Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Travailler avec des données de type XML

Version 1.0

Microsoft CERTIFIED Technology Specialist
Grégory CASANOVA

Sommaire

1 Introduction 3
2 Pré requis 4
3 XML typé et non typé 5
3.1 Non typé 5
3.2 Typé 5
4 XML et DML 7
4.1 Le DML Général 7
4.2 Le DML Spécifique 9
4.2.1 Query() 9
4.2.2 Value() 9
4.2.3 Exist() 10
4.2.4 Nodes() 11
4.2.5 Modify() 12
4.2.6 Les requêtes XQuery et XPath 13
4.3 FOR XML 13
4.4 OpenXML 15
4.5 OPENROWSET 16
5 INDEX et XML 17
6 Conclusion 18

1 Introduction

Le type de données XML est apparu dans SQL Server la première fois pour la version 2000. Ce résultat est dû à une évolution des types de stockages pour les entreprises. Le type de données XML étant de plus en plus utilisé, il était évident que Microsoft s'adapte dans son SGBDR. Dans le cas présent, ce qui nous intéresse n'est pas de pouvoir stocker les données au format XML, mais de les exploiter directement et de façon optimale grâce à SQL Server 2008. Il sera donc possible, grâce à SQL Server, d'intégrer des données contenues dans un document XML, aux tables dans un schéma relationnel, ou inversement, de produire un document XML, grâce aux données relationnelles d'une ou plusieurs tables dans une base de données. Le but de l'intégration du XML dans SQL Server, est de pouvoir stocker les données dans leur format natif, c'est pourquoi il est possible d'intégrer des données dans un schéma relationnel, aussi bien que dans leur format XML d'origine. Il n'est pas difficile de comprendre que Microsoft a considérablement augmenté sa capacité de gestion des données XML depuis la version 2000, dans le sens ou il n'était avant possible que de produire un document XML à partir de données relationnelles alors que désormais il est possible de travailler directement dessus dans leur forme native. De façon générale, les types de données, qu'ils soient relationnel ou typés XML, ne sont pas en concurrence dans votre base de données. Chacun possède ses avantages et ses inconvénients. En effet, le XML sera particulièrement adapté lorsque nous aurons besoin de faire de l'échange d'information ou encore de l'envoie de messages. Le type XML étant organisé de façon hiérarchique, Il est donc simple, du moment que la hiérarchie est connue, de retrouver des données. Les données sont typées en UTF-16 par SQL Server. Dans SQL Server, il est possible d'associer une colonne de type XML à une collection de schéma pour satisfaire les exigences du W3C en matière de XML. Dans le cas où la colonne qui contient les documents est attachée à des collections de schéma, on dira que le XML est typé, sinon il sera non typé. Pour ce cours, nous allons utiliser une table, qui possèdera une seule colonne typée XML, dont le script est le suivant :

```
CREATE TABLE Donnees_XML
(Colonne XML XML)
```

Nous n'allons pas pouvoir traiter le langage XML dans sa totalité dans ce cours, c'est pourquoi, si ce langage vous intéresse, il vous est possible de trouver de la documentation sur celui-ci, à l'adresse suivante : **http://www.w3.org/TR/REC-xml/** .

2 Pré requis

Pour pouvoir comprendre ce chapitre au mieux, vous devez connaitre :

- Les différentes opérations possibles sur une base de données, grâce au T-SQL (Chapitre 4).

3 XML typé et non typé

3.1 Non typé

Le XML, tel qu'il est définit par la norme ISO SQL-2003, se doit d'accueillir des documents XML 1.0 bien formés. Comme dit plus haut, un document non typé sera un document qui n'est pas rattaché à une collection de schémas, et donc, qui se doit d'accueillir des documents XML bien formés. Vous l'aurez compris, les données XML non typés doivent être utilisées lorsqu'il n'est pas possible de définir une collection de schémas.

3.2 Typé

Des données XML typées seront, comme énoncé précédemment, des documents liés à une collection de schémas. Puisque dans SQL Server, nous stockons les documents XML dans les colonnes typées XML, il est possible de rattacher directement une collection de schémas à une colonne particulière, afin que tous les documents devant être stockés dans la colonne soient concernés par la collection de schémas. Il ne sera donc pas possible, si une collection de schéma est lié à une colonne, d'insérer un document ne respectant pas le schéma définit. On peut donc définir les schémas comme des « contraintes d'intégrité » concernant les documents XML.

La gestion des collections de schémas XML se font par les mots clé CREATE XML SCHEMA COLLECTION, ALTER XML SCHEMA COLLECTION ET DELETE XML SCHEMA COLLECTION. Chaque collection va pouvoir contenir un ou plusieurs schémas XML, et va pouvoir être modifié après sa création, ce qui va nous permettre de s'adapter au fait de devoir ajouter ou supprimer des schémas dans la collection. La gestion des collections sera alors bien plus simple et flexible. La création d'un schéma se fait ainsi :

```
CREATE XML SCHEMA COLLECTION Schema_XML
AS
N'<?xml version="1.0" encoding="UTF-16"?>'+
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">'+
'<xs:complexType name="Client">'+
'<xs:sequence>'+
'<xs:element name="Nom"/>'+
'<xs:element name="Prenom"/>'+
'</xs:sequence>'+
'</xs:complexType>'+
'</xs:schema>';
```

Dans cet exemple, la colonne à laquelle sera rattaché le schéma devra comporter le même schéma que définit après la clause `AS`. Désormais pour rattacher ce schéma à une colonne, il faut le définir à la création de la table, comme ceci :

```
CREATE TABLE Donnees_XML
(Colonne_XML XML (Schema_XML))
```

La colonne ou est désormais défini le schéma ne pourra accepter que des valeurs formatées à ce schéma.

Note : Pour des données de type XML, il n'est pas toujours possible de définir à l'avance tout ce qu'elles vont contenir. Par exemple, on pourra ajouter un numéro de téléphone pour le compléter. Dans ce cas, le mot clé **any** nous aide dans cet objectif.

```
CREATE XML SCHEMA COLLECTION Schema_XML
AS
N'<?xml version="1.0" encoding="UTF-16"?>'+
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">'+
'<xs:complexType name="Client">'+
'<xs:sequence>'+
'<xs:element name="Nom"/>'+
'<xs:element name="Prenom"/>'+
'<xs:any namespace="autre" processContents="skip" minOccurs="0"
maxOccurs="unbounded"/>'+
'</xs:sequence>'+
'</xs:complexType>'+
'</xs:schema>';
```

Le mot clé **any** est suivit des attributs suivants :

- **Namespace :** l'espace de nom utilisé pour la validation des éléments de ce type.
- **processContents :** indique la manière dont les éléments doivent être validés par SQL Server. Trois valeurs sont possibles pour cet attribut :
  - **skip :** les éléments ne sont pas validés par rapport à un schéma.
  - **strict :** les éléments sont validés nécessairement par un schéma.
  - **lax :** les éléments sont validés par un schéma seulement si celui-ci existe dans la base.
- **minOccurs** : nombre minimal d'éléments de ce type.
- **maxOccurs :** nombre maximal d'éléments de ce type.

4 XML et DML

Pour rendre plus simple la compréhension de cette partie, nous allons supprimer la table que nous avons créée avant, puis la recréer de façon à ce qu'elle ne soit plus liée au schéma.

```
DROP TABLE Donnees_XML
GO

CREATE TABLE Donnees_XML
(catalogue XML)
GO
```

Découvrons maintenant les différentes instructions du DML appliquées à la manipulation du type de données XML.

4.1 Le DML Général

Dans un premier temps, il est nécessaire de bien comprendre certaines choses. Tout d'abord, lorsque nous appliquons une instruction du DML à une colonne de type XML, tout le contenu de l'enregistrement est manipulé, on peu donc considérer qu'on ne peut pas mettre à jour seulement une balise du document XML. Voici les résultat des quatre instructions du DML sur une colonne de type XML :

- INSERT :

```
INSERT INTO [Entreprise].[dbo].[Donnees_XML]
            ([Colonne_XML])
     VALUES
            ('<article>
             <nom>Windows Seven</nom>
             <prix>139</prix>
            </article>')
GO
```

Ici, nous insérons un article dont la définition contient un nom et un prix. Il est aussi possible d'importer les données XML depuis un fichier. Pour cela, créez un fichier que nous appellerons Cours.txt, et insérez le code suivant dans celui-ci :

```
<article>
<nom>Windows Seven</nom>
<prix>139</prix>
</article>
```

Nous utilisons le code ci-dessous :

```
INSERT INTO [Entreprise].[dbo].[Donnees_XML] ([Colonne_XML])
SELECT informations
FROM (SELECT *
FROM OPENROWSET(BULK 'C:\Users\Grégory Casanova\Desktop\Cours.txt',
SINGLE_BLOB) AS informations)
AS fichierXML(informations)
```

Grâce aux instructions OPENROWSET et BULK, le contenu du fichier dont le chemin est `'C:\Users\Grégory Casanova\Desktop\Cours.txt'` sera parsé et inséré dans la colonne de type XML de notre table.

- UPDATE :

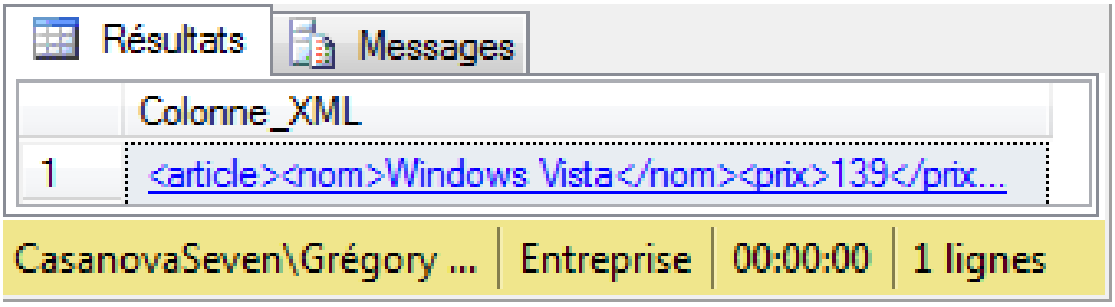
```
UPDATE [Entreprise].[dbo].[Donnees_XML]
   SET [Colonne_XML] = '<article>
            <nom>Windows Vista</nom>
            <prix>139</prix>
           </article>'
GO
```

Note : Il est important de noter qu'ici, nous changeons soit l'intégralité de l'enregistrement, soit rien. Tout l'enregistrement doit être redéfinit dans le cas d'un update simple. Il est indispensable de réécrire la totalité du code XML.

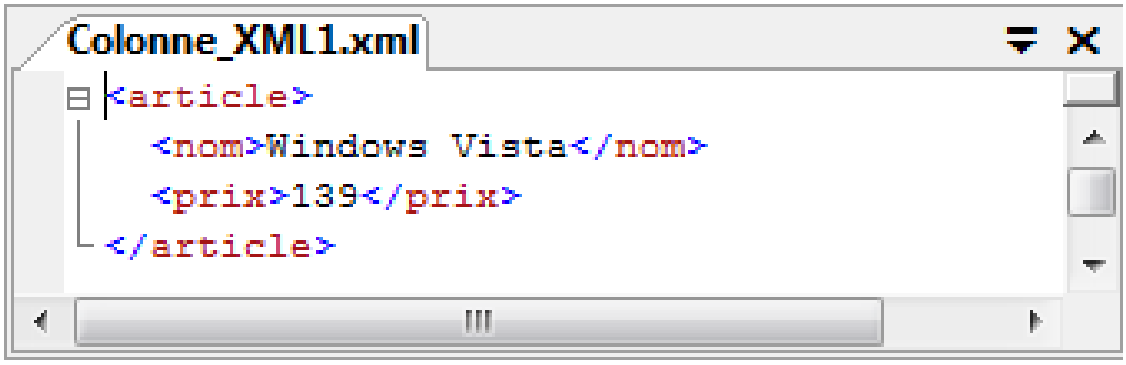
- SELECT :

```
SELECT *
FROM dbo.Donnees_XML
```

L'intégralité des données stockées dans chaque enregistrement sera sélectionnée. Avec une instruction SELECT simple, il n'est pas possible de sélectionner seulement le prix de l'article par exemple. Le résultat d'une instruction SELECT se présente de cette manière :



On remarque alors que le champ est cliquable. Le résultat « Complet » est accessible avec un clic sur celui-ci. Nous somme alors redirigés vers une nouvelle fenêtre de requête qui nous présente l'intégralité du code XML contenu dans le résultat, comme ceci :



- DELETE :

```
DELETE FROM [Entreprise].[dbo].[Donnees_XML]
GO
```

4.2 Le DML Spécifique

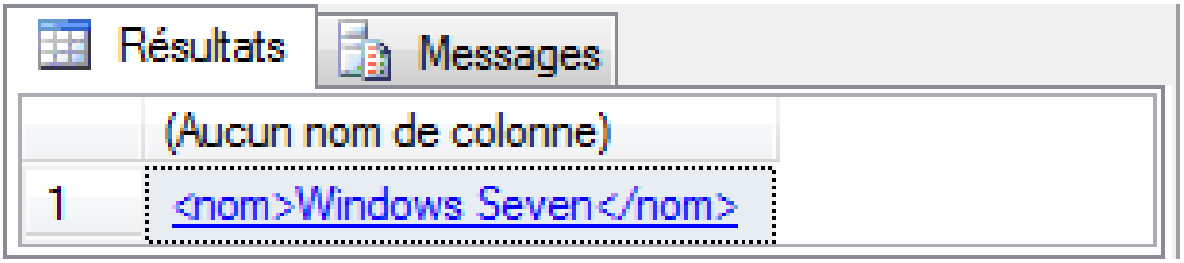
Dans la partie précédente, nous avons vue que les instructions DML générales ne peuvent toucher que l'ensemble des données d'un fichier XML, et non pas une partie de celui-ci. Il va de sois que la force principale du XML dans la base de données est de pouvoir différencier les informations dans un document, sans pour autant avoir besoin de manipuler l'enregistrement entier. C'est pourquoi il existe une méthode spécifique au type de données XML pour manipuler les données. Ces méthodes sont query(), value(), exist(), nodes() et modiify(). Voyons en détail toutes ces méthodes de manipulation des données XML.

4.2.1 Query()

Cette requête accepte en paramètre une requête de type Xquery. Elle va nous permettre de naviguer dans notre document XML, et d'appliquer une action DML sur un nœud particulier de notre document. Cette requête retourne une valeur XML non typée. Prenons un exemple :

```
SELECT catalogue.query('article/nom')
FROM Donnees_XML
```

Le résultat est le suivant :



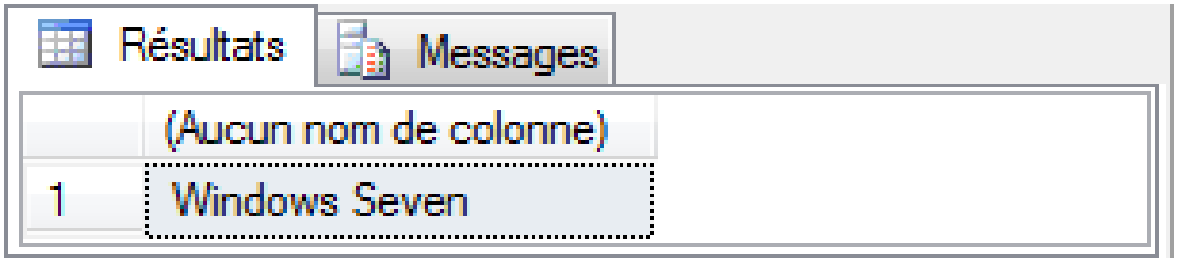
Il est aussi possible de cliquer sur le résultat de la requête pour afficher le résultat « complet » dans le gestionnaire de données XML. On retiendra que ce type de requête retourne un nœud complet du document XML et non la valeur comprise entre les balises d'un nœud de ce même document.

4.2.2 Value()

Grâce à l'instruction value(), nous allons pouvoir extraire la valeur contenue entre les balises XML d'un nœud du document XML sur lequel on travaille. Prenons un exemple :

```
USE Entreprise
SELECT catalogue.value('(article/nom)[1]', 'nvarchar(50)')
FROM Donnees_XML
```

La requête value suit le nom de la colonne de type XML où nous allons opérer. Cette requête prend 2 arguments qui sont le chemin de la valeur et un singleton, et le type de retour de la requête. La requête ci-dessus produit le résultat suivant :



On remarque alors que les balises ont disparues, et qu'il ne reste que la valeur comprise dans le nœud que nous avons placé en argument dans notre requête value().

4.2.3 Exist()

Cette instruction fait partie des instructions XQuery, et retourne une valeur scalaire de type bit. Si les données sont localisées, sa valeur est 1, si les données ne peuvent pas être localisées, sa valeur est 0 et si la valeur de l'information à trouver est NULL, alors sa valeur est NULL. Ce type d'instruction est pratique à utiliser conjointement aux autres instructions XQuery disponibles, par exemple, pour éviter la levée d'erreurs imprévues. Pour prendre un exemple plus concret, nous allons faire une insertion dans la table.

```
INSERT INTO Donnees_XML
VALUES
('<article>
  <nom>Windows Seven</nom>
  <prix>139</prix>
</article>'),
('<article>
  <nom>Windows Vista</nom>
  <prix>100</prix>
</article>'),
('<article>
  <nom>Windows XP</nom>
</article>')
```

Cette fois-ci, nous avons ajouté trois lignes dans notre colonne XML, deux contenant un prix et la troisième n'en contenant pas. Voyons maintenant le résultat d'un exemple d'utilisation de l'instruction exist() :

```
SELECT catalogue.value('(article/nom)[1]', 'nvarchar(50)')
FROM Donnees_XML
WHERE catalogue.exist('(article/prix)[1]') = 1
```

Cette requête retourne toutes les lignes pour laquelle il existe une balise XML `<prix>` non NULL. En voici le résultat :

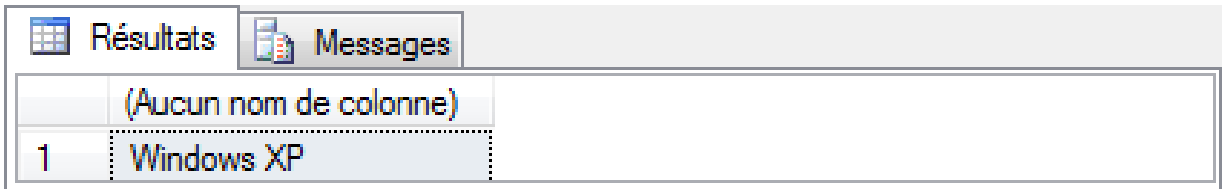
Résultats | Messages

	(Aucun nom de colonne)
1	Windows Seven
2	Windows Vista

On remarque alors que l'élément pour lequel il manquait le prix n'est pas présent, puisque la balise XML `<prix>`, n'existait pas. En revanche, si l'on passe la valeur de la requête exist() à 0, ce produit sera retourné, en remplacement des deux autre contenant une balise de prix.

```
SELECT catalogue.value('(article/nom)[1]', 'nvarchar(50)')
FROM Donnees_XML
WHERE catalogue.exist('(article/prix)[1]') = 0
```

Voici le résultat :



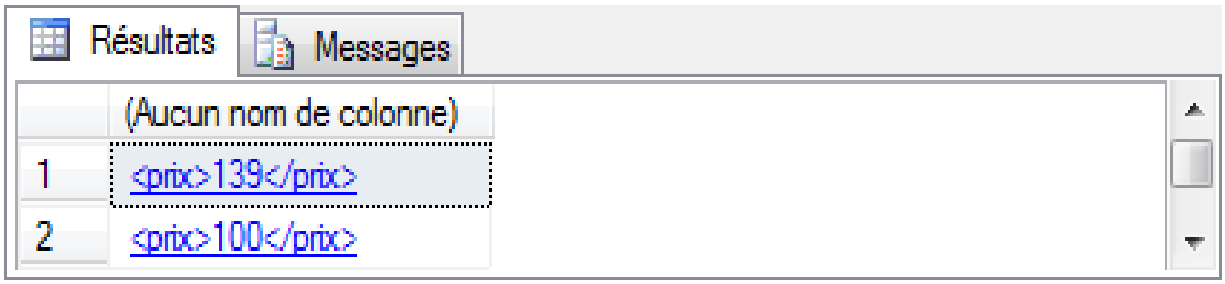
4.2.4 Nodes()

L'instruction nodes() va nous permettre de morceler notre résultat, à partir de nœuds particuliers, pour retourner seulement le prix par exemple. Dans notre exemple, nous allons morceler le résultat de rendu d'un document XML contenu dans une variable de type XML. Le résultat retournera plusieurs lignes dans lesquelles chaque prix apparaitra. Nodes() permet par conséquent de retourner une suite de lignes de résultats en définissant un nœud particulier dans le document XML.

```
DECLARE @XML XML = '<article>
  <nom>Windows Seven</nom>
  <prix>139</prix>
</article>
<article>
  <nom>Windows Vista</nom>
  <prix>100</prix>
</article>
<article>
  <nom>Windows XP</nom>
</article>'

SELECT resultats.x.query('.')
from @XML.nodes('article/prix') AS resultats(x)
```

Le résultat est le suivant :



Il est important de comprendre le fondement de cette requête. Nous aurions pue utiliser une simple requête query() pour retourner les prix du document XML, comme ceci :

```
DECLARE @XML XML = '<article>
  <nom>Windows Seven</nom>
  <prix>139</prix>
</article>
<article>
  <nom>Windows Vista</nom>
  <prix>100</prix>
</article>
<article>
  <nom>Windows XP</nom>
</article>'

SELECT @XML.query('article/prix')
```

Dont le résultat est celui-ci :

Résultats | Messages

	(Aucun nom de colonne)
1	<prix>139</prix><prix>100</prix>

Simplement, en utilisant une requête query() au lieu d'une requête nodes(), tous les prix auraient été dans la même ligne de résultat, au lieu d'être séparés comme vu dans les exemples.

4.2.5 Modify()

Comme son nom l'indique, l'instruction XQuery modify(), permet de modifier les données contenues dans un document XML, de façon ciblée, contrairement à l'instruction UPDATE qui nous permet de mettre à jour la totalité des données contenu dans la colonne. Pour ajouter, modifier ou supprimer des informations dans un document XML via la méthode modify(), on utilisera les instructions XML DML insert, replace value of et delete. Prenons un exemple :

```
DECLARE @XML1 XML =
'<ROOT>
<article>
  <nom>Windows Seven</nom>
  <prix>139</prix>
</article>
<article>
  <nom>Windows Vista</nom>
  <prix>100</prix>
</article>
<article>
  <nom>Windows XP</nom>
</article>
</ROOT>'

DECLARE @XML2 XML ='<quategorie>informatique</quategorie>'

SET @XML1.modify('insert sql:variable ("@XML2")
as last into (/ROOT/article)[1]')
SET @XML1.modify('insert sql:variable ("@XML2")
as last into (/ROOT/article)[2]')
SET @XML1.modify('insert sql:variable ("@XML2")
as last into (/ROOT/article)[3]')
```

Cette instruction va permettre de rajouter aux trois lignes, une valise quategorie pour chaque nœud article du document. Voici le résultat :

Résultats | Messages

	(Aucun nom de colonne)
1	<ROOT><article><nom>Windows Seven</nom><prix>139...

```
<ROOT>
  <article>
    <nom>Windows Seven</nom>
    <prix>139</prix>
    <quategorie>informatique</quategorie>
  </article>
  <article>
    <nom>Windows Vista</nom>
    <prix>100</prix>
    <quategorie>informatique</quategorie>
  </article>
  <article>
    <nom>Windows XP</nom>
    <quategorie>informatique</quategorie>
  </article>
</ROOT>
```

4.2.6 Les requêtes XQuery et XPath

Il existe un langage pour exécuter la plupart des requêtes possible dans SQL Server pour le format XML : Le XQuery et le XPath. Couvrir la totalité de ce langage serai trop long pour un seul chapitre, de plus, il est entièrement disponible sur le site officiel du W3C à l'addresse suivante : http://www.w3.org/TR/xquery .

4.3 FOR XML

La clause for XML permet de passer un jeu de résultat de données d'un format relationnel en un format XML au travers d'une requête de type SELECT. Cependant, malgré le fait que les informations soient extraites dans une forme XML, elles ne sont pas pour autant typées XML. Pour typer les données extraites au format XML, il faut ajouter la clause TYPE. Dans ce cas, les données extraites pourront directement insérées dans une colonne de type XML par exemple, car elles sont typées XML. Prenons un exemple concret :

```
USE Entreprise
SELECT Id_Client, Nom_client
FROM Client
FOR XML AUTO, TYPE
```

Le résultat est le suivant au travers de l'éditeur XML:

```
<Client Id_Client="1" Nom_client="CASANOVA" />
<Client Id_Client="2" Nom_client="RAVAILLE" />
<Client Id_Client="3" Nom_client="DOLLON" />
<Client Id_Client="4" Nom_client="VERGNAULT" />
<Client Id_Client="5" Nom_client="VASSELON" />
<Client Id_Client="6" Nom_client="HOLLEBEC" />
```

Si FOR XML AUTO transcrit directement les nœuds de votre format relationnel au format XML, il est possible de définir ce chemin de la manière suivante :

```
USE Entreprise
SELECT Id_Client, Nom_client
FROM Client
FOR XML PATH('Client'), TYPE
```

Le résultat est le suivant :

```
<Client>
  <Id_Client>1</Id_Client>
  <Nom_client>CASANOVA</Nom_client>
</Client>
<Client>
  <Id_Client>2</Id_Client>
  <Nom_client>RAVAILLE</Nom_client>
</Client>
<Client>
  <Id_Client>3</Id_Client>
  <Nom_client>DOLLON</Nom_client>
</Client>
<Client>
  <Id_Client>4</Id_Client>
  <Nom_client>VERGNAULT</Nom_client>
</Client>
<Client>
  <Id_Client>5</Id_Client>
  <Nom_client>VASSELON</Nom_client>
</Client>
<Client>
  <Id_Client>6</Id_Client>
  <Nom_client>HOLLEBEC</Nom_client>
</Client>
```

De plus, l'une des règles essentielle du XML est la suivante : Tout nœud doit avoir un seul nœud au niveau supérieur. C'est pourquoi il existe la clause ROOT. En voici un exemple :

```
USE Entreprise
SELECT Id_Client, Nom_client
FROM Client
FOR XML PATH('Client'),ROOT('ROOT'), TYPE
```

Le résultat est le suivant :

```
<ROOT>
  <Client>
    <Id_Client>1</Id_Client>
    <Nom_client>CASANOVA</Nom_client>
  </Client>
  <Client>
    <Id_Client>2</Id_Client>
    <Nom_client>RAVAILLE</Nom_client>
  </Client>
  <Client>
    <Id_Client>3</Id_Client>
    <Nom_client>DOLLON</Nom_client>
  </Client>
  <Client>
    <Id_Client>4</Id_Client>
    <Nom_client>VERGNAULT</Nom_client>
  </Client>
  <Client>
    <Id_Client>5</Id_Client>
    <Nom_client>VASSELON</Nom_client>
  </Client>
  <Client>
    <Id_Client>6</Id_Client>
    <Nom_client>HOLLEBEC</Nom_client>
  </Client>
</ROOT>
```

4.4 OpenXML

Cette instruction permet de traiter un document XML sous la forme d'un jeu de résultats. Il est possible d'utiliser cette instruction couplée à un SELECT, INSERT, UPDATE et DELETE. Pour utiliser cette méthode, il est necessaire de retenir l'utilisation de deux procédures stockées qui sont les suivantes :

- Sp_xml_preparedocument : permet de préparer un document texte au format XML. Cette procédure stockée retourne un identifiant qui va servir à la méthode OpenXML à retrouver les données XML créées.
- Sp_xml_removedocument : permet de libérer l'espace mémoire occupé par le document préparé par la procédure stockée précédente.

Prenons un exemple :

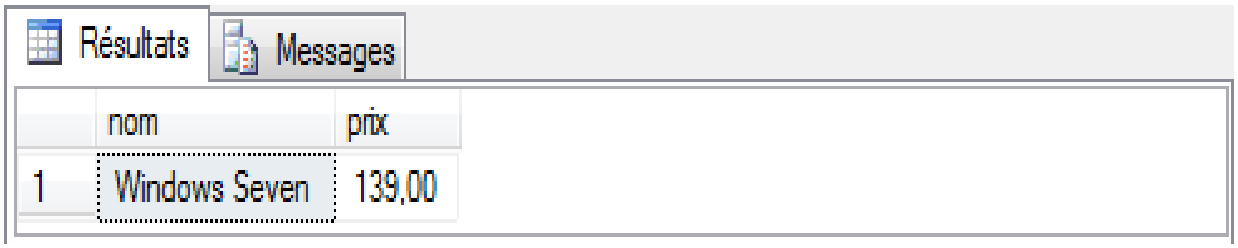
```
DECLARE @ID_XML int
DECLARE @XML1 nvarchar(500) =
'<ROOT>
<article nom="Windows Seven" prix="139"></article>
</ROOT>'

exec sp_xml_preparedocument @ID_XML OUT, @XML1

SELECT *
FROM OPENXML(@ID_XML, 'ROOT/*')
WITH (nom nvarchar(50), prix money)

exec sp_xml_removedocument @ID_XML
```

Le résultat est le suivant :



On stocke dans un premier temps notre document texte dans une variable. On le prépare ensuite grâce à la procédure stockée de préparation de documents XML. On applique ensuite OPENXML pour extraire les données contenues dans le document au format XML. Lorsque l'instruction SELECT est terminée, on utilise la procédure stockée de suppression de document XML pour libérer l'espace mémoire contenant le document préparé.

4.5 OPENROWSET

Cette méthode permet de travailler facilement avec des données présentes à l'extérieur du serveur. La source de données peut être le système de fichier Windows ou encore une autre base OLEDB. Prenons un exemple :

```
USE Entreprise
INSERT INTO Donnees_XML
SELECT INFO
FROM (SELECT *
FROM OPENROWSET (BULK 'C:\Users\Grégory Casanova\Desktop\Cours.xml',
SINGLE_BLOB) AS INFO)
AS fichierXML(INFO)
```

Dans cet exemple, on insère les données contenues dans un document XML, présent sur le système de fichier Windows, dans une table contenant une colonne typée XML.

5 INDEX et XML

Lors d'une requête, les données typées XML peuvent être volumineuses ou/et nombreuses, et donc être couteuses en ressources au niveau serveur. Un index dit principal, sur une colonne XML, est construit de la même manière qu'un index sur une colonne d'un type simple. Il existe simplement des index dit secondaires qui vont servir à accélérer la recherche de données. Les index secondaires sont définis celons des classes de requêtes fréquentes :

- PATH : Pour les requêtes portant sur le chemin d'accès.
- PROPERTY : Pour les requêtes portant sur les propriétés.
- VALUE : Pour les requêtes portant sur des valeurs.

Tout d'abord, la création d'un index primaire nécessite que la clé primaire de la table qui contient la colonne XML contienne un index organisé. On peut alors créer l'index primaire de cette façon :

```
CREATE PRIMARY XML INDEX nom_index ON nom_table(Colonne)
```

Il n'est donc possible de créer un index secondaire seulement si un index primaire est définit sur cette colonne. Voici la méthode de construction :

```
CREATE  XML INDEX nom_index_secondaire ON nom_table(Colonne)
USING XML INDEX nom_index_primaire FOR (PATH|PROPERTY|VALUE)
```

Vous l'avez compris avec la structure générale, la création de l'index secondaire nécessite le référencement de l'index primaire. Il est alors possible de choisir les trois options définis auparavent :

- PATH : Ce genre de clause améliore sensiblement les performances d'une requête contenant une clause exist().
- PROPERTY : Améliore les performances d'une requête utilisant la méthode value().
- VALUE : Améliore aussi sensiblement la méthode exist().

6 Conclusion

Le type de données XML est de plus en plus présent dans le cadre de l'entreprise, et dans le cas général dans le cadre de l'information, en particulier lors de l'interaction d'applications clientes avec la base. Toutes les informations présentées dans ce chapitre vous permettrons de transformer des données relationnelles en données XML et l'inverse, aussi bien qu'indexer une colonne de type XML.