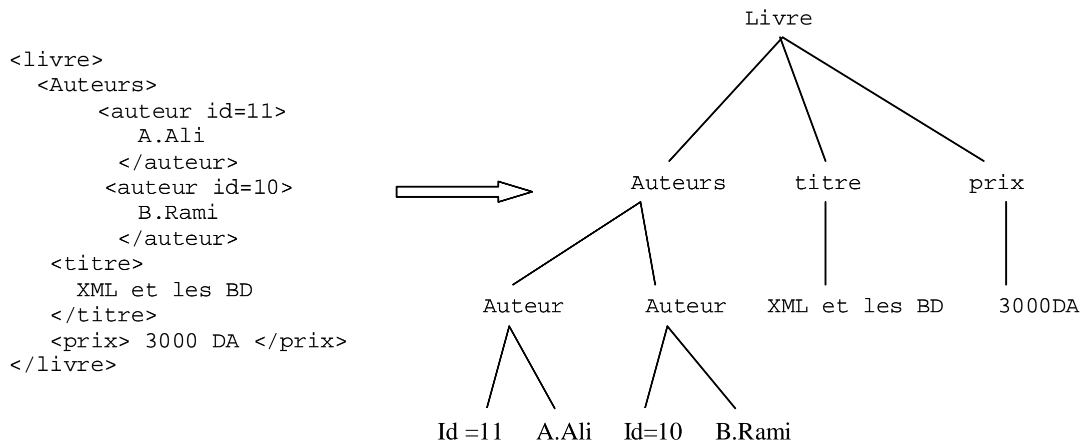

Fig.1.6 Document XML et sa structure d'arborescence

Les bases de données XML natives sont plus utiles pour le stockage des contenus orientés document [1], car elles préservent l'ordre interne du document, les instructions de traitement, les commentaires, les sections CDATA. Etc. en outre les bases XML natives supportent les langages de requêtes XML qui permettent d'effectuer des recherches difficiles à effectuer dans un langage tel que SQL.

La base de données XML native est utilisée également pour le stockage des contenus orientés données et le stockage de documents qui ne possèdent pas de DTD(s) ou tout autre type de schéma XML (les documents sans schémas)[17,1]. Cela veut dire qu'une base de données XML native peut stocker n'importe quel document XML sans configuration préalable.

Dans notre travail nous nous sommes intéressés à cette nouvelle technologie de base de données qui constitue un domaine de recherche très vaste sur plusieurs niveaux, (la mise à jour, l'indexation, le stockage…). Nous reviendrons en détail sur ce type de bases de données dans le deuxième chapitre.

6. Conception d'une base de données XML

Les différentes étapes pour la conception des bases de données relationnelles sont [13]:

- Collection des informations du domaine d'application à l'aide des utilisateurs, la collection des descriptions des processus et l'étude des applications existantes.

- Construction d’un modèle d’objets et des relations dans le domaine en utilisant par exemple UML.
- Transfert du modèle dans le schéma relationnel, par l’application des mécanismes tel que la normalisation.
- Raffinement de la conception pour s’assurer que toutes les performances requises sont satisfaites.

En principe, une approche similaire peut être appliquée aux bases de données XML(BDXML) à l’exception de la troisième étape où le modèle est transféré vers des éléments et des attributs XML, au lieu des colonnes et des tables relationnelles[13].

Les responsables des projets s'interrogent sur la différence entre une base de données XML native et une base relationnelle. Cette différence qui leur permettra de réaliser un choix approprié lors de la mise en place de leurs systèmes d'informations.

Les critères à pendre en considération dans le choix du type de base de données à utiliser pour le stockage des données XML sont [14] :

- Le premier critère à examiner est celui de la nature et de la quantité des transactions à effectuer sur les plus petites unités d’informations. S’il s’agit d’opérations d’écriture, d’insertion de nouvelles valeurs, de calculs et d’agrégations, et si les opérations sont nombreuses le choix d’un SGBD relationnel s’impose de lui-même. En revanche s’il s’agit de requêtes complexes sur les contenus, alors le choix d’une base de données XML native est plus adapté.
- Le second critère, voisin du premier concerne les fréquences de mise à jour des petites unités d’informations ; plus les mises à jour sont nombreuses est plus le choix d’un SGBDR sera pertinent
- Le troisième critère concerne les éventuels flux XML à traiter en entrée et en sortie du système. Plus ils sont nombreux et les formats de données sont variés, plus il faudra prévoir dans le cas d’un choix du SGBDR, le temps de spécification, de développement et de traitement pour convertir les données conformément au modèle de données de la base. En revanche, dans le cas d’un choix d’une base de données XML native, les flux XML peuvent être stockés, sans transformation de structure.
- Le dernier critère à prendre en compte est relatif à la complexité du ou des schémas XML utilisés. Les schémas très arborescents seront mieux traités par une base de données XML native.

Un forum a été organisé par la communauté des développeurs XML-DEV[25] en vue de discuter ces critères, et qui a été enrichi par des opinions des experts dans ce domaine .

L'implémentation d'une base de données XML dépend des caractéristiques du système d'information à développer. Ces caractéristiques peuvent être cernées dans les points suivants [28] :

- Type de base de données XML : Native vs Relationnel.
- La nature du schéma (simple vs complexe).
- La technique de l'indexation (indexation automatique vs indexation définie par l'utilisateur).
- La recherche plein texte.
- La stratégie de stockage (stockage au niveau de document vs stockage au niveau du nœud).
- Mise à jour du document entier vs mise à jour partielle.
- Sécurité.
- Système transactionnel vs système non transactionnel.
- Architecture client/serveur vs architecture intégrée.

7. Différents types de langages de requêtes

Dans une base de données XML, il existe trois types de langage de requêtes [1] : les langages de requêtes basés sur des modèles, les langages de requêtes basés sur SQL et les langages de requêtes XML.

7.1 Langages de requêtes basés sur des modèles

Les langages de requêtes les plus connus capables de renvoyer du XML depuis une base relationnelle sont ceux qui sont basés sur des modèles [1]. Dans ces langages il n'existe pas de correspondance prédéfinie entre le document et la base, au lieu de cela les ordres SELECT sont englobés dans un modèle et les résultats sont traités par le logiciel de transfert de données.

Les langages de requêtes basés sur des modèles peuvent être extrêmement flexibles. Ils se caractérisent par :

- La capacité de disposer de l'ensemble de résultats n'importe où dans le document de sortie compris comme paramètres d'ordres Select ultérieurs.
- Les constructions de programmation tel que les boucles et les instructions SI

- Les définitions de variables et de fonctions
- La possibilité de paramétrer des ordres Select à travers des paramètres http.

Les langages de requêtes basés sur des modèles sont utilisés généralement pour transférer des données depuis les BD relationnelles vers les documents XML.

Exemple :

Le modèle suivant utilise les éléments <SelectStmt> pour inclure les ordres SELECT et les valeurs $column-name pour déterminer où les résultats doivent être placés :

```
<?xml version="1.0"?>
   <InformationsVol>
      <SelectStmt>SELECT Compagnie, NumeroVol, Depart, Arrivee FROM
Vols</SelectStmt>
         <Vol>
            <Compagnie>$Compagnie</Compagnie>
            <NumeroVol>$NumeroVol</NumeroVol>
            <Depart>$Depart</Depart>
            <Arrivee>$Arrivee</Arrivee>
         </Vol>
        </InformationsVol>
```

7.2 Langages de requêtes basés sur SQL

Les langages de requêtes basés sur SQL [1] utilisent des ordres SELECT modifiés dont les résultats sont transformés en XML.

Un certains nombre de langages propriétaires de ce type son actuellement disponibles. En plus des langages propriétaires, un certain nombre de sociétés se sont réunis pour standardiser des extensions XML au langages SQL : [SQL/XML] ajoute un certain nombre de fonctions à XML, de telle façon qu'il soit possibles de construire des éléments et des attributs XML à partir de données relationnelles.

Exemple :

```
SELECT Ordres.Numero,
       XMLELEMENT(Nom "Ordre",
                XMLATTRIBUTES(Ordres.Numero AS Numero),
                XMLELEMENT(Nom "Date", Ordres.Date),
                XMLELEMENT(NAME "client", Ordres.client)) AS xmldocument
FROM Ordres
```

Cette requête construit un ensemble résultant possédant deux colonnes. La première colonne contient un numéro d'ordre des ventes et la seconde colonne contient un document XML. Il y a un document XML par ligne et il est construit d'après les données provenant de la ligne correspondante dans la table intitulée `Ordres`. Le document XML pour la ligne d'ordre de numéro 150 pourrait être par exemple :

```
<Ordre Numero ="150">
      <Date>10/29/02</Date>
      <Client>Ali Ahmed</Client>
 </Ordre>
```

7.3 Langages de requêtes XML

A la différence des langages de requêtes basés sur des modèles et des langages de requêtes basés sur SQL, qui peuvent être utilisés uniquement avec des BD relationnelles. Les langages de requêtes XML [1], peuvent être utilisés sur n'importe quel document XML, pour les utiliser avec des bases de données relationnelles, les données de la BD doivent donc être modélisées en XML ce qui permet de formuler des requêtes sur des documents XML virtuels.

Plusieurs langages de requêtes ont été développés pour l'interrogation des documents XML, citons à titre d'exemple : Lorel, Quilt, UnQL, XDuce, XML-QL, XPath, XQL, XQuery et YaTL [37]. Dans cette section nous allons présenter brièvement XPath et XQuery, les standards proposés par W3C (*World Wide Web Consortium*).

7.3.1 XML Path Language (XPath)

XPath [37] est un langage pour l'interrogation des parties d'un document XML, il utilise une syntaxe qui ressemble à celle utiliser pour adresser des parties d'un système de fichiers ou URL *"Uniform Resource Locator"*. Il fournit des fonctions pour l'interaction avec les données sélectionnées, comme la manipulation des chaînes de caractères, les nombres et les booléens.

Considérons à titre d'exemple le document XML suivant :

```
 <étudiant id="E100">
   <nom>Ahmed Nedjme Eddine</nom>
   <age>21</age>
  </étudiant>
```

La requête XPath permettant d'avoir comme résultat tout les élément fils `nom` de l'élément racine `étudiant` est comme suit : `/étudiant/nom`

7.3.2 XML Query Language (XQuery)

XQuery [37] est un langage de requêtes XML qui se repose sur XPath. XQuery est un langage procédural, où chaque requête est une expression. Il existe 7 types d'expressions qui sont : les expressions de chemins, constructeurs d'élément, l'expression FLWR(expression composée par les clauses : For, Let, Where et Return), les expressions qui incluent les opérateurs et les fonctions, les expressions de quantifications, les expressions conditionnelles, et les expressions qui testent ou modifient les types de données.

XQuery diffère du SQL dans le fait qu'il est un langage procédural et SQL est un langage déclaratif [26], un non programmeur peut apprendre SQL, car il inclut des commandes déclaratives simples. L'utilisation de XQuery implique un programme procédural dans lequel l'utilisateur définit une séquence d'action, nécessaire pour générer le résultat attendu.

Considérons l'exemple suivant d'une requête écrite en langage XQuery :

```
 FOR $b IN document("publication.xml")//article
WHERE $b/auteur = "Seif Eddine"
AND $b/année = "1998"
RETURN $b/titre
```

Cette requête permet de retourner les titres d'articles écrits par « Seif Eddine » en 1998.

7.3.3 Optimisation des requêtes XML

Les stratégies de l'optimisation des requêtes XML peuvent être classées en deux techniques principales : l'optimisation logique et l'optimisation physique [29].

L'optimisation logique signifie la réécriture des requêtes qui peut être réalisée par l'optimisation de l'expression de chemin. A l'heure actuelle, plusieurs travaux de recherche traitent ce problème [45, 46, 47,48]

L'optimisation physique regroupe :

- Les stratégies de stockage qui consistent à trouver le schéma de stockage optimal par rapport à un ensemble de requêtes fréquentes.
- Les stratégies d'indexation permettant de créer des index sur le contenu pour accélérer les recherches.

8. Exemples des bases de données XML

La majorité des bases relationnelles du marché [21] - Oracle 9i d´Oracle, Sybase ASE de Sybase, SQL Server de Microsoft, DB2 d´IBM - disposent des fonctions plus ou moins élaborées pour gérer et exploiter les documents en XML. En parallèle de ces offres, il existe des bases de données natives XML, conçues spécifiquement pour ne gérer que des contenus XML, chez Software AG (Tamino), Ixiasoft (TextML), Ipedo (XML Database), etc. Les caractéristiques de ces produits sont fournies dans [36].

La table suivante résume les principales bases de données dans le marché [16] :

Les bases de données relationnelles	
DB2 (IBM)	En vue d'accueillir des documents XML, DB2 a été complété d'un nouveau module : XML Extender. Concrètement, ce dispositif a pour but de traduire un document au format XML pour intégrer les données qu'il contient au sein de la structure relationnelle de la base.
Oracle 9i (Oracle)	Oracle 9i s'adosse à une fonction appelée XML Database Support (XDS), en vue d'intégrer directement des contenus XML dans ses colonnes. Une technologie qui lui permet aussi de décrire sa structure relationnelle au format XML, et à l'inverse des données XML sous forme relationnelle.
SQL Server (Microsoft)	La prochaine version de l'outil de Microsoft devrait supporter les données XML de manière native tout en assurant la définition de tables dans le même format.
Sybase ASE (Sybase)	La version actuelle de Sybase ASE stocke et indexe les fichiers XML au sein de sa structure. Ce qui lui permet ensuite de supporter les requêtes au format XQL lancées sur cette catégorie de contenu.
Les bases de données XML natives	
Tamino (Software AG)	Tamino XML Server couple une base de données XML à une base de données relationnelle (ODBC et JDBC). Une structure qui lui permet de constituer des documents XML bâtis à partir de sa propre structure relationnelle, mais également d'autres sources (systèmes de fichiers, etc.)
Ipedo XML Database (IPedo)	Ipedo XML Database a été conçue pour prendre en charge les documents XML au sein d'une structure dans le même format. Pour ce faire, elle propose des mécanismes d'indexation et de classement -qui s'appuient sur un système de catégorisation XML (couplant DTD et XML Schema).
TextML (IXiasoft)	La solution d'IXiasoft accueille des documents XML. L'outil s'appuie sur le "parser" d'IBM pour indexer l'ensemble des attributs XML en question sous forme de meta-données (auteur, titre, etc.). La méthode est destinée pour faciliter les développements clients.

Tableau 1.3 Bases de données XML du marché

9. Conclusion

Le modèle relationnel des bases de données fournit une solution pour la gestion des données XML, qui consiste à normaliser ces données dans des colonnes ou chaque enregistrement contient une valeur atomique. Plusieurs techniques de normalisation décrivent comment faire le mapping d'un document XML spécifié à l'aide d'une DTD ou XML schéma vers une BD relationnelle. Cependant, cette solution peut être complexe pour les concepteurs et nécessitent un temps très large dans la programmation.

La base de données ENABLED-XML supporte la gestion des données XML grâce aux types de données (LAB)(Large Object) qui permettent le stockage des données de type arbitraire, cependant ces bases de données ne supporte pas « l'aller et le retour » dans le document XML.

Une base de données XML native ne demande pas le mapping de la structure XML vers un modèle de données non XML, le coût supplémentaire pour partitionner les documents XML sur des tables multiples, et les reconstruire à partir de ces dernières constituent un traitement supplémentaire en le comparant avec la solution directe de la base de documents XML. Cependant ne pas permettre aucune forme de décomposition, induit une redondance de données qui affecte l'intégrité des données dans la base.

Chapitre II : Caractéristiques des bases de données XML natives

1. Introduction

Le modèle de données des bases de données XML natives "BDXMLN" est relativement éloigné d'un modèle relationnel "classique".Un document XML ayant une structure d'arborescence. Lors de son stockage dans une base de données XML native, un document XML est classé dans une collection. Une collection regroupe un jeu de documents XML. Une base de données peut contenir plusieurs collections. Ces collections sont organisées sous forme d'arborescence. Cette organisation est similaire à un système de fichiers : les collections correspondent aux dossiers, et les documents XML correspondent aux fichiers.

Dans ce qui suit nous allons présenter les caractéristiques des bases de données XML natives leurs avantages et leurs inconvénients. Ainsi que les outils et les techniques préconisés pour leur mise à jour et leur validation.

2. Définition d'une base de données XML native

Une base de données doit répondre à plusieurs critères pour prétendre être de type XML natif [1, 18,32] : Elle doit définir un modèle logique pour tout document XML et se baser sur ce modèle pour stocker et extraire des documents. Le modèle doit au minimum inclure les éléments, les attributs, les PCDATA et l'ordre interne du document. Quelques exemples de tels modèles sont : le modèle de données de XQuery et le glossaire XML Infoset[39]. De plus, elle doit considérer le document comme unité fondamentale de stockage. Cela change radicalement des BD relationnelles, pour qui une unité de stockage correspond à une ligne d'enregistrement d'une table donnée. Enfin, le format de stockage physique des données composant les documents n'a pas d'importance. Il pourra être de type objet, relationnel, propriétaire, ... Peu importe, tant que ce dernier n'influence pas l'intégrité des données traitées.

Cette définition exprime trois idées principales [18] :

- La base de données est spécialisée pour le stockage de documents XML, et elle stocke tous les composants du modèle XML d'une manière intacte.

- Les documents peuvent être stockés et retrouvés.
- La BDXN n'est pas destinée à remplacer les bases de données classiques.

3. Propriétés d'une base de données XML native

Des fonctions qui ne sont pas encore toutes présentes dans les bases de données XML natives figurent ci-dessous. Pourtant, elles paraissent indispensables pour une gestion efficace de documents XML. Le fait que ces fonctions ne sont pas forcément disponibles dans une base de données XML native, peut renforcer les arguments portés en faveur des bases de données relationnelles [1,30].

3.1 Collections de documents

Ce terme de collections désigne des groupes de documents. Il est ainsi possible de rassembler sous une même entité un lot de documents (par exemple tous les textes ayant trait à un certain sujet, tous les documents possédant la même DTD ou Schéma, ...). Il sera ainsi plus facile et plus rapide d'effectuer des recherches sur une collection plutôt que sur l'ensemble des documents d'une BD.

3.2 Langages de requêtes

La plupart des bases de données XML natives utilisent le langage de requête XQuery, le standard édicté par W3C pour rechercher les données, et construire des requêtes relativement complexe. Cependant certaines offrant des langages de type propriétaires.

3.3 Mises à jour et suppression

Les bases de données offrent toutes les possibilités de modifier ou de supprimer des documents entiers. Dans certaines BD on peut également, à l'aide de l'API DOM, d'en manipuler que des fragments.

3.4 Transaction et verrous

Les fonctions de transactions, comme les verrous, ne sont pas forcément disponibles sur toutes les BD XML. Les verrous s'appliquent souvent au document entier en cours de traitement. Cela peut poser des problèmes si plusieurs personnes désirent accéder au même document simultanément. Des solutions de verrous partagés en lecture seule, ou de verrous partiels permettent de contourner cet obstacle.

3.5 Application programming interface

Presque toutes les bases de données XML offrent une API. Elle se présente en général comme une interface avec des méthodes permettant de se connecter à la BD, explorer les metadata, exécuter des requêtes et retourner des résultats. Ce type d'interface est comparable à l'ODBC « Object Data Base Connectivity » qui permet sous Windows de se connecter à des sources de données relationnelles.

3.6 Round-tripping

Ce terme signifie la possibilité de pouvoir enregistrer un document et de pouvoir le restaurer par la suite dans un état strictement identique. Dans le cas des bases de données relationnelles, ce n'est parfois pas possible. Or, pour certaines applications pratiques, il est absolument nécessaire de pouvoir retrouver un document tel qu'on l'avait construit au départ (pour des raisons légales par exemple). De plus, comme les documents XML contiennent leur propre définition, le schéma ou la DTD qui leur est associé peut ne plus les reconnaître et en déduire qu'ils ont été corrompus. Cela signifie que les documents ne pourront plus être correctement interprétés par la suite.

3.7 Mise à jour des données distantes

Certaines bases de données XML peuvent inclure des données distantes dans les documents qu'elles hébergent. Cela signifie que des modifications effectuées sur les données de base (stockées à l'extérieur de la BD) seront automatiquement propagées à tous les documents XML qui y font référence.

3.8 Références externes

Un problème lors du stockage des documents XML est de savoir comment gérer les entités externes. En effet, elles pourraient soit être intégrées au document, soit rester à leur emplacement original. Cette question ne peut être résolue qu'en prenant en compte le type d'entité auquel le document fait référence. S'il s'agit d'une entité pour laquelle le document exige une valeur constamment à jour, comme une information météo par exemple, ce serait une erreur de l'intégrer au document, car celle-ci serait tout de suite périmée. Par contre, s'il s'agit de données d'archives, celles-ci peuvent être insérées dans le document de manière définitive. En effet, si elles ne le sont pas et que le contenu original était modifié, le contenu du document final lui-même pourrait ne plus être correct.

3.9 Index

Comme dans les bases de données relationnelles, les BD XML natives permettent de créer des index sur le contenu afin d'effectuer des recherches plus rapides. Certains logiciels indexent automatiquement les documents, alors que d'autres, plus souples permettent de définir exactement les éléments qu'on désire indexer.

3.10 Normalisation

La normalisation consiste à ne représenter les données qu'une seule fois dans une base de données. Cela permet d'éviter les incohérences et le risque d'avoir à faire à des données redondantes. Même si les bases de données relationnelles sont prévues pour une normalisation optimale, rien n'empêche de créer un schéma relationnel de très mauvaise qualité. Il en est de même dans les BD XML, dans lesquelles on pourra stocker aussi bien des documents très bien structurés que d'autres qui ne répondent à aucune forme logique. La façon de concevoir les documents dès le départ est donc un facteur important pour avoir des données les plus propres possibles dans la BD. Certaines données peuvent être publiées sur tous les documents, mais ne pas présenter l'avantage d'être normalisées, car elles ne représentent qu'une fraction du document entier (comme l'en-tête par exemple). D'autres données, comme des noms ou des adresses, auront quant à elles tout intérêt à ne figurer qu'à un seul endroit. Pour répondre à ce problème, il y a la possibilité de mettre des liens dans les documents avec XLink *« XML Link Language »*, une norme du W3C, ou des liens propriétaires à la plate-forme utilisée. Certains outils de requête supportent les jointures entre documents. Les données ne sont stockées qu'à un seul endroit, ce qui simplifie nettement leur gestion. Il faut toutefois noter que les bases de données natives XML ne génèrent en principe pas ces liens automatiquement. Ces derniers devront donc être créés de manière manuelle, soit par l'utilisateur soit par une application.

3.11 Intégrité référentielle

Dans une base de données relationnelle, l'intégrité référentielle est contrôlée en s'assurant que chaque clé étrangère pointe sur une clé primaire valide et existante. Dans les BD XML natives, l'intégrité consiste à vérifier que tous les liens pointent sur des documents ou des fragments de documents valides. Ces fonctions de contrôle ne sont disponibles que sur certaines bases de données XML natives.

4. Classification des bases de données XML natives

Il existe deux grandes catégories d'architectures des bases de données XML natives : les architectures basées sur le texte et celles qui sont basées sur un modèle [24,1] :

4.1 Bases de données XML natives basées sur le texte

Une base de données XML native basée sur le texte stocke [1] le document XML en tant que texte. Cela peut être un fichier dans un système de fichiers, un BLOB dans une base relationnelle, ou un format propriétaire.

Les index sont communs à toutes les bases de données XML natives basées sur le texte. Ils permettent au moteur de recherche de naviguer facilement en tout point d'un document XML quelconque. Cela procure à ce genre de base un avantage considérable en matière de vitesse quand on recherche des documents entiers ou des fragments de documents ; la base peut en effet réaliser une seule consultation de l'index, positionner la tête de lecture du disque une seule fois, puis en supposant que le fragment requis est stocké dans des octets contigus sur le disque, retrouver le document entier ou un fragment en une seule lecture. Au contraire, ré-assembler un document à partir de morceaux comme on le fait avec une base relationnelle ou certaines bases XML natives basées sur un modèle demande de multiples consultations de l'index et de nombreuses lectures de disque. Cependant ce type de base de données XML native présente l'inconvénient de re-parser à chaque fois le document [38].

4.2 Bases de données XML natives basées sur le modèle

La seconde catégorie est constituée des bases XML natives basées sur un modèle [1]. Plutôt que de stocker un document XML en tant que texte, elles construisent un modèle objet interne du document et stockent ce modèle. La manière dont le modèle est stocké dépend de la base. Certains produits stockent le modèle dans une base relationnelle ou orientée objet. Stocker par exemple le DOM dans une base relationnelle pourrait conduire à des tables du genre Éléments, Attributs, PCDATA, Entités et Références Des Entités. D'autres bases utilisent un format de stockage propriétaire adapté à leur modèle. Les bases XML natives basées sur un modèle et construites sur d'autres bases possèdent vraisemblablement des performances similaires à ces bases sous-jacentes lors de la recherche des documents, et c'est pour la raison évidente qu'elles reposent sur ces systèmes pour retrouver les données. Cependant, la conception de la base -- tout particulièrement dans le cas des bases XML natives construites sur des bases relationnelles -- laisse place à des variations. Par exemple, à partir d'une base utilisant une stricte correspondance du DOM avec un modèle objet

relationnel, il pourrait en résulter un système qui sollicite l’exécution d’instructions SELECT distinctes pour retrouver les enfants de chaque nœud.

Lorsque l’on recherche les données dans l’ordre où elles sont stockées, les bases XML natives basées sur un modèle et qui utilisent un format de stockage propriétaire possèdent vraisemblablement des performances similaires aux bases XML natives basées sur le texte. Ceci est dû au fait que la plupart de ces bases utilisent des pointeurs physiques entre les nœuds, ce qui devrait fournir des performances similaires à la recherche textuelle [1]. (La vitesse dépend aussi du format de sortie. Les systèmes basés sur le texte sont manifestement plus rapides pour renvoyer des documents textuels, tandis que les systèmes basés sur un modèle sont indubitablement plus rapides pour renvoyer des arbres DOM, en supposant que leur modèle coïncide explicitement au DOM.)

À l’instar des bases XML natives basées sur le texte, les bases natives basées sur un modèle rencontrent probablement des problèmes de performance lorsque l’on recherche et retourne des données dans un format quelconque autre que celui sous lequel ces données sont stockées, par exemple lorsque on inverse l’hiérarchie de certaines de ses parties [1]. Cependant, elle présente l’avantage de combiner de fragments de données à partir de différents documents [36,38].

5. Domaines d’utilisation des bases de données XML natives

Les principales utilisations des bases de données XML natives incluent les domaines de l’intégration de données et l’évolution des schémas [24,39], que nous allons présenter dans les sections qui suivent.

5.1 Intégration des données

La base de données XML native est bien adaptée à l'intégration de données en raison de son modèle flexible de données. En outre, le langage de requête XQuery est un langage d'intégration de données en raison de sa simplicité, et de sa capacité de joindre des données à partir de différents documents (sources de données).

Les applications d'intégration de données doivent résoudre un certain nombre de problèmes : l’accès aux données, la sécurité et la gestion des données. La section suivante présente les architectures employées pour résoudre ces problèmes: les requêtes et le mapping des schémas.

5.1.1 Architectures de requêtes

Il existe deux architectures de requêtes pour l'intégration des données avec une base de données XML natives: locale et distribuée [24]. Dans l'architecture locale, les données sont importées dans la base de données sous forme XML et consultée localement. Dans une architecture distribuée, les données se trouvent dans des sources distantes, et le moteur de recherche repartit les requêtes à travers ces sources de données. Le moteur alors compile les résultats et les renvoie à l'application.

5.1.2 Maintien des schémas hétérogènes

Le plus grand problème dans l'intégration des données est la manipulation des schémas hétérogènes. Avec des différences structurales, le même concept est représenté différemment, par exemple on peut utiliser un ou plusieurs champs pour représenter un nom avec des différences sémantiques, des concepts légèrement différents sont représentés; ceux-ci peuvent être directe (un prix est en dollars ou euros) ou raffinés (un prix inclut une réduction). La gestion des différences de schémas est difficile, bien que quelques différences ne puissent pas être complètement résolues. La base de données XML native, fournit trois architectures, pour faire face à ce problème [24]:

- **Différentes adresses dans la requête**. La requête emploie différentes fonctions pour rechercher des données à partir de différentes collections.
- **Mapping des documents vers le même schéma**. Ceci permet à des applications d'ignorer les différences entre les schémas. les données peuvent être converties au moment de chargement, lorsque elles sont stockées et interrogées localement, ou au moment de l'exécution, lorsque les requêtes distribuées sont employées. Dans le dernier cas, les conversions sont établies dans des vues de XML. La conception du schéma commun nécessite seulement d'inclure les champs à interroger.
- **Construction des index communs sur les documents.** Quelques bases de données XML natives permettent aux utilisateurs d'établir leurs propres techniques d'indexation. Par exemple, une collection d'articles utilise plusieurs éléments "Auteur" tandis que d'autres utilisent un seul élément "Auteurs". Un seul index peut être utiliser pour les éléments "Auteur" et les éléments "Auteurs". Ceci permet à des applications de chercher les articles d'un auteur.

5.2 Evolution de schémas

Dans le monde XML, l'évolution de schémas est plus rapide, car XML est employé dans des domaines extrêmement évolutifs, tels que la finance et la biologie.

La manipulation de l'évolution de schéma n'est pas facile. La solution la plus facile est de mettre à jour des données pour se conformer au nouveau schéma et de mettre à jour les applications correspondantes. Malheureusement, ce n'est pas toujours possible. Lorsque les données ne peuvent pas être mises à jour, les application doivent maintenir la compatibilité postérieure et antérieure [24], car les documents conformément aux plusieurs versions d'un schéma sont généralement stockés ensemble dans les bases de données XML natives. Les applications doivent déterminer quelle version de schéma est employée, par la vérification de la version d'un attribut ou la vérification de l'existence d'un champ particulier.

Le maintien antérieur de la compatibilité nécessite beaucoup de calcul, tel que fournir des valeurs par défaut pour des champs supplémentaires dans le nouveau schéma ou traiter chaque version d'un champ différemment. Cependant, quelques problèmes n'ont aucune solution définitive, comme le calcul de la moyenne d'un champ non trouvé dans tous les documents.

Le maintien postérieur de la compatibilité, revient à protéger les applications contre les changements postérieurs. Une stratégie consiste à ignorer tous les champs non reconnus. Malheureusement, c'est risqué, car les nouveaux champs peuvent changer la sémantique des champs existants. Une stratégie plus conservatrice consiste à traiter les documents avec un nombre de version connu. Ceci permet à des applications de continuer de travailler jusqu'à ce qu'il puisse déterminer si un changement de schéma altère le code existant.

Une stratégie qui permet d'éviter les problèmes de compatibilité postérieure et antérieure consiste à interroger uniquement les champs peu susceptibles au changement [24]. Ceci est valable lorsque les humains sont impliqués. Par exemple, un représentant de service clientèle pourrait rechercher des contrats d'un client, ou un chercheur pourrait rechercher des documents décrivant une organisation particulière. Dans les deux cas, les recherches sont faites sur des champs stables (le nom de client ou d'espèces) et le lecteur peut résoudre les différences dans les schémas.

6. Avantages et inconvénients des bases de données XML natives

Malgré que les bases de données XML natives sont en cours d'évolution, les administrateurs des bases de données les utilisent, pour les mêmes raisons qu'ils ont considéré l'emploi des bases de données relationnelles au début des années 80. Il y a vingt-

cinq ans, les bases de données relationnelles étaient lentes, non standard. Néanmoins, elles avaient toujours beaucoup d'avantages comparés aux systèmes traditionnels.
Les avantages des BDXML natives peuvent être cernés dans les points suivants [41, 34] :

- **Tout est stocké dans le même endroit.** L'avantage le plus important d'une base de données XML native est qu'elle stocke tous les documents dans un seul endroit facile a contrôlé et a recherché.
- **Vues multiples des mêmes données.** L'avantage de stocker des données dans une base de données est de permettre des vues multiples d'un même contenu. Cette fonctionnalité est supportée par les bases de données XML natives comme dans les bases de données relationnelles.
- **Exécution.** Les requêtes sur une base de données XML natives sont simples et plus rapides que les requêtes sur les documents stockés dans un système de fichiers, ceci est pour plusieurs raisons. D'abord, la base de données permet l'indexation pour accélérer la recherche. La deuxième raison est que les documents stockés dans la base de données sont pré-analysés, Par conséquent, il n'est pas besoin de vérifier que le document et bien formé lors d'une requête, ou d'établir un modèle d'objet représentant ce document. Tous ces détails sont à l'intérieur de la base de données sous une forme que le moteur de requête peut l'explorer.
  Les bases de données XML utilisent des techniques d'optimisation du temps de l'exécution, comme la réécriture des requêtes.
- **Documents très grands.** Un autre avantage des bases de données XML natives est leurs capacités de traiter de grands documents. En utilisant des outils comme SAX, XQuery, XPath, et DOM.
- **Aucun bit n'est perdu.** Les bases de données XML natives peuvent retrouver le document original. Cette fonctionnalité est critique dans certaines situations légales où il est nécessaire de reproduire le document original jusqu'au dernier byte. Cette fonctionnalité peut également être importante dans le développement des logiciels, en particulier dans l'optimisation de l'exécution.

Comme c'est le cas de toute technologie, il y a quelques désavantages avec les bases de données XML natives [41]. Tout d'abord, il s'agit d'une technologie récente qui n'a pas encore été largement testée. De plus, les bases de données XML natives sont très chères, pouvant coûter jusqu'à 240 000 euro par processeur. On voit toutefois arriver des bases de

données XML natives à des prix moins prohibitifs. La plupart d'entre elles ont été développées par la communauté open-source et arrivent désormais sous une forme commerciale. Elles restent intéressantes à considérer, car elles proposent des fonctionnalités presque identiques à celles de leurs concurrentes. Si toutefois une société prévoit d'utiliser le XML à grande échelle ou à long terme, ces inconvénients ne sont pas si importants comparés aux avantages : performances améliorées, standardisation du XML et automatisation des processus commerciaux.

Par ailleurs les bases de données XML natives souffrent de certaines limitations fonctionnelles qui sont :

- Les langages de mise à jour proposés traitent des commandes simples, et ne réalisent pas la jointure entre les documents, dans les requêtes de mise à jour [2].
- L'absence d'un support pour la validation des opérations de mise à jour et de préservation des contraintes [2, 7,8].
- Les bases de données XML natives ne permettent pas aux utilisateurs d'interroger simultanément des documents XML, et d'autres données stockées dans une base de données relationnelle [44].

Dans notre travail nous nous sommes intéressés au problème de la validation des opérations de mise à jour et de préservation des contraintes, dans les bases de données XML natives. Pour cela nous allons présenter dans les sections qui suivent, les schémas XML permettant de définir les contraintes qui doivent être respectées par les documents XML.

7. Schémas des bases de données XML natives

Un schéma XML est un ensemble de définitions et de contraintes applicables aussi bien sur le contenu que sur la structure même d'un document XML. Dans la pratique, le schéma XML sera utilisé comme support de modélisation ainsi que pour la validation des documents XML. De la même façon qu'on ne conçoit pas une base de données sans avoir définit sa structure, on ne devrait pas construire un système d'information basé sur XML sans avoir conçu la ou les schémas associés [43]. Il existe plusieurs types de schémas XML les plus utilisés sont : les DTD et XML Schema.

7.1 DTD

Les DTDs (*Document Type Definition*) [19] sont actuellement supportées par de très nombreuses applications, mais la possibilité d'y définir finement un document XML est limitée. En effet, une DTD définit très peu de types de données pour la validation du contenu, et il est également impossible d'intégrer d'autres définitions existantes puisque le support des *namespaces* différenciant les langages n'est pas supporté. Enfin, dernier défaut, le langage n'est pas exprimé en XML, ce qui est en contradiction avec la forme même du langage défini.
Une DTD a pour objectif de spécifier quatre types d'information [43]:

1. Quels sont les éléments permis dans les documents ?
2. Quel contenu peut posséder les éléments ?
3. Quels attributs peuvent être associés à quels éléments ?
4. Quelles sont les valeurs permises pour les attributs ?

Les deux premiers types d'information s'effectuent à l'aide des déclarations d'éléments, alors que les deux autres s'effectuent à l'aide des déclarations d'attributs [43] :

7.1.1 Déclarations d'éléments

Les éléments sont déclarés de cette façon dans une DTD:

```
<!ELEMENT chapitre (titre? (para|liste)*, section+)>
```

Plusieurs informations sont fournies sur une telle ligne. D'abord, chapitre est le nom de l'élément, dans cet exemple. Ensuite, le modèle de contenu est déterminé entre parenthèse. Ces modèles de contenu peuvent être des éléments (déclarés par ailleurs) ou encore des éléments et du texte (des modèles de contenu mixtes). Pour comprendre l'exemple précédent, il faut connaître la signification de ces symboles:

,	Indique une séquence, c'est-à-dire que l'élément à gauche de la virgule doit précéder l'élément à droite.
|	Indique un choix, c'est-à-dire que l'on choisit soit l'élément à gauche de la barre verticale, soit l'élément à droite.
?	Indique un élément optionnel et qui ne peut pas être répété. Autrement dit, il a une cardinalité de 0 ou 1.
*	Indique un élément qui est optionnel mais qui peut être répété. Autrement dit, il a une cardinalité de 0 ou plusieurs, sans limite.
+	Indique un élément obligatoire et qui peut être répété. Il a donc une cardinalité de 1 ou plusieurs, sans limite.

Tableau 2.1 Déclaration d'éléments dans une DTD

Un élément peut également contenir du texte. Dans un tel cas, son modèle de contenu doit nécessairement être de la sorte:

```
<!ELEMENT para (#PCDATA|ville|état|personne)*>
```

Ainsi, le mot "#PCDATA" doit être au début, les différents éléments doivent être séparés par des "ou", et l'ensemble doit être optionnel et répétable. La signification d'un tel modèle de contenu est la suivante: L'élément para peut contenir du texte ou des éléments ville, état ou personne dans n'importe quel ordre, autant d'occurrences que nécessaires, mais sans obligation de les utiliser.

Ces possibilités de structuration peuvent paraître limitées, et elles le sont. Entre autres, il n'est pas possible de contrôler le contenu textuel des champs, par exemple pour limiter à une liste de termes ou pour un format de date.

7.1.2 Déclarations d'attributs

Les DTD permettent de déceler des attributs et leurs valeurs, et de les associer aux éléments. Voici un exemple de déclaration d'attributs:

```
<!ATTLIST chapitre
     id          ID         #REQUIRED
     niveau      NMTOKEN    #IMPLIED
     etiquette   CDATA      #FIXED "chapitre"
     acces       (public|restreint) "public"     >
```

Cette déclaration peut paraître complexe, mais il faut la regarder composante par composante. Tout d'abord, elle déclare les attributs associés à l'élément chapitre. Ensuite, on retrouve quatre attributs déclarés, un par ligne dans cet exemple.

Le premier attribut se nomme *id*, son type de données est ID, c'est-à-dire qu'il doit répondre à certains critères (essentiellement composé de lettres et de chiffres, doit débuter par une lettre) et surtout que sa valeur doit être unique parmi tous les attributs de type ID dans le document, et enfin cet attribut est obligatoire, c'est-à-dire qu'il doit se retrouver dans toutes les balises d'ouverture des éléments chapitre. Ensuite, l'attribut *niveau* est un nom, semblable à un ID mais pas nécessairement unique, il n'est pas obligatoire et il n'y a pas d'autres restrictions sur son contenu. L'attribut *etiquette* est de type CDATA, c'est-à-dire qu'il peut contenir du texte quelconque, il a toujours la même valeur fixe, soit `chapitre`. Enfin, l'attribut *acces* peut posséder deux valeurs, soit `public` ou `restreint`, et par défaut il a la valeur `public`.

7.2 XML Schema

La DTD ce format descriptif, souffre de nombreuses déficiences qu'on peut les résumer dans les points suivants [19,23] :

1. Premièrement, les DTD ne sont pas au format XML. Cela signifie qu'il est nécessaire d'utiliser un outil spécial pour parser un tel fichier, différent de celui utilisé pour l'édition du fichier XML.
2. Deuxièmement, les DTD ne supportent pas les "espaces de nom. En pratique, cela implique qu'il n'est pas possible, dans un fichier XML défini par une DTD, d'importer des définitions de balises définies par ailleurs.
3. Troisièmement, le "typage" des données est extrêmement limité.

XML Schema [23] est conçu pour pallier aux limites pré-citées du DTD. XML Schema propose, en plus des fonctionnalités fournies par les DTD, des nouveautés :

- Le typage des données est introduit, ce qui permet la gestion de booléens, d'entiers, d'intervalles de temps... Il est même possible de créer de nouveaux types à partir de types existants.
- La notion d'héritage. Les éléments peuvent hériter du contenu et des attributs d'un autre élément. C'est sans aucun doute l'innovation la plus intéressante de XML Schema.
- Le support des espaces de nom.
- Les indicateurs d'occurrences des éléments peuvent être tout nombre non négatif (rappel : dans une DTD, on était limité à 0, 1 ou un nombre infini d'occurrences pour un élément).

- Les schémas sont très facilement concevables par modules.

Le but de XML Schema est de définir une classe de documents XML. Il permet de décrire les autorisations d'imbrication et l'ordre d'apparition des éléments et de leurs attributs, tout comme une DTD. Mais il permet aussi d'aller au-delà. A ce jour, XML Schema est le langage le plus complet, mais aussi le plus complexe. Il est à peu près possible de tout définir en XML Schéma, comme d'inclure d'autres définitions grâce au support des namespaces, de créer ses propres types de données en dérivant les types déjà existants, etc.

Dans les sections suivantes nous allons présenter à travers des exemples quelques fonctionnalités du langage XML Schema [23] :

7.2.1 Structure de base

Comme tout document XML, un XML Schema commence par un prologue, et un élément racine. L'élément racine est l'élément xsd:schema, et tout élément d'un schéma doit commencer par le préfixe `xsd`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"> <!--
 déclarations d'éléments, d'attributs et de types ici -->
</xsd:schema>
```

7.2.2 Déclarations d'éléments

Un élément, dans un schéma, se déclare avec la balise `<xsd:element>`. Par exemple,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"> <xsd:element
name="contacts" type="typeContacts"></xsd:element>
<xsd:element name="remarque" type="xsd:string"></xsd:element>
<!-- déclarations de types ici -->
</xsd:schema>
```

Le schéma précédent déclare deux éléments : un élément `contacts` et un élément `remarque`. Chaque élément est "typé" -c'est-à-dire qu'il doit respecter un certain format de données. L'élément `contacts` est ainsi du type `typeContacts`, qui est un type complexe défini par l'utilisateur. L'élément remarque quant à lui est du type `xsd:string` qui est un type simple prédéfini de `XML Schema`.

Chaque élément déclaré est associé à un type de données via l'attribut `type`. Les éléments pouvant contenir des élément-enfants ou possédant des attributs sont dits de type *complexe*, tandis que les éléments n'en contenant pas sont dits de type *simple*.

A. Types simples

Les types de données simples ne peuvent comporter ni attributs, ni éléments enfants. Il en existe de nombreux prédéfinis, mais il est également possible d'en "dériver" de nouveaux. Nombreux sont les types prédéfinis dans la bibliothèque de XML Schema. La liste détaillée figure sur le site du W3C.

Les types listes sont des suites de types simples (ou *atomiques*). XML Schema possède trois types de listes intégrés : `NMTOKENS`, `ENTITIES` et `IDREFS`. Il est également possible de créer une liste personnalisée, par "dérivation" de types existants. Par exemple,

```
<xsd:simpleType name="numéroDeTéléphone">
  <xsd:list itemType="xsd:unsignedByte" />
</xsd:simpleType>
```

Un élément conforme à cette déclaration serait `<téléphone>01 44 27 60 11</téléphone>`.

Il est également possible d'indiquer des contraintes plus fortes sur les types simples ; ces contraintes s'appellent des "facettes". Elles permettent par exemple de limiter la longueur de notre numéro de téléphone à 10 nombres.

Les listes et les types simples intégrés ne permettent pas de choisir le type de contenu d'un élément. On peut désirer, par exemple, qu'un type autorise soit un nombre, soit une chaîne de caractères particuliere. Il est possible de le faire à l'aide d'une déclaration d'union. Par exemple, sous réserve que le type simple `numéroDeTéléphone` ait été préalablement défini (voir précédemment), on peut déclarer...

```
<xsd:simpleType name="numéroDeTéléphoneMnémoTechnique">
  <xsd:union memberTypes="xsd:string numéroDeTéléphone" />
</xsd:simpleType>
```

Les éléments suivants sont alors des "instances" valides de cette déclaration :

```
<téléphone>18</téléphone>
<téléphone>Pompiers</téléphone>
```

B. Types complexes

Un élément de type simple ne peut contenir de sous-élément. Il est nécessaire pour cela de le déclarer de type "complexe". On peut alors déclarer, des séquences d'éléments, des types de choix ou des contraintes d'occurrences.

- **Séquences d'éléments**

L'élément xsd:sequence, permet de définir une séquence d'éléments ainsi…

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string" />
    <xsd:element name="prénom" type="xsd:string" />
    <xsd:element name="dateDeNaissance" type="xsd:date" />
    <xsd:element name="adresse" type="xsd:string" />
    <xsd:element name="adresseElectronique" type="xsd:string" />
    <xsd:element name="téléphone" type="numéroDeTéléphone" />
  </xsd:sequence>
</xsd:complexType>
```

... est équivalent à une déclaration d'élément, dans une DTD, où apparaîtrait (nom, prénom, dateDeNaissance, adresse, adresseElectronique, téléphone).

- ***Choix d'élément***

On peut vouloir modifier la déclaration de type précédente en stipulant qu'on doive indiquer soit l'adresse d'une personne, soit son adresse électronique. Pour cela, il suffit d'utiliser un élément xsd:choice :

```
<xsd:complexType name="typePersonne">
  <sequence>
    <xsd:element name="nom" type="xsd:string" />
    <xsd:element name="prénom" type="xsd:string" />
    <xsd:element name="dateDeNaissance" type="xsd:date" />
    <xsd:choice>
      <xsd:element name="adresse" type="xsd:string" />
      <xsd:element name="adresseElectronique" type="xsd:string" />
    </xsd:choice>
  </sequence>
  <xsd:element name="téléphone" type="numéroDeTéléphone" />
</xsd:complexType>
```

Ce connecteur a donc les mêmes effets que l'opérateur | dans une DTD.

- **L'élément all**

Cet élément est une nouveauté par rapport aux DTD. Il indique que les éléments enfants doivent apparaître une fois (ou pas du tout), et dans n'importe quel ordre. Cet élément xsd:all doit être un enfant direct de l'élément xsd:complexType. Par exemple...

```
<xsd:complexType>
  <xsd:all>
    <xsd:element name="nom" type="xsd:string" />
    <xsd:element name="prénom" type="xsd:string" />
    <xsd:element name="dateDeNaissance" type="xsd:date" />
```

```
    <xsd:element name="adresse" type="xsd:string" />
    <xsd:element name="adresseElectronique" type="xsd:string" />
    <xsd:element name="téléphone" type="numéroDeTéléphone" />
  </xsd:all>
</xsd:complexType>
```

... indique que chacun de ces éléments peut apparaître une fois ou pas du tout (équivalent de l'opérateur `?` dans une DTD), et que l'ordre des éléments n'a pas d'importance (cela n'a pas d'équivalent dans une DTD).

- **Indicateurs d'occurrences**

Dans une DTD, un indicateur d'occurrences ne peut prendre que les valeurs 0, 1 ou l'infini. On peut forcer un élément `sselt` à être présent 378 fois, mais il faut pour cela écrire `(sselt, sselt..., sselt, sselt)` 378 fois. XML Schema permet de déclarer directement une telle occurence, car tout nombre entier non négatif peut être utilisé. Pour déclarer qu'un élément peut être présent un nombre illimité de fois, on utilise la valeur `unbounded`. Les attributs utiles sont `minOccurs` et `maxOccurs`, qui indiquent respectivement les nombres minimal et maximal de fois où un élément peut apparaître. Le tableau suivant récapitule les possibilités :

Dans une DTD	Valeur de minOccurs	Valeur de maxOccurs
*	0	unbounded
+	1 (pas nécessaire, valeur par défaut)	unbounded
?	0	1 (pas nécessaire, valeur par défaut)
rien	1 (pas nécessaire, valeur par défaut)	1 (pas nécessaire, valeur par défaut)

Tableau 2.2 Liste des indicateurs d'occurrences

7.2.3 Déclarations d'attributs

A la différence des éléments, un attribut ne peut être que de type simple. Cela signifie que les attributs, comme avec les DTD, ne peuvent contenir d'autres éléments ou attributs. De plus, les déclarations d'attributs doivent être placées *après* les définitions des types complexes, autrement dit, après les éléments `<xsd:sequence>`, `<xsd:choice>` et `<xsd:all>`. L'exemple suivant montre la déclaration d'un attribut `maj` de type `xsd:date` (un autre type simple) qui indique la date de dernière mise à jour de la liste des contacts.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

 <xsd:element name="contacts" type="typeContacts"></xsd:element>
  <xsd:element name="remarque" type="xsd:string"></xsd:element>
  <!-- déclarations de types ici -->
```

```
  <xsd:complexType name="typeContacts">
    <!-- déclarations du modèle de contenu ici -->
    <xsd:attribute name="maj" type="xsd:date" />
  </xsd:complexType>
</xsd:schema>
```

Tout comme dans une DTD, un attribut peut avoir un indicateur d'occurrences. L'élément `attribute` d'un Schema XML peut avoir trois attributs optionnels : `use`, `default` et `fixed`. Des combinaisons de ces trois attributs permettent de paramétrer ce qui est acceptable ou non dans le document XML (attribut obligatoire, optionnel, possédant une valeur par défaut...). Par exemple, la ligne suivante permet de rendre l'attribut `maj` optionnel, avec une valeur par défaut égale à 11 octobre 2003 s'il n'apparaît pas.

```
<xsd:attribute name="maj" type="xsd:date" use="optional" default="2003-10-11" />
```

Quand l'attribut `fixed` est renseigné, la seule valeur que peut prendre l'attribut déclaré est celle de l'attribut `fixed`.

7.2.4 Espaces de nom

La notion d'espace de nom est complexe ; elle permet à un document XML quelconque d'utiliser les balises définies dans un schéma donné... quelconque.

Dans un document XML schema, la déclaration suivante : `<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">` signifie que l'espace de nom auquel ces balises font référence, là où elles sont définies, est un schéma particulier, que l'on peut consulter.

7.3 Choix du schéma à utiliser

Même si le langage DTD est le moins riche, il est toujours utilisé dans de très nombreux produits. Il est surtout utilisé pour la description de grammaire XML documentaires comme DocBook XML. Les modèles documentaires XML, des outils de publication et de gestion de contenu utilisent des DTDs. Les DTDs peuvent être utilisées dans les cas suivants [19] :

- Structure des documents XML peu évolutive.
- Structure des documents XML ne demandant qu'une validation structurelle.
- Pas de besoin d'intégration de structures XML externes.

L'utilisation de XML Schema sera préférée pour ces raisons [19] :

- Validation du contenu.
- Structure des documents XML amené à évoluer.
- Intégration de structures externes définies par des namespaces.
- Définition de langages d'échanges XML-RPC.

La procédure pour la validation des documents XML vis-à-vis le document XML Schema associé est décrite dans [27].

8. Mise à jour et validation des documents XML

Il existe trois stratégies principales permettant la mise à jour des documents XML dans une base de données XML native [18, 7,8] :

- La première stratégie consiste à utiliser des langages de mise à jour permettant d'effectuer les mises à jour dans le serveur.
- La deuxième stratégie consiste à utiliser le langage XUpdate[40], la norme proposée par XML : DB pour la mise à jour des fragments de documents XML.
- La troisième stratégie qui est adaptée par la plupart des bases de données XML natives consiste à rechercher le document XML, et le modifier en utilisant une API XML, puis le restaurer dans la base.

Différentes approches utilisées, ce qui limite l'interchangeabilité de la base de données. De ce fait, la mise à jour constitue un point de faiblesse pour les bases de données XML natives, les travaux de recherches à l'heure actuelle, se focalisent à fournir des langages de mise à jour [6, 3,10] en vue de concevoir un standard. L'optimisation des requêtes de mise à jour constitue une autre problématique que plusieurs travaux de recherches se focalisent à fournir des solutions [51,50].

Par ailleurs, les langages de mise à jour proposés ne considèrent pas la vérification des contraintes, de ce fait les applications XML utilisent l'approche de validation statique [5,49,4] qui consiste à revalider tout le document à chaque opération de mise à jour, ce qui induit un coût élevé de l'opération en terme du temps de validation. Pour faire face à ce problème les travaux de recherche proposent une autre alternative qui est l'approche de validation incrémentale, où seule la partie du document concernée directement par l'opération de mise à jour est re-vérifiée.

Dans notre contexte nous nous sommes focalisés sur la validation des documents XML. Dans ce qui suit nous allons détailler les deux techniques mentionnées ci-dessus, en

décrivant les outils associés et quelques travaux existants qui réalisent l'approche de validation incrémentale.

8.1 Validation statique des documents XML

L'exécution des requêtes de mise à jour nécessite obligatoirement la vérification de la validation des documents résultants. Ce processus est illustré dans la figure (fig2.1) :

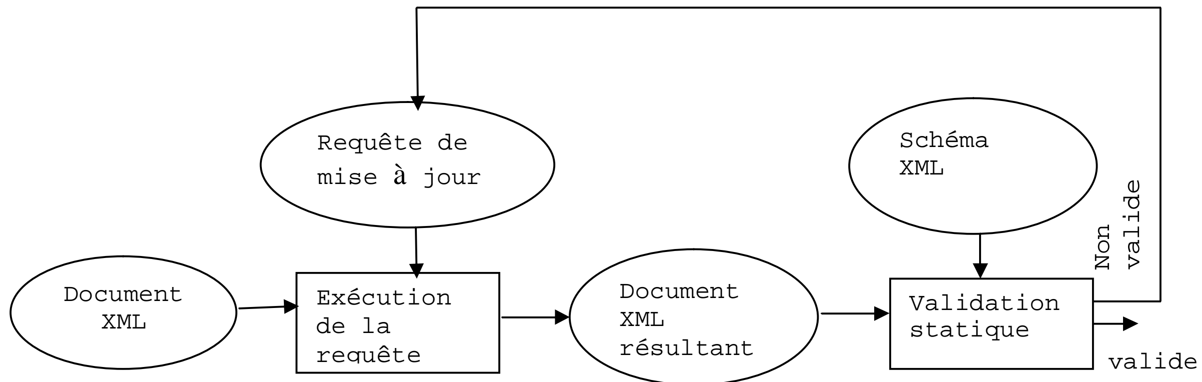

Fig.2.1 Processus de la validation statique [4]

L'approche de la validation statique utilisée par les bases de données XML natives Consiste à [4] :

1. Revalider tout le document résultant.
2. Reconsidérer le document original par l'annulation de la requête de mise à jour, si cette dernière à causer une violation des contraintes spécifiées par le schéma XML.

La revalidation des documents XML se fait à l'aide des parsers XML. Les analyseurs XML sont divisés selon l'approche qu'ils utilisent pour traiter le document. On distingue actuellement deux types d'approches :

- Les API utilisant une approche hiérarchique : les analyseurs utilisant cette technique construisent une structure hiérarchique contenant des objets représentant les éléments du document, et dont les méthodes permettent d'accèder aux propriétés. La principale API utilisant cette approche est DOM (Document Object Model)
- Les API basés sur un mode événementiel permettent de réagir à des événements (comme le début d'un élément, la fin d'un élément) et de renvoyer

le résultat à l'application utilisant cette API. SAX (Simple API for XML est la principale interface utilisant l'aspect événementiel.

8.1.1 DOM

DOM [20] est une spécification du W3C définissant la structure d'un document sous forme d'une hiérarchie d'objets, afin de simplifier l'accès aux éléments constitutifs du document.

Plus exactement DOM est un langage normalisé d'interface (API, *Application Programming Interface*), indépendant de toute plateforme et de tout langage, permettant à une application de parcourir la structure du document et d'agir dynamiquement sur celui-ci.

DOM se divise en deux spécifications :

- La spécification DOM level 1 (*DOM niveau 1*) se séparant en deux catégories
  - *Core DOM level 1*: La spécification pour les documents en général (dont XML)
  - *HTML DOM level 1*: La spécification retenant uniquement les méthodes applicables à HTML
- La spécification DOM level 2 ajoutant de nouvelles fonctionnalités comme la prise en compte des feuilles de style CSS (Cascading Style Sheets), dans la hiérarchie d'objets. La figure (fig. 2.2) montre l'arbre DOM d'un document XML

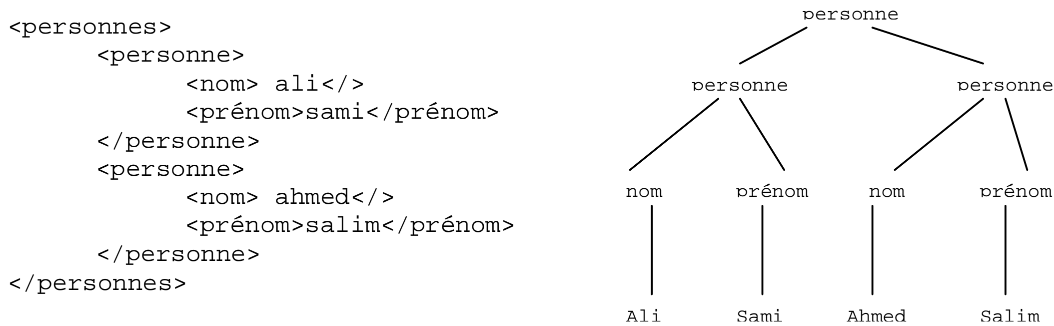

Fig.2.2 Arbre DOM du document XML

8.1.2 SAX

SAX [20] est une API basée sur un modèle événementiel, cela signifie que SAX permet de déclencher des événements au cours de l'analyse du document XML. Une application utilisant SAX implémente généralement des gestionnaires d'événements, lui permettant d'effectuer des opérations selon le type d'élément rencontré.

Soit le document XML suivant :

```
<personne>
<nom>Ahmed</nom>
<prénom>Ali</prénom>
</personne>
```

Une interface événementielle telle que l'API SAX permet de créer des événements à partir de la lecture du document ci-dessus. Les événements générés seront :

```
start document
start element: personne
start element: nom
characters: Ahmed
end element: nom
start element: prenom
characters: Ali
end element: prenom
end element: personne
end document
```

Ainsi, une application basée sur SAX peut gérer uniquement les éléments dont elle a besoin sans avoir à construire en mémoire une structure contenant l'intégralité du document.

L'API SAX définit les quatre interfaces suivantes :

- **DocumentHandler** possédant des méthodes renvoyant des événements relatifs au document :
  - *startDocument()* renvoyant un événement lié à l'ouverture du document
  - *startElement()* renvoyant un événement lié à la rencontre d'un nouvel élément
  - *characters()* renvoyant les caractères rencontrés
  - *endElement()* renvoyant un événement lié à la fin d'un élément
  - *endDocument()* renvoyant un événement lié à la fermeture du document
- **ErrorHandler** possèdant des méthodes renvoyant des événements relatifs aux erreurs ou aux avertissements
- **DTDHandler** renvoie des événements relatifs à la lecture de la DTD du document XML
- **EntityResolver** permet de renvoyer une URL lorsqu'une URI est rencontrée

8.1.3 Comparaison entre DOM et SAX

L'utilisation du DOM est appropriée lorsque les documents XML sont orientés documents, et dans le cas de développement d'un système de gestion de documents tandis que l'utilisation de SAX est approprié dans le cas où le contenu des documents XML est orienté données.

La complexité en temps de la validation statique des contraintes structurelles (les contraintes sur les éléments) en utilisant SAX et DOM est comme suit [49] :

Soit n le nombre des nœuds dans le document XML et d le nombre des nœuds dans le schéma XML associé. Le temps nécessaire pour déterminer l'état prochain de l'automate est $log\ d$. Dans le modèle SAX, les éléments dans le document XML sont visités par ordre de profondeur, c-à-d simuler un automate pour chaque élément ouvert. La complexité en temps de la validation statique en utilisant SAX est $O(n \log d)$. Pour le modèle DOM, nous pouvons valider chaque élément à la fois, à un coût de $O(n(\log n + \log d))$. Notons qu'il est possible de construire une représentation DOM d'un document tout en effectuant une validation avec SAX, $O(\log n)$ est le coût supplémentaire pour chaque insertion dans la structure de données.

8.2 Validation incrémentale des requêtes de mise à jour

Comme nous l'avons déjà mentionné, l'approche de la validation statique est inefficace car elle induit une redondance lors de la revalidation du document XML tout entier, ce qui induit un coût élevé de l'opération en terme du temps de validation, en particulier, lorsque la base de données est exposée à des mises à jour fréquentes, ou bien la taille des documents XML est importante. En vue de minimiser ce coût, les travaux de recherche se focalisent à fournir des approches de validation incrémentale qui consiste à :

- Vérifier la validation des documents avant l'exécution effective de l'opération de mise à jour.
- La vérification touche uniquement le sous arbre affecté par l'opération de mise à jour.

Dans ce qui suit nous allons présenter quelques travaux existants qui fournissent des approches pour la validation incrémentale des documents XML :

A. L'approche proposée par [8,7] :

Cette approche fournit un ensemble d'algorithmes pour la validation de la contrainte d'agrégation [7] et d'association [8], spécifiées à l'aide de XML Schema.

L'idée de base pour la validation de la contrainte d'association c'est-à-dire la validation de la contrainte d'intégrité référentielle, consiste à regrouper toutes les clés et les clés étrangères

dans un seul élément nommé *Group Key*. La limite de cette approche, est qu'elle est applicable uniquement lors d'une mise à jour d'un nœud simple.

B. L'approche proposée par [4]

Cette approche fournit des algorithmes pour la préservation de la contrainte de cardinalité et la contrainte d'apparition des attributs sous leurs éléments parent. Ces contraintes sont spécifiées à l'aide de XML Schema. SAXE est le prototype qui implémente ces algorithmes sous le langage de mise à jour XML [3].

Cette approche présente l'inconvénient d'adresser uniquement ces deux contraintes sans prise en considération, d'autres contraintes telles que l'ordre, la disjonction exclusive et l'intégrité référentielle.

C. L'approche proposée par [5]

Cette approche définit un automate d'arbre qui capture la grammaire définie par la DTD. Si l'état de l'élément racine du sous arbre affecté par l'opération de mise à jour est préservé, alors le document XML est valide du point de vue structurel. Afin de vérifier la validité des contraintes d'intégrité référentielle, l'algorithme proposé consiste à regrouper toutes les valeurs des clés dans une liste et les valeurs des clés étrangères dans une autre liste. Lors d'une opération de mise à jour, ces deux listes vont être mise à jour suivant le type de requête. Dans le cas d'une opération de suppression, leurs valeurs vont être diminuées en éliminant celles qui apparaissent dans le sous arbre à supprimer, et dans le cas d'une opération d'insertion ces valeurs vont être augmenter en ajoutant celles qui se trouvent dans le sous arbre à insérer.

La contrainte d'intégrité référentielle est préservée si les deux conditions suivantes sont satisfaites :

- Il n'y a aucune duplication d'une valeur clé.
- Toute valeur d'une clé étrangère référence une clé existante.

Exemple :

Nous allons présenter dans ce qui suit un exemple qui illustre cette approche :

Soient les règles de transition suivante :

$q_{prix}, q_{description} \rightarrow q_{produit}$

$q_{valeur} \rightarrow q_{prix}$

$\phi \rightarrow q_{valeur}$

L'application de cet automate sur un arbre XML est décrite dans la figure suivante :

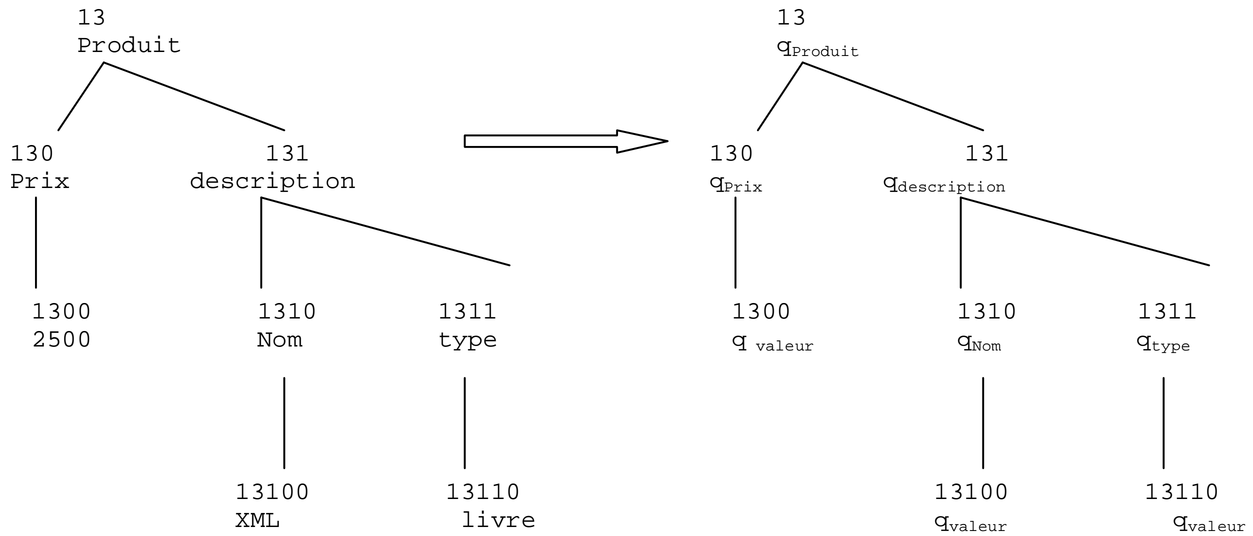

Fig.2.3 Exécution de l'automate sur l'arbre XML

Considérons à titre d'exemple la requête de l'insertion de l'élément `prix` après la position `130`. Cette requête va être rejetée car le mot généré par la concaténation des états des éléments fils de l'élément `produit` " q_{prix} q_{prix} $q_{description}$ " n'appartient pas au langage défini par les expressions régulières.

D. L'approche proposée par [49]

Similaire à sa précédente, cette approche utilise le principe de l'automate pour la validation des contraintes structurelles. En ce qui concerne la validation des contraintes d'intégrité référentielles, elle utilise une structure de données qui inclut toutes les valeurs des clés, ce qui permet de vérifier la non duplication des clés, et de s'assurer que chaque clé étrangère référence une clé existante. Pour vérifier que l'exécution d'une requête de suppression d'un élément ne va pas induire une référence vers une entité non existante, est réalisé à l'aide d'un compteur de référence. Ce compteur stocke le nombre des références vers les clés sous cet élément. Si la valeur de ce compteur est égale à zéro alors la requête de suppression est exécutée si non elle est rejetée. La suppression ou l'insertion d'une clé étrangère implique respectivement la décrémentation et l'incrémentation de la valeur du compteur.

E. Discussion

L'étude de ces travaux nous a permis d'obtenir les résultats suivants :

- La vérification des contraintes structurelles en utilisant l'automate, inclut tous les éléments fils de l'élément racine dans le sous arbre affecté par la requête de mise à jour [49,5].
- La vérification des contraintes d'intégrité référentielles dans [5,8], considère un espace de recherche contenant toutes les valeurs des clés et des clés étrangères existantes.

Dans le chapitre suivant nous allons proposer une approche de validation incrémentale telle que, la vérification des contraintes structurelles considère uniquement le sous élément à supprimer ou à insérer. Et la vérification des contraintes d'intégrité référentielle considère uniquement les valeurs des clés ou des clés étrangères correspondantes.

9. Conclusion

Les bases de données XML natives ne pourront pas remplacer les BD relationnelles. Ce n'est d'ailleurs pas leur but, puisqu'elles ont toutes les deux des objectifs différents mais parfaitement complémentaires. Alors que les BD relationnelles sont plus adaptées au stockage de "données". Les bases de données XML natives sont plus adaptées au stockage de "documents", comme explicité, elles peuvent également servir à stocker des documents qui ne sont pas totalement structurés ou même des documents n'ayant pas du tout de schéma (DTD ou de Schema). Néanmoins, les bases de données XML natives sont en cours de développement, les travaux de recherches à l'heure actuelle se focalisent sur la proposition des solutions permettant une gestion efficace des documents, telles que, la proposition des langages de mise à jour puissants, et la proposition des techniques pour la préservation de l'intégrité des documents stockés dans la base.

Chapitre III : Une approche de validation incrémentale des documents XML

1. Introduction

Au cours de ces dernières années, le besoin de stocker les documents XML dans une base de données XML native a émergé rapidement. L'idée consiste à stocker les documents sous leur forme naturelle d'arbre. Cependant, cette nouvelle technologie de base de données souffre de certaines limitations fonctionnelles, le problème de la mise à jour des documents constitue son point de faiblesse majeure.

Nous rappelons qu'il existe différentes techniques pour effectuer la mise à jour dans les bases de données XML natives [3, 6, 10,40], Les solutions proposées souffrent d'un manque de mécanisme pour la validation des document résultants. Par conséquent, après l'exécution des requêtes de mise à jour, la base de données contient des duplications des clés, des références vers des entités non existantes et beaucoup d'autres problèmes qui indiquent l'intégrité faible de la base de données.

Les contraintes qui doivent être respectées par un document XML, sont décrites à l'aide d'un schéma XML (DTD ou XML Schema). Un document XML est un document valide s'il respecte toutes les contraintes imposées dans le schéma associé. Lors d'une mise à jour d'un document XML valide, la vérification de la validité du document résultant est nécessaire, pour cela, la plupart des applications XML utilisent la validation statique dites *from scrach* [5,4,49] qui consiste à parcourir tout le document XML en visitant tous les noeuds de l'arbre, ce qui induit un coût élevé de l'opération en terme du temps de validation, en particulier lorsque les mises à jour sont fréquentes, ou bien la taille de document est grande. Du fait que cette approche est non efficace, les travaux de recherche à l'heure actuelle se focalisent à proposer des approches de validation incrémentale [5, 8, 4,49], où seules les parties concernées par la mise à jour sont re-vérifiées.

Dans ce chapitre nous proposons une approche pour la validation incrémentale. Elle est constituée d'un ensemble d'algorithmes, et des formalismes de description des contraintes. La méthode est assurément très utile, car elle permet d'optimiser la durée de validation des requêtes tout en assurant l'intégrité de la base. Dans notre contexte nous considérons la

validation des opérations d'insertion et de suppression des sous arbres vis-à-vis la préservation des contraintes d'intégrité référentielles et les contraintes structurelles.

2. Aperçu général de l'approche proposée

Nous considérons dans notre approche la validation des requêtes de mise à jour émises sur l'arbre XML, pour assurer la préservation des contraintes spécifiées à l'aide du langage XML Schema.

La validation incrémentale des contraintes ne peut être discutée sans considérer un langage de mise à jour. La proposition d'un langage de mise à jour XML n'entre pas dans l'objectif de ce travail, de ce fait nous allons nous contenter d'un ensemble d'opérations de base, qui consistent à l'insertion et la suppression des sous arbres:

Supprimer (x) **:** où "x" définit la racine de sous arbre à supprimer à partir du document XML.

Insérer(x, y) : cette opération consiste à insérer le sous arbre "y" comme un dernier fils de "x".

Inserer_apres(x, y) **:** cette opération consiste à insérer le sous arbre "y" après "x".

Le processus de la mise à jour des documents XML nécessite un mécanisme de validation incrémentale, afin de maintenir l'état cohérent du document XML modifié. Ceci est assuré par le rejet de toute requête de mise à jour menant à une violation des contraintes imposées par XML Schema, et l'acceptation de celles qui les maintiennent. Ces contraintes incluent les contraintes d'intégrité référentielles et les contraintes structurelles.

Dans notre approche nous avons utilisé les tables pour spécifier les contraintes structurelles, et nous avons défini un processus de marquage permettant de définir les valeurs des clés et des clés étrangères dans l'arbre XML.

2.1 Modèle arbre du document XML

Le document XML est une structure hiérarchique composée d'un arbre étiqueté. Chaque noeud a une position et une étiquette (par exemple, la position "1" est associée à l'étiquette *étudiant*). La figure (fig. 3.1) décrit le modèle arbre d'un document XML. Les attributs sont précédés par le symbole @ pour les distinguer des éléments simples.

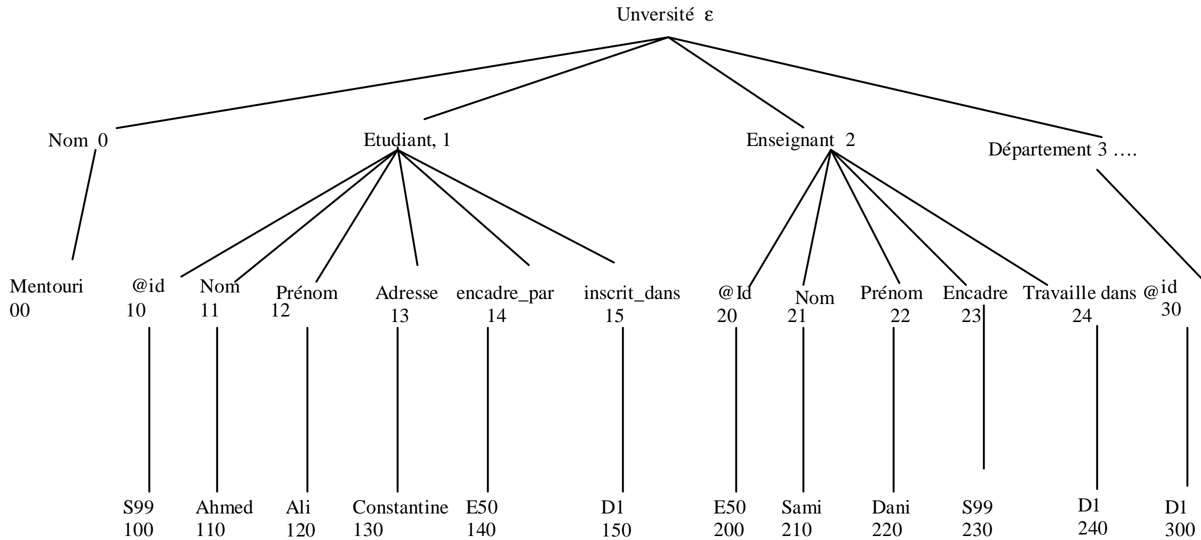

Fig.3.1 Exemple d'un document XML

Le passage d'un document XML vers le modèle d'arbre peut être réalisé à l'aide des outils bien connus comme SAX et DOM.

Définition.1 : *Soit N_a l'ensemble des nœuds dans l'arbre XML tel que : $N_a = E_a \cup A_a$.*

"E_a" l'ensemble d'éléments et "A_a" l'ensemble d'attributs dans l'arbre "a", chaque élément $e \in E_a$ ou attribut $a \in A_a$ est définit par [nom, type, valeur, position] tel que:

nom est le nom de l'élément ou l'attribut, type est le type de l'élément ou l'attribut, valeur est la valeur de l'attribut ou de l'élément simple, position est la position de l'élément ou l'attribut dans l'arbre XML

Le document XML est associé à un schéma XML (DTD ou XML Schema). Le schéma permet de spécifier les contraintes qui doivent être respectées par le document XML. Un document XML est un document valide s'il respecte toutes les contraintes définies dans le schéma XML, la validité est une propriété importante qui doit être préservée lors des requêtes de mise à jour.

Dans notre contexte nous considérons les contraintes d'intégrité référentielles et structurelles définies à l'aide de XML Schema.

2.2 Contraintes d'intégrité référentielles

La contrainte d'intégrité référentielle décrit la relation qui existe entre deux objets, qui doivent avoir la même valeur de référence lors de l'instanciation. XML Schema fournit la possibilité de spécifier la contrainte d'intégrité référentielle, à l'aide de primitives *Key et Keyref* [9]. Par exemple :

```
<key name="ENSEIGANT_Id">
      <selector xpath="ENSEIGANT"/>
      <field xpath="@id"/>
</key>
<keyref name="ÉTUDIANT _Id_ Ref" refer="ENSEIGANT_Id">
      <selector xpath="ÉTUDIANT"/>
      <field xpath="@encadre-par"/>
</keyref>
```

Ce fragment du XML Schema, montre que l'attribut `encadre-par` <**field xpath** = "@encadre_par"/> de l'élément `étudiant`, <**selector xpath**="ÉTUDIANT"/> référence l'attribut `'id'` de l'élément `enseignant`.

La contrainte d'intégrité référentielle, peut être violée lorsque la requête de mise à jour mène à :

– Une suppression d'une clé qui est référencée.

– Un ajout d'une clé qui existe déjà (duplication).

– Un ajout d'une clé étrangère qui référence une clé qui n'existe pas.

2.3 Contraintes structurelles

Les contraintes structurelles regroupent l'ensemble des contraintes d'attributs et d'éléments.

Les contraintes sur les attributs dans XML Schema, spécifient la propriété de l'adhésion de l'attribut à son élément père [9].

La déclaration *"use = required"* indique que l'attribut doit apparaître sous l'élément père, tandis que la déclaration *"use = optional"* exprime la propriété de l'adhésion faible qui existe entre l'attribut et l'élément père. Par exemple :

```
<xsd:complexType name="ENSEIGNANT Type">
<xsd:attribute name="NOM" type="xsd:string" use="required"/>
</xsd:complexType>
```

La déclaration *''use = required''* indique que l'attribut `'nom'` doit apparaître sous l'élément `enseignant`. Ce qui n'est pas le cas, dans le fragment du XML Schema suivant :

```
<xsd:complexType name="livraison Type">
<xsd:attribute name="date-livrason"type="xsd:string"use="optional"/>
</xsd:complexType>
```

La déclaration *''use = optional''* ici, exprime la propriété de l'adhésion faible, qui existe entre l'attribut `date-livraison` et l'élément `livraison`, c.à.d la date de livraison dépends de la livraison si elle est livrée, de ce fait, son occurrence n'est pas obligatoire.

La violation de la contrainte sur l'attribut, se produit lorsque la requête de mise à jour conduit à :

– Une suppression d'un attribut avec la primitive *use = "required"*.

Les contraintes sur les éléments incluent la cardinalité, l'ordre et la disjonction exclusive [9].

- La contrainte de cardinalité spécifie l'occurrence minimale et l'occurrence maximale d'un élément. Par exemple

```
<xsd:complexType name="ADDRESSE_Type">
<xsd:element name = "addresses" type = "xsd:string" minOccurs =
"1" maxOccurs = "3"/>
</xsd:complexType>
```

- La contrainte de séquence indique l'ordre des sous éléments sous leur élément père, elle est exprimée en XML Schema à l'aide de la primitive *<xsd :sequence>* . Par exemple :

```
<xsd:complexType name="PERSONNE_Type">
  <xsd:sequence>
  <xsd:element name="NOM" type="xsd:string"/>
  <xsd:element name="PRENOM" type="xsd:string"/>
  <xsd:element name="Addresses" type="ADDRESSE Type"/>
  </xsd:sequence>
</xsd:complexType>
```

- La contrainte de la disjonction exclusive indique qu'un élément père doit avoir un seul type de sous élément pour apparaître comme instances dans le document XML, cette contrainte est décrite en XML Schema à l'aide de la déclaration *<xsd : choice>*. Par exemple :

```
<xsd:complexType name="CYCLE_Type">
<xsd:choice>
<xsd:element name="Cycle long" type="xsd:string"/>
<xsd:element name="Cycle court" type="xsd:string"/>
</xsd:choice>
</xsd:complexType>
```

La violation de ces contraintes lors d'une opération de mise à jour, peut se produire dans les cas suivants :

– Un ajout d'une instance d'un élément induisant un nombre d'occurrence supérieur à la cardinalité maximale.

– Une suppression d'une instance d'un élément qui mène à un nombre d'occurrence inférieure à la cardinalité minimale.

- Un ajout d'une instance d'un élément sans respecter son ordre d'apparence (violation de la contrainte de séquence si elle est requise).
- Un ajout d'une instance d'un élément, tel que les instances qui existent ne sont pas des instances de ce nouvel élément, (si la contrainte de la disjonction exclusive est requise).

3. Spécification des contraintes

La validation des opérations de mise à jour pour la préservation des contraintes décrites ci-dessus, nécessite de :

1. Définir une stratégie permettant de capturer les contraintes structurelles définies dans le document XML Schema
2. Définir une technique permettant de concevoir l'information complète sur les valeurs des clés et des clés étrangères.

3.1 Spécification des contraintes structurelles

Pour la spécification des contraintes structurelles, qui regroupent l'ensemble des contraintes d'éléments et d'attributs, nous considérons deux tables l'une pour regrouper les contraintes d'attributs et l'autre pour regrouper les contraintes d'éléments :

- Chaque entrée de la table d'attributs est constituée du tuple <*nom_attribut ,type_attribut, contrainte_use, element_pere*> tel que:

  nom_attribut définit le nom d'attribut, *type_attribut* est le type d'attribut, *contrainte_use* définit la contrainte de l'apparition de l'attribut sous l'élément père, et *element_pere* référence l'élément père de l'attribut.

- La table d'éléments est constituée de l'ensemble des tuples :

  <*nom_element ,type_element, element_pere, contrainte_choix, rang, occurrence_min, occurrence_max*> tel que:

  nom_element définit le nom de l'élément, *type_element* définit le type de l'élément, *element_pere* définit l'élément père de cet élément, *contrainte_choix* définit si l'élément est sous la contrainte du disjonction exclusive, *rang* définit l'ordre d'apparition de l'élément tel que les éléments fils qui apparaissent sous la contrainte de séquence vont avoir un numéro de séquence qui correspond à leur ordre d'apparition, et les éléments qui n'apparaissent pas sous cette contrainte vont tous avoir la valeur de l'ordre égale à "0", et *occurrence_min*, *occurrence_max* définissent respectivement la cardinalité minimale est maximale des instances d'un élément.

Remarque : les attributs ou les éléments qui sont définis comme des clés ne vont pas être insérés dans les tables.

Exemple :

Considérons le fragment du document XML Schema suivant :

```
<xsd : complex type name= ''laboratoire_type''>
   <xsd :all>
    <xsd:element name= ''chercheur'' type =''chercheur_type'' minOccurs =
      ''1'', maxOccurs = ''unbounded''/>
      <xsd:element name = ''projet'' type = ''projet_type'' minOccurs =
      ''1'', maxOccurs = ''unbounded''/>
      <xsd:element name = ''article'' type = ''article_type'' minOccurs =
      ''1'', maxOccurs = ''unbounded''/>
   </xsd :all>
</xsd : complex type>
<xsd : complex type name = ''chercheur_type''>
<xsd : sequence>
      <xsd:element name =''nom'' type =''string'' />
      <xsd:element name ='' prenom''type =''string'' />
      <xsd:element name ='' adresse''type =''string ''/>
      <xsd: element name = '' travaille_sur'' type =''string''/>
</xsd : sequence>
      <xsd:attribut name = ''id'' type =''string'' use = ‘’required''  />
</xsd : complex type>
<xsd : complex type name =''projet_type''>
      <xsd:element  name  =''  chef_projet''  type  =''string''  use  =
      ‘’required’’ />
      <xsd:attribut name= ''id'' type =''string'' use = ''required''  />
      <xsd:attribut name =''titre'' type =''string'' use = ''required'' />
</xsd : complex type>
<xsd : complex type name =''article_type''>
      <xsd:element name =''ecrit_par ''type =''string'' />
      <xsd:attribut name= ''id'' type =''string'' use = ''required''  />
      <xsd:attribut name =''titre'' type =''string use = ''required'' />
 </xsd : complex type>
```

La table des éléments qui correspond aux déclarations d'éléments définies dans le fragment du document XML Schema, décrit ci-dessus est la suivante :

Nom_element	Type_element	Element_pere	Contrainte_choix	Rang	Occ_min	Occ_max
Laboratoire	Lboratoire_type	/	/	/	/	/
Projet	Projet_type	Laboratoire	Non requise	0	1	n
Chercheur	Chercheur_type	Laboratoire	Non requise	0	1	n
Article	Article-type	Laboratoire	Non requise	0	1	n
Nom	String	Chercheur	Non requise	1	1	1
Prénom	String	Chercheur	Non requise	2	1	1
Adresse	String	Chercheur	Non requise	3	1	1
Travaille_sur	string	Chercheur	Non requise	4	1	1
Ecrit _par	String	Article	Non requise	0	1	1
Chef_projet	String	Projet	Non requise	0	1	1

Tableau 3.1 Table d'éléments

En ce qui concerne la table d'attribut elle est constituée des entrées suivantes :

Nom_attribut	Type_attribut	Contrainte_use	Element_pere
titre	string	requise	projet
titre	string	requise	chercheur

Tableau 3.2 Table d'attributs

3.2 Spécification des contraintes d'intégrité référentielles

Nous utilisons un processus de marquage, permettant de marquer chaque élément ou attribut clé par une valeur paire ''i'', et tous les éléments ou les attributs clés étrangères correspondantes à cette clé par la valeur ''i+ 1''. Ainsi ce processus est décrit de la manière suivante:

a. Créer une table clé où chaque entrée de la table est composée d'une paire <clé,idc>où "*idc"* définit une valeur paire pour l'identification de la clé insérée dans la table.
b. Assembler les clés étrangères définies dans XML Schema dans une table où chaque entrée de la table est composée de la paire <clé_ref, idcr> / tel que :

$$Idcr = idc_{clé\ référencie} + 1.$$

c. A la fin de ce processus toutes les clés étrangères qui correspondent à une clé donnée vont avoir la même valeur de *idcr* dans la table.
d. Parcourir l'arbre XML, et marquer chaque élément ou attribut clé par la valeur définie dans la table des clés, et chaque élément ou attribut clé étrangère par la valeur définie dans la table des clés étrangères, en insérant chaque valeur d'un noeud marqué par une valeur "i" dans la liste (i) correspondante.

Exemple :

Nous considérons ce fragment d'un document XML Schema décrivant les clés et les clés étrangères correspondantes.

```
 <key name ='' id_projet''>
      <selector xpath ='' projet''>
      <field xpath='' @id''>
</key>
<keyref name ='' id_projet_ref refer ='' id_projet''>
      <selector xpath ='' chercheur''>
      <field xpath=''travail_sur''>
</keyref>
<key name ='' id_chercheur''>
      <selector xpath ='' chercheur''>
      <field xpath='' @id''>
</key>
<keyref name ='' id_chercheur_ref refer ='' id_chercheur''>
<selector xpath ='' article''>
<field xpath=''ecrit_par''>
```

```
</keyref>

<keyref name ='' id_chercheur_ref refer ='' id_chercheur''>
<selector xpath ='' projet''>
<field xpath = ''chef-projet''>
</keyref

<key name ='' id_article''>
<selector xpath ='' article''>
<field xpath='' @id''>
</key>
```

Les tables des clés et des clés étrangères qui correspondent au fragment du document XML schema sont comme suit :

Clé	Idc
Id_projet	0
Id_chercheur	2
Id_article	4

Tableau 3.3 Table des clés

Clé étrangère	Clé	Idcr
Id_projet_ref	Id_projet	1
Id_chercheur_ref	Id_chercheur	3

Tableau 3.4 Table des clés étrangères

Pour illustrer le processus de marquage permettant d'étiqueter chaque nœud clé ou noeud clé étrangère par sa valeur qui est défini dans la table (Tableau 3.4), nous considérons l'arbre XML suivant :

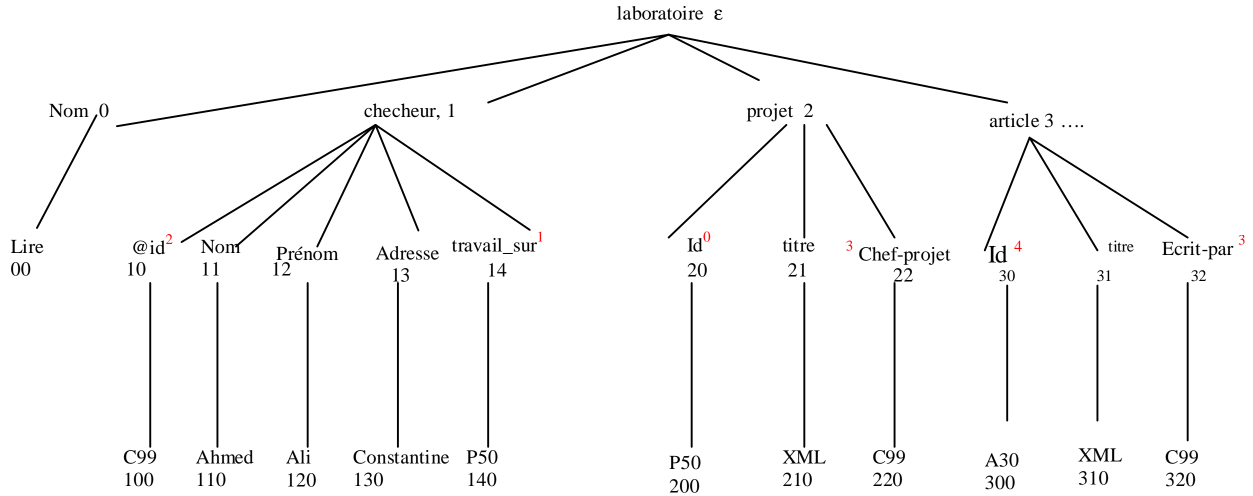

Fig.3.2 Arbre XML marqué

Les listes des valeurs des clés sont :

Liste (0) = {P50}, Liste (2) = {C99}, Liste (4) = {A30}.

Les listes des valeurs clés étrangères sont :

Liste (1) = {P50}, Liste (3) = {C99, C99}.

4 Validation de l'opération de suppression : supprimer(x)

La validation de l'opération de suppression consiste à vérifier que les contraintes structurelles et les contraintes d'intégrité référentielle sont préservées.

4.1 Vérification des contraintes structurelles

La vérification des contraintes structurelles lors d'une requête de suppression revient à vérifier les conditions suivantes :

- Si le nœud est un attribut alors il ne doit pas être sous la contrainte use = ''required''
- Si le nœud est un élément alors le nombre d'occurrences résultant de cet élément ne doit pas être inférieur à la cardinalité minimale.

Definition.2 : C*onsidérons "E" l'ensemble des éléments dans l'arbre du document XML,"TE" l'ensemble des tuples dans la table d'éléments, "A" l'ensemble des attributs dans l'arbre du document XML, et "TA" l'ensemble des tuples dans la table d'attributs.*

Soit `élément` *la fonction qui permet de faire correspondre à chaque élément "e", sa description "eid" dans la table des éléments tel que:*

élément : E®TE/ élément(e) = eid Û e.nom = eid.nom_element Ù e.type = eid.type_element.

Soit `attribut` *la fonction qui permet de faire correspondre à chaque attribut "a", sa description dans la table d'attributs aid / attribut : A®TA/ attribut(a) = aid Û a.nom = aid.nom_attribut Ù a.type = aid.type_attribut.*

La description formelle des règles précédentes peut être comme suit :

- xid.contrainte_use ≠" requise"/ tel que Attribut(x) = xid.

Cette condition permet de vérifier que l'apparition de l'attribut "x" sous son élément père n'est pas obligatoire, pour que sa suppression soit valide.

- |Ex| - 1 ≥ xid.occurrence_min /Ex ={e ∈ E_z / eid.nom_element = xid.nom_element Λ eid.type_element = xid.type_element}.(Z est l'élément père de x).

Cette condition permet de vérifier que la contrainte de la cardinalité minimale est préservée.

4.2 Vérification des contraintes d'intégrité référentielle

La vérification des contraintes d'intégrité référentielles lors d'une opération de suppression, revient à vérifier que la suppression d'une clé ne va pas induire une référence vers une entité non existante.

Définition 3 : *Soit* `marque` *la fonction qui permet de définir la marque d'un nœud x tel que :*

marque (x) = 2n si ''x'' est une clé.

marque (x) = 2n+1 si ''x'' est une clé étrangère

La description formelle de la règle est comme suit :

Proposition 1: *Soit "N_x" l'ensemble des nœuds dans le sous arbre à supprimer x.*

*" n Î N_x / si marque(n) = 2*N => n.valeurÏ liste_new(i) / i=2*N+1.*

La vérification de la règle précédente passe d'abord par l'étape suivante qui consiste à calculer la liste des valeurs liste _new (i) comme suit :

- Pour chaque nœud marqué par une valeur impaire dans le sous arbre à supprimer, supprimer la valeur de ce nœud de la liste (i) correspondante.
- liste_new (i) = liste (i) s'il n'y a aucun nœud marqué par "i" tel que "i" est impaire.

Le calcul de la liste liste_new (i) permet d'éliminer les valeurs des clés étrangères dans le sous arbre à supprimer.

4.3 Algorithmes pour la validation de l'opération de suppression

Nous allons maintenant définir les algorithmes qui correspondent aux règles définies ci-dessus.

Dans le cas où "x" est un attribut qui doit apparaître sous son élément père alors, la requête de sa suppression est rejetée.

La fonction `supprimer_attribut(x)` retourne `"true"` si l'opération de suppression est valide, et retourne `"false"` dans le cas contraire.

Valider_supprimer_attribut(x) : booléen

Début

    Si *xid.contrainte_use ¹ ''requise''* ***alors***

        Ok<=true

    Si non

        Ok<=false

    Fin si

    Retourner (Ok)

Fin si

La fonction `Valider_ supprimer_élément(x)` retourne `true` si la contrainte de cardinalité minimale est vérifiée. Si "x" est un élément complexe, cette fonction va appeler la fonction `supprimer_élément_complexe(x)`.

Valider_supprimer_élément(x) : booléen

Début

    Si *|Ex| -1>=occurrence min* ***alors***

        Ok <= true

    Sinon

        Ok <= false

    Fin si

    Si ** x est de type complexe* et Ok = true* ***alors***

        Supprimer_ élément_complexe(x)

    Finsi

    Retourner (Ok)

Fin

La fonction `supprimer_élément_complexe(x)` retourne `true` s'il n'y a aucune clé référencée dans le sous arbre à supprimer.

Supprimer_élément_complexe(x) : booléen

```
Début
    Calculer liste_new(i)
  Pour chaque noeuds fils z de x faire
   Si  marque(z)=2n  alors
        Ok<= supprimer_clé(z)
        Si Ok =false alors
           Exit()
        Fin si
     Si non
        Si z est de type complexe alors
            Ok<= supprimer_ élément_complexe(z)
        finsi
     finsi
  Fin pour
 Retourner (Ok)
Fin
```

La fonction `supprimer_clé(x)` retourne `true` si "x" est une clé non référencée et retourne `false` dans le cas inverse.

Suppression _clé(x) : booléen

```
Debut

        Si  x.valeur Ï   liste_new(2n+1) alors
                Ok<= true
        Si non
                Ok <= false
        Finsi
    Retourner (Ok)
Fin
```

5. Validation de l'opération d'insertion

Nous rappelons qu'il existe deux types d'opérations d'insertion : l'opération `insérer(x,y)`qui consiste à insérer le sous arbre "y" comme un dernier fils de "x". Et l'opération `insérer_après(x,y)` qui consiste à insérer le sous arbre "y" après "x".

5.1 Validation de l'opération d'insertion: inserer(x,y)

La validation de l'opération d'insertion consiste à vérifier que les contraintes structurelles et les contraintes d'intégrité référentielles sont préservées.

5.1.1 Vérification des contraintes structurelles

Nous supposons dans ce qui suit que le sous arbre "y" est valide, et nous vérifions si son insertion n'altère pas la validation du document.

La validation de l'opération d'insertion de l'élément "y" comme un dernier fils de l'élément "x" revient à vérifier les conditions suivantes:

1. "y" est un élément fils de "x".
2. Le nombre d'occurrences de "y" supérieur ou égale à la cardinalité maximale.
3. Si y est sous la contrainte de disjonction exclusive il doit y avoir au moins une instance de l'élément "y " dans le sous arbre "x".
4. "y" n'est pas sous la contrainte de séquence, ou bien il est une nouvelle instance du dernier élément fils de "x" ‘' last_child(x)''.

La description formelle de ces règles est comme suit :

1. yid.element_pere = x.nom_element /élément(y) = yid

Cette condition permet de vérifier que "y" est un élément fils de "x".

2. |Ey|+1 ≤ yid.occurrence_max /Ey ={e Î E_x */ élément(e) = eid Λ eid.nom_element = yid.nom_element Λ eid.type_element = yid.type_element}*

Cette condition permet de vérifier la validité de la contrainte de cardinalité maximale

3. yid.choix = "requise" => $ e Î E_x *tel que élément(e) = eid Ù eid.nom_element = yid.nom_element Ù eid.type_element = yid.type_element.*

Cette condition permet de vérifier la contrainte de la disjonction exclusive, si elle est requise.

4. yid.rang = 0 Ú (yid.nom_element = last_child(x).nom_element).

Cette condition permet de vérifier la contrainte de séquence : ''yid.rang = 0'' signifie que n'importe quel élément qui n'est pas sous la contrainte de séquence peut être inséré comme un dernier fils de "x", ou bien l'élément "y" est une nouvelle instance de l'élément last_child(x).

5.1.2 Vérification des contraintes d'intégrité référentielles

La validation de la contrainte d'intégrité référentielle lors d'une opération d'insertion d'un sous arbre "y", revient à vérifier les règles suivantes :

1. L'insertion du sous arbre ne doit provoquer une duplication des valeurs des clés.
2. L'insertion du sous arbre ne doit pas également générer des références vers des entités non existantes.

La description formelle de ces règles peut être comme suit :

Proposition 2*: soit N_y l'ensemble des noeuds dans le sous arbre* à *insérer.*

1. *" n Î N_y / si marque(n) = 2*N => n.valeurÏ È {liste(i)}/ i=2*N.*
2. *" n Î N_y / si marque(n) = 2*N +1 => n.valeurÎ liste_new(i)/ i=2*N.*

La vérification de la règle 2 passe d'abords par l'étape suivante :

Parcourir le sous arbre à insérer, et calculer la liste de valeur liste_new (i) tel que:

- Pour chaque nœud marqué par une valeur paire, ajouter la valeur de ce nœud dans la liste(i) correspondante.
- liste_new (i) = liste(i) s'il n'y a aucun nœud marqué par "i" tel que "i" est paire.

Le calcul de la liste liste_new(i) permet d'inclure les valeurs des clés dans le sous arbre à insérer du fait que la valeur de la clé étrangère peut correspondre à une clé déjà existante ou bien elle se trouve dans le sous arbre à insérer.

5.2 Validation de l'opération d'insertion: insérer_après(x,y)

L'opération de mise à jour `insérer_après(x,y)` prends en considération l'ordre dans la requête d'insertion. Les règles de vérification des contraintes d'intégrité référentielle coïncident aux celles définies pour l'opération inserer(x,y) à l'exception de la première et la quatrième règle :

1. *yid.element_pere = xid.element_pere = z*

   Cette condition permet de vérifier que "y" et "x" ayant le même élément père.

2. *$|E_y|+1 \leq$ yid.max_occ / Ey ={eÎ E_z / element(e) = eid Λ eid.nom_element = yid.nom_element Λ eid.type_element = yid.type_element}.*

   Cette condition permet de vérifier la validité de la contrainte de cardinalité maximale lors de l'insertion de sous arbre "y".

3. *yid.choix = "requise" Þ $ e Î E_z tel que élément(e) = eid Λ eid.nom_element = yid.nom_element Ù eid.type_element = yid.type_element.*

   Cette condition permet de vérifier la contrainte de la disjonction exclusive, si elle est requise.

4. *yid.rang = xid.rang Ù (yid.rang = frère_droit(x).rang))*

   Cette condition permet de vérifier si l'élément "y" respecte son ordre d'apparition lors de son insertion. Notons que frère_droit(x) est l'élément frère de "x" qui se situe à sa droite.

5.3 Algorithmes pour la validation de l'opération d'insertion

Nous allons maintenant définir les algorithmes qui correspondent aux règles définies ci-dessus. La fonction `valider_insérer(y,x)`retourne `true` si l'opération d'insertion de "x" dans "y" est valide, et elle retourne `false` dans le cas contraire. Cette fonction vérifie si l'insertion de l'élément ''x'' respecte les contraintes structurelles. Elle appelle la fonction `insérer_élément_complexe(x)` si "x" est de type complexe.

Valider_insérer (y,x) :booléen

```
Début
      Ok<= true
  Si xid.element_pere ¹ yid.nom_element alors
      Ok <= false, Exit()
  Fin si
  Si xid.contrainte_choix = '' requise'' alors

      Liste_instance<= les nœuds fils de y

      Si  x.nom_element Ï Liste_instance alors
                Ok<=false
      finsi
  Fin si
  Si yid.rang ¹ 0 Ù (yid.nom_element  ¹  last_child(x).nom_element )alors
                Ok <=false
  Finsi

  Si  |E x/  + 1 > xid.occurrence _max  alors

        Ok <= false
  Fin si
  Si   marque (x)=2n+1  alors
             Calculer liste_new(2n)
           Ok<= insérer_clé_étrangère(z)
  Fin si

  Si * x est de type complexe* etOk = true  alors

        Ok <=  Insérer_élément_complexe(x)
  Fin
  Retourner (Ok)
Fin
```

La fonction `Insérer_élément_complexe(x)` retourne la valeur `true` dans le cas où toutes les valeurs des clés sont distinctes et toutes les clés étrangères référencient des valeurs des clés existantes.

Insérer_ élément_complexe(x) : booléen

```
Début
    Calculer liste_new( 2n)
   Pour chaque noeud fils z de x  faire
     Si  marque (z)=2n alors
        Ok<= insérer_clé(z)
       Si Ok<=false alors
          Exit()
       Finsi
     sinon
      Si  marque (z)= 2n+1alors

            Ok<= insérer_clé_étrangère(z)
             Si Ok<=false alors
                 Exit()
             Finsi

      Si non
          Si z est de type complexe alors
             Ok<= insérer_élément_complexe(z)
          Fin si
      Fin si
   Fin pour
  Retourner (Ok)
Fin
```

La fonction `insérer_clé(x)` vérifie la non duplication de la valeur de la clé à insérer.

Insérer_clé(x) : booléen

```
Début
   Si  xid.valeur Î  È {liste( 2n)}alors
       Ok <= true
   Si non
         Ok <= false
   Finsi
   Retourner (Ok)
Fin
```

La fonction `insérer_clé_étrangère(x)` retourne la valeur `true` si la valeur de la clé étrangère référence une clé existante.

Insérer_clé_étrangère(x) : booléen

```
Début
   Si x.valeur Î  liste_new (2n) Ù|Ex| + 1 ≤ xid.occurrence_max  alors
                  Ok<=true
   Sinon
                  Ok<=false
   finsi
  Retourner (Ok)
Fin
```

La fonction `valider_insérer_après(x,y)` ayant le même principe que la fonction `valider_insérer_élément(x,y)`. La seule différence est que la fonction `valider insérer_après(x,y)` vérifie la validation de la contrainte de séquence lors d'une insertion aléatoire d'un élément.

Valider_insérer_après(y,x) :booléen

```
Début
    Ok<= true
  Si xid.element_pere ¹ yid.element_pere alors
    Ok <= false, Exit()
  Fin si

  Si xid.contrainte_choix = ''requise'' alors
      Liste_instance <=  les nœuds fils de y
     Si  x.nom_element Ï  Liste_instance alors
            Ok<= false, Exit()
     Fin si
  Fin si
  Si  yid.rang  ¹ xid.rang Ù (xid.rang ¹ frère_droit(x).rang) alors

            Ok <=false, Exit()

  Finsi

  Si  |Ex/  + 1 > xid.occurrence _max alors

      Ok <= false, Exit()
  Fin si
  Si   marque (x)=2n+1 alors
          Calculer liste_new(2n)
         Ok<= insérer_clé_étrangère(z)
  Fin si

  Si  x est de type complexe et Ok<= true  alors

     Ok <= Insérer_élément_complexe(x)
  Fin
 Retourner (Ok)
Fin
```

6. Etude de complexité des algorithmes

L'étude de complexité des algorithmes est discutée par considération la complexité en temps de la vérification des contraintes structurelles et la complexité en temps de la vérification des contraintes d'intégrité référentielles.

Dans la vérification des contraintes d'intégrité référentielles, le parcours des nœuds de l'arbre permettant de calculer les listes des valeurs des clés et les listes de valeurs des clés étrangères est linéaire ''O (n)'' au nombre des nœuds ''n'' dans l'arbre XML

Lors d'une requête de suppression d'un sous arbre "y", la complexité en temps pour vérifier qu'il n'y a aucune référence vers une entité non existante, est O (|y| * n) tel que : ''n'' est le nombre maximal des valeurs des clés étrangères dans les listes `liste_new(i)` tel que "i" est impaire.

Dans le cas d'une requête d'insertion d'un sous arbre "y", la complexité en temps pour vérifier qu'il n'y a aucune référence vers une entité non existante, est O (|y| * n) tel que ''n'' est le nombre maximal des valeurs des clés dans les listes `liste_new(i)` tel que "i" est paire. La vérification de la non duplication des clés est O (|y| * m) tel que ''m'' est le nombre des valeurs clés dans l'ensemble des listes `liste(i)` tel que "i" est paire.

La complexité de la validation des contraintes structurelles est linéaire au nombre des nœuds fils de l'élément racine dans le sous arbre à insérer.

7. Discussion

L'approche de validation statique ou encore naïve consiste d'abord à exécuter la mise à jour puis à revalider tout le document. Dans le cas où le document résultant n'est pas valide, la requête de la mise à jour va être annulée, et le document original va être restitué. Il est clair de noter le gain en temps fourni par notre approche. Si nous considérons par exemple une requête de suppression, d'une adresse d'une personne. L'approche de validation statique va revalider tout le document, tandis que notre approche va effectuer une validation locale de l'élément personne uniquement avant l'exécution effective de la requête.

Par ailleurs, il existe d'autres solutions réalisant l'approche de validation incrémentale, nous allons présenter dans ce qui suit une étude comparative entre notre approche et celle proposée par [5] :

Nous commençons d'abord par un petit rappel concernant le principe de l'approche [5] qui consiste à valider les contraintes spécifiées à l'aide de la DTD, en utilisant un automate définissant sa grammaire, pour la vérification des contraintes structurelles. Et deux listes : l'une pour regrouper toutes les valeurs des clés et l'autre pour regrouper toutes les valeurs des clés étrangères existantes dans le document XML.

La vérification des contraintes structurelles en utilisant cette technique consiste à exécuter l'automate sur le sous arbre concerné par la mise à jour, dans le cas où l'état de l'élément racine est préservée alors l'opération de mise à jour est valide. La complexité en temps de cette procédure est de O(n) tel que ''n'' est le nombre des nœuds dans le sous arbre.

La vérification des contraintes d'intégrité référentielles se fait en temps de O (n*m) tel que "n" est le nombre des valeurs dans la liste des clés et "m" est le nombre des valeurs dans la liste

des clés étrangères. Il est clair de noter le gain en temps fourni par notre approche lors de la vérification des contraintes d'intégrité référentielles, car elle permet d'optimiser l'espace de recherche des clés et des clés étrangères d'un espace contenant toutes les valeurs existantes vers un espace contenant uniquement les valeurs correspondantes. Ainsi que notre approche permet de vérifier les contraintes structurelles en considérant uniquement les contraintes sur l'élément à supprimer ou à insérer, au lieu d'inclure tous les éléments fils qui constituent le modèle de contenu de l'élément racine du sous arbre affecté par l'opération de mise à jour.

8. Conclusion

Les langages de mise à jour proposés jusqu'à maintenant restent limités et non complets, car ils ne supportent pas la vérification des contraintes d'intégrité référentielles et structurelles lors des opérations de mise à jour. Par conséquent le document XML modifié contient fréquemment des références vers des entités non existantes, duplication des clés et beaucoup d'autres problèmes qui indiquent l'intégrité faible de la base de données.

Dans ce chapitre, l'approche proposée se focalise sur la vérification incrémentale des contraintes référentielles et structurelles. L'ensemble des algorithmes proposés dans ce chapitre est destiné à augmenter la puissance des langages de mise à jour XML en préservant la validité des documents XML. Dans notre travail, nous avons proposé :

- Une modélisation des contraintes structurelles (spécifiées à l'aide de XML Schema) en utilisant les tables.
- Un processus de marquage permettant de définir les nœuds clés et les nœuds clés étrangères dans l'arbre XML.
- Un formalisme de validation des contraintes sous forme de règles.
- Des algorithmes pour la validation incrémentale des requêtes de mise à jour.

Chapitre IV : Etude de cas : Gestion des documents universitaires

1. Introduction

Dans ce chapitre nous allons valider les concepts présentés dans le chapitre précédent. Nous considérons l'exemple d'une collection des documents XML stockée dans une base de données XML native. Les documents XML sont associés à un document XML Schema, qui restreint leurs structures et définit les contraintes qui doivent être respectées dans ces documents.

Nous allons appliquer le processus de marquage pour étiqueter les clés et les clés étrangères, et nous allons utiliser les deux tables pour regrouper les contraintes structurelles des éléments et des attributs comme il est décrit dans le chapitre précèdent.

Nous supposons des requêtes de mise à jour que nous allons déterminer leur validation, en appliquant les algorithmes définis dans le chapitre précèdent.

2. Enoncé

Nous considérons l'ensemble de collections de documents XML, décrivant d'une manière simplifiée l'université.

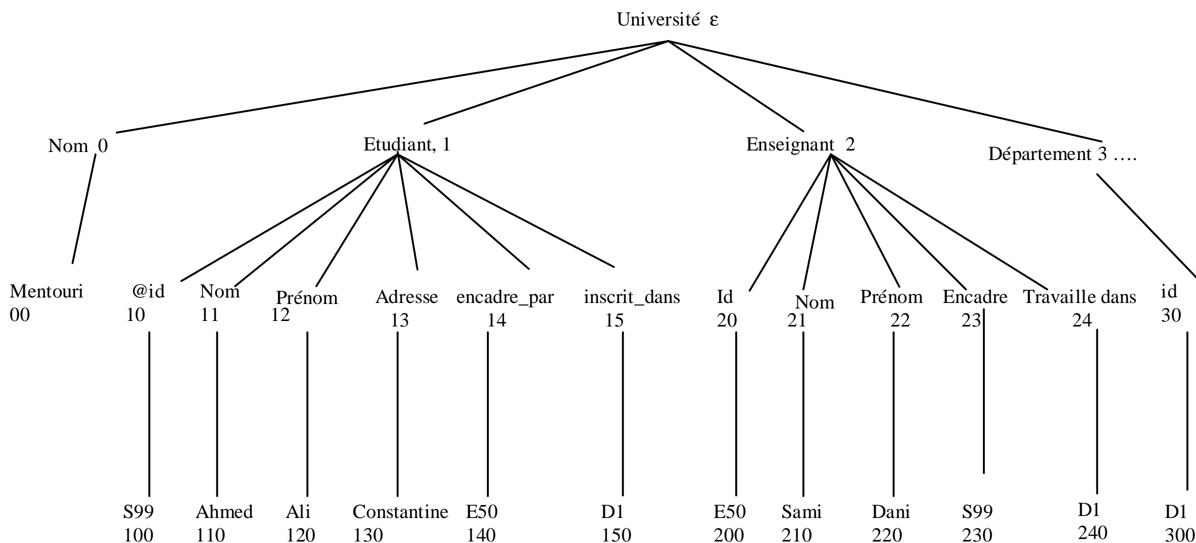

Fig.4.1 Exemple d'un document XML

Un fragment du document XML Schema qui spécifie la structure des documents XML est le suivant :

```
<xsd : complex type name ''universite_type''>
<xsd :all>
      <xsd:element name= ''étudiant'' type =''eudiant_type'' minOccurs = ''
      1 '', maxOccurs = ''unbounded''/>
      <xsd:element name= ''enseignant'' type =''enseignant_type ''minOccurs
      = '' 1 '', maxOccurs = ''unbounded''/>
      <xsd:element   name=   ''departement''   type   =''departement_type''
      minOccurs = ''1 '', maxOccurs = ''unbounded''/>
</xsd :all/>
</xsd : complex type>
<xsd : complex type name= ''étudiant_type''>
<xsd : sequence>
      <xsd:element name =''nom" type =''string'' />
      <xsd:element name ='' prenom''type =''string'' />
      <xsd:element  name=''  adresse''type  =''string  ''  minOccurs  =  ''1'',
      maxOccurs = ''3''/>
      <xsd: element name= '' encadre_par'' type =''string'' />
      <xsd: element name= ''inscrit_dans'' type =''string'' />
</xsd : sequence>
      <xsd:attribut name= ''id'' type =''string''use = ''required'' />
</xsd : complex type>
<xsd : complex type name= ''enseignant_type''>
<xsd : sequence>
      <xsd:element name = ''nom’’ type = ''string '' />
      <xsd:element name = ''prenom''type =''string''  />
      <xsd:element  name  =''adresse''  type  =''string  ''  minOccurs  =  ''1'',
      maxOccurs = ''3''  />
      <xsd:element  name=  ''  encadre''type  =''string''  minOccurs  =  ''1'',
      maxOccurs = ''unbounded'' />
      <xsd:element  name=  ''travaille_dans  =  ''  type  =''string''  use  =
      ''required'' />
</xsd : sequence>
      <xsd:attribut name= ''id'' type = ''string '' use = ''required'' />
</xsd : complex type>
      <xsd : complex type name= '’departement''>
  <xsd : choice>
      <xsd:element  name=  ''  laboratoire''  type  =''string''  minOccurs  =''
      1'', maxOccurs ='' unbounded ''/>
      <xsd:element  name   ''centre_recherche  ''  type  =''string''  minOccurs
      =''1 '', maxOccurs = ''unbounded''/>
  </xsd : choice>
      <xsd:attribut name ='' id'' type =''string'' use = ''required''/>
</xsd : complex type>
```

Fig.4.2 Document XML Schema

A partir du document XML Schema nous allons construire la table d'éléments suivante :

Nom_element	Type_element	Element_pere	Contrainte_choix	Rang	Occ_min	Occ_max
Université	Universite_type	/	/	/	/	/
Étudiant	Étudiant_type	Université	non requise	0	1	n
Enseignant	Enseignat_type	Université	non requise	0	1	n
Département	Departement_type	Université	non requise	0	1	n
Nom	String	étudiant	non requise	1	1	1
prénom	String	étudiant	non requise	2	1	1
Adresse	String	étudiant	non requise	3	1	3
Encadre_par	String	étudiant	non requise	4	1	1
Inscrit_dans	String	étudiant	non requise	5	1	1
Nom	String	enseignant	non requise	1	1	1
Prénom	String	enseignant	non requise	2	1	1
adresse	String	enseignant	non requise	3	1	3
travaille_dans	String	enseignant	non requise	4	1	1
encadre	String	enseignant	non requise	5	1	n
Laboratoire	String	département	requise	0	1	n
Centre_ recherche	String	département	requise	0	1	n

Tableau 4.1 Table d'éléments

Dans ce cas nous n'avons pas une table d'attributs, les attributs existants sont des clés définies dans le fragment de document XML Schema suivant :

```
<key name ='' id_etudiant''>
      <selector xpath ='' etudiant''>
      <field xpath='' @id''>
</key>
<keyref name ='' id_etudiant_ref refer ='' id_etudiant''>
      <selector xpath ='' enseignant''>
      <field xpath=''encadre''>
</keyref>
<key name ='' id_enseignant''>
      <selector xpath ='' enseignant''>
      <field xpath='' @id''>
</key>
<keyref name ='' id_enseignant_ref refer ='' id_enseignant''>
<selector xpath ='' etudiant''>
<field xpath=''encadre_par''>
</keyref>
<key name ='' id_departement''>
<selector xpath ='' departement''>
<field xpath='' @id''>
</key>
<keyref name ='' id_departement_ref '' refer =''
id_departement'>
<selector xpath ='' enseignant''>
<field xpath=''travaille_dans''>
</keyref>
<keyref name ='' id_departement_ref refer ='' id_departement'>
<selector xpath ='' etudiant''>
<field xpath=''inscrit-en''>
</keyref>
```

Fig.4.3 Fragment d'un document XML Schema

Les deux tables clés et clés étrangères correspondantes aux déclarations key et keyref, dans le document XML schema de la figure (4.3) sont les suivantes :

Clé	Idc
Id_étudiant	0
Id_enseigant	2
Id_departement	4

Tableau 4.2 Table des clés

Clé étrangère	Clé	idcr
Id_étudiant_ref	Id_étudiant	1
Id_enseigant_ref	Id_enseigant	3
Id_departement_ref	Id_departement	5

Tableau 4.3 Table des clés étrangères

L'application du processus de marquage vu dans le chapitre précèdent sur l'arbre XML est illustrée dans la figure (Fig.4.4) :

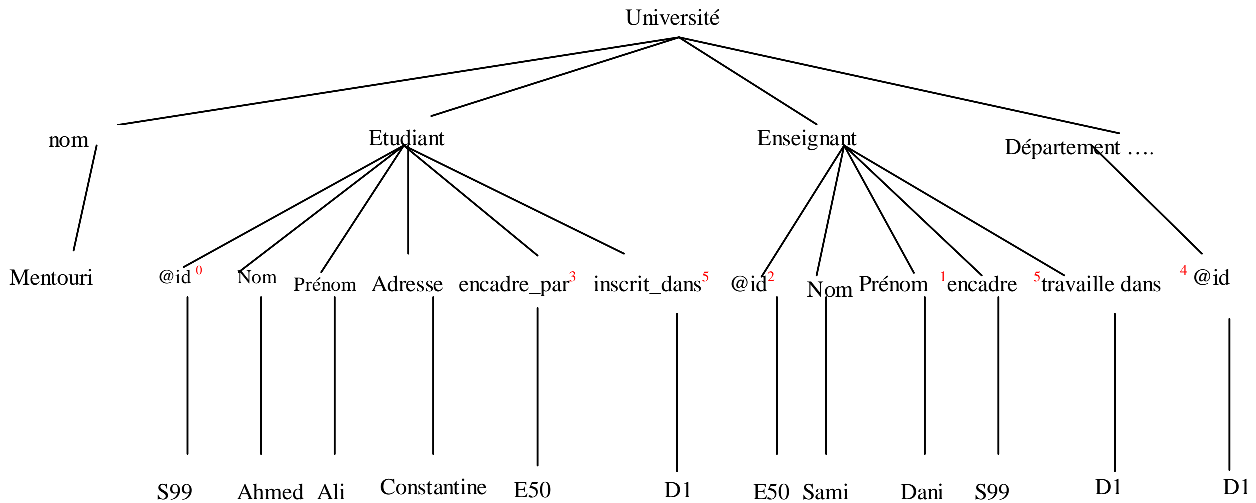

Fig.4.4 Arbre représentant un document XML marqué

Le parcours de l'arbre XML marqué va engendrer les listes de valeurs suivantes:

$Liste_0$= {S99}, $Liste_1$ = {S99}, $Liste_2$= {E50}, $Liste_3$= {E50}, $Liste_4$={D1},

$Liste_5$= {D1,D1}.

3. Exemples de requêtes de mise à jour

Nous considérons dans ce qui suit des requêtes de mise à jour, que nous allons déterminer leur validation en utilisant les algorithmes proposés.

3.1 Requêtes de suppression

Soit la requête de suppression `supprimer(étudiant)` qui consiste à supprimer le sous arbre dont la racine est étudiant. Le processus de la validation de cette requête est illustré par le schéma suivant :

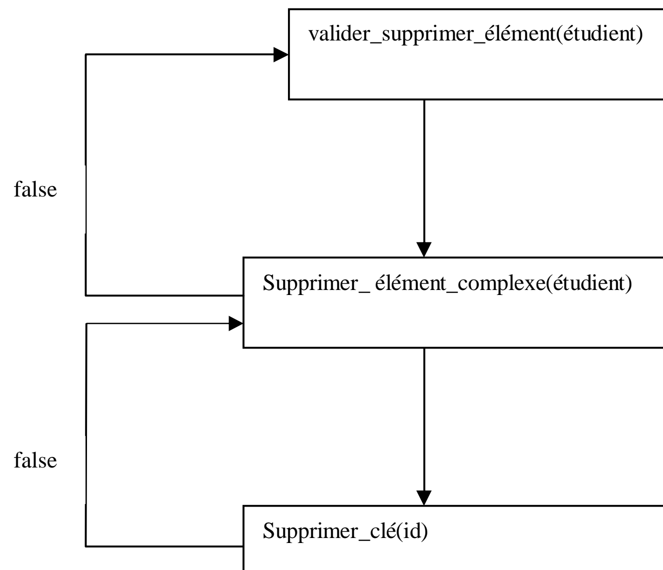

Fig.4.5 Validation de la requête supprimer(étudiant)

La fonction `valider_supprimer_élément(étudiant)`vérifie que la suppression de l'élément étudiant ne va pas induire une violation de la contrainte de cardinalité minimale.

La fonction `supprimer_élément_complexe(étudiant)` permet de parcourir le sous arbre dont la racine est étudiant et vérifier que toute valeur d'une clé (nœud marquée par une valeur paire) n'existe pas dans la liste des clés étrangères correspondante. Le nœud `id` est une clé, il est marqué par une valeur paire qui est '2'. Sa valeur qui est égale à 'E50' existe dans la liste(3) donc cette clé est référencie, de ce fait la fonction `supprimer_clé(id)` retourne la valeur `false`. La valeur `false` indique que la requête de suppression de l'élément étudiant va être rejetée.

3.2 Requêtes d'insertion

Soit la requête de l'insertion `insérer(université,enseignant)`qui permet d'insérer le sous arbre dont la racine est enseignant comme un dernier fils de l'élément université. Ceci peut être schématiser comme suit :

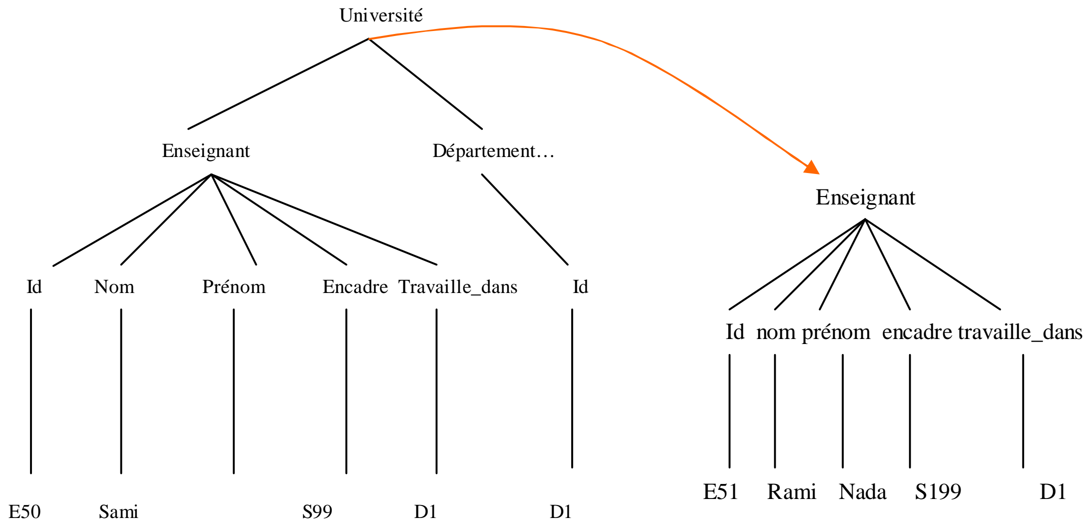

Fig.4.6 Requête insérer(université, enseignant)

La validation de cette requête suit les étapes illustrées dans la figure suivante la valeur booléenne retournée décide l'exécution effective ou le rejet de l'opération.

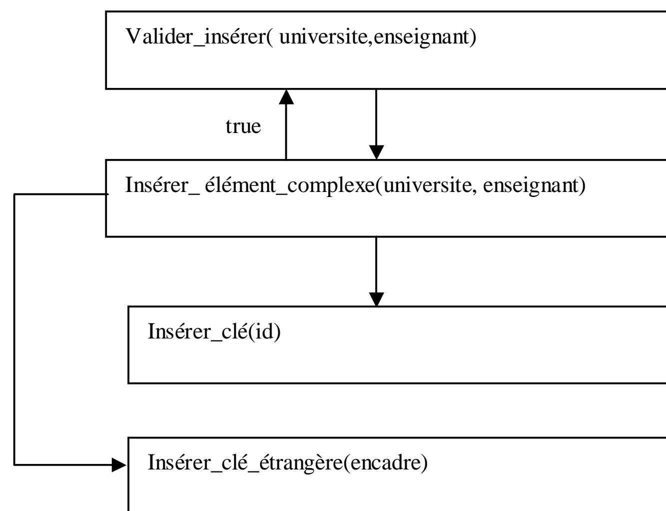

Fig.4.7 Processus de validation de la requête insérer(université,enseignant)

La fonction `valider_insérer(université,enseignant)` vérifie les contraintes structurelles de l'élément enseignant, ce dernier n'est pas sous la contrainte de disjonction exclusive, le nombre maximal de ses instances est égal à "n", ainsi son rang est égal à zéro ce qui signifie qu'il n'est pas sous la contrainte de séquence.

La fonction `insérer_élément_complexe(université, enseignant)` appelle la fonction `inserer_clé(id)` pour vérifier que la valeur E51 n'est pas dupliquée, et la fonction `insérer_clé_étrangère(encadre)` pour vérifier que la clé étrangère *encadre* référence une valeur clé existante.

Considérons la requête de l'insertion `insérer(département, centre_recherche)` qui permet d'insérer l'élément `centre_recherche` à l'extrémité droite des sous éléments de l'élément département.

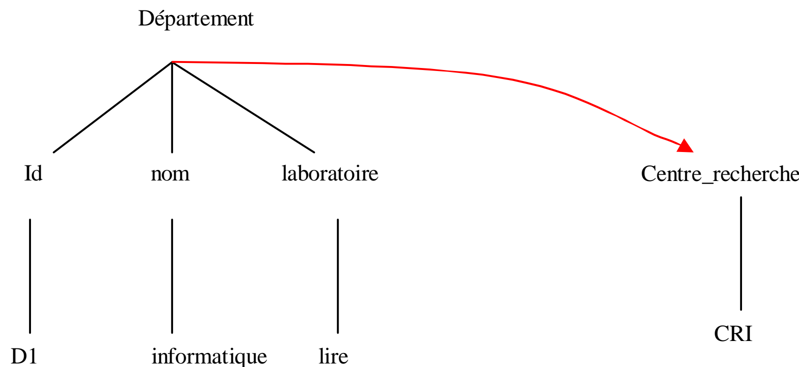

Fig.4.8 Requête insérer (département, centre_recherche)

Si la fonction `valider_insérer(département,centre_recherche)`retourne la valeur `true` alors la requête est exécutée mais si par contre elle retourne la valeur `false`, alors elle est rejetée.

L'élément *centre_recherche* est sous la contrainte de disjonction exclusive et il n'y a aucune instance déjà existante de cet élément sous l'élément département, cependant la valeur booléenne retournée par la fonction `valider_insérer(département, centre_recherche)` égale à `false` indiquant que cette opération d'insertion est rejetée car elle ne préserve par la contrainte de disjonction exclusive.

Le dernier exemple de requête de mise à jour qu'on considère dans cette étude de cas est la requête `insérer_après(adresse, encadre)`.

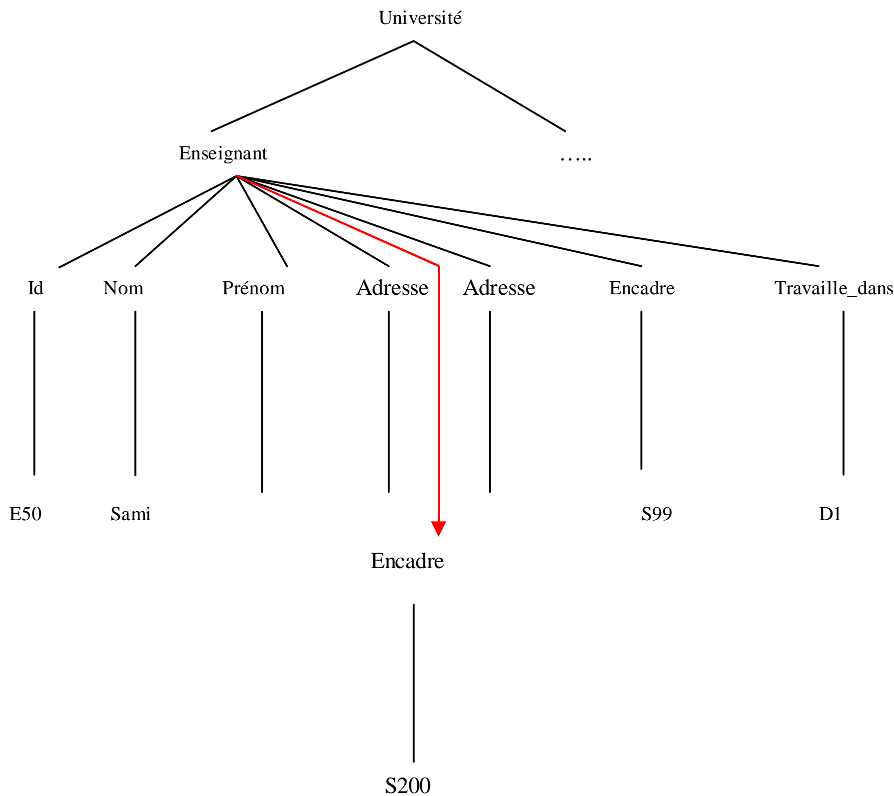

Fig.4.9 Requête insérer_après(adresse, encadre)

L'appel de la fonction `valider_insérer_après(adresse, encadre)` permet de valider la requête de l'insertion, le processus de validation est illustrée par le schéma suivant :

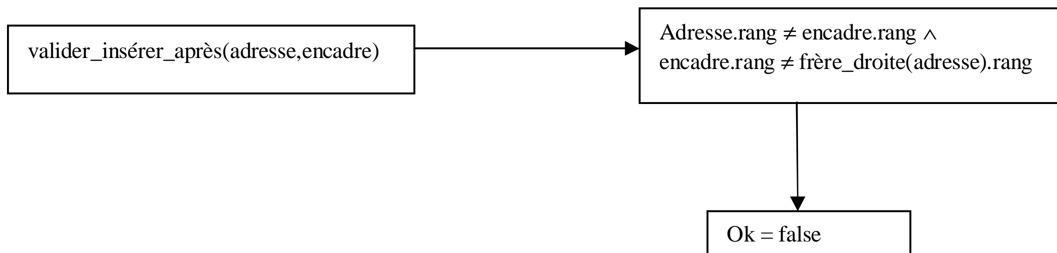

Fig.4.10 Validation de la requête insérer_après(adresse, encadre)

La fonction `Valider_insérer_après(adresse,encadre)` retourne une valeur booléenne égale à `false`, car l'élément `encadre` ne respecte pas son ordre d'apparition, du fait que son

rang est différent de celui de l'élément adresse et de l'élément frère_droite(adresse) qui est également une instance de l'élément adresse :

Adresse.rang ≠ encadre.rang ∧ encadre.rang ≠ frère_droite(adresse).rang

Cela signifie que cette requête va induire l'insertion de l'élément encadre entre deux instances de l'élément adresse, ce qui résulte une violation de la contrainte de séquence imposée, car toutes les instances de l'élément adresse doivent apparaître avant celles de l'élément encadre. Donc la requête est rejetée.

4. Conclusion

L'étude de cas présentée, nous a montré l'utilité de l'approche de validation incrémentale proposée, pour la mise à jour des documents XML. Les exemples des requêtes de mise à jour cernent d'une manière assez complète les différents cas qui peuvent mener à la violation des contraintes. Les algorithmes proposés ont déterminé à chaque fois l'exécution effective ou le rejet de la requête.

Conclusion générale

1. Bilan du travail

Le format XML, de part sa simplicité et sa flexibilité est devenu le format adapté par l'intégralité des applications dans le Web pour la représentation et l'échange de données. Le nombre accru des documents XML générés a révélé un besoin inhérent pour leur stockage. Cependant, il existe principalement deux stratégies pour le stockage des documents : les bases de données Enabled-XML et les bases de données XML natives.

A l'heure actuelle, les travaux de recherche se focalisent à fournir des solutions aux problèmes liés à la gestion des documents XML stockés sous forme native. Cette nouvelle technologie de stockage dont la mise à jour constitue son point de faiblesse principal. En particulier le maintien de l'intégrité de la base lors de sa mise à jour. Dans ce travail nous nous sommes intéressés à la validation des requêtes de mise à jour par considération d'une approche de validation incrémentale.

Nous avons d'abord étudié les stratégies de stockage de chaque type de document XML, à savoir un contenu orienté données et un contenu orienté documents. Ainsi que les différents langages de requête permettant de rechercher des données XML. Nous avons également étudié d'une manière détaillée les bases de données XML natives, et le problème de la validation statique des requêtes de mise à jour dans ce type de base de données. Ensuite, nous avons présenté un certain nombre de travaux de recherche qui proposent des solutions en se basant sur le principe de la validation incrémentale.
Cette parade nous a fourni les fondements théoriques et techniques nécessaires pour notre travail, dans lequel nous avons proposé :

- Une modélisation des contraintes structurelles (spécifiées à l'aide de XML Schema) en utilisant deux tables (table d'éléments et table d'attributs).
- Un processus de marquage permettant de définir les nœuds clés et les nœuds clés étrangères dans l'arbre XML.
- Un formalisme de validation des contraintes sous forme de règles.
- Des algorithmes pour la validation incrémentale des requêtes de mise à jour.

Notre approche optimise le temps de la validation des documents résultants des requêtes de mise à jour, par comparaison avec l'approche de validation statique. Dans notre approche nous considérons que les documents XML sont associés au XML Schema, alors que peu de travaux considèrent ce type de schéma.

Les algorithmes proposés permettent d'optimiser l'espace de recherche des clés et des clés étrangères d'un espace contenant toutes les valeurs existantes vers un espace contenant uniquement les valeurs correspondantes. Ainsi ils permettent une validation des contraintes structurelles en tenant compte des contraintes sur l'élément affecté directement par l'opération de mise à jour au lieu d'utiliser un automate qui détermine la validation des contraintes structurelles en incluant tous les éléments fils dans le sous arbre affecté par la requête.

2. Perspectives

Le sujet dans lequel nous nous sommes lancés dans cette étude est très novateur. Cependant, plusieurs aspects pourraient être analysés :

- L'évaluation de l'approche proposée par une étude expérimentale.
- L'intégration des algorithmes proposés dans un langage de mise à jour XML, tel que l'extension de XQuery pour la mise à jour [3].
- L'extension de l'approche proposée pour traiter les requêtes de mise à jour "globales". Actuellement, notre approche vérifie les contraintes au niveau atomique, c-à-d, chaque opération de mise à jour atomique sur un élément simple ou complexe dans le document XML est permise si elle mène à un document valide. Comme prochaine étape, nous explorerions le concept de la mise à jour globale, c-à-d, une série de mises à jour sont seulement permises d'être exécutées, si l'effet global de leur exécution mène à un document valide.

Bibliographie

[1] *R. Bourett "XML et les bases de données", http://www.rpbourret.com, 2003.*

[2] *P. Amornsinlaphachai, M. Akhtar and N.Rossiter, "Updating XML using object relational database", Springer Verlag belin Heidelberg, 2005, pp 155-160.*

[3] *I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.S. Weld ,''Updating XML ", ACM SIGMOD, September 2001, pp 413-424.*

[4] *B. Kane, H. Su, and E. A. Rundensteiner "Consistently Updating XML Documents using Incremental Constraint Check Queries",WIDM'02, ACM, 2002 .pp 1-8.*

[5] *M. Abrao, B. Bouchou, and M. Ferari "Updates and incremental validation of XML document", International Conference on Database Programming Languages (DBPL), 2003.*

[6] *G. Wang, M, Liu and L. Lu "Extending XML-RL with update", IDEAS, 2003,pp 1-10.*

[7] *E. Pardede, W. Rahayu and D. Taniar "Preserving constraint for aggregation relationship updates", IAWTIC, 2004, pp 494-501.*

[8] *E. Pardede, WJ. Rahayu and D. Taniar "Preserving referential constraint for association relationship updates", IFIP, LNCS, 2004, pp 256-263.*

[9] *L. Feng, T. Dillon and E.Chang "A semantic network-based design methodology for XML documents", ACM Transactions on Information System, Vol.20, No 4, 2002, pp 390-421.*

[10] *Z. Chen, T.W Ling, M. Liu, G. Dobbie "XTree: A Declarative Query Language for XML Documents", to appear, 2005.*

[11] *D. McGoveran, The age of the XML Database, EAI Journal, October 2001.*

[12] *M. Champion, ''Storing XML in databases'', EAI Journal, 2001, pp 18-21.*

[13] *M. Kay, ''XML Databases'', www.softwareag.com , March 2003, pp 53-55.*

[14] *L. Soual, ''Base de données XML et relationnelles'', http://www.campus xml.org/ 2003.*

[15] *B. Amann, ''Bases de données et XML'', http://www.aristote.asso.fr, 2005.*

[16] *A. C. Damais, ''Panorama: les bases de données XML" http://solutions.journald net.com/ 2002.*

[17] *G. Lapis, ''XML and Relational Storage–Are they mutually exclusive?'', IBM Corporation XTech ,2005.*

[18] *K. Staken, ''Introduction to native XML database'' http://www.xml .com, 2001.*

[19] *''Faisons le point sur les langages de schéma XML'',http://www.clever- age.com/veille/clever-link/faisons-le-point-sur-les-langages-de-schema-xml.html*

[20] *''Parser du XML : les API DOM et SAX'', http://www. commentcamarche.net*

[21] *M. Varandat, '' Bases de données XML : pour faire quoi ?'' http://www.indexel .net /bin/doc, 2002.*

[22] *Y. Marccoucs, ''Bien forme versus validité des documents XML'', http://www.mapageweb .umontreal.ca/marcoux ,2005.*

[23] *G. Chagnon, ''Initiation au Schémas XML'', http://gilles.chagnon.free.fr/cours/xml/schema .html , 2005.*

[24] *R. Bourret,''Going native: Use cases for native XML databases'', http ://www.rpbourret .com 2005.*

[25] *L. Dodds, "XML and Databases Follow Your Nose'', http://www.xml.com/pub/a/2001/10/24 /follow-yr-nose.html.*

[26] *M. Liotta and C. Preimesberger, ''Native XML databases resolve XML document retrieval issues", http://builder.com.com/5100-6388-1051795.html*

[27] *D. Vohra, ''XML Document Validation with an XML Schema'', http://www.onjava.com/pub/ a/onjava/2004/09/15/schema-validation.html*

[28] *J. Snelson, ''All XML Databases are Equal'', http://www.parthcomp.com 2005.*

[29] *B. Amann, ''Evaluation des requêtes XML'', http://www .aristote.asso.fr 2006.*

[30] *E. Anken, ''Les bases de données XML natives'', http://www.syms.ch/xml-campu s.ch /xmlcampus/articles.*

[31] *R. Bourret, '' Mapping DTDs to Databases'', http://www.rpbourret.com 2005.*

[32] *M. Cyrenne, ''When should you use a Native XML Database?'', http://www .edocmagazine .com/articlenew, November/December 2002.*

[33] *D. Mertz, ''XML Matters: Putting XML in context with hierarchical, relational, and object-oriented models'', http://www.ibm.com/developerworks /xml/library*

[34] *E. R. Harold, ''Managing XML data: Native XML databases'', http://www.ibm.com, 2005.*

[35] *R. Bourret, ''Mapping W3C Schemas to Object Schemas to Relational Schema'' http://www .rpbourret.com , March, 2001.*

[36] *R. Bourret, ''XML Database Products: Native XML Databases'', http://www .rpbourret.com*

[37] *D. Obasanjo, ''An exploration of XML in database management systems'', http://www.25 hoursaday.com/StoringAndQueryingXML.html, 2001.*

[38] *''Choosing an XML Database Solution'', http://www.cincom com 2002.*

[39] *B. Verhaegen,''Requêtes OLAP sur une base de données XML native''Académie Universitaire WALLONIE BRUXELLES, 2005.*

[40] *"XML:DB, Xupdate-XML Update Language" http://www.xmldb.org/xupdate 2004.*

[41] *M. Biggs, '' e-business : faut-il opter pour des bases de données XML natives ?'', TechUpdate ZDNet US, 10 décembre 2001.*

[42] *H. Karine, ''Parser un fichier XML'', http://users.etu.info.unicaen.fr, 2000.*

[43] *M. Sévigny, ''Les DTD et les schémas'' http://www.ajlsm.com/formation/xml, 2002.*

[44] *J. Shanmugasundaram, R,Krishnamurthy, I. Tatarinov, '' A general technique for querying XML documents using a relational databases system'', Sigmod Record, Vol 30, No 3, 2001, pp 20-26.*

[45] *G. Wang and M. Liu, J. Xu Yu, B. Sun, G. Yu, and J. Lv, H. Lu, ''Effective Schema-Based XML Query Optimization Techniques'', (IDEAS'03), IEEE, 2003. pp 1-6.*

[46] *P. Chippimolchai, V. Wuwongse and C. Anutariya.''Towards Semantic Query Optimization for XML Databases'', (ICDE '05), 1084-4627/05, IEEE, 2005.*

[47] *D. Che and Y. Liu, ''Efficient Minimization of XML Tree Pattern Queries'', (NWeSP'05), IEEE, 0-7695-2452-4/05, 2005.*

[48] *S. Bőttcher, R. Steinmetz, ''Embedding XML Schema Constraints in Search-Based Intersection Tests for XPath Query Optimization'', Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05) 1529-4188/05 © 2005 IEEE.*

[49] *D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet,'' Efficient Incremental Validation of XML Documents'', Proceedings of the 20th International Conference on Data Engineering (ICDE'04),1063-6382/04 © 2004 IEEE.*

[50] *C. Li, T.W. Ling, M. Hu, ''Efficient Processing of Updates in Dynamic XML Data'', Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), 8-7695-2570-9/06 © 2006 IEEE.*

[51] *T. Amagasa, M. Yoshikawa, '' QRS: A Robust Numbering Scheme for XML Documents'', Proceedings of the 19th International Conference on Data Engineering (ICDE'03), 1063-6382/03 © 2003 IEEE.pp 705-707*