Persistance XML avec Hibernate

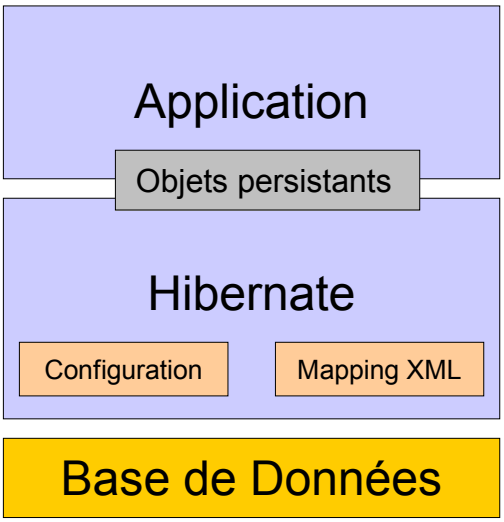
1 Introduction

Hibernate s'est récemment fait connaître comme une solution efficace pour assurer la persistance d'applications développées en Java dans des bases de données relationnelles. Reprenant des principes proches des concepts énoncées dans les spécifications de la norme JDO (Java Data Object), Hibernate propose une approche plus simple qui conserve des performances correctes.
Document Object Model (DOM) s'est imposé comme l'interface de facto pour la manipulation en mémoire d'arbres XML (avec une implémentation dans le langage Java depuis la version 1.4). DOM est utilisé par un ensemble d'outils et de normes s'articulant autours de la manipulation d'arbre (tel que XSLT), la modification d'éléments (comme XUpdate), la recherche de nœud (XPath, XQuery) et la validation de grammaire (XML Schema, RelaxNG).
La combinaison d'une implémentation DOM avec le moteur de persistance Hibernate offre un compromis capable de proposer des fonctionnalités proches des bases de données XML natives avec des performances correctes.

2 Hibernate

2.1 Présentation

Hibernate est une librairie de persistance d'objets Java dans une base de données relationnelle. A l'aide d'un mapping, n'importe quel objet Java classique (Plain Old Java Object, ou POJO) est automatiquement pris en charge par Hibernate pour sauvegarder et restituer les valeurs contenues dans les instances. Hibernate ne nécessite aucune modification du code source de l'application, et fonctionne à l'aide des mécanismes de réflexivité du langage Java. Le mapping indique les propriétés persistantes de la classes nécessaires pour lire et écrire les états de l'instance. Par introspection, Hibernate identifie les méthodes de la classe qui correspondent à celles définies dans le mapping avec la base de données. Ces méthodes sont automatiquement appelées par Hibernate qui se charge de peupler les objets, lors de la récupération des instances, ou à l'inverse de remplir la base en cas de sauvegarde. Hibernate encapsule certains objets à l'aide de la classe Proxy en Java, pour justement intercepter les appels qui nécessiteraient la restitution d'instance, ce, afin de minimiser l'utilisation de l'espace mémoire et le nombre de requêtes à la base de données.



Hibernate propose plusieurs fonctionnalités, ce qui le rend fort attractif pour le type d'application que nous allons présenter :

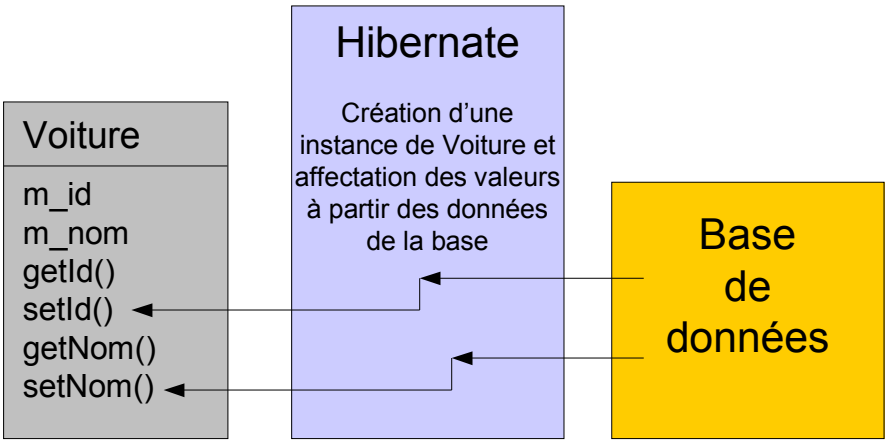
- Il n'interfère pas avec le code existant : le mécanisme de réflexion disponible dans le langage Java, offre la possibilité d'interagir avec des classes existantes sans nécessairement en connaître la structure au préalable. Hibernate va même plus loin, puisque dans ses dernières versions il utilise les fonctions de la librairie CGLIB qui assure la génération de bytecodes Java à chaud pour implémenter les classes chargées de la persistance des données.
- Il propose un mécanisme de chargement à la demande (type 'lazy') des données en collection (très pratique pour les structures en arbre qui ne nécessitent pas l'intégralité de la hiérarchie en mémoire).
- La sauvegarde de données ne s'effectue que sur les modifications et par bloc (ce qui réduit le nombre de requêtes), et de façon transactionnelle.
- Il ne fait pas de pooling d'instances, et ne nécessite donc pas de mécanisme de synchronisation avancée (mais utilise un système de cache au niveau de la récupération des données)

Hibernate est conçu pour se greffer facilement sur des applications existantes. La configuration permet de définir l'utilisation de différentes implémentations de TransactionFactory, gestionnaire de cache, datasource etc, qui lui assure une grande souplesse et une bonne intégration avec les serveurs d'application existants tels que Tomcat ou JBoss.

2.2 Utilisation d'Hibernate

2.2.1 Définition du mapping POJO/SGBD

Dans l'exemple ci dessous, nous définissons le mapping de persistance dans une base relationnelle pour l'objet Voiture. A aucun moment la structure de la classe 'Voiture' n'est modifiée pour implémenter le mécanisme de persistance.




```
public class Voiture {

	private Long m_id;
	private String m_nom;

	private void setId( Long id ) {
	    m_id = id;
	}
	public Long getId() {
	    return m_id;
	}

	void setNom( String str ) {
	    m_nom = str;
	}
	public String getNom() {
	    return m_nom;
	}
}
```

```
<class
	name="Voiture"
	table="voiture_db"
	>

<id name="id" type="java.lang.Long" column="id_db">
	<generator class="increment"/>
</id>

<property
	name="nom"
	type="java.lang.String"
	column="nom_db"
/>
</class>
```

Tel que nous le définissons dans le mapping, les valeurs 'id' et 'nom' de l'objet 'Voiture' sont sauvegardées dans la base, dans les champs 'id_db' et 'nom_db' de la table 'voiture_db'. L'attribut 'name' du fichier de mapping indique les méthodes get/set associées à la persistance. Par exemple pour le champ 'nom_db' de la base, les méthodes getNom/setNom, identifiées à l'aide la valeur 'nom' de l'attribut 'name', seront appelées respectivement lors de la sauvegarde et de la restitution des données persistantes.

2.2.2 Persistance des données

Hibernate permet à travers son API de programmation, d'assurer la persistance des objets auxquels sont associés un mapping. Hibernate se charge, en autres, de traiter les problèmes spécifiques au mapping Objet/Relationnel liés au polymorphisme et à la gestion des relations multivaluées. Hibernate propose également d'utiliser HQL, un langage de requête proche de SQL qui s'affranchit de toute dépendance syntaxique avec un type de base de données. HQL offre notamment la possibilité d'associer directement des classes d'objet dans les requêtes, Hibernate se chargeant d'effectuer la conversion en SQL propre à la base de données utilisée.

Exemple de sauvegarde de données :

```
Voiture voiture = new Voiture();
...
Transaction tx = session.beginTransaction();
session.save( voiture );
tx.commit();
```

Dans l'exemple ci dessus, les valeurs contenues dans l'instance 'voiture' seront automatiquement sauvegardées de façon transactionnelle lors de l'exécution de l'instruction 'session.save'.

Exemple de récupération de données :

```
List list = session.find( "from Voiture where nom='"+  panhard +"'" );
Voiture voiture = (Voiture)list.get(0);
```

Ici, nous récupérons une liste de voitures ayant pour nom 'panhard'.

3 Présentation de l'interface Java 'Document Object Model'

Document Object Model (DOM) est une spécification du W3C qui vise à définir une interface pour parcourir un arbre d'objets représentable sous forme de document XML. L'interface DOM est intégrée dans le langage Java depuis la version 1.4 (via les packages JAXP, Java API for XML Processing) et permet de créer, modifier, déplacer, et supprimer les nœuds d'un arbre.

DOM (contenu dans les packages 'org.w3c.dom'), malgré quelques imperfections, s'est rapidement imposé comme standard, ce qui a permis le développement d'un grand nombre d'outils s'articulant autour des fonctionnalités disponibles, les premières librairies ayant vu le jour étant des moteurs de translation (XSLT), suivis des langages de requêtes (XPath, XQuery) ou de mise à jour (XUpdate) ou encore des outils de validation de schéma (comme IsoRelax).

Différentes implémentations de l'interface DOM existent, à commencer par celle proposée par le langage Java. Compte tenu de la simplicité de l'interface, il est facile de développer sa propre implémentation.

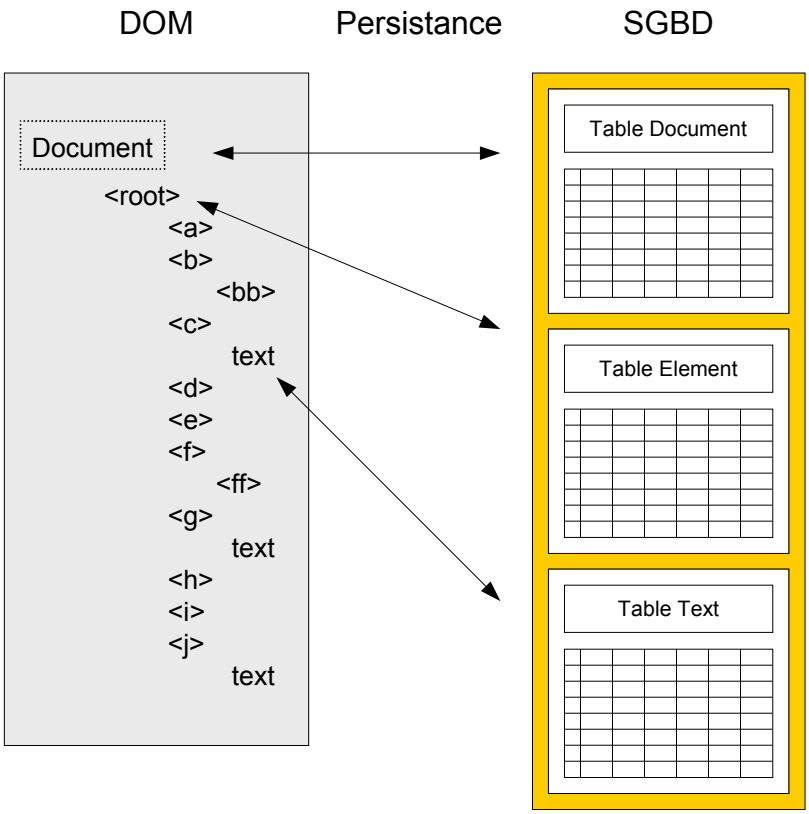
La manipulation de l'arbre s'effectue essentiellement à travers les interfaces 'Element' et 'Text' qui étendent l'interface 'Node'. Un 'Element' peut contenir un ensemble de nœuds (essentiellement d'autres 'Element' ou des nœuds 'Text'). Un 'Text' véhicule, bien entendu,

des données de type textuel et en théorie ne doit pas avoir d'enfant (en théorie car l'interface nœud étendue par 'Text' propose la méthode de manipulation de 'Child'). L'intégralité d'un document XML est géré au niveau de l'implémentation de l'interface 'Document' qui propose notamment la fonction d'import, seul point assurant la copie de fragments entre deux documents (éventuellement d'implémentation différente).

4 Mapping DOM-SGBDR via Hibernate

Comme nous l'avons présenté plus haut, Hibernate assure le mapping des objets Java sans avoir à intervenir sur le code existant. Par conséquent il est possible de proposer une persistance pour une implémentation de DOM en toute transparence à l'aide d'Hibernate.

L'interface DOM manipulant une structure hiérarchique, elle se prête pas vraiment à un mapping dans une base relationnelle (ce qui d'ailleurs justifie l'adoption d'une base de données XML native) . Cependant, comme nous allons le voir, Hibernate offre un compromis relativement efficace et simple pour économiser les requêtes et sous requêtes nécessaires pour récupérer une arborescence (et n'utiliser que le nécessaire).



Pour illustrer le mapping DOM-SGBDR nous prendrons comme exemple une représentation simpliste d'implémentation de DOM.

Afin de restituer la structure en arbre du modèle DOM, nous utilisons des relations 'many-to-one' et 'one-to-many' entre les nœuds. Dans l'exemple ci dessous nous présentons un mapping pour l'implémentation d'une classe abstraite 'AbstractNode' de l'interface 'Node'.

```
<class
	name="AbstractNode"
	table="abstractnodes"
	dynamic-insert="true"
	dynamic-update="true"
	>
	<id name="id" type="java.lang.Long" column="id">
		<generator class="increment"/>
	</id>

	<many-to-one name="pParent"
		class="AbstractNode"
		column="parent_id"
	/>

</class>
```

Le schéma de la table SQL (PostgreSQL) associée sera

```
CREATE TABLE abstractnodes (
	  id integer NOT NULL,
	  document_id integer,
	  parent_id integer,
	  list_index integer
};
```

La déclaration contenue dans l'éléments 'many-to-one' permet d'associer le nœud parent du nœud actuel.

De même pour établir la relation père/fils entre nœuds dans l'arbre, nous déclarons une liste de nœud au niveau d'une classe abstraite 'AbstractParentNode' dans le mapping. A noter qu'ici, la classe 'AbstractParentNode' dérive la classe 'AbstractNode' ce qui se traduit ici par l'écriture d'un élément 'joined-subclass' assurant la jointure entre les deux tables de données pour les deux classes respectives.

```
<joined-subclass
	name="AbstractParentNode"
	table="abstractparentnodes"
>
	<key column="id"/>

	<list name="pChilds" cascade="all" lazy="true">
		<key column="parent_id"/>
		<index column="list_index"/>
		<one-to-many class="AbstractNode"/>
	</list>
</joined-subclass>
```

La liste 'pChilds' sera peuplée de nœuds implémentant la classe abstraite 'AbstractNode' évoquée plus haut. La relation 'one-to-many' permet d'indiquer qu'un nœud parent contient plusieurs fils (ce qui rejoint la relation 'many-to-one' présentée plus haut, ou plusieurs nœud peuvent avoir un seul parent). La liste conserve l'ordre des nœuds fils qu'elle contient, ce qui se traduit par l'utilisation du champs 'list_index' dans la table 'abstractnode' utilisé par la classe 'AbstractNode'.

Dans la déclaration de la liste, l'attribut 'lazy' est positionné à 'true' ce qui signifie que les données de la liste seront chargées si nécessaire (tentative d'accès). Cette fonctionnalité permet d'éviter le chargement intégral d'un arbre de données en mémoire et bien évidemment d'effectuer un nombre optimal de requêtes avec la base.

5 Utilisation des XPath, XQuery et XUpdate

5.1 XPath

5.1.1 Présentation

XPath est un langage permettant d'identifier une partie d'un document XML. XPath 1.0 a été conçu à l'origine pour être utilisé par XSL et XPointer, pour finalement être décorrélé de ces deux normes et utilisé indépendamment en tant que langage de requête. À l'aide d'une expression XPath, il est possible de spécifier un nœud ou une liste de nœuds en fonction d'une position relative ou absolue (à partir de la racine du document) dans un document.

Par exemple, si nous considérons le document suivant :

```
<root>
      <a type="1">text1</a>
      <a type="2">text2</a>
      <b>text3</b>
      <c>
            <d>text5</d>
      </c>
      <c>
            <d>text6</d>
            <e>text7</e>
      </c>
      <f>
            <d>text8</d>
            <e>text9</e>
      </f>
</root>
```

Une expression XPath s'écrira de la façon suivante :

```
/root/a/text()
```

Cette expression désigne les nœuds textes situés dans les éléments <a> imbriqués dans l'élément <root> à savoir : 'text1' et 'text2'.

```
//e/text()
/root//e/text()
```

Dans les exemples ci dessus nous utilisons la notation '//' pour indiquer que nous ne prenons pas en compte les ancêtres de l'élément pour l'évaluation du XPath. Par conséquent, cette expression désignera les nœuds textes : 'text7' et 'text9'.

```
/root/a[1]/text()
```

Les accolades permettent de spécifier une contrainte sur la position du nœud dans l'arbre. Dans l'exemple ci dessus, le premier élément <a> seulement est considéré lors de l'évaluation de l'expression. L'évaluation de l'expression renverra le nœud texte : 'text1'.

De même, il est également possible de limiter la sélection en spécifiant une valeur requise pour un attribut des éléments.

```
/root/a[@type='1']/text()
```

Ici nous indiquons que l'élément <a> doit avoir l'attribut type ayant la valeur '1' (similaire à une clause 'where' dans une expression SQL), et donc après évaluation de l'expression, le nœud texte retenu sera : 'text1'.

Sur le même principe, des requêtes un peu plus complexes peuvent être écrites :

```
//c[d/text()='text6'][e/text()='text7']
```

Ici nous recherchons l'ensemble des éléments <c> qui comprennent les sous-éléments <d> contenant le nœud texte 'text6' et les sous-éléments <e> avec le nœud texte 'text7'.

Tel que nous le présentons dans les exemples ci-dessus, XPath apparaît comme un langage de requêtes capable de fournir des listes de résultats en fonction du chemin souhaité. D'une certaine manière, XPath associé à un document XML structuré de façon pertinente, couvre une parties des fonctionnalités attendues dans une base de données.

5.1.2 XPath, DOM et Hibernate

Plusieurs implémentations des spécification permettent l'utilisation de requêtes XPath sur des documents supportant l'interface DOM.
Pour le langage Java, les librairies telles que Xalan ou Saxon supportent l'évaluation d'expression XPath à partir de documents basés sur DOM.

Appliquées à une implémentation de DOM, de telles librairies ne profitent pas d'optimisations capables d'accélérer le traitement des requêtes. En effet, dans le cas d'une requête ciblant un nombre restreint de réponses, la construction des résultats pourra nécessiter un parcours linéaire complet des données.

```
/root/a[@type='1']/text()
```

Dans cet exemple, l'évaluation de l'expression, sur une implémentation DOM standard, se traduira par une lecture de tous les attributs type des éléments <a> contenus dans l'élément <root>.

Dans le cas d'une implémentation DOM persistante à travers un mapping Hibernate, l'utilisation de requêtes HQL (Hibernate Query Language) sur une base de données relationnelle indexée (sur les noms des éléments, les valeurs contenues dans les attributs et les nœuds textes) apporte des gains en performance à partir d'un certain volume traité. Contrairement à la démarche de mapping transparent proposée dans les chapitres précédents (pour persister la structure de données DOM), l'utilisation de requête HQL nécessite une intervention au niveau du code des implémentations.

Au niveau même de l'implémentation de l'interface 'Element' de DOM, deux méthodes peuvent être réécrites :

```
public NodeList getElementsByTagName(String name);
public NodeList getElementsByTagNameNS(String namespace, String name);
```

Ces deux méthodes renvoient une liste de sous-éléments en fonction du nom et éventuellement de l'espace de noms ('namespace') fournis en paramètres.

Ces méthodes sont notamment utilisées (par Saxon et Xalan) lors d'une évaluation d'un XPath incorporant une recherche exhaustive sur un élément donné.

```
/root//e/text()
```

Ici, l'ensemble des sous-éléments <e> de l'élément <root> doit être récupéré.

Afin de limiter le parcours séquentiel des sous-éléments et par extension un chargement en mémoire de tous les éléments, nous pouvons utiliser la requête HQL pour chaque niveau hiérarchique de l'arbre XML :

```
List list = session.find( "from Elements where name='e'" );
```

L'exécution de cette requête, permet notamment de charger uniquement en mémoire les éléments <e>.

Dans le cas d'évaluation d'expressions plus complexes, il est alors possible d'intervenir à même le code des différentes implémentations XPath. L'utilisation de requêtes HQL réduit alors fortement le parcours séquentiel des éléments dans la structure arborescente.

Par exemple pour l'évaluation de l'expression :

```
/root/a[@type='1']/text()
```

Nous effectuerons la requête HQL permettant d'obtenir la liste des éléments <a> avec l'attribut 'type' contenant la valeur '1'.

```
List list = session.find( "select e
     from ElementImpl as e, AttributImpl as a
     where e.name='a'
     and e.parent_id='"+ rootId +"'
     and a.parent_id=id
     and a.name='type'
     and a.value='1'
" );
```

Bien entendu, un requête HQL traduisant un XPath devient rapidement lourde dans le cas d'une évaluation faisant intervenir des tests sur un grand nombre de nœuds de la hiérarchie.

5.2 XQuery

5.2.1 Présentation

XQuery est un langage de requête permettant de représenter sous forme de fragments de document le résultat d'une recherche dans une structure XML.
La recommandation 1.0 (encore au stade de working draft) normalisée par le W3C, s'appuie fortement sur la recommandation XPath 2.0 et introduit la notion de schéma dans la validation des structures manipulées.
XQuery repose principalement sur l'évaluation d'expressions XPath pour identifier les liste de nœuds à traiter lors de l'évaluation de requêtes.
Le résultat d'une requête XQuery est structuré sous la forme d'un fragment de document qui est construit en fonction des valeurs récupérées et du nombre d'items manipulés.

Dans l'exemple ci dessous, nous effectuons une requête XQuery simple qui récupère deux listes de nœuds textes et les retourne dans une liste de couples d'élément <value-a> et <value-b>.

```
FOR $a IN /root/a/text()
FOR $b IN /root/b/text()
RETURN
     <value-a>{$a}</value-a>
     <value-b>{$b}</value-b>
```

Par ailleurs XQuery simplifie l'écriture d'expression XPath faisant intervenir des clauses conditionnelles (type where).
Par exemple la requête :

```
FOR $b IN //book
WHERE $b/year/text() = "1998"
RETURN
    <author>{$b/author}</author>
```

Sera équivalente à celle ci :

```
FOR $b IN //book[year/text='1998']/author
RETURN
    <author>{$b}</author>
```

Dans l'exemple ci dessus l'intérêt de la syntaxe proposée par XQuery n'est pas flagrant (mis à part la construction de la réponse dans une structure XML).
Cependant XQuery devient particulièrement simple et efficace dans le cas de requêtes plus complexes.

Par exemple, si nous supposons que nous avons deux fichiers contenant une liste de livres de deux boutiques différentes. La requête suivante permet d'obtenir simplement les titres des livres communs aux deux boutiques mais moins chers dans la boutique 1 :

```
FOR   $a IN document(livres-boutique1.xml)//book,
      $b IN document(livres-boutique2.xml)//book
WHERE
      $a/ISBN = $b/ISBN
AND   $a/prix < $b/prix

RETURN
    <titre>{$a/titre}</titre>
```

Au delà de ces exemples simples, le langage XQuery offre un grand nombre de fonctionnalités que nous détaillerons pas, telles que :

- la prise en compte du schéma dans l'évaluation d'un XPath (XPath 2.0)
- la validation d'un schéma de résultat (pour s'assurer que la fragment de document retourné respect un schéma donné)
- les branchements conditionnels (if, then, else) dans la construction de résultats
- la définition de fonctions et par extension d'évaluation récursive

5.2.2 Utilisation d'Hibernate

Comme nous l'avons illustré plus haut, XQuery repose sur XPath pour la sélection des listes de nœuds à traiter avec cependant, la possibilité de décomposer des XPath en évaluation successives. Si bien qu'en règle générale, une évaluation de requêtes XQuery repose sur une expression XPath globale qui sera affinée à travers les clauses 'where'.
Par conséquent il est préférable d'éviter de récupérer en mémoire l'intégralité des résultats des requêtes XPath.

Hibernate propose des mécanismes avancés de chargement de données à la demande pour le traitement d'un grand volume de données. Ce mécanisme repose sur l'utilisation d'un itérateur Java qui déroule le résultat de la requête au fur et à mesure et qui ne charge les données qu'en cas d'accès.

Si nous reprenons l'exemple d'XPath présenté dans paragraphe précédent, nous obtenons :

```
Iterator iterator = session.iter( "select e
     from ElementImpl as e, AttributImpl as a
     where e.name='a'
     and e.parent_id='"+ rootId +"'
     and a.parent_id=id
     and a.name='type'
     and a.value='1'
" );
while( iterator.hasNext() ){
     Element element = (Element)iterator.next(); // chargement des données
     if( ... ){
     }
}
```

Ici, le traitement est effectué en itérant sur les résultats de la requête sans préchargement en mémoire des objets.

5.3 XUpdate

XUpdate est un langage conçu pour assurer la mise à jour de documents XML, fonctionnalité que ne propose pas XQuery. La norme XUpdate ratifié par le groupe de travail XML :DB en 2000 est encore à l'état de working draft. XUpdate est utilisé dans des bases de données XML native telle que Xindice ou X-Hive/DB. Tout comme XQuery, XUpdate s'appuie sur les expressions XPath pour sélectionner l'emplacement dans un document XML des nœuds à mettre à jour.

Le langage XUpdate propose les fonctions suivantes : insert (insertion), append (ajout en fin de liste), update (mise jour), remove (suppression) et rename (renommer).

Par exemple l'expression ci dessous, met à jour le contenu du nœud texte désigné par le XPath "/books/book[1]/author" (l'auteur du 1er livre dans la liste des 'books').

```
<xupdate:update select="/books/book[1]/author">
     Emile Zola
</xupdate:update>
```

Après évaluation de l'expression, l'auteur prendra pour valeur : 'Emile Zola'.

XUpdate est relativement basique et, dans l'état actuel, ne s'avère pas aussi riche et puissant que XQuery. XUpdate ne propose pas encore la validation de schéma, les traitements conditionnels, la définition de block de transactions etc.

6 Benchmarks

6.1 Dispositif

Afin d'illustrer le fonctionnement d'Hibernate nous proposons d'effectuer des mesures de performance sur une implémentation de DOM utilisant le mécanisme de persistance proposé par Hibernate.

Pour effectuer nos mesures, nous utilisons la librairie open-source XMLPersistence 1.2-rc1 (http://xml-persistence.net) mappée sur une base de données relationnelle PostgreSQL 7.4 à l'aide d'Hibernate 2.1.1, le tout fonctionnant sur une machine Linux 2.4.25/P4-2.4GHz .

Nous insistons sur le fait qu'un grand nombre d'optimisations Hibernate ne sont pas encore exploitées par la librairie XMLPersistence.

Pour nos tests, nous utiliserons une structure simpliste de type :

```
<books>
      <book>
            <ISBN></ISBN>
            <author></author>
            <year></year>
      </book>
</books>
```

En insertion nous comparons une requête XUpdate à une requête SQL directe.

Requête XUpdate utilisée :

```
<xupdate:insert-after select='/books/book[last()]'>
<xupdate:element name='book'>
<title>Title</title>
<author>Auteur</author>
<ISBN>123456789</ISBN>
</xupdate:element>
</xupdate:insert-after>
```

Requête SQL utilisée :

```
INSERT INTO testtable (id, title, author, isbn) VALUES (?,?,?,?)
```

En sélection nous comparons une requête XPath à une requête SQL directe.

Requête XPath utilisée :

```
/books/book[./author/text()='Auteur1']
```

Requête SQL utilisée :

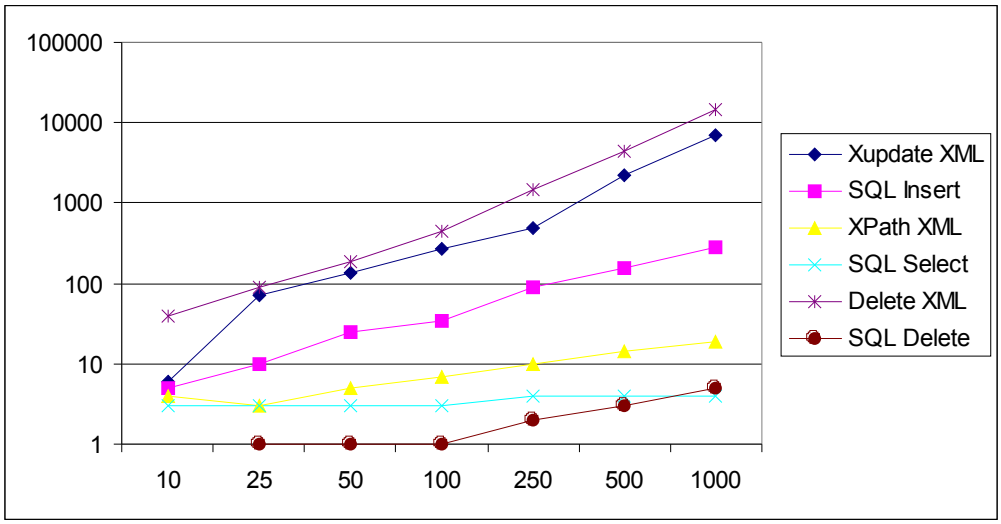
```
SELECT * FROM testtable WHERE author=?
```

En suppression nous effectuons un ‘delete’ Hibernate sur la racine du document et une requête SQL équivalente.

6.2 Benchmark

Nous avons effectué les mesures en miliseconde pour un nombre d’entrées variables (10, 25, 50, 100, 250, 500, 1000).

	10	**25**	**50**	**100**	**250**	**500**	**1000**
XUpdate XML	6	72	136	271	491	2247	7094
SQL Insert	5	10	25	34	90	158	282
XPath XML	4	3	5	7	10	14	19
SQL Select	3	3	3	3	4	4	4
Delete XML	40	90	185	443	1446	4407	14868
SQL Delete	0	1	1	1	2	3	5



Le graphique ci dessus présente une progression linéaire en fonction du nombre d’entrée pour chaque type de requête. En revanche nous constations sans surprise un écart de performance significatif entre les requêtes SQL directe et les requêtes dans la structure XML, du aux différentes couches logicielles et la démultiplication de requêtes.

La structure XML est particulièrement pénalisante lors de la suppression qui se traduit par un grand nombre de requêtes dans la base (une par niveau de profondeur).

7 Conclusion

Au travers de cet article, nous avons présenté une utilisation originale de la librairie de persistance Hibernate en mélangeant les notion de bases de données relationnelles et de bases de données XML.

En s'appuyant sur un mapping objet dans une base de données relationnelle nous avons rapidement restitué les principales caractéristiques proposées par les bases de données XML. L'utilisation des fonctions proposées par Hibernate nous a permis de proposer des optimisations assurant des performances correctes.

Bien entendu, la complexité d'une structure hiérarchique telle que nous l'avons implémenté ne souffre pas la comparaison avec une approche reposant sur une base de données relationnelle en terme de performance, et plus particulièrement lors de l'écriture de données. Nous avons clairement mis en évidence la principale faiblesse de notre modèle reposant sur la transcription d'un arbre dans un ensemble de tables.

Cependant, à défaut d'être extrêmement performante, l'utilisation de la persistance XML dans un SGBD se justifie : en tant que stratégie de coexistence ou de transition d'un SGBD vers une BD-XML, pour la robustesse et la maturité d'un SGBD traditionnel par opposition à une BD-XML native récente, pour un besoin fortement axé en lecture (ou la contrainte de performance est alors parallélisable en cluster).

Dans tous les cas, la richesse, la simplicité et l'efficacité de ses fonctions font d'Hibernate un moteur de persistance puissant et complet.

8 Références

Hibernate : http://www.hibernate.org/
XML : http://www.w3.org/XML/
DOM : http://www.w3.org/DOM/
XPath : http://www.w3.org/TR/xpath
XSL : http://www.w3.org/TR/xsl/
XQuery : http://www.w3.org/TR/xquery/
XUpdate : http://www.xmldb.org/xupdate/
JAPX : http://java.sun.com/xml/jaxp/
Xerces : http://xml.apache.org/xerces-j/
Xalan : http://xml.apache.org/xalan-j/
Saxon : http://saxon.sourceforge.net/
XMLPersistence : http://www.xml-persistence.net/