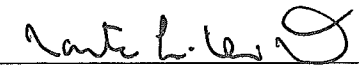
XVERTER: ARMAZENAMENTO E CONSULTA DE DADOS XML EM SGBDs

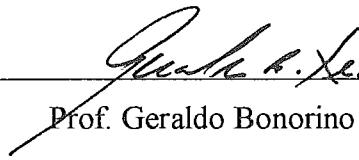
Humberto José Vieira Junior

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

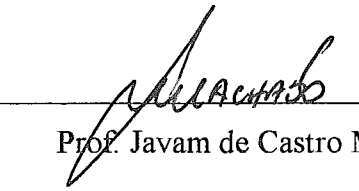
Aprovada por:



Profª. Marta Lima Queirós Mattoso, D.Sc.



Prof. Geraldo Bonorino Xexéo, D.Sc.



Prof. Javam de Castro Machado, Dr.

RIO DE JANEIRO, RJ – BRASIL

DEZEMBRO DE 2002

VIEIRA JUNIOR, HUMBERTO JOSÉ

XVerter: Armazenamento e Consulta de Dados XML em SGBDs [Rio de Janeiro] 2002

XII, 92 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Armazenamento de Dados XML
2. Consulta de Dados XML

I. COPPE/UFRJ II. Título (série)

Agradecimentos

Aos meus pais que me deram o suporte necessário para que eu conseguisse chegar até aqui.

À Vivian por compreender os finais de semana perdidos para que este trabalho pudesse ser concluído.

Aos meus amigos que me incentivaram a entrar no mestrado e a concluir mais esta etapa da minha vida.

À Prof. Marta Mattoso por ter sido uma excelente orientadora, sempre dando novas idéias e ajudando em tudo que fosse possível. Também agradeço o grande incentivo dado para que eu ingressasse no mestrado.

À Gabriela Ruberg por ter me ajudado bastante na tese e nos artigos, sempre com ótimas sugestões.

A todos os meus professores da graduação e do mestrado, principalmente à Prof. Maria Luiza que me fez tomar gosto por Banco de Dados ao lecionar a disciplina Banco de Dados I na graduação. Foram eles também que me mostraram os caminhos para me tornar um bom profissional no futuro.

Ao Prof. Jano de Souza por oferecer oportunidade nos projetos Coppetec para que eu pudesse pôr em prática os conhecimentos adquiridos na Universidade.

Aos Profs. Geraldo Xexéo e Javam Machado por terem aceitado participar desta banca.

À Patrícia Leal por estar sempre disposta a ajudar em quaisquer circunstâncias.

Às secretárias do PESC, sempre atenciosas e prestativas.

À CAPES pelo apoio financeiro.

A todas as outras pessoas que de uma forma ou de outra contribuíram para a execução deste trabalho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

XVERTER: ARMAZENAMENTO E CONSULTA DE DADOS XML EM SGBDs

Humberto José Vieira Junior

Dezembro / 2002

Orientadora: Marta Lima Queirós Mattoso

Programa: Engenharia de Sistemas e Computação

Com a consolidação do padrão XML como a *lingua franca* para a troca de informações na *Web*, tornaram-se essenciais mecanismos para consulta sobre bases de dados XML. Por esta razão, a W3C especificou uma linguagem padrão de consulta para dados XML, a XQuery. Estes dados XML podem estar armazenados nas mais diversas fontes de dados, desde as mais simples, como um sistema de arquivos, até chegar às mais sofisticadas, como os SGBDs. Estes SGBDs, por sua vez, podem ser nativos, relacionais ou baseados em objetos. Por este motivo, técnicas de armazenamento de dados XML em SGBDs vêm sendo amplamente estudadas na literatura com o objetivo de aproveitar o potencial de uma tecnologia bem estabelecida, como a dos SGBDs relacionais e baseados em objetos. Desta forma, é possível recuperar estes dados XML através da linguagem de consulta do SGBD utilizado. Todavia, esta não é uma boa solução, já que obriga o usuário a conhecer o esquema de representação dos documentos XML no SGBD e mais uma linguagem de consulta, além da XQuery.

O objetivo deste trabalho é propor uma solução para a realização de consultas XQuery sobre uma base de dados XML armazenada em um SGBDOR. São propostas regras de tradução automática da XQuery sobre o documento XML para a SQL3. O documento XML é armazenado através do formato padrão de representação DOM. As regras são implementadas em um arquivo XSL. Desta maneira, pudemos utilizar o XSLT, que é um outro padrão, para transformar uma representação XML intermediária da XQuery de entrada na SQL3 correspondente de saída.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

XVERTER: STORING AND QUERYING XML DATA WITH DBMS

Humberto José Vieira Junior

December / 2002

Advisor: Marta Lima Queirós Mattoso

Department: Computer and Systems Engineering

As the standard XML became the *lingua franca* for information exchange in the Web, querying mechanims for XML databases turned out to be necessary. Because of that, W3C has specified a standard query language for XML data, the XQuery language. These XML data may be stored in distinct data sources, from a file system to a sofisticated DBMS. These DBMS can be native, relational or object-based. Therefore, storage techniques are being widely studied in the literature to take advantage of a well established technology, such as the relational and object-based DBMS. So, it is possible to access these XML data through the DBMS query language. However, it is not a good solution, since the user needs to know the representation schema of the XML documents in the DBMS and an additional query language, besides the XQuery language.

The main purpose of this work is to propose a solution that will enable querying the XQuery language over an XML database stored into an ORDBMS. Rules of automatic translation from XQuery over an XML document to the SQL3 language are proposed. The XML document is stored through the use of standard DOM specification as DOM classes. These rules are implemented in an XSL archive. So, we were able to use XSLT, another standard, to translate an intermediate XML representation of the initial XQuery to its correspondent SQL3 query.

Índice:

Índice de Figuras

Índice de Tabelas

Capítulo 1 - Introdução

1.1 - Motivação

Com a consolidação do padrão XML (E*xtensible Markup Language*) como a *lingua franca* para a troca de informações na *Web*, tornaram-se essenciais mecanismos para consulta sobre bases de dados XML. Com o objetivo de solucionar este problema, o W3C XML *Query Working Group* (WORLD WIDE WEB CONSORTIUM (W3C), 2002b) propôs a XQuery (WORLD WIDE WEB CONSORTIUM (W3C), 2002c), uma linguagem de consulta para dados XML derivada principalmente da linguagem *Quilt* (CHAMBERLIN, ROBIE et al., 2000), mas que herda características de outras linguagens. Através da linguagem XQuery, os documentos XML podem ser consultados de forma declarativa, sendo necessário, para isso, um processador de consultas que, ao executá-las, navegue sobre os elementos dos documentos XML.

Estes documentos XML podem estar armazenados nas mais diversas fontes de dados, desde as mais simples, como um sistema de arquivos, até chegar às mais sofisticadas, como os Sistemas Gerenciadores de Bases de Dados (SGBDs). Estes SGBDs, por sua vez, podem ser nativos, relacionais ou baseados em objetos. Existem inúmeros trabalhos sobre estas estratégias para armazenamento de documentos XML (FEGARAS & ELMASRI, 2001; FLORESCU & KOSSMAN, 1999; MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001; RUNAPONGSA & PATEL, 2002; SHANMUGASUNDARAM, SHEKITA et al., 2001; SHANMUGASUNDARAM, TUFTE et al., 1999; TATARINOV, VIGLAS et al., 2002; TIAN, DEWITT et al., 2002). Estes trabalhos são favoráveis ao armazenamento de documentos XML em SGBDs tradicionais, devido às vantagens obtidas pelo armazenamento de dados estruturados legados e dados XML em um único sistema, aliado ao aproveitamento das funcionalidades típicas de um SGBD. Todavia, podemos identificar vários problemas nas propostas apresentadas para o armazenamento de documentos XML em SGBDs. Primeiramente, a grande maioria dessas propostas utiliza o modelo de dados relacional (FLORESCU & KOSSMAN, 1999; MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001; SHANMUGASUNDARAM, SHEKITA et al., 2001; SHANMUGASUNDARAM, TUFTE et al., 1999; TATARINOV, VIGLAS et al., 2002), que oferece poucos recursos

para representar características próprias dos dados XML, como relacionamentos aninhados e ordenação das estruturas. Em segundo lugar, os esquemas de representação de documentos XML não são genéricos, com exceção de (MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001), tornando pouco provável o conhecimento prévio do esquema criado no SGBD para armazenar os documentos XML e dificultando o acesso declarativo aos dados. Como outra opção, temos o modelo de dados orientado a objetos (OO) (FEGARAS & ELMASRI, 2001; TIAN, DEWITT et al., 2002), cuja semântica é muito semelhante à do modelo de dados XML. As abordagens propostas por (FEGARAS & ELMASRI, 2001) apresentam um esquema proprietário, ou seja, criado pelos autores, assim como a abordagem baseada em objetos proposta em (TIAN, DEWITT et al., 2002). Portanto, da mesma maneira que em algumas abordagens relacionais, é necessário um prévio conhecimento das estruturas criadas para possibilitar um acesso aos dados. Além destes, há um trabalho que armazena os dados XML em um SGBDOR, porém também é utilizado um esquema proprietário para realizar este armazenamento (RUNAPONGSA & PATEL, 2002). Uma saída para estes problemas é a utilização de um esquema genérico, ou seja, que sempre possui as mesmas estruturas, independente do documento XML armazenado. Na literatura, existem alguns trabalhos que propõem esta abordagem, como (MANOLESCU, FLORESCU et al., 2001) e (MANOLESCU, FLORESCU et al., 2000) para os modelos relacionais e uma das propostas de (FEGARAS & ELMASRI, 2001), que utiliza um modelo de dados baseado em objetos.

Uma vez armazenados estes dados XML nos SGBDs, os usuários são capazes de manipulá-los, independente da abordagem escolhida. Entretanto, para ter acesso aos dados, é necessário que se conheça o esquema, ou seja, as estruturas que foram criadas no banco. Obviamente esta não é uma boa alternativa, já que obriga o usuário a conhecer previamente as estruturas internas criadas para manipular os dados XML. Portanto, para que o usuário não seja obrigado a obter este conhecimento, é necessário que haja algum mecanismo transparente que transforme uma determinada consulta sobre os dados XML, fornecida pelo usuário, na linguagem-alvo de consulta do SGBD utilizado. Esta solução é a ideal para o usuário já que ele precisa apenas conhecer a linguagem de consulta do XML para poder manipular os dados, além é claro do conhecimento sobre os documentos a serem consultados. E como já existe uma linguagem padrão de consulta do XML, que é a XQuery, e o usuário tem familiaridade

com esta linguagem, já que trabalha com XML, não é preciso ter nenhum conhecimento adicional para conseguir recuperar as informações desejadas.

A execução transparente de consultas escritas na linguagem XQuery em um SGBD tradicional (que não é nativo de XML) requer um processo automático de tradução para a linguagem-alvo do SGBD hospedeiro. Esta tarefa não é trivial, pois é necessário que o tradutor conheça as estruturas do esquema utilizado de representação para armazenar os documentos XML no SGBD. Assim, a escolha do esquema de representação num SGBD é fundamental para o sucesso da execução de consultas XQuery. Encontramos na literatura poucos trabalhos que fazem a tradução da linguagem de consulta do XML para a linguagem de consulta do SGBD. Como exemplos, podemos citar (MANOLESCU, FLORESCU et al., 2001) que utiliza o modelo relacional e (FEGARAS & ELMASRI, 2001) que usa o modelo de dados baseado em objetos. Além deles, há o XPERANTO (CAREY, FLORESCU et al., 2000; CAREY, KIERNAN et al., 2002) que não é um tradutor especial, já que seu objetivo é realizar uma publicação de dados armazenados em um SGBDOR no formato XML. Assim, XPERANTO traduz consultas XML-QL, que é uma linguagem de consulta para XML, sobre visões XML de dados originalmente estruturados e já integrantes do esquema do SGBD de armazenamento. Desta forma, a tradução na verdade é um retorno ao mapeamento realizado previamente de dados estruturados para XML. O trabalho apresentado em (SHANMUGASUNDARAM, SHEKITA et al., 2000) apresenta o mesmo enfoque do XPERANTO, porém publica os dados armazenados em SGBDRs. A adoção do modelo de dados relacional torna essa tradução complexa e muito ineficiente para consultas recursivas, pois o mapeamento resulta em comandos SQL (*Structured Query Language*) verborrágicos contendo um número excessivo de junções. Isto pode ser constatado no trabalho de (MANOLESCU, FLORESCU et al., 2001), que realiza uma tradução da XQuery para a SQL. O número elevado de junções é uma conseqüência da disparidade semântica existente entre os modelos relacional e XML, o que torna ainda mais complexo o processo de tradução. Por outro lado, o modelo de dados orientado a objetos oferece maior semântica para a representação de relacionamentos e algoritmos específicos para a execução de consultas contendo expressões de caminho. *Fegaras* e *Elmasri* (FEGARAS & ELMASRI, 2001) apresentam uma nova linguagem de consulta XML, dita XML-OQL, e a partir dela propõem duas alternativas para tradução desta linguagem para a OQL (*Object Query Language*). A primeira solução realiza a tradução sem ter conhecimento do esquema

dos documentos XML, utilizando uma representação genérica, ao contrário da segunda alternativa apresentada, que usa uma representação não genérica dos dados. Apesar de usar um mapeamento mais direto que no modelo relacional, em (FEGARAS & ELMASRI, 2001) não é adotada a linguagem padrão XQuery, o que dificulta o processo de tradução. Além disso, na segunda abordagem é utilizado um esquema não genérico para o armazenamento dos dados XML e é apresentada uma álgebra bastante complexa, o que torna pouco viável a implementação do respectivo tradutor. O problema destes tradutores está em usar uma representação proprietária. Além disso, o trabalho de (MANOLESCU, FLORESCU et al., 2001) utiliza o modelo de dados relacional e as abordagens apresentadas em (FEGARAS & ELMASRI, 2001) não utilizam padrões.

1.2 - Objetivos

O objetivo do nosso trabalho é propor uma arquitetura para o armazenamento de documentos XML em um SGBD objeto-relacional (SGBDOR) e para a execução transparente de consultas escritas na linguagem XQuery. O cerne da nossa proposta é o tradutor *XVerter*, o qual utiliza um conjunto de regras de transformação para converter consultas XQuery sobre documentos XML em comandos da linguagem SQL3 sobre as classes do DOM. O tradutor *XVerter* é baseado na criação do esquema do DOM (*Document Object Model*) (WORLD WIDE WEB CONSORTIUM (W3C), 2002a) em um SGBDOR. Assim, quando o documento XML é estruturado em classes, obtém-se um mapeamento correto entre a estrutura em árvore para a estrutura de classes. Como conseqüência, a linguagem SQL3 pode ser usada de modo equivalente à XQuery, através da correspondência entre a estrutura em árvore do documento XML e a estrutura de classes do DOM. Uma vantagem considerável da nossa abordagem é que o DOM consiste em um padrão amplamente utilizado no universo XML para a manipulação de documentos em memória principal por linguagens de programação, como *Java*, além de preservar as características originais do documento, como a ordenação das estruturas. Uma outra vantagem do DOM é que ele possui um formato parecido com a abordagem baseada em objetos proposta em (TIAN, DEWITT et al., 2002) e que apresenta um bom desempenho na execução das consultas. Vale ressaltar também que não encontramos trabalhos sobre a implementação do formato DOM sobre classes de um SGBD.

De acordo com a arquitetura proposta para o *XVerter*, a sua primeira etapa consiste em transformar a consulta XQuery de entrada em uma representação intermediária que seja equivalente à consulta recebida. A saída desta primeira etapa será uma representação XML para possibilitar a utilização do XSLT (*Extensible Stylesheet Language Transformations*) (WORLD WIDE WEB CONSORTIUM (W3C), 2002d), que é um outro padrão proposto pelo W3C. Desta forma, representamos nossas regras de transformação através do XSL e utilizamos o XSLT para transformar a representação XML intermediária na SQL3 de saída. Com isto, grande parte da solução fica dentro do universo XML. Portanto, estamos propondo uma solução automática e transparente para tradução de consultas XQuery em consultas SQL3 utilizando o formato DOM de armazenamento em SGBDORs.

Nossa proposta foi validada experimentalmente no GOA (MATTOSO, 2000), um protótipo de SGBD orientado a objetos (SGBDOO) desenvolvido na COPPE/UFRJ, através da tradução de diversas consultas XQuery que exploram características definidas no *benchmark* (SCHMIDT, WAAS et al., 2002) , como as expressões *Flower*, as expressões de caminho e o operador '//'.

Maiores detalhes quanto ao armazenamento de documentos XML no GOA são encontrados em (MATTOSO, CAVALCANTI et al., 2002), que foi um artigo apresentado na Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software (SBES'2002). Já um resumo do *XVerter*, que é o tema desta tese, foi publicado no Simpósio Brasileiro de Banco de Dados (SBBD'2002) (VIEIRA, RUBERG et al., 2002).

1.3 - Organização

O restante desta tese está organizado da seguinte forma: o Capítulo 2 analisa diversas propostas de armazenamento de um documento XML em SGBDs, incluindo o formato DOM, que é utilizado na nossa arquitetura. Este capítulo de revisão da literatura não inclui uma introdução sobre XML, XQuery nem XSLT, pois este estudo foi realizado anteriormente e se encontra em dois relatórios técnicos: (VIEIRA, GONÇALVES et al., 2002b) e (VIEIRA, GONÇALVES et al., 2002a). No Capítulo 3, detalhamos o conjunto de regras de transformação que constituem o *XVerter* além de apresentarmos alguns exemplos de tradução da XQuery para o SQL3. O Capítulo 4

apresenta a arquitetura proposta e a implementação que fizemos do *XVerter*, incluindo o armazenamento, geração da representação XML intermediária e a transformação utilizando o XSLT. O Capítulo 5 apresenta o *benchmark* utilizado para a validação da nossa proposta e as características nele presentes. Além disso, discute quais destas características são suportadas pelo *XVerter* e por dois processadores de consulta XQuery: Quip (SOFTWARE AG, 2002b) e XQuery Demo (MICROSOFT .NET HOME, 2002). Finalmente, o Capítulo 6 conclui esta dissertação, apresentando as contribuições e os trabalhos futuros.

Capítulo 2 - Estratégias de Armazenamento de Documentos XML

A estratégia mais simples para o armazenamento de documentos XML consiste na geração e manutenção de arquivos de texto no sistema operacional, respeitando o formato original de cada documento. Esta alternativa apresenta inúmeras desvantagens quanto à flexibilidade e ao desempenho em consultas elaboradas aos dados. Por isso, é recomendável que bases de documentos XML sejam beneficiadas pelas funcionalidades disponíveis em sistemas de bancos de dados (MCHUGH, 2000).

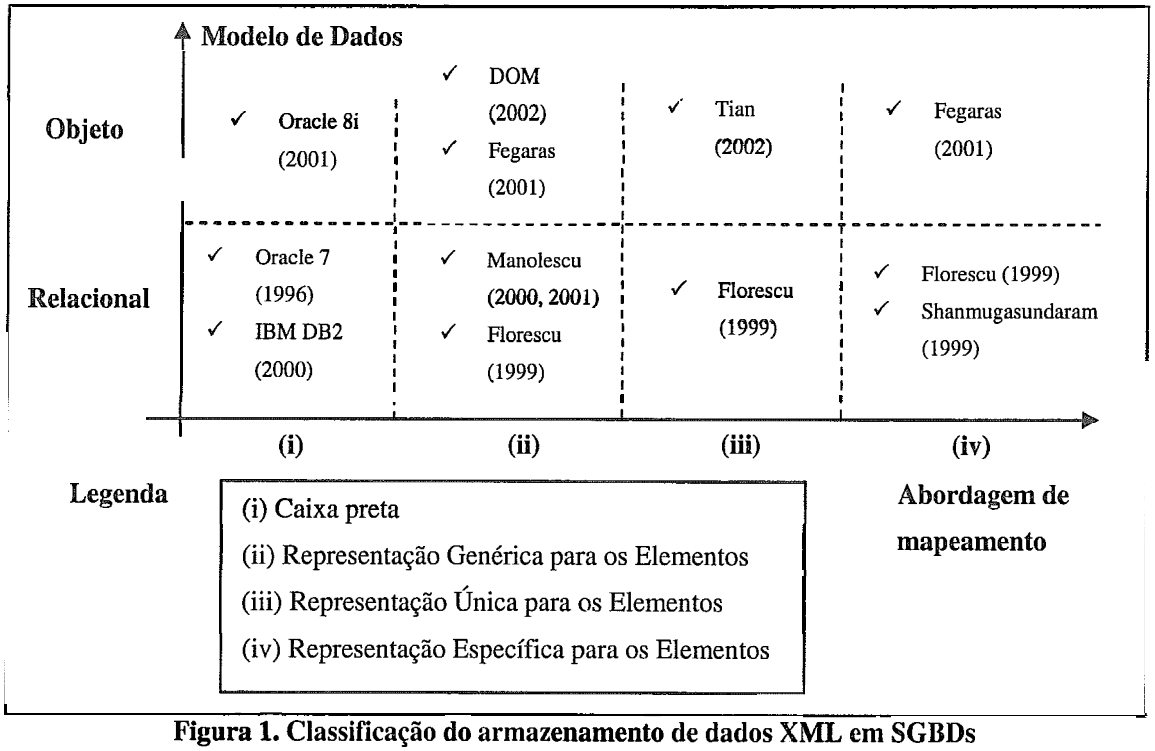


Figura 1. Classificação do armazenamento de dados XML em SGBDs

Na literatura, existem inúmeras propostas para realizar o mapeamento de documentos XML em SGBDs. Em alguns destes trabalhos, é proposto um mapeamento para os SGBDs relacionais (SGBDRs) (FLORESCU & KOSSMAN, 1999; MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001; SHANMUGASUNDARAM, SHEKITA et al., 2001; SHANMUGASUNDARAM, TUFTE et al., 1999; TATARINOV, VIGLAS et al., 2002); em outros, são apresentadas abordagens de mapeamento para os SGBDs baseados em Objetos (FEGARAS & ELMASRI, 2001; TIAN, DEWITT et al., 2002); e em outros, são utilizados os

SGBDORs (RUNAPONGSA & PATEL, 2002). Uma outra alternativa existente é armazenar os documentos XML em SGBDs Nativos (SOFTWARE AG, 2002a). Neste caso, não há necessidade de mapeamento, já que estes SGBDs surgiram especialmente para realizar este tipo de armazenamento.

Por serem propostas recentes, dos últimos três anos, a terminologia é muito variada tornando difícil a análise comparativa entre eles. Propomos aqui uma classificação destas propostas levando em consideração o modelo de representação do SGBD e a abordagem de mapeamento da árvore XML para a estrutura de dados do SGBD. Analisando estes trabalhos, conseguimos identificar quatro diferentes classificações para as abordagens de mapeamento existentes: (i) caixa preta; (ii) representação genérica para os elementos; (iii) representação única para os elementos; (iv) representação específica para os elementos. Para cada uma destas quatro abordagens, podemos utilizar os SGBDRs ou os SGBDs baseados em Objetos. A Figura 1 apresenta algumas das alternativas existentes na literatura considerando estes dois aspectos.

2.1 - Abordagens para Mapeamentos de Documentos XML em SGBDs

Um documento XML é geralmente representado por uma árvore rotulada, onde os nós são os elementos e os arcos, os relacionamentos entre os elementos e sub-elementos ou atributos. Identificamos quatro classes de mapeamento de uma árvore XML para as estruturas de dados do SGBD. Nas próximas seções detalhamos estas classes, que serão denominadas por:

(i) caixa preta;

(ii) representação genérica para os elementos;

(iii) representação única para os elementos;

(iv) representação específica para os elementos.

2.1.1 - Mapeamento Caixa Preta

Esta alternativa é a mais simples, consistindo no armazenamento do documento XML no SGBD sem nenhum tipo de mapeamento, ou seja, como uma caixa preta. Esta abordagem é útil quando o objetivo é apenas utilizar o SGBD como um mero

repositório de dados. Uma outra opção é utilizar arquivos texto para armazenar os documentos XML sem realizar nenhum tipo de mapeamento, ou seja, os documentos XML originais são exatamente iguais às suas representações em um sistema de arquivos.

Na literatura, podemos identificar a utilização de tipos especiais nos SGBDs, como os CLOBs, para realizar o mapeamento Caixa Preta. O uso destes tipos torna esta abordagem semelhante em muitos aspectos ao armazenamento de arquivos de texto no sistema operacional, portanto não oferecendo muitas vantagens devido à falta de flexibilidade. Apresentamos rapidamente dois exemplos do uso de tipos especiais, sendo uma abordagem relacional e outra baseada em Objetos.

2.1.1.1 - Oracle

O SGBD comercial Oracle (ORACLE, 2002) inicialmente previa o armazenamento dos documentos XML em colunas do tipo CLOB, sem nenhum recurso adicional para consultas sobre os dados armazenados. Uma evolução desta solução foi a criação de um tipo especial de coluna para armazenar estes documentos, denominada XMLType, que pode ser encontrada no Oracle 9i. Ao executar consultas que utilizam este tipo de coluna, o processador utilizará alguns recursos oferecidos por este uso, tais como o *parser* XML e as funções embutidas. Um filtro sobre uma expressão de caminho é um exemplo deste tipo de função.

2.1.1.2 – DB2

O SGBD IBM DB2 (CHAUDHURI & SHIM, 2001; CHENG & XU, 2000) possui três diferentes tipos de coluna para armazenar documentos XML: XML CLOB, para grandes documentos XML; XMLVARCHAR, para pequenos documentos XML; e XMLFile, que armazena o documento XML como um arquivo em um sistema local de arquivos.

2.1.2 - Mapeamento com Representação Genérica para os Elementos

Este mapeamento adota um esquema genérico para representar os elementos dos documentos XML, o qual independe da estrutura do documento armazenado. Nesta abordagem existe um mapeamento indireto de elementos dos documentos XML para

estruturas genéricas que indiretamente armazenam estes elementos. Dizemos que o mapeamento é direto quando cada elemento corresponde a uma tabela ou classe no SGBD, ao contrário do que ocorre no mapeamento indireto. Nesta representação genérica não é gerado um número alto de estruturas de dados e não há conflito de nomes de elementos XML. Entretanto, um grande problema acontece no momento em que o usuário tem necessidade de consultar os dados. Como não foi utilizado um mapeamento direto dos elementos, o usuário é obrigado a conhecer este esquema genérico de armazenamento para que, através dele, tenha acesso aos elementos dos documentos XML. Os trabalhos que adotam essa abordagem de mapeamento em geral definem um esquema proprietário para a representação dos dados (FEGARAS & ELMASRI, 2001; MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001) muitas vezes dependente de recursos do SGBD hospedeiro. Com a utilização de um esquema proprietário, o conhecimento prévio das estruturas utilizadas por parte do usuário se torna ainda menos provável.

Identificamos na literatura algumas abordagens genéricas para representar os documentos XML em SGBDRs (FLORESCU & KOSSMAN, 1999; MANOLESCU, FLORESCU et al., 2000; MANOLESCU, FLORESCU et al., 2001). Para os SGBDs baseados em objetos, além do formato padrão DOM para manipulação de documentos XML, identificamos uma outra proposta que é apresentada em (FEGARAS & ELMASRI, 2001).

2.1.2.1 – A abordagem de Manolescu e Florescu

Na abordagem genérica proposta em (MANOLESCU, FLORESCU et al., 2000), são criadas as tabelas presentes na Figura 2. Deste modo, o conjunto de tabelas desta abordagem independe do documento XML armazenado, evitando a criação de um número excessivo de tabelas caso o documento XML possua uma enorme diversidade de nomes de elementos. Existe um outro trabalho semelhante a este (MANOLESCU, FLORESCU et al., 2001) que, apesar de apresentar algumas diferenças em relação à representação genérica da Figura 2, possui a mesma idéia principal. Neste trabalho de 2001, algumas tabelas foram excluídas (`Tag`, `ElemContent`, `ElemAttribute`, `ElemDoc`, `Word` e `Contains`) e outras inseridas (`URI`, `ProcInstr`, `QName`, `NameSpace`, `Comment`, `Child` e `TransClosure`).

Document	(docID, docURL)	Element	(elID, tagID)
Value	(valID, value)	ElemContent	(parentID, childID, valID, index)
Tag	(tagID, valID)	ElemAttribute	(elID, attID, valID)
Attribute	(attID,valID, type,isRequired)	ElemDoc	(elemID, docID)
Word	(wordID, word)	Contains	(elID, wordID, depth, tag)

Figura 2. Esquema do formato Genérico

2.1.2.2 – A abordagem de Florescu e Kossman (*Edge* com tabelas separadas)

Esta abordagem é apresentada em (FLORESCU & KOSSMAN, 1999), que foi um dos trabalhos pioneiros neste importante problema de armazenamento de documentos XML. No trabalho mencionado, os autores assumem que os documentos XML podem ser representados por um grafo direcionado e ordenado, onde cada elemento XML é representado por um nó nomeado com o seu OID. Assume-se que cada elemento XML possui um identificador, mas caso não tenha, o sistema automaticamente gerará um. Os relacionamentos entre elementos e sub-elementos são representados através de arestas no grafo e são nomeados com o nome do sub-elemento. Já os valores do documento são representados como folhas do grafo. Este trabalho não diferencia sub-elementos de atributos, portanto o documento original não pode ser reconstruído a partir dos dados relacionais.

```
<!DOCTYPE Dept SYSTEM "Dept.dtd">
<Dept dept_id="dept1">
  <Student student_id="123">
    <Name>St1</Name>
    <Enroll>CS10</Enroll>
    <Enroll>CS20</Enroll>
  </Student>
  <Student student_id="124">
    <Name>St2</Name>
  </Student>
</Dept>
```

Figura 3. Documento XML de exemplo

O formato *Edge* é uma das três formas apresentadas em (FLORESCU & KOSSMAN, 1999) de realizar o mapeamento das arestas e consiste em armazená-las em uma única tabela, onde cada tupla representa um arco do grafo do documento. Para cada uma destas propostas, existem duas formas de armazenar os valores dos documentos XML. A primeira consiste em armazenar estes valores em tabelas separadas enquanto a outra os armazena junto com as arestas (*Inlining*). O mapeamento com representação genérica para os elementos corresponde ao formato *Edge* com tabelas separadas.

A Figura 4 apresenta um exemplo de mapeamento para esta abordagem considerando o documento XML da Figura 3. Cada tabela separada corresponde a um

tipo de dados utilizado no documento XML, que pode ser inteiro, *string*, data, ... Cada uma destas tabelas possui dois campos, sendo que o primeiro corresponde a um identificador do elemento ou atributo e o segundo armazena o seu valor.

Edge

SourceID	Ordinal	Tag	Flag	TargetID
1	1	Dept	ref	2
2	0	dept_id	string	v1
2	1	Student	ref	3
2	2	Student	ref	4
3	0	student_id	int	v2
3	1	Name	string	v3
3	2	Enroll	string	v4
3	3	Enroll	string	v5
4	0	student_id	int	v6
4	1	Name	string	v7

Vint

vID	Value
v1	dept1
v3	St1
v4	CS10
v5	CS20
v7	St2

Vstring

vID	Value
v2	123
v6	124

Figura 4. Exemplo de mapeamento *Edge* com tabelas separadas

2.1.2.3 – A abordagem DOM

O W3C definiu o DOM como o formato padrão para manipulação de documentos XML. O esquema completo das classes do DOM nível 2 é visualizado na Figura 5 enquanto a Tabela 1 apresenta a descrição destas classes.

Conforme pode ser visto na Figura 5, praticamente todas as classes do DOM herdam da classe `Node`. Esta classe, por sua vez, possui dois atributos, denominados `name` e `value`, que serão herdados por todas as outras classes. Além destes atributos, há um outro denominado `childrenNode`, que representa um auto-relacionamento da classe `Node`. Logo, as outras classes também herdarão este atributo. Desta forma, a navegação entre as classes do DOM é realizada através do atributo `childrenNode`.

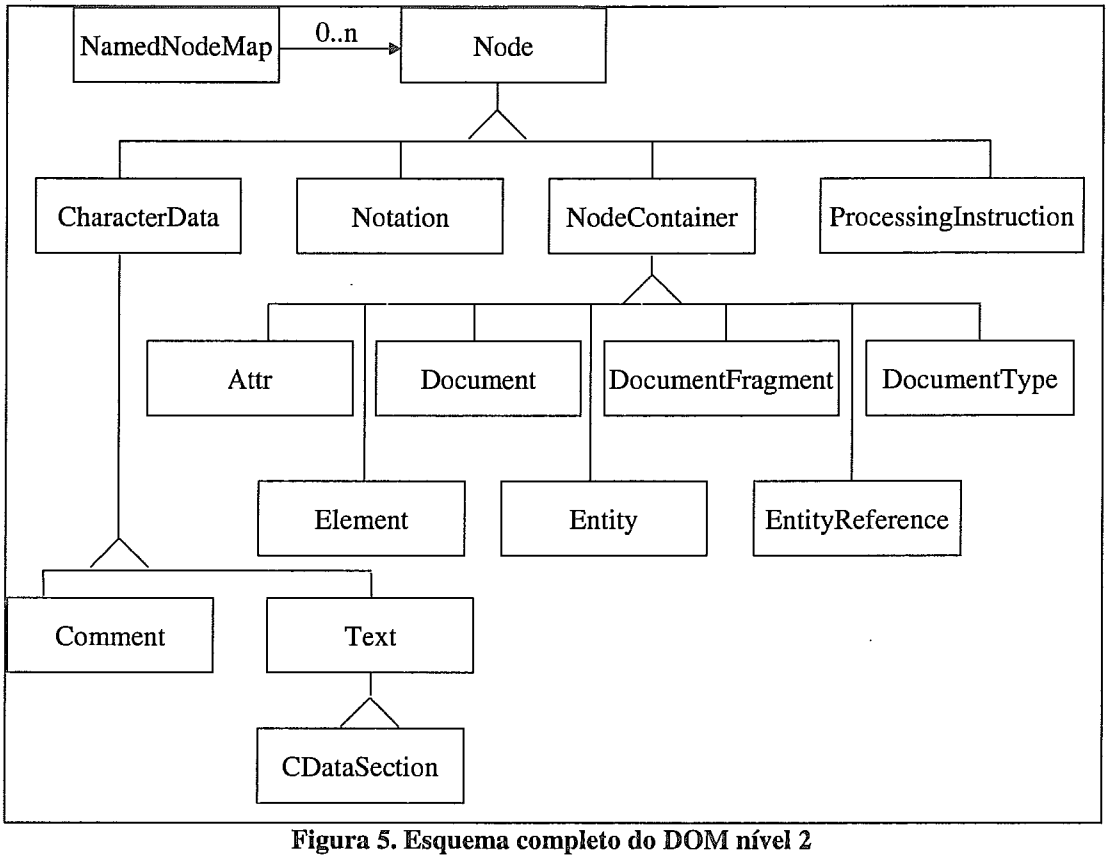


Figura 5. Esquema completo do DOM nível 2

Tabela 1. Descrição das classes do DOM

Classe	Descrição
Attr	Representa um atributo de um determinado elemento
CdataSection	Representa uma seção XML CData
CharacerData	Representa uma string genérica no DOM
Comment	Representa um comentário XML
Document	Representa um documento XML
DocumentFragment	Representa uma árvore de nós que geralmente não é uma árvore completa
DocumentType	Representa o DTD na árvore do documento
Element	Representa um elemento na árvore DOM
Entity	Representa uma entidade XML
EntityReference	Representa uma referência à uma entidade XML
NamedNodeMap	Contém uma coleção de nós que podem ser acessados pelo nome
Node	É a classe pai de todos os nós da árvore DOM
NodeContainer	Adiciona a capacidade de ter nós filhos
Notation	Representa uma notação XML
ProcessingInstruction	Representa o processamento de uma instrução XML
Text	Representa o dado textual do documento XML

2.1.2.4 – A abordagem de Fegaras e Elmasri

Em (FEGARAS & ELMASRI, 2001), o objetivo é transformar uma linguagem de consulta proposta por eles, a XML-OQL, em uma OQL que será executada em um SGBDOO. Os autores apresentam duas abordagens para realizar esta tradução: a

primeira armazena os dados de modo indireto, utilizando a abordagem genérica mostrada na Figura 1, e a segunda faz uma mapeamento direto, utilizando a representação específica apresentada na mesma figura. Nesta seção, entraremos em detalhes na primeira solução, já que é um exemplo de representação genérica para os elementos XML.

Conforme dito acima, os autores deste trabalho propõem uma extensão da OQL para lidar com dados XML, sendo chamada de XML-OQL. Maiores detalhes sobre esta linguagem podem ser encontrados na seção 3.3.2. Como o objetivo deste capítulo é apresentar diferentes abordagens de armazenamento de documentos XML, daremos ênfase a esta parte do trabalho proposto em (FEGARAS & ELMASRI, 2001).

```
enum attribute_kind {CDATA, IDref, ID,
IDrefs};
enum element_kind {TAG, PCDATA};

union attribute_type switch (attribute_kind)
{  case CDATA:     string           value;
   case IDref:     string           id_ref;
   case IDrefs:    list <string>    id_refs;
   case ID:        string           id;
}

struct attribute_binding
{  string          name;
   attribute_type  value; };
```

```
struct node_type {
  string                        name;
  list <attribute_binding>      attributes;
  list <XML_element>            content;
};

union element_type switch (element_kind)
{   case TAG:     node_type     tag;
    casePCDATA:   string        data;
};

class XML_element (extent Elements)
{attribute   element_type   element; };
```

Figura 6. Estruturas genéricas para o modelo de objetos

A Figura 6 apresenta as estruturas criadas ao utilizarmos esta abordagem. Conforme podemos perceber, os tipos de atributo podem ser `CDATA`, `IDref`, `ID` e `IDrefs` enquanto os tipos de elemento possíveis são `TAG` e `PCDATA`. Há somente uma classe de nome `XML_element`, que contém um atributo de nome `element` e tipo `element_type`. Este por sua vez é uma união entre elementos do tipo `TAG` ou `PCDATA`. No primeiro caso, o atributo adicionado à união (`tag`) é do tipo `node_type` enquanto no segundo, o atributo (`data`) é do tipo `string`. O atributo `node_type` pode ser uma `string` (`name`), um `list` `<attribute_binding>` (`attributes`) ou um `list` `<XML_element>` (`content`). Por sua vez, o `attribute_binding` pode ser uma `string` (`name`) ou um `attribute_type` (`value`). No último caso, ele é uma união dos quatro possíveis tipos de atributos: `CDATA`, `IDref`, `IDrefs` ou `ID`.

2.1.3 - Mapeamento com Representação Única para os Elementos XML

No caso do uso de uma única estrutura de dados para representar os elementos nomeados de um documento XML, a simplicidade da estrutura de representação possui como desvantagem a inflexibilidade para o armazenamento do esquema XML. Este mapeamento também pode ser considerado genérico e indireto, porém devido à rigidez da estrutura única, optamos por criar outra categoria de representação. Encontramos três abordagens de mapeamento na literatura que se encaixam na classificação de representação única para os elementos: *Edge* com *Inlining* e Universal com *Inlining* nos SGBDs Relacionais e uma abordagem baseada em Objetos. A palavra *Inlining*, conforme explicado na seção 2.1.2.2, significa que os valores dos elementos e atributos estão armazenados juntamente com as *tags*.

2.1.3.1 – A abordagem de Florescu e Kossman (*Edge* com *Inlining*)

Conforme explicado na seção 2.1.2.2, o formato *Edge* armazena as arestas do grafo correspondente ao documento XML em uma única tabela, onde cada tupla representa um arco do grafo. Além disso, os valores dos documentos XML podem ser armazenados de duas formas distintas. Na primeira, os valores são armazenados em tabelas separadas, conforme descrito na seção 2.1.2.2, e na segunda, estes valores são armazenados juntamente com as arestas, sendo esta abordagem denominada *Inlining*. O mapeamento com representação única para os elementos corresponde ao formato *Edge* com *Inlining*.

A Figura 7 apresenta um exemplo de mapeamento para esta abordagem considerando o documento XML da Figura 3. O desempenho da abordagem *Edge* com *Inlining* também foi analisado em (TIAN, DEWITT et al., 2002).

SourceID	Tag	Ordinal	TargetID	Data
1	Dept	1	2	NULL
2	dept_id	0	0	"dept1"
2	Student	1	3	NULL
2	Student	2	4	NULL
3	student_id	0	0	"123"
3	Name	1	0	"St1"
3	Enroll	2	0	"CS10"
3	Enroll	3	0	"CS20"
4	student_id	0	0	"124"
4	Name	1	0	"St2"

Figura 7. Exemplo de Mapeamento *Edge* com *Inlining*

2.1.3.2 – A abordagem de Florescu e Kossman (Universal com *Inlining*)

Esta abordagem também é apresentada em (FLORESCU & KOSSMAN, 1999) e seu conceito é melhor entendido a partir da definição de uma outra abordagem de mapeamento. Resumidamente, tabelas Binárias correspondem a uma fragmentação horizontal da tabela do formato *Edge*, onde cada tabela representa um *tag*. Já a tabela Universal corresponde a um *outer join* entre estas tabelas Binárias.

Da mesma forma que ocorre para o formato *Edge*, a abordagem de tabela Universal também apresenta duas formas de armazenar os valores dos documentos XML, em tabelas separadas ou junto com as arestas (*Inlining*). Aquela que possui uma representação única para os elementos é a Universal com *Inlining*. A Figura 8 apresenta mapeamento obtido com esta abordagem utilizando o documento XML da Figura 3.

source	$ord_{student}$	$targ_{student}$	ord_{name}	$targ_{name}$	ord_{enroll}	$targ_{enroll}$	...
1	1	2	null	null	null	null	...
1	2	3	null	null	null	null	...
2	null	null	1	St1	null	null	...
2	null	null	null	null	1	CS10	...
2	null	null	null	null	2	CS20	...
3	null	null	2	St2	null	null	...
...	...	...	...	...	...	...	...

Figura 8. Exemplo de Documento XML com Mapeamento Universal com *Inlining*

2.1.3.3 – A abordagem de Tian e DeWitt

Em (TIAN, DEWITT et al., 2002) é apresentada uma abordagem única para o armazenamento orientado a objetos. Para este mapeamento, é proposto o seguinte formato para os objetos:

`(offset, length, flag, tag, parent, prev, next, opt_child, opt_attr, opt_text)`

Offset	Record
0	Length = 40, Dept, parent = nil, prev = nil, next = nil, first_child = 40, last_child = 140, Attr(dept_id = "dept1")
40	Length = 40, Student, parent = 0, prev = nil, next = 140, first_child = 80, last_child = 120, Attr(student_id = "123")
80	Length = 20, Name, parent = 40, prev = nil, next = 100, no children, no attribute, #PCDATA = "St1"
100	Length = 20, Enroll, parent = 40, prev = 80, next = 120, no children, no attribute, #PCDATA = "CS10"
120	Length = 20, Enroll, parent = 40, prev = 100, next = nil, no children, no attribute, #PCDATA = "CS20"
140	Length = 40, Student, parent = 0, prev = 40, next = nil, first_child = 180, last_child = 180, Attr(student_id = "124")
180	Length = 20, Name, parent = 140, prev = nil, next = nil, no children, no attribute, #PCDATA = "St2"

Figura 9. Exemplo de Documento XML com Mapeamento para Objetos

O `offset` é utilizado como um identificador, enquanto o `length` armazena o tamanho total do objeto. O campo `flag` contém bits que indicam se o objeto possui os campos `opt_child`, `opt_attr` ou `opt_text`. O `tag`, como o nome sugere, armazena o nome do elemento XML. O atributo `parent` guarda o OID (`offset`) do nó pai enquanto o `opt_child` contém os OIDs do primeiro e último filhos, desde que o objeto tenha filhos. Os campos `prev` e `next` possuem o OID do irmão anterior e o próximo. Portanto, uma lista duplamente encadeada é representada por estes atributos. O campo `opt_attr` armazena o par (`name`, `value`) de cada atributo do elemento XML. E por fim, o atributo `opt_text` contém o texto do elemento XML. A Figura 9 apresenta o mapeamento correspondente a esta abordagem, utilizando o documento XML da Figura 3.

2.1.4 - Mapeamento com Representação Específica para os Elementos XML

Um esquema de representação mais flexível é o enfoque do mapeamento com representação específica para os elementos, onde cada elemento XML possui uma estrutura própria no SGBD, tal que os atributos de um elemento são os atributos da estrutura equivalente. Este é um mapeamento quase que direto entre os elementos XML e as suas estruturas correspondentes no SGBD, sejam elas tabelas ou classes. Entretanto, é necessário um atributo adicional para representar o conteúdo (texto) do elemento XML, além de atributos de referência para os relacionamentos entre os elementos. Essa abordagem otimiza o acesso a elementos de um mesmo tipo, mas dificulta bastante a reconstrução do documento XML original, pois pode gerar um número alto de estruturas no SGBD (contendo poucas instâncias), além de requerer razoável conhecimento sobre as estruturas de armazenamento do documento XML para a elaboração de consultas. Além disso, o armazenamento adequado de elementos com tipos distintos (em níveis e documentos diferentes e com atributos próprios) mas com o mesmo nome é um problema adicional deste mapeamento.

Observe que tanto na representação específica quanto na representação única (seção 2.1.3), é difícil representar certas características importantes dos dados XML, como a ordem em que os elementos aparecem no documento.

Na literatura, existem duas abordagens de mapeamento que podemos classificar como uma representação específica para os elementos XML em SGBDRs: Binário (ou

Atributo) e DTD. Um dos mapeamentos baseados em objetos apresentados em (FEGARAS & ELMASRI, 2001) também é um exemplo de representação específica para os elementos. Além destas abordagens, também identificamos um mapeamento para SGBDORs que utiliza esta representação específica para os elementos (RUNAPONGSA & PATEL, 2002).

2.1.4.1 – A abordagem de Florescu e Kossman (Binário ou Atributo)

Esta abordagem, também apresentada em (FLORESCU & KOSSMAN, 1999), corresponde a uma fragmentação horizontal das tuplas que correspondem às arestas da tabela do formato *Edge* (apresentado na seção 2.1.2.2). Nesta abordagem, cada tabela representa um *tag* e, portanto, todas as arestas com o mesmo nome irão pertencer à mesma tabela.

B_{name}

source	ord	val_{int}	val_{string}	target
3	1	null	St1	null
4	1	null	St2	null

B_{enroll}

source	ord	val_{int}	val_{string}	target
3	2	null	CS10	null
3	3	null	CS20	null

$B_{student}$

source	ord	val_{int}	val_{string}	target
2	1	null	null	3
2	2	null	null	4

B_{dept}

source	ord	val_{int}	val_{string}	target
1	1	null	null	2

$B_{student\_id}$

source	ord	val_{int}	val_{string}	target
3	0	123	null	null
4	0	124	null	null

B_{dept_id}

source	ord	val_{int}	val_{string}	target
1	0	null	dept1	null

Figura 10. Exemplo de Mapeamento Binário com tabelas separadas

Da mesma forma que ocorre para os formatos *Edge* e Universal, o mapeamento Binário também apresenta duas formas de armazenar os valores dos documentos XML, em tabelas separadas ou junto com as arestas (*Inlining*). Tanto o Binário com tabelas separadas quanto o Binário com *Inlining* são exemplos de mapeamento com representação específica para os elementos. A Figura 10 apresenta um exemplo de mapeamento Binário com tabelas separadas considerando o documento XML da Figura 3. Conforme pode ser observado na figura, cada elemento do documento passa a possuir uma tabela específica para armazenar os seus dados. Neste exemplo, são criadas seis tabelas que são nomeadas concatenando a letra "B" (Binário) com o nome do elemento que aparece no documento. O `source` e o `target` servem para fazer o

relacionamentos entre as tabelas e o `ord` serve para armazenar a ordem de um elemento em determinado nó.

A abordagem Atributo citada em (TIAN, DEWITT et al., 2002) corresponde ao formato Binário de (FLORESCU & KOSSMAN, 1999). (CHAUDHURI & SHIM, 2001) apresenta outro exemplo de mapeamento para esta abordagem de armazenamento.

2.1.4.2 – A abordagem de Shanmugasundaram e Tufte (DTD)

No formato DTD, proposto em (SHANMUGASUNDARAM, TUFTE et al., 1999) e citado em (TIAN, DEWITT et al., 2002), são criadas tabelas de acordo com os relacionamentos existentes entre os elementos no DTD. Cada tupla em uma tabela possui um identificador e uma coluna para identificar o seu elemento pai. Um elemento XML que só pode aparecer uma vez no elemento pai deve ser adicionado como uma coluna na tabela que representa o seu pai. Já para os elementos que aparecem mais de uma vez no elemento pai, é criada uma outra tabela. Caso o DTD contenha ciclos, uma tabela separada é criada para quebrar este ciclo. A Figura 12 apresenta o mapeamento para esta abordagem considerando o DTD da Figura 11 que corresponde ao documento XML da Figura 3.

```
<?xml?>
<!ELEMENT Dept (Student*)>
<!ATTLIST Dept dept_id ID #REQUIRED>
<!ELEMENT Student (Name, Enroll*)>
<!ATTLIST Student student_id ID #REQUIRED>
<!ELEMENT Name #PCDATA>
<!ELEMENT Enroll #PCDATA>
```

Figura 11. DTD correspondente ao documento XML da Figura 3

Dept

ParentID	ID	Dept_id
1	2	"dept1"

Enroll

ParentID	ID	TEXT
3	5	"CS10"
3	6	"CS20"

Student

ParentID	ID	Student_id	Name
2	3	"123"	"St1"
2	4	"124"	"St2"

Figura 12. Exemplo de Mapeamento DTD

2.1.4.3 – A abordagem de Fegaras e Elmasri

Conforme explicado na seção 2.1.2.4, o trabalho proposto por (FEGARAS & ELMASRI, 2001) apresenta duas abordagens para traduzir uma linguagem proposta por eles, a XML-OQL, em uma OQL. Na primeira, os dados são armazenados sem ter o

conhecimento do esquema, ao contrário da segunda abordagem. Nesta seção, o enfoque será sobre a segunda solução, já que é um exemplo de representação específica para os elementos XML. Não entraremos em detalhes sobre a linguagem XML-OQL pois ela será mais bem apresentada na seção 3.3.2.

t::=	any	an arbitrary XML element	s ::=	ID
|	()	identity	|	primitive_type
|	v[t]	labeled (tagged) type	|	any
|	{v_1: s_1, ... , v_n: s_n} t	a type with attributes	|	path
|	t_1, t_2	concatenation	|	path*
|	t_1 | t_2	alternation	path::=	XML_type_name
|	t*	repetition	|	v
|	t?	optionality	|	path.v
|	primitive-type	integer, string, image, etc		

Figura 13. Sintaxe dos tipos XML

Além de estender a OQL para lidar com dados XML, os autores também estenderam a ODL para que ela fosse capaz de manipular dados XML. A esta extensão foi dado o nome XML-ODL, que será utilizada para traduzir as consultas escritas em XML-OQL. Os tipos XML são definidos em XML-ODL utilizando uma declaração de tipos especiais da forma: `typedef XML[t] type_name`; onde a sintaxe de `t` está descrita na Figura 13, onde `t`, t_1 e t_2 são tipos XML; `v`, v_1, ..., v_n são nomes; e `s`, s_1, ..., s_n são tipos de atributo. Esta construção define um novo tipo XML com nome `type_name`. Desta forma, `type_name` pode aparecer em qualquer lugar onde um tipo ODL estiver sendo esperado. Como exemplo da utilização da XML-ODL, a Figura 14 mostra um exemplo de DTD e sua equivalente declaração utilizando o XML-ODL.

```
<!ELEMENT bib (vendor*)>
<!ELEMENT vendor (name, email, book*)>
<!ATTLIST vendor id ID #REQUIRED>
<!ELEMENT boook (title, publisher?, year?, price, author+)>
<!ATTLIST book ISBN ID #REQUIRED>
<!ATTLIST book related_to IDrefs>
<!ELEMENT author (firstname?, lastname)>
```

(a) DTD

```
typedef XML [bib[vendor[ {id:ID}
  (name[string], email[string],
   book[ { ISBN: ID, related_to: bib.vendor.book.ISBN* }
         (title[string], publisher[string]?,
          year[integer]?, price[integer],
         author[firstname[string]?, lastname[string]],
         author[firstname[string]?, lastname[string]]* )
       ]* )
]*]] bibliography;
```

(b) XML-ODL

Figura 14. Exemplo de DTD e seu equivalente utilizando XML-ODL

Chegamos agora ao ponto que nos interessa, que é a criação do esquema de classes. A Figura 15 mostra as regras para mapear os tipos XML para ODL. Não entraremos em detalhes na explicação de cada uma destas regras, pois foge do nosso propósito.

T([[any]], ρ)	= list ⟨ XML_element ⟩	(T1)
T([[()]],ρ)	= struct {}	(T2)
T([[v[t]]], ρ)	= T([[t]], ρ, v)	(T3)
T([[{v_1:s_1,...,v_n:s_n}t]],ρ) where s_k = ID	= a reference to a new class C: class C (extent C key v_k) { attribute T([[t]], ρ) info; attribute S([[s1]]) v_1;...; attribute S([[s_n]]) v_n; }; bind path ρ, v_k to C in σ and path ρ, v_k to { v_1: s_1, ..., v_n: s_n}t in δ	(T4)
T([[{v_1:s_1,...,v_n:s_n}t]],ρ)	= struct {T([[t]], ρ) info; S([[s_1]]) v_1;...;S([[s_n]])v_n;}	(T5)
T([[t_1,t_2]], ρ)	= struct {T([[t_1]], ρ) fst; T([[t_2]], ρ) snd;}	(T6)
T([[t_1|t_2]], ρ)	=struct{union_kind tag;T([[t_1]],ρ) left;T([[t_2]],ρ) right;}	(T7)
T([[t*]], ρ)	= list ⟨ T([[t]], ρ) ⟩	(T8)
T([[t?]], ρ)	= T([[t]], ρ)	(T9)
T([[primitive_type]], ρ)	= primitive_type	(T10)
S([[primitive_type]])	= primitive_type	(T11)
S([[path]])	= σ[path]	(T12)
S([[path]]*)	= list ⟨ σ[path] ⟩	(T13)
S([[s]])	= string *otherwise*	(T14)

Figura 15. Mapeamento XML-ODL para ODL

Apresentamos então um exemplo para ilustrar este mapeamento. A Figura 16 mostra a ODL para o esquema cujo DTD está definido na Figura 14.

```
class C2 (extent _C2 key id) {
    //vendor's name and email
    attribute struct{ struct{ string fst; string snd; } fst;
    // books by this vendor
                        list ⟨ C1 ⟩ snd; } info;
    attribute string id; };
class C1 (key ISBN) {
    // title and publisher
    attribute struct{ struct { struct { struct {struct{ string fst, string snd; } fst;
                                              integer snd; } fst;          // year
                                    integer snd; fst;                       // price
                               struct{string fst;string snd;} snd;} fst; // first author
                        list ⟨struct {string fst; string snd;}⟩ snd;} info; // coauthors
attribute string ISBN;
attribute list ⟨ C1 ⟩ related_to; };
```

Figura 16. ODL para o exemplo da Figura 14

2.2 - Comparação entre as Abordagens de Mapeamento

De acordo com a classificação que propusemos no eixo x da Figura 1, há quatro abordagens distintas de mapeamento para os documentos XML. Nesta seção, faremos uma análise comparativa entre estas quatro abordagens.

Enquanto a representação específica é direta, já que cada elemento corresponde a uma tabela ou classe no SGBD, as representações genérica e única fazem um mapeamento indireto. A diferença entre estas duas é que na representação genérica, os elementos são mapeados para algumas estruturas genéricas que indiretamente armazenam estes elementos enquanto que, na representação única, todos os elementos do documento são armazenados num único objeto, com uma estrutura interna que representa o grafo dos sub-elementos. Já a abordagem caixa preta não realiza nenhum tipo de mapeamento, armazenando todo o documento XML no seu formato original.

Enquanto que a representação genérica e a abordagem caixa preta mantêm de forma natural a organização do grafo do documento XML original, os demais formatos precisam de estruturas auxiliares para compor este documento. Além disso, a representação genérica e a abordagem caixa preta adotam um agrupamento de hierarquia de sub-elementos, enquanto que os demais provêem agrupamento por tipo, ou seja, todos os elementos do mesmo tipo estão agrupados. Conseqüentemente, assim como em SGBDs esse armazenamento possui impacto no processamento de consultas, as consultas de navegação são mais eficientes na representação genérica e na caixa preta, e consultas sobre tipos específicos, nas demais abordagens. A vantagem da representação específica em relação às demais abordagens consiste em ter mapeamento direto entre elementos e estruturas no SGBD. Nas demais, com exceção do caixa preta, o acesso aos elementos se dá através de tabelas ou classes genéricas, sobre as quais o elemento é buscado.

Em (TIAN, DEWITT et al., 2002) é realizada uma comparação experimental de desempenho de acesso entre algumas abordagens de mapeamento: caixa preta (Texto) representação única (*Edge* com *Inlining* e uma abordagem baseada em objetos) e representação específica (DTD e Atributo (Binário)). Os resultados apresentados por Tian et al. comprovam as observações anteriores, onde uma das representações específicas (DTD) se destaca em termos de consultas sobre elementos específicos. Além disso, possui bom desempenho em diversas consultas, desde que não envolvam processamento recursivo de SQL. Por outro lado, quando a reconstrução do XML é necessária, as representações específicas levam duas ordens de grandeza a mais no tempo de processamento que a representação única, sendo que esse tempo é zero na abordagem caixa preta. Por outro lado, o desempenho da abordagem caixa preta é vantajoso a custo da manutenção de muitos índices.

Uma outra questão importante está relacionada à linguagem de consulta que será utilizada. Caso o usuário deseje consultar o documento XML através da SQL, a representação ideal é a específica, pois rapidamente o usuário irá identificar os elementos individualmente. Por outro lado, estabelecer os relacionamentos entre os elementos não será trivial e o problema está no grande número de junções necessárias.

Caso o usuário queira fazer consultas em XQuery, a abordagem caixa preta é a ideal, desde que o sistema possua um bom processador de consultas XQuery, já que, ao contrário dos demais, ele não poderá aproveitar os algoritmos de consultas do SGBD. Para conciliar XQuery com SQL, é necessário que haja um tradutor automático para as estruturas do formato escolhido, entretanto essa tarefa não é trivial. No trabalho de (TIAN, DEWITT et al., 2002), as traduções XQuery para SQL foram realizadas manualmente. Esta complexidade decorre da distância estrutural entre a representação do documento XML em árvore e a representação dos elementos em tabelas normalizadas.

Por sua vez, a representação única possui a visível desvantagem de desempenho por armazenar tudo em uma mesma estrutura, apresentando um elevado grau de redundância e valores nulos. As consultas que retornam muitos resultados possuem um péssimo desempenho nesta representação.

Já a abordagem de mapeamento com representação genérica apresenta a vantagem de possuir sempre o mesmo esquema, independente do documento XML armazenado. Isto evita um número excessivo de estruturas criadas caso o documento armazenado possua uma enorme diversidade de nomes de elementos. Todavia, a manipulação destes dados se torna mais complexa visto que obriga o usuário a conhecer as tabelas e os seus relacionamentos, o que não ocorre por exemplo na representação específica, onde cada elemento possui uma estrutura com o mesmo nome.

2.3 - Impacto dos Diferentes Mapeamentos nos Modelos de Dados

De acordo com a classificação que propusemos no eixo y da Figura 1, podemos identificar dois modelos de dados para armazenar os documentos XML: Relacional ou baseado em Objetos. Nesta seção, analisamos o impacto da utilização de cada um deles ao armazenar estes documentos.

O emprego de SGBDs tradicionais (que não são nativos de XML) para o armazenamento de documentos XML é considerado vantajoso por usufruir da segurança e do desempenho de tecnologias amplamente consolidadas, além de permitir a aplicação do universo de recursos advindos do XML em sistemas legados. Entretanto, nesse caso não há um mapeamento direto do modelo de dados XML para o modelo de dados do SGBD hospedeiro, o que requer a identificação e o devido ajuste das possíveis discrepâncias. Esse processo pode incorrer em perdas semânticas, como a ordem dos elementos no documento XML, e no aumento da complexidade das estruturas utilizadas para representar a base XML. Tal complexidade compromete a clareza e o desempenho do acesso aos dados. Um outro aspecto decorrente desse hiato entre os modelos de dados é a dificuldade de tradução da linguagem padrão de consulta sobre dados XML (XQuery) para a linguagem-alvo do SGBD hospedeiro. Um problema adicional está na dificuldade de reconstrução do documento original.

Em SGBDRs, o mapeamento de um documento XML pode implicar na criação de várias estruturas adicionais (colunas para chaves estrangeiras, tabelas normalizadoras para atributos multivalorados, etc.). O modelo de dados relacional privilegia o armazenamento físico eficiente das estruturas de dados, sacrificando para isso o suporte à definição e à manipulação de relacionamentos entre essas estruturas. Para o processamento de expressões de caminho envolvendo vários relacionamentos entre diferentes elementos, comuns na linguagem XQuery, esse enfoque requer comandos SQL verborrágicos e com um número excessivo de junções (MANOLESCU, FLORESCU et al., 2001; SHANMUGASUNDARAM, TUFTE et al., 1999; TIAN, DEWITT et al., 2002).

Por outro lado, os SGBDs baseados em objetos oferecem a riqueza semântica do modelo de dados OO, com maior transparência para a representação de relacionamentos através de identificadores de objetos (OIDs) gerados automaticamente e de atributos de referência para tais OIDs. Além disso, possuem a capacidade de representação da herança, o que facilita a representação dos nós e suas especializações. Esses diferenciais reduzem significativamente o impacto da transição entre os modelos de dados XML e OO, diminuindo o número de estruturas auxiliares como as relacionais e facilitando a tradução da linguagem de consulta.

Já os SGBDs nativos XML não precisam realizar mapeamento algum, pois os documentos XML são armazenados em uma estrutura própria para estes tipos de documento. Desta forma, a manipulação sobre estes dados tende a funcionar de uma

maneira mais eficaz em comparação ao armazenamento em SGBDs não nativos. Entretanto, as vantagens obtidas com o armazenamento de dados estruturados legados e dados XML em um mesmo sistema são perdidas. Além disso, refazer diversos algoritmos de representação e acesso, já robustos e eficientes nos SGBDs tradicionais, para o modelo XML, pode comprometer essa eficiência.

2.4 - Árvores DOM para Armazenar Documentos XML

Conforme visto nas seções anteriores, o uso de um SGBD baseado em objetos combinado com um esquema genérico de representação dos dados é uma alternativa bem adequada para realizar o armazenamento dos documentos XML, já que precisaremos manipular estes dados posteriormente. Neste contexto, acreditamos que o formato DOM oferece a melhor relação entre a semântica oferecida para a representação dos documentos XML e a eficiência de acesso aos dados. A árvore DOM possui a vantagem considerável de ser um padrão para a manipulação de documentos XML através do formato OO de representação de dados. Este formato é amplamente utilizado em memória principal, com poder de representação das características do XML (por exemplo, é conservada a ordenação dos elementos). Por usar uma estrutura de dados orientada a objetos, a árvore DOM permite o emprego de algoritmos simples, eficientes e bastante conhecidos para a manipulação dos elementos XML. A principal desvantagem da árvore DOM para o armazenamento de documentos XML em SGBDs decorre do baixo desempenho das consultas que recuperam elementos XML de um mesmo tipo, visto que tais elementos estão espalhados pela árvore. Todavia, estruturas de indexação da árvore DOM podem ser utilizadas para minimizar este problema. É importante ressaltar que, pelo uso intensivo de relacionamentos e operações de navegação, a implementação do DOM em SGBDRs é pouco factível.

O nosso objetivo é aproveitar a infra-estrutura oferecida por SGBDORs, como também tornar o processo de transformação o mais simples possível. Para tanto, é necessário um formato de armazenamento que nos permita acessar um esquema de representação conhecido e reconstruir com facilidade o documento XML ou parte dele. Como o formato DOM se enquadra nestas características, optamos por utilizar esta abordagem para o armazenamento de documentos XML. Através da criação e

povoamento de classes da árvore DOM num SGBDOR é possível consultar os dados armazenados através da SQL3.

Entretanto, nesta estratégia ainda persiste a necessidade de conhecimento prévio dos tipos que compõem a árvore DOM para a elaboração de consultas SQL sobre os documentos XML armazenados. Portanto, é necessário um mecanismo que transforme uma consulta sobre documentos XML escrita em XQuery (que deve ser a mais natural ao usuário do documento XML) em uma consulta sobre a árvore DOM escrita na linguagem do SGBD. Neste trabalho, nós apresentamos regras de tradução da linguagem XQuery para a linguagem SQL3 e mostramos que as características da árvore DOM facilitam significativamente esse processo, conforme o conjunto de regras propostas no Capítulo 3.

2.4.1 - Análise do SAX

O SAX (*Simple API for XML*) (SAX PROJECT, 2002) foi um outro formato analisado para representar os documentos XML em estruturas de linguagens de programação. No entanto, como esta abordagem não monta uma árvore e o acesso aos dados é feito de maneira serial, é bastante complicado acessar elementos aleatórios no documento XML. O SAX é categorizado como baseado em eventos enquanto o DOM se encaixa na categoria dos baseados em árvore. E como o nosso objetivo principal é executar consultas para retornar dados aleatórios armazenados em documentos XML, a utilização desta abordagem foi descartada.

Este formato é adequado para aplicações onde é necessário retornar uma informação particular no documento, pois não há uma sobrecarga adicional para montar uma árvore. Um exemplo simples para ilustrar esta situação é de uma aplicação que tem como objetivo buscar os dados de determinado livro em um documento XML. Como o acesso é feito serialmente, assim que o livro desejado é encontrado, o processamento pára (o SAX permite isso) e o dado é retornado para o usuário. Podemos utilizar o SAX também para aplicações onde não é necessário retornar o documento inteiro, mas sim parte dele. Caso uma aplicação precise retornar os livros de determinado autor, o SAX percorre o documento e retorna apenas os livros solicitados. Neste exemplo, a construção de uma árvore é desnecessária e não traz nenhum benefício para a aplicação.

2.4.2 - Manipulação de Dados no Formato DOM

A seguir, explicamos como acessar o conteúdo de um documento XML armazenado na árvore DOM e o procedimento para recuperar elementos da árvore. O esquema completo do DOM, nível 2, já foi apresentado na Figura 5.

Considere que as classes com sufixo `Impl` correspondem à implementação das classes DOM, conforme definido pela API Xerces (THE APACHE XML PROJECT, 2002). A classe `DocumentImpl`, como o próprio nome sugere, armazena informações sobre os diferentes documentos XML armazenados na árvore DOM. Já a classe `ElementImpl` é responsável por armazenar os elementos XML, deixando para a classe `TextImpl` guardar o seu conteúdo. A classe `AttrImpl` guarda, além do nome do atributo, o seu valor. A navegação entre as classes é realizada através do atributo `childrenNode`, que pertence à classe `NodeImpl`. Todas as classes citadas anteriormente são herdadas de `NodeImpl`. Os nomes de elementos e atributos XML ficam armazenados no atributo `name` e seus conteúdos no atributo `value`. A descrição de todas as classes está definida na Tabela 1.

Portanto, uma consulta sobre determinado documento começa sobre o `DocumentImpl` e continua através do atributo `childrenNode`, alcançando desta forma a raiz do documento. Prosseguindo através do `childrenNode`, acessamos os elementos filhos da raiz e assim sucessivamente, até chegarmos aos nós-folhas do documento. É importante ressaltar que os nomes e os valores de atributos XML encontram-se em um mesmo nó da árvore, ao contrário do que ocorre com os elementos. Desta forma, para acessar o conteúdo de determinado elemento, precisamos navegar mais uma vez pelo `childrenNode`, para só então recuperar o seu valor.

A Figura 17 apresenta um exemplo de representação do DOM utilizando o documento XML da Figura 3. É importante lembrar que as classes do DOM herdam de uma classe chamada `NodeImpl` que contém os atributos `name` e `value`, além do atributo de navegação entre classes denominado `childrenNode`. Portanto, todas as outras classes herdarão estes atributos. A navegação pela árvore correspondente a este documento começa pela classe que armazena as informações sobre o documento, que é a `DocumentImpl`. No seu atributo `name`, armazenamos o nome deste documento, que é "*Dept.xml*". Podemos então acessar os filhos dos objetos desta classe navegando através do atributo `childrenNode`. A classe `DocumentImpl` possui apenas um

filho, que é um objeto da classe `ElementImpl`, que por sua vez está armazenando o nome da raiz do documento XML, que é "*Dept*", através do seu atributo `name`. Continuando o percurso na árvore através do atributo `childrenNode`, alcançamos os três filhos do objeto que armazena a raiz do documento, sendo um atributo, armazenado na classe `AttrImpl`, e dois elementos, armazenados na classe `ElementImpl`. A classe `AttrImpl` armazena o nome do atributo e o seu valor no mesmo objeto, respectivamente nos atributos `name` e `value`. Nesse caso, `name` é igual a "*dept_id*" e `value` equivale à "*dept1*". Já os outros dois filhos pertencem à classe `ElementImpl`, e armazenam no atributo `name` o nome do elemento que correspondem, que nesse caso é "*Student*". Navegando a partir destes dois filhos através do atributo `childrenNode`, alcançamos os quatro filhos do primeiro objeto, sendo um atributo ("*student_id*") e três elementos ("*Name*", "*Enroll*", "*Enroll*"); e os dois filhos do segundo objeto, sendo um atributo ("*student_id*") e um elemento ("*Name*"). Navegando a partir destes elementos através do atributo `childrenNode`, alcançamos seus filhos, que são objetos da classe `TextImpl`, e armazenam os seus valores. No exemplo, estão armazenados quatro objetos da classe `TextImpl`: "*St1*", "*CS10*", "*CS20*" e "*St2*", que são filhos dos elementos "*Name*", "*Enroll*", "*Enroll*" e "*Name*", respectivamente. Já os nomes dos atributos, "*student_id*" são armazenados na classe `AttrImpl` junto com os seus valores, "*123*" e "*124*".

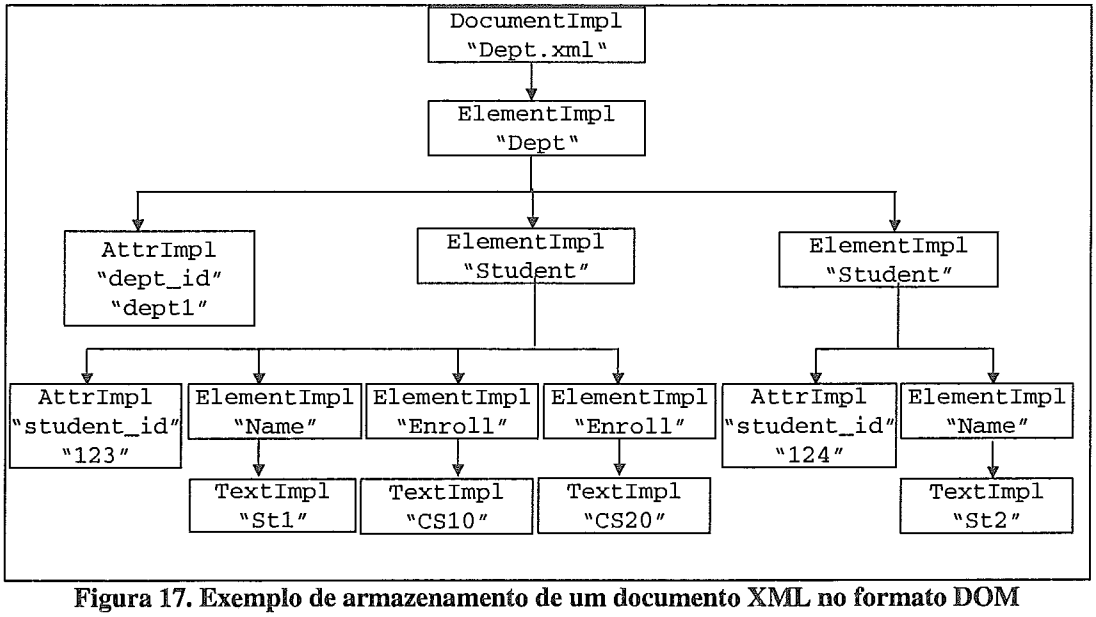


Figura 17. Exemplo de armazenamento de um documento XML no formato DOM

Capítulo 3 - Transformação XQuery -> SQL3

A execução de consultas no SGBD é uma forma simples que os usuários têm de acessar os dados armazenados. Contudo, caso os dados em questão sejam documentos XML armazenados em SGBDs não nativos, é difícil oferecer esta facilidade com transparência. Conforme visto no Capítulo 2, existem inúmeras maneiras de armazenar estes documentos XML em um SGBD. Independente da abordagem de armazenamento escolhida, o usuário necessita conhecer as estruturas criadas, sejam elas tabelas ou classes, para poder realizar consultas sobre a base. Obviamente, obrigar o usuário a conhecer as estruturas de armazenamento criadas não é uma boa solução. Além disso, a utilização das linguagens de consulta dos SGBDs não é a mais apropriada para consultar dados XML, já que um mapeamento foi feito para armazenar os dados XML no SGBD alvo. Portanto, seria mais natural e coerente a utilização de uma linguagem de consulta própria do XML para acessar os dados armazenados, pois podemos supor que aqueles usuários que estão armazenando documentos XML conhecem uma linguagem de consulta para acessar as informações necessárias. Conforme definido em (WORLD WIDE WEB CONSORTIUM (W3C), 2002c), a linguagem padrão para consulta sobre dados XML é a XQuery.

Portanto, como há uma linguagem de consulta padrão para o XML, é interessante permitir que os usuários a utilizem para acessar os dados armazenados. Desta forma, eles não precisam conhecer as estruturas criadas no processo de armazenamento dos documentos e ainda podem utilizar uma linguagem de consulta que é familiar aos usuários do XML.

Com a escolha desta alternativa para consultar os dados armazenados, um outro problema surge. Como a maioria dos SGBDs (com exceção é claro dos Nativos XML) não possui um processador XQuery, um processo de transformação deve ser efetuado na consulta XQuery de entrada com o objetivo de convertê-la para a linguagem de consulta do SGBD alvo. Dependendo da escolha da abordagem de armazenamento e do modelo de dados escolhido, relacional ou baseado em objetos, esta transformação geralmente é uma tarefa complexa.

No Capítulo 2, nós identificamos quatro abordagens para o armazenamento de documentos XML. Como precisamos transformar uma XQuery de entrada em uma linguagem alvo de consulta de um SGBD, levando em consideração o formato de

armazenamento, precisamos escolher uma abordagem que facilite o processo de transformação. Analisando as abordagens apresentadas, podemos concluir que a representação genérica para os elementos é a mais adequada para o nosso propósito, já que as estruturas criadas são sempre as mesmas, independente do documento XML armazenado. A abordagem caixa preta foi descartada por utilizar um tipo de coluna especial, sem apresentar nenhum tipo de mapeamento. Já na representação única para os elementos, a desvantagem é a composição da consulta para recuperar os dados, já que apenas uma estrutura de dados é utilizada no mapeamento. E por fim, a representação específica para os elementos não é uma boa solução já que as estruturas criadas variam com os documentos armazenados, dificultando o processo de tradução.

Na seção 3.1, detalhamos a arquitetura proposta no nosso trabalho, com a utilização do formato DOM de armazenamento e de um modelo de dados baseado em objetos. Na seção 3.2, explicamos alguns pontos importantes do nosso trabalho em relação à XQuery, à SQL3 e ao DOM. Na seção 3.3, discutimos alguns trabalhos que realizam a transformação da XQuery para uma linguagem alvo de consulta. Após isso, apresentamos algumas definições sobre expressões de caminho na seção 3.4 e definimos as regras propostas para realizar a transformação da XQuery de entrada para a SQL3 de saída na seção 3.5. Finalmente, mostramos alguns exemplos destas transformações na seção 3.6.

3.1 - Arquitetura Proposta

A Figura 18 apresenta uma arquitetura que consiste na tradução da linguagem XQuery para uma linguagem alvo de consulta. A idéia inicial consiste no envio de um documento XML ao Módulo de Armazenamento, que é o responsável por armazenar os documentos XML em um SGBD seguindo uma determinada abordagem. O usuário entra então com a consulta escrita em XQuery que é submetida a um tradutor automático dando a impressão que existe um processador nativo de consultas XML. O tradutor, por sua vez, aplica as regras de transformação necessárias para a tradução da XQuery de entrada para a linguagem alvo de consulta. Finalmente, esta consulta é executada sobre o SGBD correspondente, retornando o resultado para o usuário.

Portanto, com esta arquitetura genérica, podemos explorar a potencialidade de execução otimizada de consultas em SGBDs já existentes e a utilização da linguagem

XQuery, que é a linguagem de consulta padrão para o XML. É importante ressaltar que isto é feito de maneira transparente, evitando que o usuário necessite conhecer as estruturas de armazenamento dos documentos XML. Desta forma, o usuário continua a trabalhar somente no universo XML, que é o mais natural para ele, enquanto o tradutor se encarrega de realizar a transformação da XQuery para a linguagem alvo de consulta.

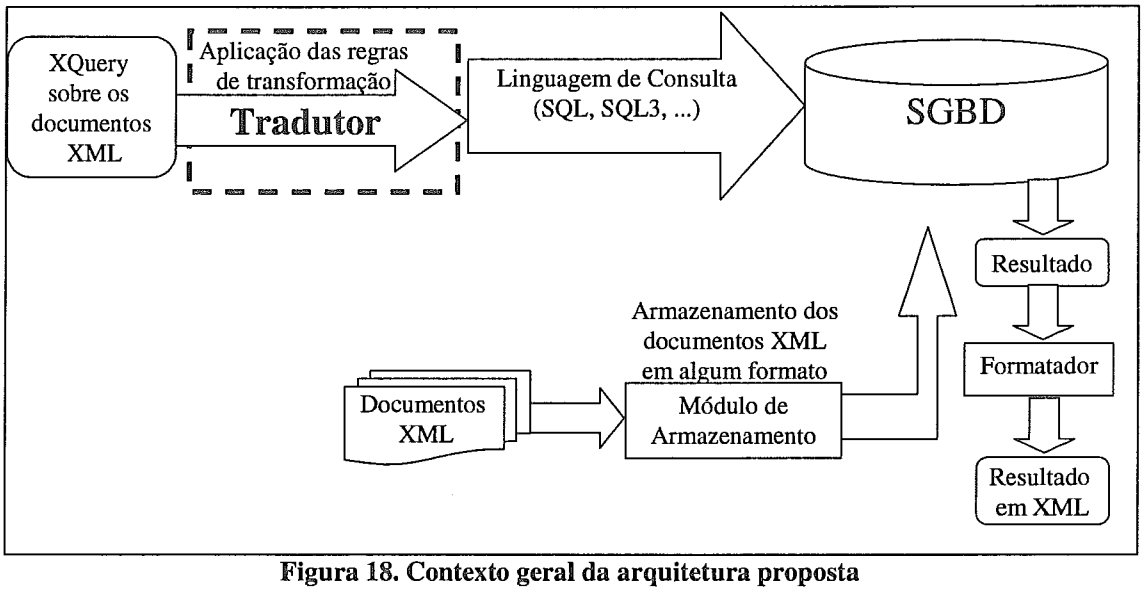


Figura 18. Contexto geral da arquitetura proposta

Conforme explicado anteriormente, concluímos que a utilização de uma abordagem genérica para armazenamento dos elementos aliada a um modelo de dados baseado em objetos oferece vantagens significativas para o nosso objetivo principal. Portanto, a "instância" da arquitetura proposta que utilizamos possui uma linguagem de consulta baseada em objetos, a SQL3, e um SGBDOR, permitindo executar neste SGBD a linguagem de consulta de saída do tradutor, que denominamos *XVerter*. Além disso, nesta "instância" o Módulo de Armazenamento utiliza o formato DOM para persistir os documentos XML no SGBD. Acreditamos que a combinação DOM + SGBDOR facilita o processo de transformação da XQuery para a SQL3, devido às semelhanças em suas semânticas.

3.2 - XQuery x SQL3 x DOM

A XQuery é uma linguagem bastante poderosa e fornece uma enorme variedade de funcionalidades para consultar dados XML. Porém, devido às suas características, não existe um mapeamento direto entre a XQuery e a linguagem de consulta típica de

SGBDORs – a SQL3. Isto quer dizer que algumas consultas XQuery requerem construções adicionais à SQL3 para que o mapeamento seja satisfeito. O nosso trabalho engloba a tradução de um conjunto significativo das operações de consulta da XQuery, com exceção principalmente das expressões condicionais (por limitações do SQL3, que é uma linguagem de consulta essencialmente declarativa). As principais características da linguagem XQuery, as quais são apresentadas neste trabalho, são:

- ✓ Expressões FLWR (FOR-LET-WHERE-RETURN);
- ✓ Navegação sobre as expressões de caminho;
- ✓ Uso do operador "//".

Analisamos essas características inovadoras através de experimentos baseados no *benchmark* (SCHMIDT, WAAS et al., 2002) sobre processadores XQuery, visando avaliar o potencial desta linguagem e as principais implementações existentes. O Capítulo 5 apresenta mais detalhes sobre este experimento. Com base nesta análise, verificamos que os sistemas existentes concentram-se no suporte a um conjunto limitado de características da linguagem XQuery.

Apesar de estarmos limitando a XQuery de entrada àquelas características citadas anteriormente, precisamos mostrar de alguma forma que este sub-conjunto pode ser mapeado para uma SQL3 correspondente. É sabido que uma consulta XQuery consegue retornar qualquer informação desejada dos documentos XML. Por conseguinte, o sub-conjunto que estamos tratando na nossa proposta também é capaz disto. Outra informação importante é que o formato DOM consegue representar totalmente os documentos XML em suas estruturas sem nenhuma perda semântica. O DOM possui, entre outras, a capacidade de representação e de reconstrução total dos documentos XML. Isto quer dizer que podemos utilizar o formato DOM para armazenar os dados e depois gerar um documento de saída que será exatamente igual ao documento original de entrada. Além disso, o DOM também permite uma reconstrução parcial do documento original. Ou seja, a partir de um determinado elemento, o DOM permite a navegação através dos seus filhos construindo um documento que é um fragmento do documento XML original. Como estamos criando as estruturas do formato DOM em um SGBDOR, podemos utilizar a SQL3 para navegar sobre estas estruturas e recuperar os objetos necessários. Já que o DOM possui a capacidade de representar os documentos XML sem perda semântica, a SQL3 é capaz de recuperar quaisquer dados que estejam armazenados neste formato. Portanto, visto que: (i) a XQuery consegue recuperar qualquer informação de um documento XML; (ii) Mapeamos o formato DOM

em disco no SGBDOR; (iii) o DOM é capaz de representar os documentos XML de forma completa, sem perda semântica; (iv) e a SQL3 permite expressar caminhos na árvore DOM para recuperar quaisquer objetos armazenados; concluímos que a linguagem SQL3 consegue expressar consultas da linguagem XQuery, desde que os dados estejam armazenados na árvore DOM. Por conseguinte, o sub-conjunto de características da XQuery que estamos tratando em nossa proposta é capaz de ser mapeado para a SQL3, cujo resultado da consulta será o mesmo da XQuery de entrada.

3.3 - Trabalhos Relacionados

Dentre os diversos trabalhos analisados, existem dois que se destacam por apresentar uma solução para transformar uma linguagem de consulta do XML em uma linguagem alvo de consulta e que utilizam a representação genérica para os elementos, ou seja, a mesma abordagem escolhida por nós. Um deles utiliza o modelo de dados relacional e o outro um modelo de dados baseado em objetos. Primeiro discutiremos a solução apresentada por (MANOLESCU, FLORESCU et al., 2001), que utiliza o modelo de dados relacional e a XQuery como consulta de entrada. Em seguida, analisaremos uma das soluções propostas em (FEGARAS & ELMASRI, 2001), que usa um modelo de dados baseado em objetos e uma linguagem de consulta XML proprietária.

3.3.1 - Uma Solução Relacional

A Figura 19 apresenta a arquitetura proposta em (MANOLESCU, FLORESCU et al., 2001) para executar consultas XQuery sobre fontes de dados heterogêneas.

Neste trabalho, os autores apresentam uma pequena introdução sobre a linguagem XQuery antes de iniciar o detalhamento sobre a arquitetura proposta. A primeira etapa desta arquitetura consiste em normalizar a consulta XQuery de entrada a fim de prepará-la para a etapa posterior. Para tal, são apresentadas regras de normalização para aplicar sobre a consulta inicial, desde que isso seja possível. Como há uma discrepância entre os modelos relacional e XML, nem sempre a XQuery pode ser normalizada. Os autores então definem um sub-conjunto de características da XQuery que poderá ser traduzido para o SQL. Este sub-conjunto consiste em:

(i) Expressões de caminho, incluindo acesso aos filhos, descendentes e atributos além dos elementos referenciados;

(ii) Construtores de elemento, cujas *tags* e dados são constantes ou vêm de expressões de caminho ou de expressões FLWR que podem ser traduzidas;

(iii) Expressões FLWR;

(iv) Operações aritméticas, lógicas, de conjunto e lista sobre coleções de itens XML do mesmo tipo.

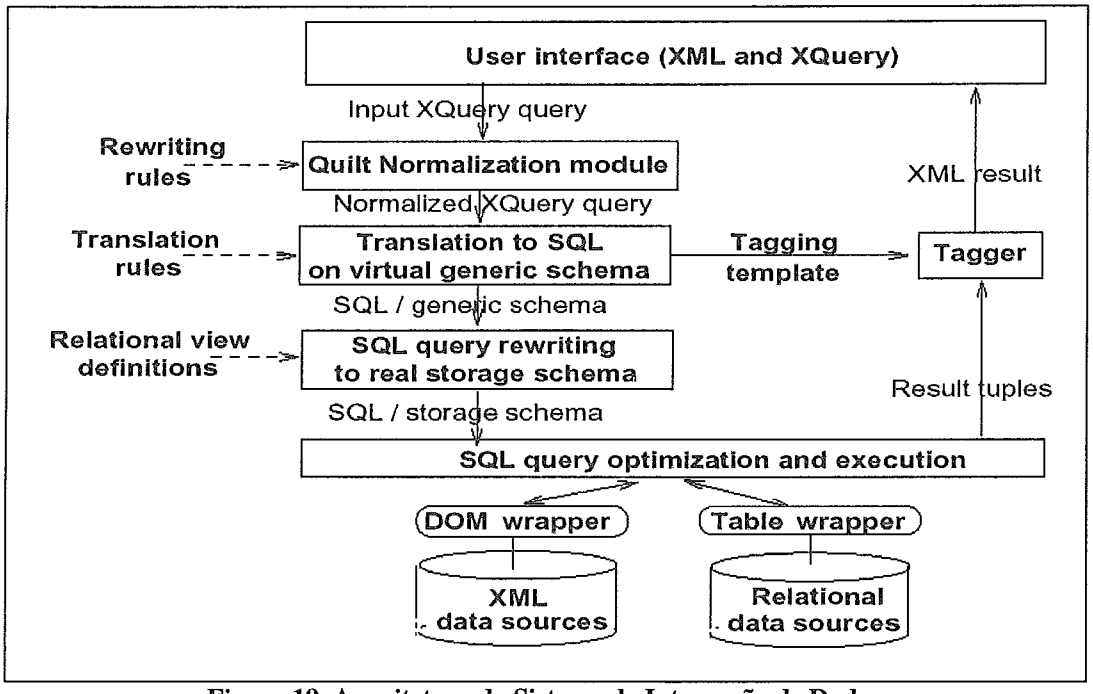


Figura 19. Arquitetura do Sistema de Integração de Dados

A consulta XQuery normalizada passa então por um processo de tradução para o SQL, através da aplicação de um conjunto de regras. Finalmente, a consulta SQL resultante da tradução é reescrita (ainda em SQL) sobre o esquema virtual para que seja então otimizada e executada sobre as fontes heterogêneas. O formato de armazenamento utilizado neste trabalho, de acordo com a nossa classificação, é uma representação genérica para os elementos usando o modelo relacional. Esta abordagem consiste na criação de um conjunto de tabelas que armazenarão os elementos, atributos, enfim, toda a estrutura dos documentos XML, além é claro dos dados neles contidos. Além disso, este conjunto de tabelas é invariável, ou seja, é sempre o mesmo independente do documento XML que está sendo armazenado. Esta abordagem de armazenamento foi apresentada no Capítulo 2, na Figura 2.

Como podemos perceber, existem inúmeras semelhanças entre a proposta apresentada em (MANOLESCU, FLORESCU et al., 2001) e a nossa. A primeira é em relação à abordagem de armazenamento. Ambas as propostas utilizam uma representação genérica para os elementos, apesar de utilizarem modelos de dados distintos. Além disso, as duas propostas possuem a capacidade de receber como entrada uma consulta XQuery e transformá-la em uma linguagem alvo para executar sobre as fontes correspondentes. Durante este processo de transformação, outras semelhanças se tornam evidentes. Em ambas as propostas, é definido um sub-conjunto da XQuery que poderá ser tratado pelos módulos componentes da arquitetura. Conforme pôde ser observado no início desta seção, o sub-conjunto suportado pela proposta de Manolescu et. al. é bem semelhante às características suportadas pelo *XVerter*. Ambas suportam expressões de caminho e expressões *Flower*, porém ainda não têm regras definidas para o tratamento das funções CONTAINS, EMPTY, BEFORE, entre outras. Além disso, nos dois casos são aplicadas regras de normalização na consulta XQuery de entrada com o objetivo de prepará-la para uma etapa posterior. E por fim, na parte mais importante das duas arquiteturas, são apresentadas regras de tradução para transformar a consulta normalizada na linguagem alvo. Esta linguagem alvo no nosso caso é a SQL3, já que estamos propondo a utilização de um SGBDOR, enquanto na outra é a linguagem SQL, pois o SGBD em questão é o relacional.

```
for x in document("med.xml")/medical/patient,
    y in document("med.xml")//patientSSno,
    z in x/name
where x/@SSno = y
return z

select e3.elID as $z
from Document d1, URI u1, Value v1, Element e1, QName q1,
     Value v2, Child c1, Element e2, QName q2, Value v3,
     Attribute a1, Value v4, Value v5, Child c2, Element e3,
     QName q3, Value v6, TransClosure tc1, Element e4,
     QName q4, Value v7, Child c3, Value v8
where d1.docURIID = u1.uriID and u1.uriValID = v1.valID and
      v1.value = "med.xml" and d1.rootElemId = e1.elID and
      e1.elQNameID = q1.qNameID and q1.qnLocalID = v2.valID
      and v2.value = "medical" and c1.parentID = e1.elID and
      c1.childID = e2.elID and e2.elQNameID = q2.qNameID and
      q2.qnLocalID = v3.valID and v3.value = "patient" and
      a1.attrElID = e2.elID and a2.attrNameID = v4.valID and
      v4.value = "SSno" and a1.attrValID = v5.valID and
      c2.parentID = e2.elID and c2.childID = e3.elID and
      e3.elQNameID = q3.qNameID and q3.qnLocalID = v6.valID
      and v6.value = "name" and d1.rootElemId = tc2.parentID
      and tc2.childID = e4.elID and e4.elQNameID = q4.qNameID
      and q4.qnLocalID = v7.valueID and
      v7.value = "patientSSno" and c3.parentID = e3.elID and
      c3.childValID = v8.valID and v5.value = v8.value
```

Figura 20. Exemplo de tradução da XQuery para SQL

Contudo, também existem algumas diferenças fundamentais nas duas propostas. Enquanto a nossa utiliza um modelo de dados baseado em objetos, a proposta de (MANOLESCU, FLORESCU et al., 2001) usa o modelo de dados relacional. Este fato resulta em diferenças na consulta resultante do processo de tradução. Devido às diferenças de semântica entre o modelo relacional e o modelo XML, a SQL resultante é geralmente enorme, com um elevado número de junções, fato este que compromete o tempo de execução desta consulta. A Figura 20 ilustra a complexidade desta tradução, mostrando a XQuery de entrada e a SQL resultante da transformação. Como estamos utilizando o modelo OR, a nossa proposta não apresenta este problema. Já que o DOM é capaz de representar totalmente o grafo correspondente ao documento XML, precisamos apenas navegar entre os nós através dos atributos de referência para recuperarmos a informação desejada.

Outra diferença se encontra no módulo de tradução. Enquanto a nossa proposta utiliza o XSLT para realizar a transformação, mantendo desta forma grande parte da solução no universo XML, a proposta de (MANOLESCU, FLORESCU et al., 2001) não apresenta uma solução deste tipo, tornando necessário o uso de uma linguagem de programação como Java ou C++ para realizar esta tradução. Como o XSLT é uma solução coerente e natural para representar transformações de documentos XML, optamos por este caminho. Desta forma, o resultado do módulo de normalização será uma representação XML da consulta XQuery de entrada. Após isto, serão aplicadas regras de tradução para gerar como resultado a SQL3 de saída. Portanto, a solução fica mais robusta pois qualquer alteração nas regras precisa ser feita apenas no documento XSL. Já na outra solução, uma modificação pode ser bastante custosa devido à necessidade de alterar um programa feito em uma linguagem de programação qualquer, o que nem sempre é uma tarefa simples.

3.3.2 - Uma Solução Baseada em Objetos

Em (FEGARAS & ELMASRI, 2001), o objetivo é transformar uma linguagem de consulta proposta por eles, a XML-OQL, em uma OQL para ser executada em SGBDOOs. Conforme já explicado no Capítulo 2, duas soluções são apresentadas no trabalho: uma realiza a tradução sem ter conhecimento da estrutura do documento armazenado e a outra possui as informações do esquema para transformar a XML-OQL

de entrada. Daremos atenção especial à primeira solução pois é a que mais se aproxima da nossa proposta, já que também utiliza uma representação genérica para os elementos.

Os autores começam apresentando a extensão da OQL proposta por eles e denominada XML-OQL. A XML-OQL é essencialmente a OQL tradicional com duas características adicionais: uso de expressões de caminho em XML e construção de dados em XML. Pela sintaxe desta linguagem, os elementos XML são construídos da seguinte forma, para m, n ≥ 0:

<tag a_1 = u_1 ... a_m = u_m> e_1, ..., e_n </tag>

Esta expressão constrói um elemento XML com um nome, tag, com atributos a_1 ..., a_m e sub-elementos e_1, ..., e_n como conteúdo. Cada atributo a_i é associado ao resultado da expressão u_i . Os dados XML podem ser acessados diretamente de um SGBD através do nome, retrieve("bibliography") ou podem ser baixados de um arquivo local ou da *Web* utilizando uma URL, como em document(http://www.acm.org/xml/journals.xml).

Tabela 2. Navegação sobre expressões de caminho em XML

Expressão de caminho	Descrição
e.A	Projeção sobre a *tag* de nome A
e._	Projeção sobre qualquer *tag*
e.*	Todos os sub-elementos de e em qualquer profundidade
e.@A	Projeção sobre o atributo A de e
e[\v→e']	Os sub-elementos de e que satisfazem e'
e[e']	O sub-elemento de e na posição e'

A estrutura da árvore dos dados XML pode ser navegada utilizando expressões de caminho apresentadas na Tabela 2, onde e e e' são expressões da XML-OQL, A é uma *tag* ou o nome de um atributo e v é uma variável. O filtro utilizado difere do filtro de elementos utilizado pelo XPath. Para cada sub-elemento v de e, é associada uma variável v para o sub-elemento e avaliado o predicado e'. O resultado contém todos os sub-elementos de e que satisfazem e'. Outro detalhe que é um pouco diferente da XPath, se deve ao fato que a projeção e.A da XML-OQL retorna apenas o conteúdo de A, ao passo que na XPath (e/A), os nomes da *tag* também aparecem no resultado. A sintaxe da XML-OQL também permite referência a IDrefs, conforme pode ser

observado pelo exemplo da Figura 21(a) através da expressão `r.title`. O DTD do documento utilizado se encontra na Figura 21(b), onde os elementos que não aparecem descritos são do tipo #PCDATA.

```
select list <bib>
               <author> b.author.lastname </author>,
               <title> b.title </title>,
               <related> select list <title> r.title </title>
                         from r in b.@related_to
               </related>
            </bib>
from bs in retrieve("bibliography").bib.vendor.book,
b in bs
where b.year > 1995 and
count(b.author) > 2 and
b.title like "%computer%"
```

(a) Consulta XML-OQL

```
<!ELEMENT bib (vendor*)>
<!ELEMENT vendor (name, email, book*)>
<!ATTLIST vendor id ID #REQUIRED>
<!ELEMENT boook (title, publisher?, year?, price, author+)>
<!ATTLIST book ISBN ID #REQUIRED>
<!ATTLIST book related_to IDrefs>
<!ELEMENT author (firstname?, lastname)>
```

(b) DTD do documento XML utilizado

Figura 21. Exemplo de uma consulta XML-OQL

Uma vez explicada a XML-OQL, podemos discutir melhor a primeira solução proposta pelos autores, onde é feita uma tradução sem ter conhecimento do esquema do documento XML armazenado. A abordagem de armazenamento utilizada nesta solução já foi apresentada no Capítulo 2, na Figura 6.

(U1) $e._ \rightarrow any\_projection(e)$

(U2) $e.* \rightarrow wildcard\_projection(e)$

(U3) $e.@A \rightarrow attribute\_projection(e, "A")$

(U4) $$\frac{e : list\langle XML\_element >}{e.A \rightarrow tag\_projection(e, "A")}$$

(U5) $$\frac{e : list\langle attribute\_type\rangle, a \in \{A, @A, \_, *\}}{e.a \rightarrow deref(e).a}$$

(U6) $$\frac{e : list\langle XML\_element\rangle, e' : \text{int } eger}{e[e'] \rightarrow list(e[e'])}$$

(U7) $e[\backslash v \rightarrow e'] \rightarrow select\ list\ v\ from\ v\ in\ e\ where\ e'$

Figura 22. Tradução das expressões de caminho XML-OQL para OQL

A Figura 22 apresenta as regras de tradução de expressões de caminho da XML-OQL para OQL. A notação `e→e'` indica que `e` é traduzida para `e'`; `e:t` indica que a expressão `e` possui tipo `t`; e uma fração corresponde a uma assertiva condicional. As funções `tag_projection` e `wildcard_projection` estão definidas na Figura 23. A função `any_projection` é similar à `tag_projection`, possuindo a condição `if-then-else` sempre verdadeira. As assinaturas das funções

attribute_projection e deref, que acham um elemento XML cujo ID é igual ao seu valor IDref, também se encontram na Figura 23.

```
define tag_projection (e: list <XML_element>, tag_name: string) : list <XML_element> as
select list y
from x in e,
         y in (case x.element of
                  PCDATA: list(),
                  TAG: if x.element.tag.name = tag.name
                            then x.element.tag.content
                            else list()
                  end );
```

```
define wildcard_projection (e: list <XML_element>) : list <XML_element> as
e + (select list y
     from x in e,
          y in (case x.element of
                  PCDATA: list(),
                  TAG: wildcard_projection(x.element.tag.content)
                  end ) );
```

```
attribute_projection (e: list <XML_element>, aname: string): list <attribute_type>
```

```
deref (idrefs: list <attribute_type>): list <XML_element>
```

Figura 23. Funções utilizadas

Fazendo uma comparação desta solução com a nossa proposta, podemos perceber que ambas utilizam uma representação genérica para os elementos em conjunto com o modelo de dados baseado em objetos. Uma das diferenças está no fato desta solução utilizar uma linguagem de consulta para XML criada pelos autores, a XML-OQL, ao contrário da nossa proposta que usa uma linguagem de consulta padrão, a XQuery. Portanto, esta é uma grande desvantagem da solução de (FEGARAS & ELMASRI, 2001), pois obriga os usuários a conhecer esta linguagem proprietária de consulta. Um outro ponto negativo desta solução é que o usuário necessita conhecer o esquema de classes criado para consultar os dados, já que é utilizada uma abordagem de mapeamento proprietária. Este problema não ocorre com a nossa proposta, visto que estamos utilizando um formato de armazenamento padrão, como o DOM. Além disso, a XML-OQL não parece ser tão poderosa quanto a XQuery, não apresentando certas características como o operador "//" e funções do tipo CONTAINS, EMPTY e BEFORE. Desta forma, a tradução desta linguagem para a OQL fica facilitada, porém é incapaz de representar grande parte das características presentes na XQuery.

3.4 - Expressões de Caminho

DEFINIÇÃO 1 (Expressão de caminho): Uma expressão de caminho (EC) representa um percurso por uma seqüência EC:root/e[1][p1]/.../e[n][pn] de

elementos XML relacionados, onde root indica a raiz do documento XML para o qual EC é analisada e os termos da forma e[i], 1≤i≤n, representam os elementos XML nomeados tal que e[0]=root e e[i-1] é pai de e[i]. O comprimento de EC é determinado pelo número de elementos envolvidos, sendo denotado por n. Podem ser definidos predicados aninhados sobre os elementos e/ou atributos de elementos contidos em uma EC. Cada predicado aninhado pi, 1≤i≤n, é opcional e pode ser escrito como uma combinação lógica de termos da forma (e[i]|a[i])<*operador*><*valor*>, onde a[i] representa um atributo do elemento e[i].

DEFINIÇÃO 2 (*Radical de uma Expressão de Caminho*): Denotamos ECr como o radical de uma expressão de caminho ECx de comprimento n, tal que ECr é uma expressão de caminho de comprimento m e ECr ⊂ ECx, m≤n.

3.5 - Regras de Transformação

Uma consulta em XQuery possui como estrutura básica uma expressão FLWR da forma:

FOR [declaração de cursores]

LET [atribuição de cursores]

WHERE [predicados de seleção]

RETURN [projeção do resultado].

Um expressão FLWR denotada por Q pode conter um conjunto de ECs, representado por S_Q={EC1,...,ECw}, w≥0. Na conversão desta estrutura para a representação tradicional SELECT-FROM-WHERE da SQL3, nós devemos observar as seguintes regras:

1. O resultado da consulta corresponde à projeção feita na cláusula RETURN da XQuery. Esse resultado é descrito na cláusula SELECT da SQL3;
2. As cláusulas FOR, LET, WHERE e RETURN da XQuery definirão os cursores a serem especificados na cláusula FROM da SQL3. Nessas cláusulas são declaradas as ECs contidas em S_Q, onde a quantidade de cursores é determinada pelo comprimento de cada ECx∈ S_Q, denotado por nx, tal que:

a. Para cada ECx , 1≤x≤w, é declarado um cursor dx que representa a raiz do documento XML correspondente, da forma "dx in DocumentImpls";

b. Para cada elemento nomeado $e[i] \in ECx$, $1 \leq x \leq w$ e $1 \leq i \leq nx$, é declarado um cursor c_i da forma "c_i in c_{i-1}.childrenNode", onde c_0=dx. No caso de existirem várias ECs na consulta, os cursores serão identificados por ca_i, cb_i , ..., e assim sucessivamente para cada EC declarada;

c. Para cada elemento XML nomeado $e[i]$, $1 \leq i \leq nx$, de cada $ECx \in S_Q$, $1 \leq x \leq w$, cujo valor é acessado na consulta e para cada atributo $a[i]$, $1 \leq i \leq nx$, especificado em cada $ECx \in S_Q$, $1 \leq x \leq w$, é declarado um cursor c_{i+1} da forma "c_{i+1} in c_i.childrenNode";

3. A cláusula WHERE da XQuery corresponde quase diretamente à cláusula WHERE da SQL3. Neste caso, observamos também que, além dos predicados definidos sobre elementos e atributos XML, teremos predicados derivados de:

a. Declaração dos documentos XML na cláusula FOR da XQuery;

b. Declaração de cada elemento XML nomeado nas ECs, ou seja, para cada $e[i] \in ECx$, $1 \leq i \leq nx$, $ECx \in S_Q$, $1 \leq x \leq w$, é necessário declarar um predicado aninhado pi referente ao nome de $e[i]$.

c. Declaração de junções de ECs relativas à cláusula LET.

Passo de Otimização (Redução): Em uma consulta Q, duas expressões de caminho Ecx e Ecy, tal que $ECx \in S_Q$ e $ECy \in S_Q$, podem compartilhar os cursores correspondentes se possuírem um radical comum ECr. Os cursores compartilhados são os determinados pela estrutura de ECr.

Em consulta XQuery sobre uma árvore DOM de um documento XML, cuja altura (número de níveis de elementos XML da árvore) é representada por h, assumimos a seguinte representação:

⇒ $e[i.1]$, ..., $e[i.k]$: nomes dos k diferentes elementos XML que estão armazenados em um mesmo nível i, $1 \leq i \leq h$, da árvore DOM;

⇒ $a[i.1]$, ..., $a[i.k]$: k diferentes atributos de um elemento $e[i]$, $1 \leq i \leq h$, da árvore DOM.

3.5.1 - Tradução de uma Expressão de Caminho Explícito

A Figura 24 apresenta um exemplo que transforma uma EC simples da XQuery, de comprimento n, em um comando da SQL3. A parte da linha 1 da XQuery que faz

referência ao nome do documento tem como equivalentes a linha 2 da SQL3 (decorrente da regra 2a) e a linha 5 (aplicação da regra 3a). Aplicando as regras 2b e 2c, o restante da linha 1 da XQuery (que define a EC) corresponde às linhas 3 e 4 da SQL3, respectivamente (onde são declarados os cursores que navegam na árvore DOM através do atributo *childrenNode*). As linhas 6 e 7 da SQL3 são montadas a partir da regra 3b. Por fim, a linha 2 da XQuery (referente à projeção do resultado) equivale à linha 1 da SQL3, após ser aplicada a regra 1.

É importante ressaltar a necessidade de declarar todos os elementos XML nomeados na EC da XQuery na cláusula WHERE da SQL3. Um retorno errado poderá acontecer caso o `e[i]`, $1 \leq i \leq n$, esteja presente mais de uma vez no documento XML e seja filho de diferentes nós-pais (por exemplo, `e[y]` e `e[z]`). Se quisermos consultar o `e[i]` filho de `e[y]` e não declararmos um predicado referente ao elemento `e[y]` na cláusula WHERE, o resultado desta consulta poderá retornar também o `e[i]` filho de `e[z]`.

```
XQuery
1 for $x in document("doc.xml")/e[1]/e[2]/…/e[n]
2 return $x

SQL3
1 select c_{n+1}.value
2 from d1 in DocumentImpls,
3 c1 in d1.childrenNode, c2 in c1.childrenNode,…,
4 c_{n+1} in c_n.childrenNode
5 where (d1.name = "doc.xml") and
6 (c1.name = "e[1]") and (c2.name = "e[2]") and … and
7 (c_n.name = "e[n]")
```

Figura 24. Expressão de caminho explícito

Também precisamos declarar o nome do documento na cláusula WHERE porque vários documentos podem estar armazenados na mesma base. Caso ocultemos o nome do documento pesquisado, o resultado poderá retornar os elementos de um outro documento XML que também satisfizerem às condições da consulta. Verifique que a árvore DOM precisa ser percorrida até o nível máximo da expressão de caminho, já que a nossa abordagem de armazenamento não extrai um esquema onde classes representam cada elemento no documento XML (e o acesso somente aos elementos nomeados da EC não é possível). Neste exemplo, são criados `n+1` cursores c_i para percorrer os `n` níveis de elementos da EC contida na XQuery, sendo o último cursor necessário para acessar o valor do elemento armazenado no nível `n`.

3.5.2 - Tradução de uma Expressão de Caminho Explícito com Atributos

O exemplo da Figura 25 tem apenas uma diferença em relação ao exemplo anterior: a presença de atributos XML na EC. Como observamos no exemplo anterior, o conteúdo de um elemento `e[i]` fica armazenado em um nó no nível `i+1` da árvore DOM. Do mesmo modo, o nome e o conteúdo de um atributo `a[i]` (pertencente ao elemento `e[i]`) encontram-se armazenados no nível `i+1` da árvore DOM. Desta forma, `n+1` cursores devem ser declarados na cláusula FROM da SQL3.

```
XQuery
1 for $x in document("doc.xml")/e[1]/…/e[n]/@a[n]
2 return $x

SQL3
1 select cn.value
2 from d1 in DocumentImpls,
3 c1 in d1.childrenNode, c2 in c1.childrenNode,…,
4 cn in cn-1.childrenNode, cn+1 in cn.childrenNode
5 where (d1.name = "doc.xml") and
6 (c1.name = "e[1]") and (c2.name = "e[2]") and … and
7 (cn.name = "e[n]") and
8 (cn+1.name = "@a[n]")
```

Figura 25. Expressão de caminho explícito com atributos

3.5.3 - Tradução de XQuery contendo Várias Expressões de Caminho

O exemplo da Figura 26 consiste em uma generalização dos exemplos anteriores. Neste caso, mais de uma EC está presente na consulta XQuery, como podemos verificar nas linhas 2 a 5. Deste modo, de acordo com a regra 2, precisamos fazer referência a todas elas na cláusula FROM da SQL3 para podermos representar os predicados de consulta existentes na cláusula WHERE da XQuery. Porém, considerando o nosso passo de otimização, não é necessário declarar todas as ECs completamente, já que elas possuem um radical `ECr` em comum (o radical que começa em `e[1]` e segue até `e[n-1]` está presente em todas as ECs da consulta), cuja declaração equivale à linha 4 da SQL3. As linhas 3 a 8 representam as declarações das diferentes ECs presentes na XQuery, conforme determinado pelas regras 2b e 2c. Seguindo a regra 3b, os predicados correspondentes à consultas são declarados nas linhas 10 a 14 da SQL3. Através da regra 3a, a linha 5 da XQuery equivale à linha 1 da SQL3. A declaração do documento na linha 1 da XQuery equivale à linha 2 da SQL3 (aplicando a regra 2a) e à linha 9 (ao seguirmos a regra 3b). Finalmente, após aplicada a regra 1, a linha 5 da XQuery corresponde à linha 1 da SQL3.

Embora o comando SQL3 resultante na Figura 26 seja um pouco longo, já que o caminho deve ser explicitado totalmente tanto para os predicados de consulta quanto para o "esquema" XML (nomes dos elementos), no caso da representação relacional sem o uso do DOM, o comando SQL ficaria bem maior devido às inúmeras junções, como pode ser observado em (MANOLESCU, FLORESCU et al., 2001).

```
XQuery
1 for $x in document("doc.xml")/e[1]/e[2]/.../e[n-1]
2 where $x/e[n.1] = "constante[1]" and
3 $x/e[n.2] = "constante[2]" and ... and
4 $x/e[n.(k-1)] = "constante[k-1]"
5 return $x/e[n.k]

SQL3
1 select cz_{n+1}.value
2 from d1 in DocumentImpls,
3 ca1 in d1.childrenNode, ca2 in ca1.childrenNode,…,
4 ca_{n-1} in ca_{n-2}.childrenNode,
5 ca_n in ca_{n-1}.childrenNode, ca_{n+1} in ca_n.childrenNode,
6 cb_n in ca_{n-1}.childrenNode, cb_{n+1} in cb_n.childrenNode,
7 cc_n in ca_{n-1}.childrenNode, cc_{n+1} in cc_n.childrenNode,…,
8 cz_n in ca_{n-1}.childrenNode, cz_{n+1} in cz_n.childrenNode
9 where (d1.name = "doc.xml") and
10 (ca1.name = "e[1]") and (ca2.name = "e[2]") and … and
11 (ca_n.name = "e[n.1]") and (ca_{n+1}.value = "constante[1]") and
12 (cb_n.name = "e[n.2]") and (cb_{n+1}.value = "constante[2]") and ... and
13 (cy_n.name = "e[n.(k-1)]") and (cy_{n+1}.value = "constante[k-1]") and
14 (cz_n.name = "e[n.k]")
```

Figura 26. XQuery contendo várias expressões de caminho

3.5.4 - Tradução do Operador "//"

Em todos os exemplos vistos até agora, o caminho completo das ECs está declarado na consulta XQuery. Porém, é bastante comum em uma consulta XQuery a especificação de um determinado elemento nomeado no documento XML que pode ser encontrado em qualquer nível de profundidade na árvore. É exatamente este o objetivo do operador "//", onde `root/e[1]/.../e[n]//e[j]` indica que, partindo de `root/e[1]/.../e[n]`, desejamos alcançar todos os elementos com nome `e[j]` independente dos caminhos existentes entre eles e a raiz da árvore DOM, ou mesmo do nível onde `e[j]` se encontra. O termo conhecido nesse problema é o comprimento do radical de elementos nomeados, representado por `n`. A tradução do operador "*" seria um sub-conjunto da regra definida para traduzir o operador "//". Apresentamos então o caso mais complexo, que é o operador "//".

Para traduzir o "//" para SQL3 não é suficiente declarar na cláusula WHERE apenas aqueles elementos que são nomeados na EC. Necessitamos descobrir a altura máxima da árvore para definirmos quantos cursores serão declarados na cláusula FROM e para construirmos os predicados correspondentes na cláusula WHERE. Portanto, esta

tradução será realizada em dois passos: (i) descobrir a altura da árvore; e (ii) construir a consulta resultante.

No nosso procedimento de armazenamento no formato DOM, utilizamos o campo "*value*" da classe *DocumentImpl* para guardar a altura máxima da árvore DOM, considerando os nós que armazenam elementos XML. Desta forma, teremos acesso a esta informação apenas com a consulta trivial mostrada na Figura 27.

```
select dl.value from dl in DocumentImpls where (dl.name = "doc.xml")
```

Figura 27. Consulta para descobrir a altura máxima da árvore DOM

Após a execução desta consulta, temos o resultado da altura máxima da árvore (desconsiderando os nós de conteúdo e os nós de atributos XML) e estamos aptos para construir a consulta final. Apresentamos duas regras diferentes para a tradução do operador "//". A primeira não utiliza o operador UNION e a segunda usa este operador na tradução. A Figura 29 mostra a tradução de uma EC contendo o operador "//" em uma árvore DOM com altura `h` sem utilizar o operador UNION e a Figura 29 mostra a tradução usando este operador.

```
XQuery
1 for $x in document("doc.xml")/e[1]/e[2]/.../e[n]//e[j]
2 return $x

SQL3
1 select c_{n+2}.value, c_{n+3}.value,..., c_{h+1}.value
2 from dl in DocumentImpls,
3 c1 in dl.childrenNode, c2 in c1.childrenNode,..., c_n in c_{n-1}.childrenNode,
4 c_{n+1} in c_n.childrenNode,..., c_h in c_{h-1}.childrenNode, c_{h+1} in c_h.childrenNode
5 where (dl.name = "doc.xml") and
6 (c1.name = "e[1]") and (c2.name = "e[2]") and ... and (c_n.name = "e[n]") and
7 ((c_{n+1}.name= "e[j]") or (c_{n+2}.name= "e[j]") or ... or
8  (c_{h-1}.name= "e[j]") or (c_h.name= "e[j]"))
```

Figura 28. Expressão de caminho utilizando o operador "//" sem o uso do UNION

Na Figura 28, utilizando as regras 2b e 3b respectivamente, as linhas 3 e 6 da SQL3 correspondem aos elementos declarados na linha 1 da XQuery que começa em `e[1]` e vai até `e[n]`. Pelas regras 2b e 2c, a linha 4 da SQL3 declara os cursores necessários para percorrer a árvore DOM conforme o critério de busca por `e[j]`. Já as linhas 7 e 8, através da regra 3b, definem os predicados destes cursores, utilizando o operador lógico "`or`" para indicar que o elemento `e[j]` pode estar em qualquer nível da árvore DOM, desde o nível `n+1` até o nível `h`. Aplicando a regra 1, a linha 1 da SQL3 equivale à linha 2 da XQuery e inclui na projeção todas as possibilidades de resposta, desde o nível `n+1` até o `h`. E finalmente, conforme já visto nos exemplos anteriores, através da aplicação das regras 2a e 3a , construímos respectivamente as linhas 2 e 5 da SQL3, que correspondem à declaração do documento XML.

A Figura 29 mostra uma tradução semelhante à anterior, porém o uso do UNION evita a existência de nulos no retorno do resultado. Neste caso, o número de *sub-selects* da consulta será equivalente à diferença entre a altura da árvore e o nível onde e[n] se encontra. Ou seja, cada *sub-select* corresponde ao resultado de um nível da árvore. Todos estes sub-resultados são então agregados gerando o resultado final apenas em uma coluna, ao contrário da solução que não utiliza o UNION.

```
XQuery
1 for $x in document("doc.xml")/e[1]/e[2]/.../e[n]//e[j]
2 return $x

SQL3
1 select c_{n+2}.value
2 from c1 in d1.childrenNode, c2 in c1.childrenNode,..., c_n in c_{n-1}.childrenNode,
3 c_{n+1} in c_n.childrenNode, c_{n+2} in c_{n+1}.childrenNode
4 where (d1.name = "doc.xml") and
5 (c1.name = "e[1]") and (c2.name = "e[2]") and ... and (c_n.name = "e[n]") and
6 (c_{n+1}.name= "e[j]")
7 union
8 select c_{n+3}.value
9 from c1 in d1.childrenNode, c2 in c1.childrenNode,..., c_n in c_{n-1}.childrenNode,
10 c_{n+1} in c_n.childrenNode, c_{n+2} in c_{n+1}.childrenNode, c_{n+3} in c_{n+2}.childrenNode
11 where (d1.name = "doc.xml") and
12 (c1.name = "e[1]") and (c2.name = "e[2]") and ... and (c_n.name = "e[n]") and
13 (c_{n+2}.name= "e[j]")
14 union
15 ...
16 union
17 select c_{h+1}.value
18 from c1 in d1.childrenNode, c2 in c1.childrenNode,..., c_n in c_{n-1}.childrenNode,
19 c_{n+1} in c_n.childrenNode,..., c_h in c_{h-1}.childrenNode, c_{h+1} in c_h.childrenNode
20 where (d1.name = "doc.xml") and
21 (c1.name = "e[1]") and (c2.name = "e[2]") and ... and (c_n.name = "e[n]") and
22 (c_h.name= "e[j]")
```

Figura 29. Expressão de caminho utilizando o operador "//" com o uso do UNION

3.5.5 - Tradução do LET

A cláusula LET de uma expressão FLWR representa uma atribuição de cursores, que basicamente consiste na junção de ECs. Portanto, esta cláusula é tratada de maneira semelhante à declaração das ECs presentes na cláusula FOR. O exemplo da Figura 30 ilustra a utilização do LET. As linhas 5, 6 e 9 da SQL3 equivalem à declaração da EC no LET, através da aplicação das regras 2b, 2c e 3b respectivamente. Da mesma forma, as linhas 3, 4 e 8 correspondem à EC declarada no FOR. As linhas 2 e 7 da SQL3 equivalem aos documentos declarados no FOR e LET, nas linhas 1 e 2 da XQuery, ao serem aplicadas as regras 2a e 3a respectivamente. A linha 3 da XQuery equivale à linha 1 da SQL3, através da regra 1. E finalmente, a regra 3c define a linha 10 da SQL3 que equivale ao final da linha 2 da XQuery (`[f[m]=$x]`).

```
XQuery
1 for $x in document("doc1.xml")/e[1]/e[2]/.../e[n]
2 let $y := document("doc2.xml")/f[1]/f[2]/.../f[m-1][f[m]=$x]
3 return $y/f[m]

SQL3
1 select cb_{m+1}.value
2 from d1 in DocumentImpls, d2 in DocumentImpls,
3 ca1 in d1.childrenNode, ca2 in ca1.childrenNode,..., ca_n in ca_{n-1}.childrenNode,
4 ca_{n+1} in ca_n.childrenNode,
5 cb1 in d2.childrenNode, cb2 in cb1.childrenNode,...,  cb_m in cb_{m-1}.childrenNode,
6 cb_{m+1} in cb_m.childrenNode,
7 where (d1.name = "doc1.xml") and (d2.name = "doc2.xml") and
8 (ca1.name = "e[1]") and (ca2.name = "e[2]") and ... and (ca_n.name = "e[n]") and
9 (cb1.name = "f[1]") and (cb2.name = "f[2]") and ... and (cb_m.name = "f[m]") and
10 (ca_{n+1}.value = cb_{m+1}.value)
```

Figura 30. Utilização do LET

3.6 - Exemplos de Mapeamento

Apresentamos a seguir algumas consultas testadas no *XVerter*, que correspondem aos cinco exemplos básicos de tradução explicados na seção 3.5: EC explícito, EC explícito com atributos, consulta com várias expressões de caminho, uso do operador "//" e uso da cláusula LET. As consultas são apresentadas na Tabela 1, onde a coluna "Mapeamento" representa a figura onde os termos e os resultados da tradução são exibidos. Foram utilizados dois documentos XML para os testes de tradução XQuery->SQL3 (`recados.xml` e `cartas.xml`), apresentados na Figura 31 e na Figura 32, respectivamente.

Tabela 3. Consultas de exemplo do tradutor *XVerter*

Consulta	*Texto*	*Mapeamento*
Q1	Retornar os dias de todos os recados do documento recados.xml.	**Figura 33**
Q2	Listar os identificadores menores de 5 de todos os recados do documento recados.xml.	**Figura 34**
Q3	Listar as pessoas para as quais Jane enviou recados no documento recados.xml.	**Figura 35**
Q4	Listar todos os assuntos descendentes de cartas no documento cartas.xml.	**Figura 36**
Q5	Listar as pessoas que enviaram cartas no documento cartas.xml e que também foram remetentes no documento recados.xml.	**Figura 37**

```
<recados>
 <recado id="1">
  <para>Joao</para>
  <de>Jane</de>
  <cabecalho>Lembrete</cabecalho>
  <corpo>Nao se esqueca de mim no fim de semana.</corpo>
  <data>
    <dia>01</dia>
    <mes>02</mes>
    <ano>2001</ano>
  </data>
 </recado>
 <recado id="2">
  <para>Maria</para>
  <de>Jane</de>
  <cabecalho>Colegio</cabecalho>
```

```
  <corpo>Nao se esqueca do dever de casa.</corpo>
  <data>
    <dia>01</dia>
    <mes>03</mes>
    <ano>2001</ano>
  </data>
 </recado>
 <recado id="3">
  <para>Joao</para>
  <de>Maria</de>
  <cabecalho>Almoco</cabecalho>
  <corpo>Hoje eu vou almocar em casa.</corpo>
  <data>
    <dia>02</dia>
    <mes>02</mes>
    <ano>2001</ano>
  </data>
 </recado>
</recados>
```

Figura 31. recados.xml

```
<cartas>
 <carta id="1">
  <para>Joao</para>
  <de>Maria</de>
  <assunto>Viagem</assunto>
 </carta>
 <carta id="2">
  <para>Jane</para>
  <de>Joao</de>
  <assunto>Empresa</assunto>
 </carta>
 <bilhete id="1">
  <conteudo>
   <para>Maria</para>
   <de>Jane</de>
   <assunto>Carro</assunto>
  </conteudo>
 </bilhete>
</cartas>
```

Figura 32. cartas.xml

```
XQuery
FOR $n IN document("recados.xml")/recados/recado/data/dia
RETURN $n

SQL3
SELECT c5.value
FROM d1 in DocumentImpls, c1 in d1.childrenNode, c2 in c1.childrenNode,
c3 in c2.childrenNode, c4 in c3.childrenNode, c5 in c4.childrenNode
WHERE (d1.name = "recados.xml") and (c1.name="recados") and
(c2.name="recado") and (c3.name = "data") and (c4.name = "dia")

Resultado
{ [ "01" ] [ "01" ] [ "02" ] }
```

Figura 33. Tradução da consulta Q1

```
XQuery
FOR $n IN document("recados.xml")/recados/recado/@id
WHERE $n < 5
RETURN $n

SQL3
SELECT c2.value
FROM d1 in DocumentImpls,c1 in d1.childrenNode, c2 in c1.childrenNode,
c3 in c2.childrenNode
WHERE (d1.name = "recados.xml") and
(c1.name = "recados") and (c2.name = "recado") and (c3.name = "id") and
(c3.value < "5")

Resultado
```

```
{ [ "1" ] [ "2" ] [ "3" ] }
```

Figura 34. Tradução da consulta Q2

```
XQuery
FOR $n IN document("recados.xml")/recados/recado
WHERE $n/de = "Jane"
RETURN $n/para

SQL3
SELECT cb4.value
FROM d1 in DocumentImpls, ca1 in d1.childrenNode, ca2 in ca1.childrenNode,
ca3 in ca2.childrenNode, ca4 in ca3.childrenNode, cb3 in ca2.childrenNode,
cb4 in cb3.childrenNode
WHERE (d1.name = "recados.xml") and (ca1.name = "recados") and
(ca2.name = "recado") and (ca3.name = "de") and (ca4.value = "Jane") and
(cb3.name = "para")

Resultado
{ [ "Joao" ] [ "Maria" ] }
```

Figura 35. Tradução da consulta Q3

```
XQuery
FOR $n IN document("cartas.xml")/cartas//assunto
RETURN $n

SQL3
SELECT c3.value, c4.value, c5.value
FROM d1 in DocumentImpls, c1 in d1.childrenNode, c2 in c1.childrenNode,
c3 in c2.childrenNode, c4 in c3.childrenNode, c5 in c4.childrenNode
WHERE (d1.name = "cartas.xml") and (c1.name = "cartas") and
((c2.name = "assunto") or (c3.name = "assunto") or (c4.name = "assunto"))

Resultado
{ [ "null" "Viagem" "null" ] [ "null" "Empresa" "null" ] [ "null" "null" "Carro" ] }
```

Figura 36. Tradução da consulta Q4

```
XQuery
FOR $x in document("recados.xml")/recados/recado/de
LET $y := document("cartas.xml")/cartas/carta[de=$x]
RETURN $y/de

SQL3
SELECT cb4.value
FROM d1 in DocumentImpls, d2 in DocumentImpls,
ca1 in d1.childrenNode, ca2 in ca1.childrenNode, ca3 in ca2.childrenNode, ca4 in ca3.childrenNode,
cb1 in d2.childrenNode, cb2 in cb1.childrenNode, cb3 in cb2.childrenNode, cb4 in cb3.childrenNode
WHERE (d1.name = "recados.xml") and (d2.name = "cartas.xml") and
(ca1.name = "recados") and (ca2.name = "recado") and (ca3.name = "de") and
(cb1.name = "cartas") and (cb2.name = "carta") and (cb3.name = "de") and (ca4.value = cb4.value)

Resultado
{ [ "Maria" ] }
```

Figura 37. Tradução da consulta Q5

Neste capítulo, damos destaque ao projeto e implementação da arquitetura *XVerter*, com ênfase no tradutor, no armazenamento dos documentos XML e na execução da consulta no SGBD. A Figura 38 corresponde a um detalhamento da arquitetura genérica do tradutor conforme a Figura 18, apresentando todos os passos necessários no processo de transformação de uma XQuery para uma linguagem alvo de consulta. Estes passos são aqueles contidos na linha pontilhada da figura. É importante ressaltar, conforme podemos visualizar na figura, que esta arquitetura é genérica, podendo ser utilizada por qualquer aplicação, desde que sejam definidas a abordagem de armazenamento compatível (ou na linha) com o modelo de classes do DOM e a linguagem alvo de consulta.

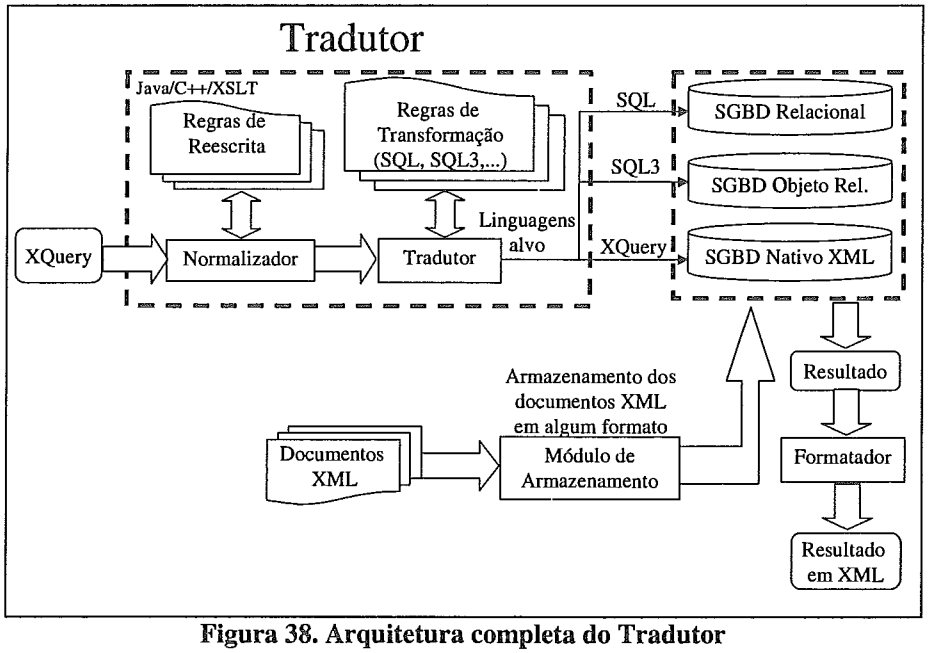


Figura 38. Arquitetura completa do Tradutor

Quando o tradutor recebe uma consulta XQuery de entrada, o módulo de normalização aplica algumas regras de reescrita nesta consulta transformando-a em uma representação XML intermediária da XQuery inicial. Este passo pode ser implementado através de qualquer linguagem de programação, não apresentando maiores complexidades. Esta representação intermediária é passada então para o módulo de tradução que irá realizar a transformação propriamente dita para a linguagem alvo de consulta, através da aplicação de algumas regras pré-definidas. Estas regras, por sua vez, estão implementadas em um documento XSL, que é uma solução coerente e natural

para transformar documentos XML em algum formato de saída. Com isso, podemos utilizar o XSLT para transformar a representação XML intermediária da XQuery de entrada em uma linguagem alvo de consulta. Por fim, podemos executar esta consulta resultante em um SGBD correspondente. É importante ressaltar que a geração de uma representação intermediária aliada ao uso do XSLT foi uma forma encontrada para mantermos a solução no universo XML, além de deixarmos a solução de modo mais flexível possível. Poderíamos utilizar diretamente uma linguagem de programação para transformar a XQuery de entrada em uma linguagem alvo de consulta, mas o custo embutido para realizar alterações nesta solução seria muito elevado. Qualquer modificação na abordagem de armazenamento ou na linguagem alvo de consulta teria como conseqüência a implementação de um novo tradutor, sem a possibilidade de nenhum aproveitamento da implementação anterior. A nossa solução permite uma reutilização parcial do que estiver implementado, bastando adaptar a geração do documento intermediário e as regras implementadas no documento XSL para a nova abordagem de armazenamento e/ou linguagem de consulta, além de ser uma solução modularizada.

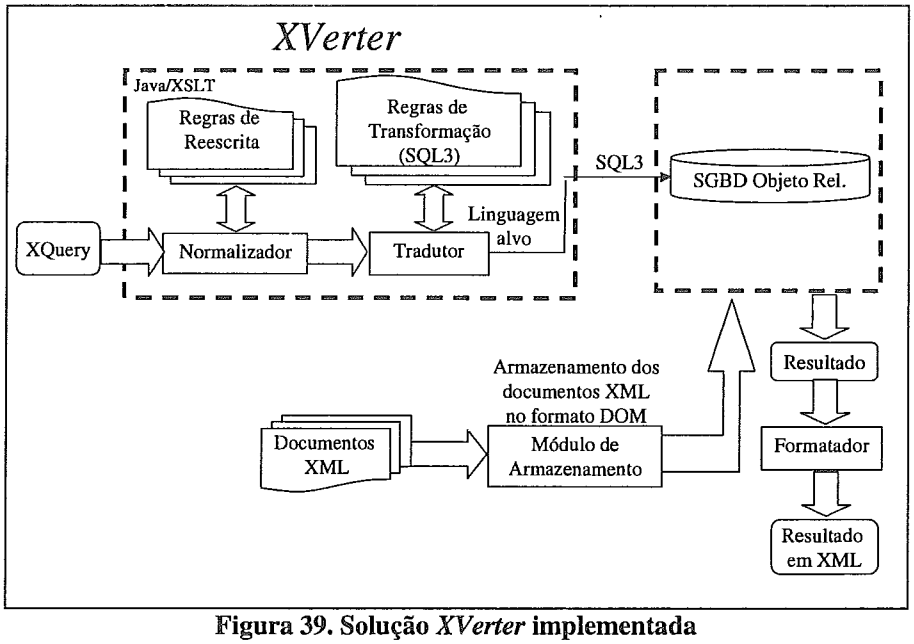


Figura 39. Solução *XVerter* implementada

Conforme explicado no Capítulo 3, decidimos utilizar o formato DOM de armazenamento e um SGBDOR para executar consultas SQL3 que serão geradas pelo *XVerter*. A solução implementada está apresentada na Figura 39.

Como podemos perceber, a nossa solução apresenta três passos principais no processo de transformação de uma consulta XQuery em uma SQL3 de saída:

(i) Armazenamento dos documentos XML em um SGBDOR;

(ii) Geração da representação XML intermediária a partir da XQuery de entrada;

(iii) Aplicação das regras de transformação utilizando XSLT.

Nas próximas seções, explicaremos com detalhes cada uma destas etapas.

4.1 - Armazenamento dos documentos XML

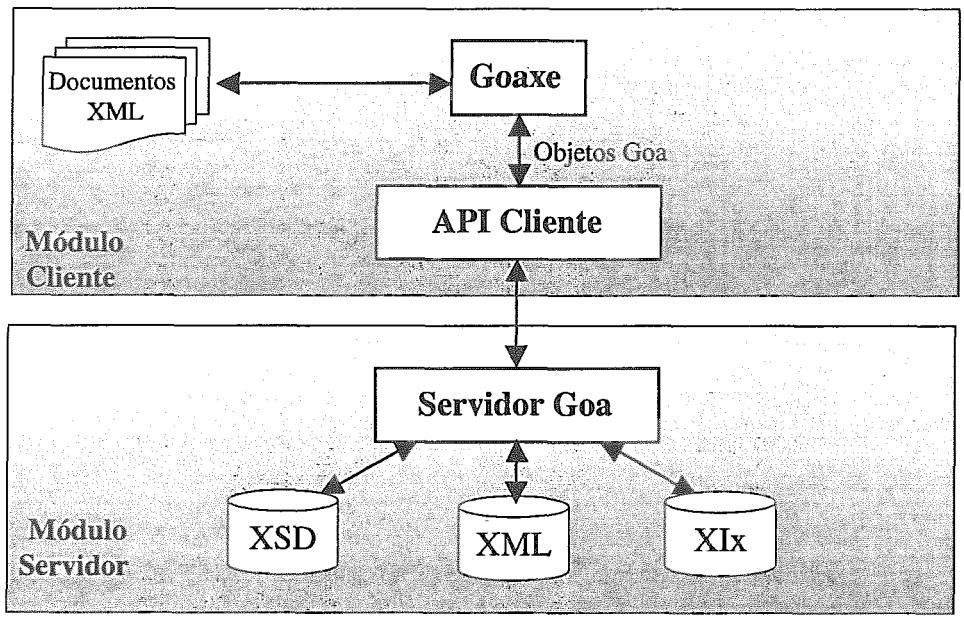


Figura 40. Arquitetura do GOA XML

Para o armazenamento dos documentos XML, foi utilizado o GOA, que é um protótipo de SGBDOO totalmente desenvolvido na COPPE-UFRJ e em constante evolução. Apesar de estarmos propondo a utilização de um SGBDOR, podemos realizar os testes no GOA pois não estamos usando nenhuma característica que seja puramente OO. Ou seja, estamos validando as regras através da execução de consultas OQL mas que poderiam estar sendo executadas em um SGBDOR utilizando SQL3 sem nenhuma diferença.

```
CREATE_CLASS Root - Roots;
CREATE_CLASS NamedNodeMapImpl Root NamedNodeMapImpls;
CREATE_CLASS NodeImpl Root NodeImpls name V50 value V50;

CREATE_CLASS NotationImpl NodeImpl NotationImpls;
```

```
CREATE_CLASS ProcessingInstructionImpl NodeImpl ProcessingInstructionImpls;
CREATE_CLASS NodeContainerImpl NodeImpl NodeContainerImpls;
CREATE_CLASS CharacterDataImpl NodeImpl CharacterDataImpls;

CREATE_CLASS AttrImpl NodeContainerImpl AttrImpls;
CREATE_CLASS EntityImpl NodeContainerImpl EntityImpls;
CREATE_CLASS DocumentImpl NodeContainerImpl DocumentImpls;
CREATE_CLASS EntityReferenceImpl NodeContainerImpl EntityReferenceImpl;
CREATE_CLASS DocumentTypeImpl NodeContainerImpl DocumentTypeImpls;
CREATE_CLASS DocumentFragmentImpl NodeContainerImpl DocumentFragmentImpls;
CREATE_CLASS ElementImpl NodeContainerImpl ElementImpls;

CREATE_CLASS CommentImpl CharacterDataImpl CommentImpls;
CREATE_CLASS TextImpl CharacterDataImpl TextImpls;

CREATE_CLASS CDataSectionImpl TextImpl CDataSectionImpls;

CREATE_RELATIONSHIP nodes * NamedNodeMapImpl NodeImpl namedNodeMapImpl 1;
CREATE_RELATIONSHIP attrElement 1 NamedNodeMapImpl ElementImpl attributes 1;
CREATE_RELATIONSHIP childrenNode * NodeImpl NodeImpl parentNode 1;
CREATE_RELATIONSHIP docType 1 DocumentImpl DocumentTypeImpl typeDocs 1;
```

Figura 41. Script de criação das classes do DOM no GOA

Para permitir este armazenamento, a API Cliente GOA XML *Enabler* (*Goaxe*) (MATTOSO, CAVALCANTI et al., 2002) foi desenvolvida por diversos alunos da linha de Banco de Dados da COPPE. Apesar de já haver uma versão do cliente GOA em C++, foi necessário desenvolver um outro cliente em Java para permitir a utilização de uma API (THE APACHE XML PROJECT, 2002) que manipula os documentos XML no formato DOM. Desta forma, o cliente Java foi adaptado para que pudesse utilizar esta API no armazenamento de documentos XML no GOA. A Figura 40 apresenta a arquitetura desenvolvida para habilitar o GOA a manipular documentos XML. Portanto, o *Goaxe* é o responsável por receber como entrada os documentos XML e armazená-los no GOA como objetos. Conforme já citado anteriormente, estamos utilizando uma abordagem genérica para os elementos, mais especificamente o DOM. Desta forma, o *Goaxe* é o encarregado em transformar os dados XML de entrada em objetos que possam ser armazenados no GOA, utilizando as estruturas criadas para utilizar o DOM. Uma vez criados os objetos, API Cliente é a responsável por fazer a comunicação com o servidor via *Socket* e transferir os objetos criados pela *Goaxe*. Como a *Goaxe* é capaz de realizar o mapeamento de qualquer documento XML para objetos GOA, podemos concluir que o servidor GOA pode estar armazenando quaisquer tipos de bases XML, desde uma base de índices (XIx) até uma base de esquemas (XSD), passando por uma base de documentos XML instanciados a partir de esquemas XML. O servidor GOA está implementado na linguagem C++ e a API cliente está disponível em C++ e Java. Porém, esta capacidade de armazenar documentos XML só está presente na versão do cliente em Java.

No Capítulo 2, foi apresentado o esquema de classes do DOM e um exemplo de mapeamento de um documento XML nestas estruturas. Portanto, torna-se necessária a criação destas classes para que o *Goaxe* seja capaz de realizar um *parser* nos documentos XML e criar os objetos correspondentes nas classes do DOM. A Figura 41 apresenta o *script* de criação destas classes utilizando a nomenclatura do GOA.

Neste momento, a API *Goaxe* já se encontra habilitada para manipular os documentos XML de entrada para criar os objetos no GOA. O documento é então carregado para a memória para posteriormente começarmos a criar os objetos correspondentes no GOA. O documento, neste momento, começa a ser percorrido em profundidade com o percurso em pré-ordem e criando objetos da classe *GoaObject*, que contém todos os objetos que correspondem aos elementos e dados do documento XML. A criação dos objetos termina quando todos os nós do documento tiverem sido percorridos. A Figura 42 apresenta o modelo de classes da API Cliente GOA, que contém a classe *GoaObject* citada anteriormente. Outros detalhes em relação ao armazenamento de documentos XML no GOA podem ser encontrados em (MATTOSO, CAVALCANTI et al., 2002).

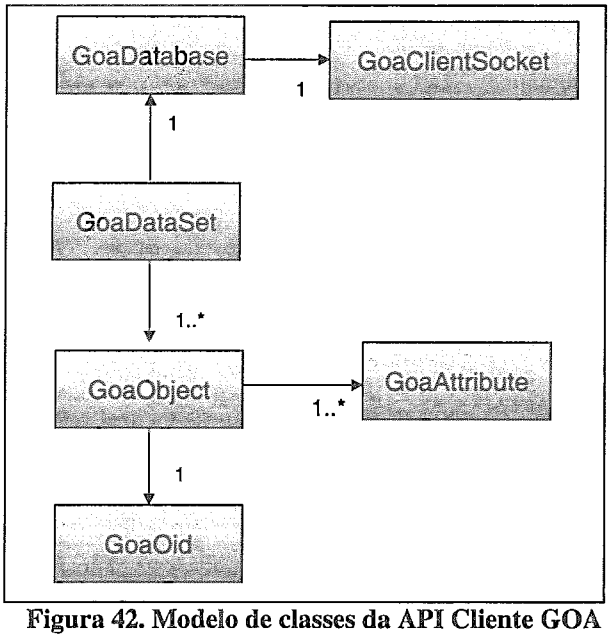


Figura 42. Modelo de classes da API Cliente GOA

Ao término da criação destes objetos, todo o documento XML já está mapeado nas classes do DOM, podendo ser enviados para o servidor realizar a persistência. Uma vez armazenados no servidor, já somos capazes de realizar qualquer consulta para

recuperarmos a informação desejada, desde que conheçamos a estrutura de classes do DOM.

4.2 - Geração da representação XML intermediária

A seção 4.1 explicou o processo de armazenamento dos documentos XML no GOA. Apesar de já termos acesso a todos os dados armazenados através de consultas sobre a base, ainda somos obrigados a conhecer a estrutura de classes do DOM para podermos recuperar as informações corretamente através da linguagem de consulta do SGBD. Nesta seção e na seguinte, nós resolvemos este problema através da tradução das consultas XQuery de entrada em consultas SQL3 que serão executadas no GOA sobre as classes do DOM.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CONECTIVO (#PCDATA)>
<!ELEMENT CONECTIVOS (CONECTIVO)>
<!ELEMENT CURSOR-F (#PCDATA)>
<!ELEMENT CURSOR-R (#PCDATA)>
<!ELEMENT CURSOR-W (#PCDATA)>
<!ELEMENT DOCUMENT-F (#PCDATA)>
<!ELEMENT DOCUMENTS-F (DOCUMENT-F)>
<!ELEMENT ELEMENT (#PCDATA)>
<!ATTLIST ELEMENT
        TYPE (real | virtual) #IMPLIED>
<!ELEMENT ELEMENT-F (#PCDATA)>
<!ELEMENT ELEMENT-R (#PCDATA)>
<!ELEMENT ELEMENT-W (#PCDATA)>
<!ELEMENT ELEMENTS-F (ELEMENT-F+)>
<!ELEMENT ELEMENTS-R (ELEMENT-R+)>
<!ELEMENT ELEMENTS-W (ELEMENT-W+)>
<!ELEMENT FOR (PATH-F)>
<!ELEMENT LET EMPTY>
<!ELEMENT PATH (ELEMENT+)>
<!ELEMENT PATH-AUX (ELEMENT+)>
<!ELEMENT PATH-F (CURSOR-F, DOCUMENTS-F, ELEMENTS-F)>
<!ELEMENT PATH-R (CURSOR-R, ELEMENTS-R)>
<!ELEMENT PATH-W (CURSOR-W, ELEMENTS-W, VALUE-W)>
<!ELEMENT PATHS (PATH)>
<!ELEMENT PATHS-AUX (PATH-AUX+)>
<!ELEMENT PATHS-W (PATH-W+)>
<!ELEMENT RETURN (PATH-R)>
<!ELEMENT VALUE-W (#PCDATA)>
<!ATTLIST VALUE-W
        OPERATOR-W CDATA #REQUIRED>
<!ELEMENT WHERE (CONECTIVOS, PATHS-W)>
<!ELEMENT XQUERY (PATHS, PATHS-AUX, FOR, LET, WHERE, RETURN)>
```

Figura 43. DTD da representação da XQuery no formato XML intermediário

Conforme explicado anteriormente, nós decidimos criar uma representação XML intermediária da consulta XQuery de entrada para sermos capazes de utilizar o XSLT para realizar a transformação. Desta forma, a solução fica mais flexível e

modular, tornando menos complexa a tarefa de realizar alguma alteração tanto na abordagem de armazenamento quanto na linguagem alvo de consulta.

O esquema desta representação XML intermediária foi construído com o objetivo de facilitar a aplicação das regras de transformação da etapa seguinte. Portanto, dependendo da abordagem de armazenamento utilizada e da linguagem alvo de consulta, esta representação intermediária pode sofrer pequenas alterações em seu esquema. A Figura 43 apresenta o DTD proposto por nós para representar em formato XML a consulta XQuery de entrada.

A Figura 44 mostra um algoritmo para transformar a consulta XQuery inicial no seu formato XML intermediário equivalente.

```
InsereTagEntrada(XQUERY)
Lê a cláusula FOR
InsereTagEntrada(PATHS)
InsereTagEntrada(PATH)
Se não existir operador "//" então
  Para cada elemento e pertencente à expressão de caminho faça
    InsereTagEntrada(ELEMENT, TYPE, "real")
    InsereElemento(e)
    InsereTagSaida(ELEMENT)
Senão
  n = Altura da árvore - nº elementos distintos na expressão de caminho + 1
  Para cada elemento e pertencente à expressão de caminho faça
  Se operador = "/" então
    InsereTagEntrada(ELEMENT, TYPE, "real")
    InsereElemento(e)
    InsereTagSaida(ELEMENT)
  Senão Se operador = "//" então
     Repetir n vezes
       InsereTagEntrada(ELEMENT, TYPE, "virtual")
       InsereElemento(e)
       InsereTagSaida(ELEMENT)
InsereTagSaida(PATH)
InsereTagSaida(PATHS)
InsereTagEntrada(PATHS-AUX)
Se não existir operador "//" então
  Para cada elemento e pertencente à expressão de caminho faça
    InsereTagEntrada(PATH_AUX)
    InsereTagEntrada(ELEMENT, TYPE, "real")
    InsereElemento(e)
    InsereTagSaida(ELEMENT)
    InsereTagSaida(PATH_AUX)
Senão
  n = Altura da árvore - nº elementos distintos na expressão de caminho + 1
  {Existem n expressões de caminhos distintas ao utilizarmos o "//"
  Quando um nível precisar ser representado, utilizamos o símbolo "∃"
  A expressão de caminho a/b//c possui os seguintes caminhos possíveis, considerando
  como 4 a altura da árvore: a/b/c ou a/b/∃/c}
  Para cada expressão de caminho possível faça
    InsereTagEntrada(PATH_AUX)
    Para cada elemento e pertencente à expressão de caminho faça
      Se e = "∃" então
        InsereTagEntrada(ELEMENT, TYPE, "virtual")
        InsereElemento("")
        InsereTagSaida(ELEMENT)
      Senão
        InsereTagEntrada(ELEMENT, TYPE, "real")
        InsereElemento(e)
        InsereTagSaida(ELEMENT)
    InsereTagSaida(PATH_AUX)
InsereTagSaida(PATHS-AUX)
InsereTagEntrada(FOR)
```

```
InsereTagEntrada(PATH-F)
InsereTagEntrada(CURSOR-F)
InsereElemento(cursor) (cursor lido da cláusula FOR)
InsereTagSaida(CURSOR-F)
InsereTagEntrada(DOCUMENTS-F)
InsereTagEntrada(DOCUMENT-F)
InsereElemento(documento) (documento lido da cláusula FOR)
InsereTagSaida(DOCUMENT-F)
InsereTagSaida(DOCUMENTS-F)
InsereTagEntrada(ELEMENTS-F)
Se não existir operador "//" na expressão de caminho então
  Para cada elemento e pertencente à expressão de caminho com exceção do último faça
    InsereTagEntrada(ELEMENT-F)
    InsereElemento(e)
    InsereTagSaida(ELEMENT-F)
Senão
  Enquanto não encontrar o operador "//" faça
    InsereTagEntrada(ELEMENT-F)
    InsereElemento(e)
    InsereTagSaida(ELEMENT-F)
  c = altura da árvore - nº elementos distintos na expressão de caminho
  Repetir c vezes
    InsereTagEntrada(ELEMENT-F)
    InsereElemento("")
    InsereTagSaida(ELEMENT-F)
InsereTagSaida(ELEMENTS-F)
InsereTagSaida(PATH-F)
InsereTagSaida(FOR)
Lê a cláusula LET
InsereTagEntrada(LET)
// Não implementado
InsereTagSaida(LET)
InsereTagEntrada(CONECTIVOS)
  n = Altura da árvore - nº elementos distintos na expressão de caminho + 1
For i = 1 to n faça
  Para cada conectivo pertencente à cláusula WHERE faça
    InsereTagEntrada(CONECTIVO)
    InsereElemento(conectivo) (conectivo lido do predicado de consulta)
    InsereTagSaida(CONECTIVO)
    (Conectivos para separar às diversas possibilidades da posição
     de cada elemento ao utilizar o "//")
  Se i <> n então
    InsereTagEntrada(CONECTIVO)
    InsereElemento(OR)
    InsereTagSaida(CONECTIVO)
InsereTagSaida(CONECTIVOS)
InsereTagEntrada(PATHS-W)
Se não existir operador "//" na expressão de caminho então
  Para cada predicado de consulta faça
    InsereTagEntrada(PATH-W)
    InsereTagEntrada(CURSOR-W)
    InsereElemento(cursor) {cursor lido da cláusula FOR}
    InsereTagSaida(CURSOR-W)
    InsereTagEntrada(ELEMENTS-W)
    InsereTagEntrada(ELEMENT-W)
    InsereElemento(e) {inserir o último elemento da expressão de caminho}
    InsereTagSaida(ELEMENT-W}
    InsereTagSaida(ELEMENTS-W)
    InsereTagEntrada(VALUE-W, OPERATOR-W, "operador") {operador lido do predicado}
    InsereElemento(valor) {valor presente a direita do operador}
    InsereTagSaida(VALUE-W)
    InsereTagSaida(PATH-W)
Senão
  d = Altura da árvore - nº elementos a direita do "//" na expressão de caminho
  Para i = 1 até d faça
    Para cada predicado de consulta faça
      InsereTagEntrada(PATH-W)
      InsereTagEntrada(CURSOR-W)
      InsereElemento(cursor) {cursor lido da cláusula FOR}
      InsereTagSaida(CURSOR-W)
      InsereTagEntrada(ELEMENTS-W)
      Repetir  i-1 vezes
        InsereTagEntrada(ELEMENT-W)
        InsereElemento("")
        InsereTagSaida(ELEMENT-W)
      InsereTagEntrada(ELEMENT-W)
      InsereElemento(e) {inserir o último elemento da expressão de caminho}
```

```
      InsereTagSaida(ELEMENT-W)
      InsereTagSaida(ELEMENTS-W)
      InsereTagEntrada(VALUE-W, OPERATOR-W, "operador") {operador lido do predicado}
      InsereElemento(valor) {valor presente a direita do operador}
      InsereTagSaida(VALUE-W)
      InsereTagSaida(PATH-W)
InsereTagSaida(PATHS-W)
InsereTagSaida(WHERE)
Lê a cláusula RETURN
InsereTagEntrada(RETURN)
InsereTagEntrada(PATH-R)
InsereTagEntrada(CURSOR-R)
InsereElemento(cursor) {cursor lido da cláusula FOR}
InsereTagSaida(CURSOR-R)
InsereTagEntrada(ELEMENTS-R)
Para cada elemento e pertencente à expressão de caminho da cláusula RETURN faça
  InsereTagEntrada(ELEMENT-R)
  InsereElemento(e)
  InsereTagSaida(ELEMENT-R)
InsereTagSaida(ELEMENTS-R)
InsereTagSaida(PATH-R)
InsereTagSaída(RETURN)
InsereTagSaida(XQUERY)
```

Figura 44. Algoritmo para construção do documento XML intermediário

Para exemplificar a simplicidade desta representação, daremos um exemplo de uma consulta XQuery de entrada e sua correspondente representação XML intermediária. Conforme pode ser visto na Figura 45, esta representação nada mais é que uma maneira diferente de escrever a XQuery de entrada, com algumas modificações que nos ajudarão ao implementarmos as regras de transformação no documento XSL. É importante ressaltar que ao gerar esta representação intermediária, já sabemos qual a altura da árvore, já que armazenamos esta informação em uma classe do DOM. Este é um dos exemplos mais complicados implementados pelo *XVerter*, já que contém o operador "//" e dois predicados de consulta na cláusula WHERE.

```
for $n in document("recados.xml")/recados//data/dia
where ($n > 1) and ($n < 5)
return $n
```

(a) Consulta XQuery

```
<XQUERY>
  <PATHS>
    <PATH>
      <ELEMENT>recados</ELEMENT>
      <ELEMENT>data</ELEMENT>
      <ELEMENT>data</ELEMENT>
      <ELEMENT>dia</ELEMENT>
    </PATH>
  </PATHS>
  <PATHS-AUX>
    <PATH-AUX>
      <ELEMENT TYPE="real">recados</ELEMENT>
      <ELEMENT TYPE="real">data</ELEMENT>
      <ELEMENT TYPE="real">dia</ELEMENT>
    </PATH-AUX>
    <PATH-AUX>
      <ELEMENT TYPE="real">recados</ELEMENT>
      <ELEMENT TYPE="virtual"/>
      <ELEMENT TYPE="real">data</ELEMENT>
      <ELEMENT TYPE="real">dia</ELEMENT>
    </PATH-AUX>
  </PATHS-AUX>
```

```
<FOR>
  <PATH-F>
    <CURSOR-F>$n</CURSOR-F>
    <DOCUMENTS-F>
      <DOCUMENT-F>recados.xml</DOCUMENT-F>
    </DOCUMENTS-F>
    <ELEMENTS-F>
      <ELEMENT-F>recados</ELEMENT-F>
      <ELEMENT-F/>
    </ELEMENTS-F>
  </PATH-F>
</FOR>
<LET/>
<WHERE>
  <CONECTIVOS>
    <CONECTIVO>and</CONECTIVO>
    <CONECTIVO>or</CONECTIVO>
    <CONECTIVO>and</CONECTIVO>
  </CONECTIVOS>
  <PATHS-W>
    <PATH-W>
      <CURSOR-W>$n</CURSOR-W>
      <ELEMENTS-W>
        <ELEMENT-W>dia</ELEMENT-W>
      </ELEMENTS-W>
      <VALUE-W OPERATOR-W="&gt;">1</VALUE-W>
    </PATH-W>
    <PATH-W>
      <CURSOR-W>$n</CURSOR-W>
      <ELEMENTS-W>
        <ELEMENT-W>dia</ELEMENT-W>
      </ELEMENTS-W>
      <VALUE-W OPERATOR-W="&lt;">5</VALUE-W>
    </PATH-W>
    <PATH-W>
      <CURSOR-W>$n</CURSOR-W>
      <ELEMENTS-W>
        <ELEMENT-W>data</ELEMENT-W>
        <ELEMENT-W>dia</ELEMENT-W>
      </ELEMENTS-W>
      <VALUE-W OPERATOR-W="&gt;">1</VALUE-W>
    </PATH-W>
    <PATH-W>
      <CURSOR-W>$n</CURSOR-W>
      <ELEMENTS-W>
        <ELEMENT-W>data</ELEMENT-W>
        <ELEMENT-W>dia</ELEMENT-W>
      </ELEMENTS-W>
      <VALUE-W OPERATOR-W="&lt;">5</VALUE-W>
    </PATH-W>
  </PATHS-W>
</WHERE>
<RETURN>
  <PATH-R>
    <CURSOR-R>$n</CURSOR-R>
  </PATH-R>
</RETURN>
</XQUERY>
```

(b) Representação XML Intermediária

Figura 45. Exemplo de uma XQuery e sua correspondente representação XML intermediária

Esta etapa de transformar a consulta de entrada XQuery em uma representação XML intermediária está sendo feita manualmente, pois não se encontra implementada. Porém, a complexidade desta implementação é baixa, sendo basicamente a reescrita de uma XQuery no formato XML intermediário. Para a nossa arquitetura, seria interessante desenvolver esta implementação em Java, para manter todo o cliente GOA no mesmo

ambiente. Porém, conforme dito anteriormente, esta etapa pode ser implementada utilizando qualquer linguagem de programação.

4.3 - Aplicação das regras de transformação

Ao final da etapa descrita na seção 4.2, o tradutor *XVerter* precisará realizar a transformação propriamente dita, que é a sua tarefa principal. Para tal, ele associará um documento XSL que contém as regras de transformação à representação XML intermediária resultante da etapa anterior. A seguir, mostraremos alguns trechos do documento XSL que foi construído, analisando os principais pontos do código. É importante ressaltar que a saída do tradutor *XVerter* obedece à sintaxe do SGBD utilizado no experimento, que no nosso caso é o GOA. Outro detalhe é que a cláusula LET não se encontra implementada, assim como as consultas XQuery contendo várias expressões de caminho distintas.

A Figura 46 apresenta o início da tradução, onde primeiro será traduzida a cláusula RETURN da XQuery e em seguida as cláusulas FOR e WHERE. A explicação para esta inversão no processamento se deve ao fato de a cláusula RETURN ser a responsável por gerar a projeção da SQL3. Por esta razão, ela é a primeira a ser processada. Em seguida, a cláusula FOR será processada, sendo responsável por construir a cláusula FROM e uma parte da cláusula WHERE da SQL3. E finalmente, processaremos a cláusula WHERE da XQuery, que completará a geração da cláusula WHERE da SQL3, que foi iniciada pela cláusula FOR.

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="XQUERY">
  <xsl:apply-templates select="RETURN"/>
  <xsl:apply-templates select="FOR"/>
  <xsl:apply-templates select="WHERE"/>
</xsl:template>
```

Figura 46. Trecho de código do XSL

A Figura 47 apresenta o início da construção da SQL3 através da cláusula RETURN da XQuery. Como podemos perceber, a cláusula SELECT foi inserida na sintaxe do GOA e o comando `<xsl:apply-templates/>` foi inserido para continuar o processamento da representação XML intermediária da XQuery.

```
<!--Inicio da transformacao da clausula RETURN-->
<xsl:template match="RETURN">
  <xsl:text>SELECT resultado(</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>)</xsl:text>
</xsl:template>
```

Figura 47. Trecho de código do XSL

A construção da cláusula SELECT da SQL3 pode ser basicamente de dois tipos: a primeira quando o operador "//" é utilizado na XQuery de entrada e o segundo quando este operador não é utilizado. No primeiro caso, conforme explicado durante a apresentação das regras no Capítulo 3, a projeção terá que levar em consideração que o elemento que se encontra logo após o operador, pode estar presente em um ou mais níveis da árvore. A Figura 48(a) e a Figura 48(b) apresentam os *templates* que servem como base para a construção dos dois tipos de cláusula SELECT descritos acima.

```
<!--Template para a construcao da projecao ao utilizar o "//"-->
<xsl:template name="projecoesVirtuais">
  <!--Declaracao dos parametros-->
  <xsl:param name="contador"/>
  <xsl:param name="tipoElementoAnterior"/>
  <!--Declaracao das variaveis-->
  <xsl:variable name="cursorElemento">c</xsl:variable>
  <!--Corpo do template-->
  <xsl:if test="$tipoElementoAnterior = 'virtual'">
    <xsl:text>, </xsl:text>
  </xsl:if>
  <xsl:text>result</xsl:text>
  <xsl:value-of select="$contador"/>
  <xsl:text>: </xsl:text>
  <xsl:value-of select="$cursorElemento"/>
  <xsl:value-of select="$contador"/>
  <xsl:text>.value</xsl:text>
</xsl:template>
```

(a) Template de projeção utilizando o operador "//"

```
<!--Template para a construcao da projecao sem utilizar o "//"-->
<xsl:template name="projecoesReais">
  <!--Declaracao dos parametros-->
  <xsl:param name="contador"/>
  <!--Declaracao das variaveis-->
  <xsl:variable name="cursorElemento">c</xsl:variable>
  <!--Corpo do template-->
  <xsl:text>result:</xsl:text>
  <xsl:value-of select="$cursorElemento"/>
  <xsl:value-of select="$contador"/>
  <xsl:text>.value </xsl:text>
</xsl:template>
```

(b) Template de projeção sem utilizar o operador "//"

Figura 48. Trecho de código do XSL

Uma vez construída a cláusula SELECT, podemos continuar o processamento a fim de gerar a cláusula FROM da SQL3 a partir da cláusula FOR da XQuery. A Figura 49 apresenta o código que inicia esta construção, primeiro gerando os cursores

referentes aos documentos utilizados na XQuery e depois construindo os cursores referentes aos elementos da consulta de entrada.

```
<!--Inicio da transformacao da clausula FOR-->
<xsl:template match="FOR">
  <xsl:text> FROM </xsl:text>
  <xsl:apply-templates select="PATH-F/DOCUMENTS-F"/>
  <xsl:apply-templates select="ancestor::XQUERY/PATHS"/>
</xsl:template>
```

Figura 49. Trecho de código do XSL

O *template* que é o responsável pela declaração dos cursores dos documentos e dos elementos pode ser visualizado na Figura 50. Este *template* constrói expressões do tipo "d_i in DocumentImpls" e "c_i in c_{i-1}.childrenNode", sendo c_0 = d. Desta forma, declaramos todos os cursores necessários que serão utilizados pelas cláusulas SELECT e WHERE da SQL3.

```
<!--Template para a construcao das declaracoes dos cursores-->
<xsl:template name="cursores">
  <!--Declaracao dos parametros-->
  <xsl:param name="cursor"/>
  <xsl:param name="contadorAtual"/>
  <xsl:param name="contadorAnterior"/>
  <!--Declaracao das variaveis-->
  <xsl:variable name="cursorElemento">c</xsl:variable>
  <xsl:variable name="cursorDocumento">d</xsl:variable>
  <!--Corpo do template-->
  <xsl:choose>
    <xsl:when test="$cursor='d' ">
      <!--Declaracao do cursor da raiz do documento (d1 in DocumentImpls)-->
      <xsl:value-of select="$cursor"/>
      <xsl:value-of select="$contadorAtual"/>
      <xsl:text> in DocumentImpls </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <!--Declaracao dos cursores dos elementos-->
      <!--Virgula para separar as declaracoes dos cursores-->
      <xsl:text>, </xsl:text>
      <xsl:value-of select="$cursor"/>
      <xsl:value-of select="$contadorAtual"/>
      <xsl:text> in </xsl:text>
      <xsl:choose>
        <xsl:when test="$contadorAtual=1">
          <!--Declaracao do primeiro cursor(c1 in d1.childrenNode)-->
          <xsl:value-of select="$cursorDocumento"/>
          <xsl:value-of select="$contadorAtual"/>
          <xsl:text>.childrenNode </xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <!--Declaracao dos cursores seguintes-->
          <xsl:value-of select="$cursorElemento"/>
          <xsl:value-of select="$contadorAnterior"/>
          <xsl:text>.childrenNode </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figura 50. Trecho de código do XSL

Além de ser a responsável por gerar a cláusula FROM da SQL3, a cláusula FOR da XQuery também constrói uma parte da cláusula WHERE da consulta de saída. Conforme explicado nas regras de transformação, precisamos declarar na cláusula WHERE da SQL3 todos os documentos e elementos que aparecem na consulta XQuery de entrada. A Figura 51 mostra o *template* utilizado para construir estas expressões na consulta SQL3 de saída. Estas expressões serão do tipo "(d$_i$.name = "nomeDocumento")" e "(c$_i$.name = "nomeElemento")".

```
<!--Template para a construcao das condicoes definidas implicitamente na clausula
FROM(.name) e explicitamente na clausula WHERE(.value)-->
<xsl:template name="condicoes">
  <!--Declaracao dos parametros-->
  <xsl:param name="cursor"/>
  <xsl:param name="contador"/>
  <xsl:param name="nome"/>
  <xsl:param name="atributo"/>
  <xsl:param name="operador"/>
  <xsl:param name="tipo"/>
  <!--Corpo do template-->
  <xsl:if test="$tipo = 'real'">
    <xsl:text> ( </xsl:text>
    <xsl:value-of select="$cursor"/>
    <xsl:value-of select="$contador"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="$atributo"/>
    <xsl:value-of select="$operador"/>
    <xsl:text>"</xsl:text>
    <xsl:value-of select="$nome"/>
    <xsl:text>"</xsl:text>
    <xsl:text> ) </xsl:text>
  </xsl:if>
</xsl:template>
```

Figura 51. Trecho de código do XSL

Finalmente, para terminar a geração da SQL3 de saída, falta apenas construir os predicados de consulta presentes na cláusula WHERE da XQuery de entrada. A Figura 52 mostra o início deste processamento.

```
<!--Inicio da transformacao da clausula WHERE-->
<xsl:template match="WHERE">
  <xsl:apply-templates select="PATHS-W"/>
</xsl:template>
```

Figura 52. Trecho de código do XSL

O *template* utilizado para a geração destes predicados é o mesmo daquele utilizado para construir as expressões relativas à declaração dos documentos e elementos que aparecem na XQuery de entrada, e pode ser visualizado na Figura 51. A única diferença é que na geração das expressões relativas à cláusula WHERE da XQuery de entrada, são construídas expressões do tipo "(c$_i$.value = "valor")",

que não eram geradas na outra parte do processamento. Expressões do tipo "`(ci.name = "nomeElemento")`" também são construídas nesta etapa da transformação.

Vimos, de uma maneira geral, a forma pela qual o documento XSL foi construído para gerar como saída uma SQL3 utilizando o XSLT. O código completo deste documento pode ser encontrado no Apêndice A.

Capítulo 5 - Avaliação do XVerter

No Capítulo 4, detalhamos a implementação da arquitetura e do tradutor do *XVerter*, incluindo o armazenamento, geração da representação intermediária e finalmente a transformação desta representação na SQL3 de saída. Portanto, para validar esta implementação, decidimos utilizar algum *benchmark* com o intuito de verificar se a nossa proposta satisfaz às principais características da XQuery. Existem na literatura alguns *benchmarks* para XML, a saber X-Mach (BOHME & RAHM, 2001), X-OO7 (BRESSAN, DOBBIE et al., 2001) e XMark (SCHMIDT, WAAS et al., 2001; SCHMIDT, WAAS et al., 2002) cada um focando em características diferentes. O X-Mach é voltado para desempenho em concorrência, que não é o nosso foco. O XOO7 é voltado para OO, assim achamos que poderia ser favoravelmente tendencioso, já que a característica principal do *XVerter* são as expressões de caminho. O escolhido foi o X-Mark (SCHMIDT, WAAS et al., 2002) por ser mais abrangente e se dedicar a avaliar desafios da XQuery, e não necessariamente consultas típicas. Também foi publicado um relatório técnico (SCHMIDT, WAAS et al., 2001) sobre o X-Mark, antes mesmo dele possuir este nome.

Como a XQuery é uma linguagem relativamente nova e a sua implementação ainda está evoluindo nos produtos que possuem a capacidade de processá-la, decidimos também avaliar a XQuery em dois dos inúmeros processadores existentes: Quip (SOFTWARE AG, 2002b) e XQuery Demo (MICROSOFT .NET HOME, 2002). Escolhemos o Quip pelo fato de a sua empresa, AG Software, ser a mesma que desenvolve o Tamino (SOFTWARE AG, 2002a), um dos mais famosos SGBDs Nativos de XML existentes. E o XQuery Demo foi o outro escolhido por ser o processador desenvolvido pela Microsoft.

Nas seções seguintes, faremos uma descrição do *benchmark* XMark utilizado e analisaremos o *XVerter*, o Quip e o XQuery Demo nos baseando nas características que o XMark apresenta como sendo as principais da XQuery.

5.1 - Descrição do *Benchmark* XMark

Para um *benchmark* ter a capacidade de analisar determinada linguagem, é necessário que seja utilizado um modelo para que as consultas sejam baseadas nele.

Apresentamos a seguir, o conjunto de entidades que compõem o modelo utilizado no *benchmark*, que é de um *site* de leilões na Internet. As principais entidades são: *person*, *open auction*, *closed auction*, *item* e *category*. Os relacionamentos entre eles são expressos através de referências, com exceção de *annotations* e *descriptions*, cujos conteúdos são textos em linguagem natural. A Tabela 4 apresenta uma breve descrição das entidades citadas acima, a Figura 53 mostra o esquema hierárquico do modelo e na Figura 54 estão apresentadas as referências para realizar a conexão entre as sub-árvores do esquema.

Tabela 4. Descrição das principais entidades do modelo do *benchmark*

Entidade	Descrição
Person	É caracterizado pelo nome, *email*, telefone, endereço, *homepage*, número do cartão de crédito, perfil com seus interesses e um conjunto de leilões em aberto que ele tem acesso.
Open Auction	É um leilão que ainda está em aberto. Suas propriedades são um *status*, o histórico de ofertas com referências para o licitante e o vendedor, a oferta atual, a média de aumento no valor da oferta, o tipo de leilão, o intervalo de tempo permitido para que a oferta seja aceita, o *status* da transação e a referência para o item que está sendo vendido.
Closed Auction	É um leilão que já terminou. Suas propriedades são o vendedor (uma referência para a pessoa), o comprador (uma referência para a pessoa), uma referência para o item respectivo, o preço, a quantidade de itens vendidos, a data que a compra foi efetuada, o tipo de transação e as anotações que foram feitas antes, durante e depois do processo de leilão.
Item	É o objeto que está à venda ou que já foi vendido. Cada item possui um único identificador e possui propriedades como pagamento (cartão de crédito, dinheiro, ...), uma referência para o vendedor, uma descrição, ... A cada item é associado a uma região do mundo, que representa o seu pai.
Category	Possui um nome e uma descrição; são utilizados para implementar classificação dos itens.

Os autores também desenvolveram um gerador de documentos XML baseado no esquema do *benchmark*, chamado *xmlgen*. Este gerador possui a capacidade de construir documentos com diferentes tamanhos, capacidade esta que é controlada por um fator (fator 1 equivale a um documento de aproximadamente 100Mb) e funciona em qualquer plataforma. Para gerar os textos das entidades que correspondem à linguagem natural, foram utilizadas as 17 mil palavras mais comuns das peças de Shakeaspeare. Para gerar os endereços, foram usadas listas telefônicas disponíveis e assim sucessivamente para os outros elementos.

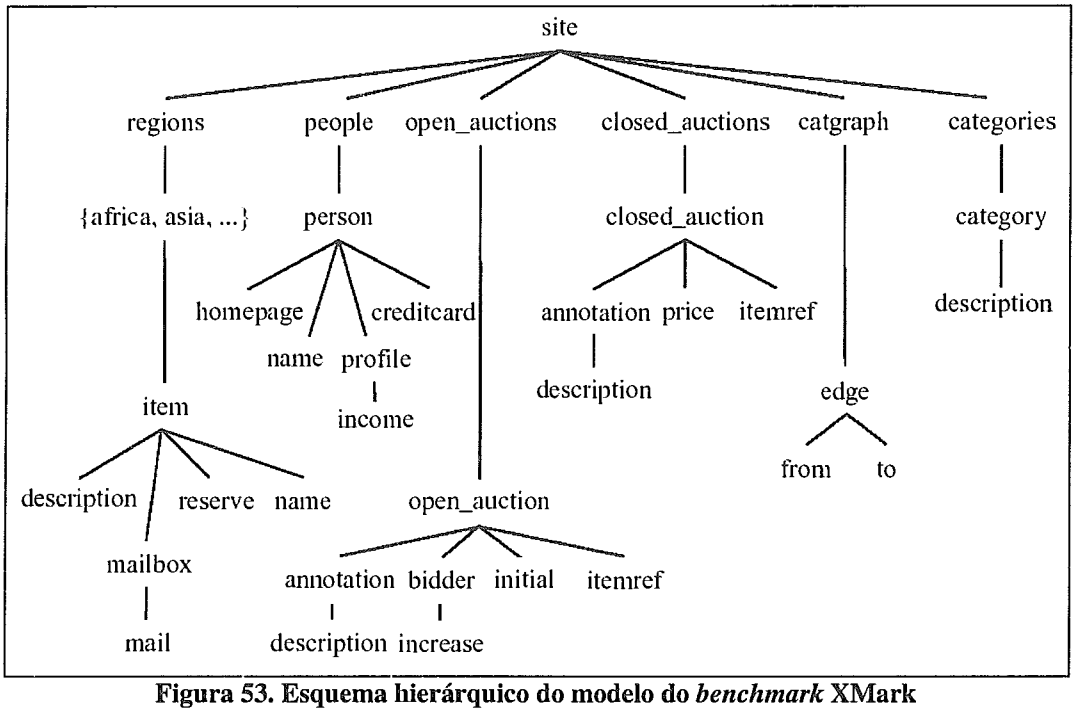


Figura 53. Esquema hierárquico do modelo do *benchmark* XMark

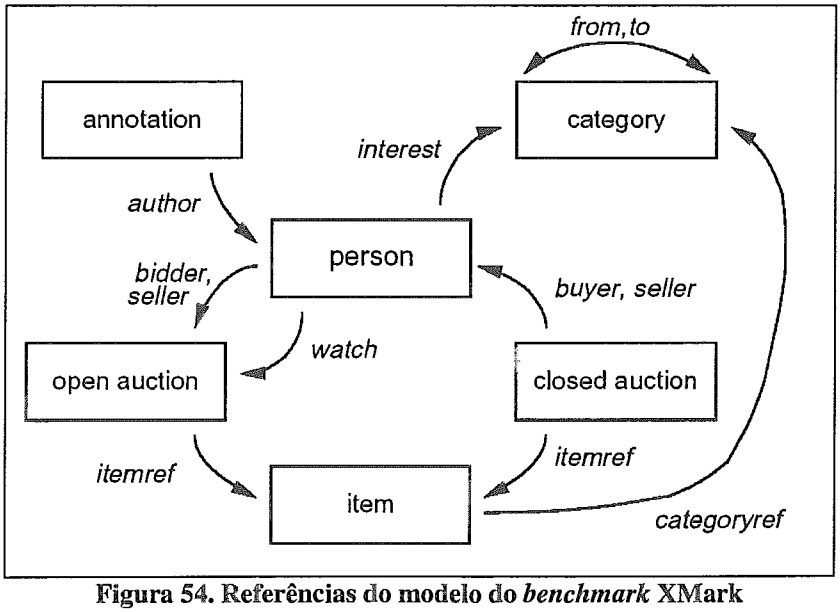


Figura 54. Referências do modelo do *benchmark* XMark

Na seção 5.2, apresentamos as consultas que compõem o *benchmark* e as análises referentes ao *XVerter* e aos dois processadores que utilizamos para os testes.

5.2 - Consultas do *Benchmark* XMark

Com o objetivo de avaliar da melhor forma os processadores da XQuery, os autores apresentam vinte consultas, divididas em quatorze características significativas da linguagem. Para cada uma destas características, apresentamos uma análise do nosso tradutor *XVerter* e dos dois processadores utilizados para o teste, que são o Quip e o XQuery Demo, conforme explicado anteriormente.

Outro ponto a ser ressaltado é que ao contrário de outros *benchmarks* que contêm consultas típicas e mais freqüentes, o XMark pretende ser um *benchmark* que vem a cobrir todo o espectro de características da XQuery.

É importante ressaltar que o tradutor *XVerter* possui regras para a tradução do núcleo da linguagem XQuery, que são a navegação sobre as expressões de caminho, as expressões FLWR e o operador "//". Dentre estas, não estão implementadas as consultas que possuem várias expressões de caminho distintas e a tradução do operador LET. Entretanto, somos capazes de executar consultas sobre as classes do DOM para recuperar quaisquer informações necessárias, já que o DOM consegue representar totalmente os documentos XML.

5.2.1 - Consulta por valor exato

Esta é uma das características mais simples presentes no *benchmark* e consiste em testar a capacidade em percorrer um caminho e acessar um valor de determinado elemento ou atributo. A Figura 55 apresenta a consulta Q1 sugerida para avaliar esta característica.

```
Q1. Retornar o nome do item que possui ID igual a 'item20748' e está registrado na
América do Norte.
```

```
FOR $b IN document("auction.xml")/site/regions/namerica/item[@id="item20748"]
RETURN $b/name/text()
```

Figura 55. Consulta Q1

O *XVerter* atende a esta característica, porém, a consulta Q1 utiliza duas expressões de caminho distintas A regra 3 é responsável por traduzir expressões de

caminho para a navegação de classes DOM. Entretanto a implementação de duas expressões de caminho no arquivo XSL não foi ainda contemplada. Sendo assim, essa consulta não será traduzida de modo correspondente. Contudo, com a pequena modificação na XQuery apresentada na Figura 56, o tradutor *XVerter* consegue transformá-la para a SQL3 correspondente. É importante ressaltar que as regras para tradução de consultas com várias expressões de caminho distintas já se encontram definidas, faltando apenas a sua implementação.

```
FOR $b IN document("auction.xml")/site/regions/namerica/item/@id
WHERE $b = "item20748"
RETURN $b
```

Figura 56. Consulta Q1 modificada

Esta característica também se encontra implementada no Quip, porém a consulta Q1 não funciona pois o processador por algum motivo se confunde com o elemento *item*, achando que ele é uma palavra reservada.

No XQuery Demo, a consulta Q1 funcionou normalmente.

5.2.2 - Acesso ordenado

Este item avalia a capacidade de lidar com a ordem dos elementos nos documentos XML, que é uma característica importante ao manipularmos dados XML. O *benchmark* apresenta três consultas para avaliar este item, que estão apresentadas na Figura 57, Figura 58 e Figura 59.

```
Q2. Retornar os aumentos iniciais de todos os leilões em aberto.
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
RETURN <increase> $b/bidder[1]/increase/text() </increase>
```

Figura 57. Consulta Q2

```
Q3. Retornar o primeiro e o atual aumentos de todos os leilões em aberto cujo aumento
atual seja no mínimo duas vezes o aumento inicial.
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
WHERE $b/bidder[0]/increase/text() * 2 <= $b/bidder[last()]/increase/text()
RETURN <increase first=$b/bidder[0]/increase/text()
last=$b/bidder[last()]/increase/text()>
```

Figura 58. Consulta Q3

```
Q4. Listar as reservas dos leilões em aberto onde certa pessoa inseriu uma oferta antes
de outra pessoa.
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
WHERE $b/bidder/personref[id="person18829"] BEFORE
$b/bidder/personref[id="person10487"]
RETURN <history> $b/initial/text() </history>
```

Figura 59. Consulta Q4

O *XVerter* ainda não contém regras para tratar operadores específicos de acesso ordenado aos elementos. Porém, como o DOM armazena os documentos preservando a sua estrutura original, é possível que regras sejam definidas para tratar esta característica da XQuery.

Já o Quip consegue executar a consulta Q2 desde que a expressão de caminho contida na *tag name* esteja entre chaves, conforme especificação da XQuery definida em (WORLD WIDE WEB CONSORTIUM (W3C), 2002c). Como nenhuma consulta do *benchmark* faz uso destas chaves, iremos ignorar este comentário para as demais consultas, apesar de ser necessária a sua inclusão para que elas executem corretamente. Outro detalhe é que o primeiro elemento é representado por [1] na especificação da XQuery, enquanto na consulta Q3 do *benchmark* ele é representado pelo [0]. Apesar de alterar estes detalhes, o processador não conseguiu executar a consulta Q3, pois a operação de multiplicação na cláusula WHERE deu erro. O Quip também não conseguiu construir um atributo no resultado complexo da cláusula RETURN. A consulta Q4 não pôde ser executada pois a cláusula BEFORE não é reconhecida.

Para que esta característica seja suportada pelo XQuery Demo, é necessário que a expressão de caminho presente antes do [] esteja entre parênteses. Com esta modificação, a consulta Q2 executou corretamente, mas a Q3 não, por possuir uma multiplicação na cláusula WHERE e um acesso ordenado ao construir um atributo na cláusula RETURN. O BEFORE da consulta Q4 também não funcionou.

5.2.3 - *Casting*

Esta característica avalia a capacidade de transformar um tipo de dados em outro. A Figura 60 mostra a consulta do *benchmark* referente a este item.

```
Q5.  Quantos itens vendidos custam mais de 40?
COUNT (FOR $i IN document("auction.xml")/site/closed_auctions/closed_auction
WHERE $i/price/text() >= 40
RETURN $i/price)
```

Figura 60. Consulta Q5

Na verdade, não é preciso gerar nenhuma regra no *XVerter* para que esta característica seja atendida. O seu funcionamento dependerá do SGBD que está executando a consulta possuir esta capacidade ou não.

O Quip não atende a esta característica. O COUNT até funciona, desde que esteja escrito com letras minúsculas, porém o *casting*, que é o objetivo de teste deste item, não funciona.

Os mesmos comentários feitos para o Quip se aplicam para o XQuery Demo.

5.2.4 - Expressões de caminho regulares

Expressões de caminho são fundamentais para qualquer linguagem de consulta para XML ou dados semi-estruturados. Estas consultas avaliam a capacidade que os processadores de consulta têm em otimizar expressões de caminho e não percorrer partes irrelevantes da árvore. A Figura 61 e a Figura 62 apresentam consultas de exemplo de expressões de caminho regulares.

```
Q6. Quantos itens aparecem em todos os continentes?
FOR $b IN document("auction.xml")/site/regions
RETURN COUNT ($b//item)
```

Figura 61. Consulta Q6

```
Q7. Quantas partes de texto descritivo há no nosso arquivo?
FOR $p IN document("auction.xml")/site
LET $c1 := count($p//description),
$c2 := count($p//mail),
$c3 := count($p//email),
$sum := $c1 + $c2 + $c3
RETURN $sum;
```

Figura 62. Consulta Q7

A implementação do *XVerter* atual é capaz de traduzir expressões de caminho regulares com utilização do operador "//". Entretanto, como a função COUNT não foi considerada na definição das regras, as consultas Q6 e Q7 precisam ser alteradas para serem traduzidas pelo nosso tradutor. A consulta mostrada na Figura 63, que é uma alteração da Q6, é capaz de ser traduzida pelo *XVerter*.

```
FOR $b IN document("auction.xml")/site/regions//item
RETURN $b
```

Figura 63. Consulta Q6 modificada

Já no Quip, a consulta Q6 apresenta o mesmo problema da Q1, pois possui na sua expressão de caminho o elemento *item*. Caso seja colocado um outro elemento em seu lugar, a consulta funciona corretamente, desde que também a palavra COUNT esteja com letras minúsculas. A consulta Q7 executou normalmente, mas o resultado retornado não foi o esperado. Para a contagem do número de elementos *description*, foi

considerado apenas uma expressão de caminho, ao invés de percorrer todo o documento XML à procura de todos os elementos *description* presentes em qualquer nível.

A consulta Q6 funciona no XQuery Demo desde que o COUNT esteja com letras minúsculas. A consulta Q7 também não apresentou problemas e ao contrário do Quip, retornou o resultado correto.

5.2.5 - Navegação através das referências

Referências são uma parte do XML que permite relacionamentos mais ricos que aqueles definidos pela hierarquia dos elementos nas estruturas. As consultas mostradas na Figura 64 e na Figura 65 são os exemplos contidos no *benchmark* para avaliar esta característica.

```
Q8. Listar os nomes das pessoas e o número de itens que eles compraram (junção entre
person e closed auction).
FOR $p IN document("auction.xml")/site/people/person
LET $a := FOR $t IN document("auction.xml")/site/closed_auctions/closed_auction
WHERE $t/buyer/@person = $p/@id
RETURN $t
RETURN <item person=$p/name/text()> COUNT ($a) </item>
```

Figura 64. Consulta Q8

```
Q9. Listar os nomes das pessoas e os nomes dos itens que foram comprados na Europa.
(junção entre person, closed auction e item)
FOR $p IN document("auction.xml")/site/people/person
LET $a := FOR $t IN document("auction.xml")/site/closed_auctions/closed_auction
LET $n := FOR $t2 IN document("auction.xml")/site/regions/europe/item
WHERE $t/itemref/@item = $t2/@id
RETURN $t2
WHERE $p/@id = $t/buyer/@person
RETURN <item> $n/name/text() </item>
RETURN <person name=$p/name/text()> $a </person>
```

Figura 65. Consulta Q9

As consultas Q8 e Q9 obviamente contêm mais de uma expressão de caminho para poderem realizar as junções. Porém, conforme dito anteriormente, esta funcionalidade ainda não está implementada no *XVerter* e conseqüentemente estas consultas ainda não podem ser traduzidas pelo nosso tradutor. Vale lembrar mais uma vez que as regras para tratamento de consultas XQuery com mais de uma expressão de caminho distintas foram definidas no nosso trabalho, porém não implementadas. Como uma limitação das nossas regras, permitimos apenas uma expressão de caminho na cláusula LET e não uma expressão FLWR, como está definido nas consultas Q8 e Q9.

O Quip consegue executar a consulta Q8 desde que seja retirada a construção do atributo na cláusula RETURN e que o COUNT esteja com letras minúsculas. Já a

consulta Q9 não funcionou devido ao problema do elemento *item* já descrito anteriormente.

No XQuery Demo, a consulta Q8 executou quando colocamos o COUNT com letras minúsculas e a Q9 rodou sem problemas.

5.2.6 - Construção de resultados complexos

Esta característica permite uma construção de resultados complexos na cláusula RETURN de acordo com as necessidades do usuário. A Figura 66 mostra um exemplo que constrói resultados deste tipo.

```
Q10. Listar todas as pessoas de acordo com suas preferências; utilizar o francês nas
tags do resultado.
FOR $i IN DISTINCT
document("auction.xml")/site/people/person/profile/interest/@category
LET $p := FOR $t IN document("auction.xml")/site/people/person
WHERE $t/profile/interest/@category = $i
RETURN <personne>
<statistiques>
<sexe> $t/gender/text() </sexe>,
<age> $t/age/text() </age>,
<education> $t/education/text()</education>,
<revenu> $t/income/text() </revenu>
</statistiques>,
<coordonnees>
<nom> $t/name/text() </nom>,
<rue> $t/street/text() </rue>,
<ville> $t/city/text() </ville>,
<pays> $t/country/text() </pays>,
<reseau>
<courrier> $t/email/text() </courrier>,
<pagePerso> $t/homepage/text()</pagePerso>
</reseau>,
</coordonnees>
<cartePaiement> $t/creditcard/text()</cartePaiement>
</personne>
RETURN <categorie>
<id> $i </id>,
$p
</categorie>
```

Figura 66. Consulta Q10

A construção de resultados complexos não foi considerada em nosso tradutor, porém seria possível tratá-la já que teríamos apenas que mostrar os dados de forma diferente. O conteúdo das *tags* pode ser recuperado através das consultas sobre as classes do DOM. Após isto, bastaria fazer uma rearrumação do resultado de saída.

No Quip, a consulta Q10 não funcionou pois o DISTINCT não está implementado. Se o retirarmos da consulta, ela executa normalmente.

O mesmo comentário pode ser aplicado ao XQuery Demo.

5.2.7 - Junções por valor

Este item tem como objetivo avaliar a capacidade de processar as junções por valor, ao contrário das junções apresentadas na seção 5.2.5. As consultas Q11 e Q12, mostradas respectivamente na Figura 67 e na Figura 68, são exemplos de utilização da junção por valor.

```
Q11. Para cada pessoa, listar o número de itens atualmente a venda cujo preço não
ultrapasse 0,02% da renda dessa pessoa.
FOR $p IN document("auction.xml")/site/people/person
LET $l := FOR $i IN document("auction.xml")/site/open_auctions/open_auction/initial
WHERE $p/profile/@income > (5000 * $i/text())
RETURN $i
RETURN <items name=$p/profile/@income> COUNT ($l) </items>
```

Figura 67. Consulta Q11

```
Q12. Para cada pessoa mais rica que a média, listar o número de itens atualmente a venda
cujo preço não ultrapasse 0,02% da renda dessa pessoa.
FOR $p IN document("auction.xml")/site/people/person
LET $l := FOR $i IN document("auction.xml")/site/open_auctions/open_auction/initial
WHERE $p/profile/@income > (5000 * $i/text())
RETURN $i
WHERE $p/profile/@income > 50000
RETURN <items person=$p/profile/@income> COUNT ($l) </person>
```

Figura 68. Consulta Q12

As consultas Q11 e Q12 apresentam características que o nosso tradutor ainda não suporta. Apresentamos na Figura 69, um exemplo de consulta XQuery que é suportado pelo *XVerter* e contém uma junção por valor. Este exemplo é a consulta Q12 modificada. Como o GOA não faz *casting*, a consulta fará a comparação entre *strings*, e não entre os números.

```
FOR $p IN document("auction.xml")/site/people/person
WHERE $p/profile/@income > 50000
RETURN $p/profile/@income
```

Figura 69. Consulta Q12 modificada

O Quip não consegue executar a consulta Q11 mesmo colocando o COUNT em minúsculas e tirando a construção do atributo dos resultados complexos. O problema neste caso é a operação de multiplicação na cláusula WHERE do LET. Um comentário análogo pode ser feito para a consulta Q12, adicionando a impossibilidade de fazer o *casting* da cláusula WHERE da consulta mais externa.

No XQuery Demo acontecem os mesmos problemas, com exceção da construção do atributo nos resultados complexos.

5.2.8 - Reconstrução

Escolher uma abordagem para armazenar documentos XML em SGBDs para posterior reconstrução destes documentos é a característica testada neste item. A consulta da Figura 70 testa a capacidade de reconstruir partes do documento XML original.

```
Q13. Listar os nomes dos itens que estão registrados na Austrália e suas descrições.
FOR $i IN document("auction.xml")/site/regions/australia/item
RETURN <item name=$i/name/text()> $i/description </item>
```

Figura 70. Consulta Q13

O *XVerter* possui a capacidade de tratar esta característica, porém o resultado complexo da consulta Q13 não pode ser gerado. A Figura 71 apresenta uma modificação na consulta Q13 que a faz ser suportada pelo nosso tradutor.

```
FOR $i IN document("auction.xml")/site/regions/australia/item
RETURN $i/description
```

Figura 71. Consulta Q13 modificada

No Quip, a consulta Q13 também possui o problema de acessar o elemento de nome *item*. Se colocarmos outro elemento na expressão de caminho e excluirmos a criação do atributo na cláusula RETURN, a consulta executa sem problemas.

A consulta Q13 executa normalmente no XQuery Demo.

5.2.9 - Texto completo

Este item testa uma busca por texto através dos documentos XML. A consulta Q14, que pode ser visualizada na Figura 72, apresenta um exemplo para este tipo de busca.

```
Q14. Retornar os nomes de todos os itens cuja descrição contenha a palavra 'gold'.
FOR $i IN document("auction.xml")/site//item
WHERE CONTAINS ($i/description,"gold")
RETURN $i/name/text()
```

Figura 72. Consulta Q14

Esta característica não foi considerada em nosso trabalho por ser uma necessidade secundária da XQuery. Contudo, é possível construir uma regra para traduzir o CONTAINS da XQuery para o LIKE da SQL3, já que são funções que possuem a mesma finalidade.

No Quip, de novo aparece o problema do elemento *item*. Entretanto, mesmo testando com outra expressão de caminho, a consulta não roda devido à função CONTAINS. Com letras minúsculas, o processador dá uma mensagem dizendo que o tipo de um dos argumentos está errado.

O XQuery Demo conseguiu executar a consulta Q14, desde que a função CONTAINS esteja com letras minúsculas.

5.2.10 - Percurso do caminho

Esta característica tem como objetivo avaliar a navegação sobre expressões de caminho sem a utilização do operador "//", que era o objetivo de uma característica apresentada anteriormente. As consultas da Figura 73 e da Figura 74 são exemplos de uso desta navegação.

```
Q15. Imprimir o conteúdo dos elementos keyword com ênfase nas anotações dos leilões que
já terminaram.
FOR $a IN document("auction.xml")/site/closed_auctions/closed_auction/annotation/\
description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
RETURN <text> $a <text>
```

Figura 73. Consulta Q15

```
Q16. Retornar os IDs dos vendedores dos leilões que tiveram um ou mais elementos keyword
em ênfase.
FOR $a IN document("auction.xml")/site/closed_auctions/closed_auction
WHERE NOT EMPTY ($a/annotation/description/parlist/listitem/parlist/\
listitem/text/emph/keyword/text())
RETURN <person id=$a/seller/@person />
```

Figura 74. Consulta Q16

O *XVerter* também suporta esta característica, entretanto o resultado complexo da consulta Q15 não pode ser gerado. A Figura 75 mostra esta consulta modificada e que o nosso tradutor consegue transformar.

```
FOR $a IN document("auction.xml")/site/closed_auctions/closed_auction/annotation/\
description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
RETURN $a
```

Figura 75. Consulta Q15 modificada

Já a consulta Q16 utiliza a função EMPTY que não foi considerada na definição das regras do *XVerter*, mas que poderia ser traduzida utilizando a função NOT IN da SQL3.

A utilização do elemento *text* da consulta Q15 no Quip impede a sua execução, assim como a da consulta Q16. Porém, se excluirmos este elemento da expressão de

caminho, a consulta Q15 funciona normalmente, ao contrário da Q16, mesmo transformando a construção do atributo em um elemento. Apesar de a função EMPTY estar implementada, a consulta passa a não executar mais quando o NOT é inserido.

O mesmo problema do elemento *text* acontece na execução das consulta Q15 e Q16 no XQuery Demo. Com outra expressão de caminho, a execução não apresenta problemas, desde que na consulta Q16 a função EMPTY esteja com letras minúsculas e sem o NOT, que não está implementado.

5.2.11 - Elementos não existentes

Esta característica avalia a capacidade dos processadores de consulta lidarem com aspectos semi-estruturados dos dados XML, principalmente aqueles elementos que estão declarados opcionais no DTD. A consulta Q17 que está mostrada na Figura 76 é um exemplo deste item.

```
Q17. Quais pessoas não possuem uma homepage?
FOR $p IN document("auction.xml")/site/people/person
WHERE EMPTY($p/homepage/text())
RETURN <person name=$p/name/text()/>
```

Figura 76. Consulta Q17

Conforme explicado anteriormente, o XVerter não trata a função EMPTY, por não ser uma característica essencial da linguagem XQuery.

Já no Quip, a consulta Q17 executa normalmente se nós transformarmos a construção do atributo da cláusula RETURN em um elemento.

A consulta Q17 no XQuery Demo também executa ao colocarmos o EMPTY com letras minúsculas.

5.2.12 - Aplicação de funções

A capacidade de lidar com as UDFs (*User Defined Function*) é o principal objetivo desta característica. A Figura 77 apresenta uma consulta que utiliza uma função.

```
Q18. Converter a moeda utilizada em todos os leilões em aberto para outra moeda.
FUNCTION CONVERT ($v)
{
RETURN 2.20371 * $v -- convert Dfl to Euros
}
FOR $i IN document("auction.xml")/site/open_auctions/open_auction/
RETURN CONVERT($i/reserve/text())
```

Figura 77. Consulta Q18

Por também se tratar de uma característica secundária, o *XVerter* não trata esta funcionalidade.

Apesar de o Quip possuir a capacidade de criar funções, não foi possível executar a consulta Q18, mesmo depois de colocar a definição da função na sintaxe do Quip.

O mesmo comentário se aplica ao XQuery Demo.

5.2.13 - Ordenação

O comando SORTBY da XQuery tem o mesmo efeito do ORDER BY do SQL e tem como objetivo ordenar o resultado. A Figura 78 apresenta um exemplo de uso do SORTBY.

```
Q19. Ordenar os itens alfabeticamente pela sua localização.
FOR $b IN document("auction.xml")/site/regions//item
LET $k := $b/name/text()
RETURN <item name=$k> $b/location/text() </item>
SORTBY (.)
```

Figura 78. Consulta Q19

Apesar de não existirem regras do *XVerter* para tratar a ordenação, a definição destas regras não seria complicada já que, conforme dito acima, o SORTBY da XQuery equivale ao ORDER BY da SQL3. Portanto, esta característica poderia ser implementada pelo nosso tradutor.

A consulta Q19 utiliza novamente o elemento *item* e por esta razão não é possível executá-la no Quip. No entanto, trocamos a expressão de caminho, continuamos a utilizar o SORTBY e a consulta funcionou normalmente.

No XQuery Demo, o SORTBY não está implementado.

5.2.14 - Agregação

Esta característica avalia a capacidade de agregar os resultados de saída e pode ser testada através da consulta da Figura 79.

```
Q20. Agrupar os clientes através da sua renda e apresentar como saída a cardinalidade de
cada grupo.
<result>
<preferred>
COUNT (document("auction.xml")/site/people/person/profile[@income >= 100000])
</preferred>,
<standard>
COUNT (document("auction.xml")/site/people/person/profile[@income < 100000
and @income >= 30000])
</standard>,
```

```
<challenge>
COUNT (document("auction.xml")/site/people/person/profile[@income < 30000])
</challenge>,
<na>
COUNT (FOR $p in document("auction.xml")/site/people/person
WHERE EMPTY($p/@income)
RETURN $p)
</na>
</result>
```

Figura 79. Consulta Q20

O *XVerter* não implementa funções de agregação, mas assim como para a ordenação, esta característica poderia ser implementada devido às semelhanças entre estas funções na linguagem XQuery e na SQL3.

Já no Quip, a existência de predicados de consulta que exigem *casting* impede esta consulta de executar. Caso retiremos estes predicados e coloquemos o COUNT com letras minúsculas, a consulta Q20 poderá ser executada.

No XQuery Demo, as mesmas observações podem ser feitas.

5.3 - Considerações Finais sobre a Avaliação

Conforme pode ser observado pela análise da seção 5.2, o *XVerter* possui as regras necessárias para tratar as características essenciais da XQuery, como a consulta por valor exato, *casting*, expressões de caminho regulares, navegação através das referências, junções por valor, reconstrução e percurso do caminho. Além disso, possui a capacidade de ser estendido para atender a algumas outras características como acesso ordenado, texto completo, ordenação e agregação. A tradução destas funcionalidades não parece ser complexa, visto que o CONTAINS, o SORTBY e o COUNT da XQuery possuem correspondentes no SQL3. Da mesma forma, podemos representar o acesso ordenado através da utilização do tipo de dados *array* da SQL3. Já a característica de lidar com elementos não existentes também poderia ser suportada pelo *XVerter*, mas não com a facilidade das características anteriores. A função EMPTY da XQuery equivale à construção NOT IN da SQL3, e com uma adaptação nas regras existentes poderíamos fazer com que o *XVerter* também suportasse esta característica.

Vamos analisar agora as outras duas características que faltam: construção de resultados complexos e aplicação de funções. A construção de resultados complexos nada mais é que uma reescrita do resultado da consulta, sendo praticamente uma questão de formatação. Portanto, esta construção poderia ser implementada por um módulo que seria chamado após a execução desta consulta, uma vez que o resultado

desta execução já nos traria todos os dados necessários para a formatação do resultado conforme descrito na cláusula RETURN da XQuery. Já a aplicação de funções é um refinamento da linguagem XQuery que dependendo do SGBD alvo, também poderia ser traduzido, desde que este SGBD suporte tal característica.

Em relação à implementação das regras propostas, apenas as consultas com mais de uma expressão de caminho e com cláusula LET ainda não são possíveis de serem executadas pelo *XVerter*.

Num trabalho recente, que realiza o armazenamento de documentos XML em SGBDORs (RUNAPONGSA & PATEL, 2002), são definidas seis consultas para avaliar o desempenho de consultas, consideradas típicas pelos autores, frente à estratégia de armazenamento proposta. Cada consulta definida é focada em determinada característica: *flattening*, expressão de caminho, seleção, múltiplas seleções, seleção com sub-níveis e acesso ordenado. Destas, o *XVerter* apenas não suporta o acesso ordenado, porém, como explicado na seção 5.2.2, regras podem ser definidas para atender a esta característica. Cabe ressaltar que Runapongsa e Patel não fazem a tradução automática de XQuery para SQL3 e sim manualmente.

Concluindo, a nossa proposta trata grande parte das características apresentadas no *benchmark* e no trabalho de (RUNAPONGSA & PATEL, 2002), e que consideramos serem as principais da linguagem XQuery. É importante ressaltar que nem os processadores da linguagem XQuery são capazes de executar todas as características presentes no *benchmark*, conforme pudemos constatar através dos testes realizados.

Capítulo 6 - Conclusões

O número elevado de documentos que vêm sendo criados no formato XML levam à necessidade de armazenamento e acesso. Estes documentos XML podem estar armazenados nas mais diversas fontes de dados, desde as mais simples, como um sistema de arquivos, até chegar às mais sofisticadas, como os Sistemas Gerenciadores de Bases de Dados (SGBDs). Estes SGBDs, por sua vez, podem ser nativos, relacionais ou baseados em objetos. Existem inúmeros trabalhos sobre as estratégias para armazenamento de documentos XML em SGBDs relacionais e objeto-relacionais.

A solução proposta nesta dissertação para o armazenamento e consulta sobre os dados XML em SGBDORs tem inúmeras vantagens. Já que estamos propondo o armazenamento em um SGBD consolidado, aproveitamos todo o seu potencial nos aspectos importantes de um SGBD, como a execução eficiente de consultas. Além disso, dados já armazenados em um SGBD podem conviver com novos dados XML, sem a necessidade de utilização de um segundo SGBD para esta finalidade.

Outro aspecto positivo da nossa abordagem é a utilização do modelo objeto-relacional em vez do relacional, que é o foco da maioria dos trabalhos existentes. Nestes trabalhos, são apresentados SQLs complexos e de difícil tradução, ao contrário do que nossa abordagem proporciona com a utilização da combinação SGBDOR e formato DOM. Uma outra vantagem é o uso de dois padrões amplamente conhecidos no universo XML, que são a linguagem XQuery e o DOM. Portanto, a nossa proposta não utiliza esquemas de representação proprietários nem linguagens de consulta adaptadas para facilitar a transformação, como ocorre com outros trabalhos relacionados. A facilidade para a recuperação total ou parcial do documento XML também favoreceu a escolha do esquema de representação DOM. Vale ressaltar que, embora o DOM seja um formato bastante conhecido, não encontramos trabalhos que exploram a estrutura de árvore DOM no armazenamento de documentos XML em SGBDs. Este formato é geralmente utilizado para a manipulação de documentos XML na memória principal.

A arquitetura proposta para a tradução da linguagem de consulta XQuery é genérica e flexível para a geração de diferentes linguagens-alvo. Isto contribui bastante para a consolidação do XML como a *lingua franca* de ambientes distribuídos e heterogêneos. Além disso, o uso da linguagem XSLT para expressar as regras de transformação do *XVerter* torna o processo de tradução mais coerente com o padrão

XML, oferecendo maior expressividade e extensibilidade à solução proposta. O tempo gasto no processo de tradução baseada no formato DOM é irrisório, pois não são necessárias conexões adicionais ao SGBD além da consulta propriamente dita.

É importante ressaltar que a linguagem XQuery é bastante poderosa, possuindo operadores, como o "//', que não existem nos SGBDs tradicionais. Desta forma, apesar de a tradução de expressões de caminho regulares que possuem o operador "//" ser complexa, nós conseguimos definir regras para realizar esta transformação. Este processo de transformação de uma linguagem de consulta XML para a linguagem do SGBD não é trivial, e isto pode ser constatado através do número reduzido de trabalhos que realiza esta tradução automática. Mesmo nestes existentes, apenas o núcleo base da XQuery é contemplado. Assim, funções do tipo CONTAINS, BEFORE e EMPTY não são mencionados.

Conforme pode ser observado pela análise realizada na seção 5.2, nem os próprios processadores de XQuery atendem a todas as características presentes no *benchmark*. A nossa proposta suporta grande parte delas e tem potencial para atender às outras características restantes, bastando para isso estender as regras já existentes. Uma destas características não suportada atualmente é a formatação de resultados complexos. Porém, a nossa solução consegue recuperar todas as informações necessárias da árvore DOM, faltando apenas a inserção de *tags* para que o resultado seja retornado como um documento XML.

Por fim, a solução apresentada neste trabalho para o armazenamento de documentos XML e a tradução de consultas XQuery é simples de ser implementada em qualquer SGBDOR, pois as únicas modificações que precisam ser efetuadas são a criação de um mecanismo de importação dos documentos XML para o formato DOM e, eventualmente, o ajuste das regras apresentadas neste trabalho.

6.1 - Contribuições

Uma das contribuições do nosso trabalho foi a apresentação de uma classificação das inúmeras propostas de armazenamento de documentos XML existentes. A partir da análise das diversas propostas, pudemos caracterizar quatro abordagens para representar os dados XML e para cada uma delas, podemos utilizar o modelo de dados relacional ou baseado em objetos. Uma segunda contribuição foi a utilização do formato padrão

DOM para armazenamento dos documentos XML em disco, ao contrário de vários fabricantes que o utilizam em memória. Com a utilização deste formato, não é necessário conhecer a estrutura do documento XML que está sendo armazenado, já que as estruturas criadas no SGBD serão sempre as mesmas. Esta é, portanto, uma outra questão importante do nosso trabalho. Uma outra contribuição foi a arquitetura proposta para o *XVerter*. Para que seja utilizada uma nova abordagem de armazenamento e/ou uma nova linguagem-alvo de consulta, é necessário adaptar e definir novas regras de transformação e implementá-las no arquivo XSL. Aliás, esta foi uma outra contribuição do nosso trabalho, pois permite a utilização de um outro padrão do XML para realizar a transformação através do XSLT. Desta forma, grande parte da solução é mantida no universo XML. E finalmente, a principal contribuição do nosso trabalho foi a definição das regras para transformar uma consulta de entrada em XQuery, que também é padrão, em uma linguagem de saída para ser executada em SGBDORs, a SQL3.

6.2 - Trabalhos Futuros

Como trabalhos futuros, pretendemos estender as regras já existentes para permitir que o *XVerter* suporte algumas outras características secundárias que estão presentes no *benchmark* discutido no Capítulo 5, já que as características principais já são suportadas. Um outro trabalho futuro é formatar o resultado de saída do SGBDOR para transformá-lo em um documento XML. Desta forma, o *XVerter* passaria a possuir a capacidade de construir resultados complexos, que é uma característica presente no *benchmark* que analisamos.

Pretende-se que o *XVerter* seja utilizado em diversas aplicações e projetos em andamento na COPPE-Sistemas / UFRJ, como aplicações de petróleo, bioinformática, projetos em mineração de dados e processamento de consultas XML em ambientes com distribuição.

BOHME, T.,RAHM, E., 2001, "XMach-1: A Benchmark for XML Data Management". In: *Datenbanksysteme in Büro, Technik und Wissenschaft (German Database Conference) (BTW'2001)*, pp. 264-273, Oldenburg, German.

BRESSAN, S., DOBBIE, G., LACROIX, Z., et al, 2001, "XOO7: Applying OO7 Benchmark to XML Query Processing Tool". In: *International Conference on Information and Knowledge Management (CIKM'2001)*, pp. 167-174, Atlanta, USA.

CAREY, M., FLORESCU, D., IVES, Z., et al, 2000, "XPERANTO: Publishing Object-Relational Data as XML". In: *Third International Workshop on the Web and Databases (WebDB'2000)*, pp. 105-110, Dallas, USA.

CAREY, M., KIERNAN, J., SHANMUGASUNDARAM, J., et al, 2002, "XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents". In: *26th International Conference on Very Large Data Bases (VLDB'2000)*, pp. 646-648, Cairo, Egypt.

CHAMBERLIN, D., ROBIE, J., FLORESCU, D., 2000, "Quilt: An XML Query Language for Heterogeneous Data Sources". In: *Third International Workshop on the Web and Databases (WebDB'2000)*, pp. 1-25, Dallas, USA.

CHAUDHURI, S.,SHIM, K., 2001, "Storage and Retrieval of XML Data Using Relational Databases". In: *27th International Conference on Very Large Data Bases (VLDB'2001)*, Roma, Italy.

CHENG, J.,XU, J., 2000, "XML and DB2". In: *16th International Conference on Data Engineering (ICDE'2000)*, pp. 569-573, San Diego, California, USA.

FEGARAS, L.,ELMASRI, R., 2001, "Query Engines for Web-Accessible XML Data". In: *27th International Conference on Very Large Data Bases (VLDB'2001)*, pp. 251-260, Roma, Italy.

FLORESCU, D.,KOSSMAN, D., 1999, "Storing and Querying XML Data using an RDBMS". In: *IEEE Data Engineering Bulletin*, v. 22, pp. 27-34.

MANOLESCU, I., FLORESCU, D., KOSSMAN, D., 2001, "Answering XML Queries Over Heterogeneous Data Sources". In: *27th International Conference on Very Large Data Bases (VLDB'2001)*, pp. 241-250, Roma, Italy.

MANOLESCU, I., FLORESCU, D., KOSSMAN, D., et al, 2000, "Agora: Living with XML and Relational". In: *26th International Conference on Very Large Data Bases (VLDB'2000)*, pp. 623-626, Cairo, Egypt.

MATTOSO, M., 2000, "GOA++: Gerente de Objetos Armazenados". In: http://www.cos.ufrj.br/~goa, Accessed in 21/08/2002.

MATTOSO, M., CAVALCANTI, M., PINHEIRO, R., et al, 2002, "Gerência de Documentos XML no GOA". In: *XVI Simpósio Brasileiro de Engenharia de Software (SBES'2002) - Sessão de Ferramentas*, pp. 402-407, Gramado, Brasil.

MCHUGH, J., 2000, *Data Management and Query Processing for Semistructured Data*, Tese de D. Sc., Department of Computer Science, Stanford University, USA.

MICROSOFT .NET HOME, 2002, "XML Query Language Demo". In: http://xmlqueryservices.com, Accessed in 06/11/2002.

ORACLE, 2002, "The New XMLType Datatype". In: http://otn.oracle.com/products/oracle9i/daily/nov08.html, Accessed in 24/09/2002.

RUNAPONGSA, K.,PATEL, J., 2002, "Storing and Querying XML Data in Object-Relational DBMSs*". In: *International Conference on Extending Database Technology (EDBT'2002) Workshops XMLDM*, pp. 266-285, Prague, Czech Republic.

SAX PROJECT, 2002, "About SAX". In: http://www.saxproject.org/, Accessed in 24/09/2002.

SCHMIDT, A., WAAS, F., KERSTEN, M., et al, 2002, "XMark: A Benchmark for XML Data Management". In: *28th International Conference on Very Large Data Bases (VLDB'2002)*, Hong Kong, China.

SCHMIDT, A., WAAS, F., KERSTEN, M., et al, 2001, *The XML Benchmark Project.* Relatório Técnico INS-R0103. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands.

SHANMUGASUNDARAM, J., SHEKITA, E., BARR, R., et al, 2000, "Efficiently Publishing Relational Data as XML Documents". In: *26th International Conference on Very Large Data Bases (VLDB'2000)*, pp. 65-76, Cairo, Egypt.

SHANMUGASUNDARAM, J., SHEKITA, E., KIERNAN, J., et al, 2001, "A General Technique for Querying XML Documentos using a Relational Database System". In: *Special Interest Group on Management of Data (SIGMOD) Record*, v. 30, pp. 20-26.

SHANMUGASUNDARAM, J., TUFTE, K., HE, G., et al, 1999, "Relational Databases for Querying XML Documents: Limitations and Opportunities". In: *25th International Conference on Very Large Data Bases (VLDB'1999)*, pp. 302-314, Edinburgh, Scotland, United Kingdom.

SOFTWARE AG, 2002a, "Tamino XML Server". In: http://www.softwareag.com/tamino/, Accessed in 21/08/2002a.

SOFTWARE AG, 2002b, "XQuery Prototype Quip". In: http://www.softwareag.com/developer/quip/default.htm, Accessed in 06/11/2002b.

TATARINOV, I., VIGLAS, S., BEYER, K., et al, 2002, "Storing and Querying Ordered XML Using a Relational Database System". In: *Special Interest Group on Management of Data (SIGMOD'2002)*, Wisconsin, USA.

THE APACHE XML PROJECT, 2002, "Package org.apache.xerces.dom". In: http://xml.apache.org/xerces-j/apiDocs/org/apache/xerces/dom/package-summary.html, Accessed in 24/09/2002.

TIAN, F., DEWITT, D., CHEN, J., et al, 2002, "The Design and Performance Evaluation of Alternative XML Storage Strategies". In: *Special Interest Group on Management of Data (SIGMOD) Record*, v. 31.

VIEIRA, H., GONÇALVES, F. C., MATTOSO, M., 2002a, *XQuery, XML Schema e XSL*. Relatório Técnico Submetido. COPPE Sistemas/UFRJ.

VIEIRA, H., GONÇALVES, F. C., POSSATO, R., et al, 2002b, *XML: Definição, Representação e Armazenamento*. Relatório Técnico ES-585/02. COPPE Sistemas/UFRJ.

VIEIRA, H., RUBERG, G., MATTOSO, M., 2002, "XVerter: Armazenamento e Consulta de Dados XML em SGBDs". In: *XVII Simpósio Brasileiro de Banco de Dados (SBBD'2002)*, pp. 224-238, Gramado, Brasil.

WORLD WIDE WEB CONSORTIUM (W3C), 2002a, "Document Object Model (DOM)". In: http://www.w3.org/DOM, Accessed in 21/08/2002a.

WORLD WIDE WEB CONSORTIUM (W3C), 2002b, "W3C World Wide Web Consortium". In: http://www.w3.org, Accessed in 21/08/2002b.

WORLD WIDE WEB CONSORTIUM (W3C), 2002c, "XML Query". In: http://www.w3.org/Query, Accessed in 21/08/2002c.

WORLD WIDE WEB CONSORTIUM (W3C), 2002d, "XSL Transformations (XSLT)". In: http://www.w3.org/TR/xslt, Accessed in 21/08/2002d.

Apêndice A

Este apêndice apresenta o código contido no arquivo XSL que contém as regras para a transformação de uma XQuery em uma SQL3.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <!--Inicio da declaracao das variaveis globais-->
    <xsl:variable name="paths" select="XQUERY/PATHS/PATH"/>
    <xsl:variable name="contPaths" select="count($paths/ELEMENT)"/>
    <xsl:variable name="paths-aux" select="XQUERY/PATHS-AUX"/>

    <!--Clausula FOR-->
    <xsl:variable name="elements-f" select="XQUERY/FOR/PATH-F/ELEMENTS-F"/>
    <xsl:variable name="contElements-f" select="count($elements-f/ELEMENT-F)"/>
    <xsl:variable name="documents-f" select="XQUERY/FOR/PATH-F/DOCUMENTS-F"/>

    <!--Clausula LET-->
    <xsl:variable name="elements-l" select="XQUERY/LET/PATH-L/ELEMENTS-L"/>
    <xsl:variable name="contElements-l" select="count($elements-l/ELEMENT-L)"/>
    <xsl:variable name="documents-l" select="XQUERY/LET/PATH-L/DOCUMENTS-L"/>

    <!--Clausula WHERE-->
    <xsl:variable name="paths-w" select="XQUERY/WHERE/PATHS-W"/>
    <xsl:variable name="contPaths-w" select="count($paths-w/PATH-W)"/>
    <xsl:variable name="conectivos" select="XQUERY/WHERE/CONECTIVOS"/>

    <!--Clausula RETURN-->
    <xsl:variable name="elements-r" select="XQUERY/RETURN/PATH-R/ELEMENTS-R"/>
    <xsl:variable name="contElements-r" select="count($elements-r/ELEMENT-R)"/>
    <xsl:variable name="atributosVirtuais-r" select="XQUERY/PATHS/PATH/ELEMENT[@TYPE='virtual']"/>

    <!--Contador do numero de atributos virtuais-->
    <xsl:variable name="contAtributosVirtuais-r" select="count($atributosVirtuais-r)"/>

    <!--Fim da declaracao das variaveis globais-->

    <!--Inicio da declaracao dos templates-->

    <!--Template para a construcao das declaracoes dos cursores-->
    <xsl:template name="cursores">
        <!--Declaracao dos parametros-->
        <xsl:param name="cursor"/>
        <xsl:param name="contadorAtual"/>
        <xsl:param name="contadorAnterior"/>
        <!--Declaracao das variaveis-->
        <xsl:variable name="cursorElemento">c</xsl:variable>
        <xsl:variable name="cursorDocumento">d</xsl:variable>
        <!--Corpo do template-->
        <xsl:choose>
            <xsl:when test="$cursor='d' ">
                <!--Declaracao do cursor da raiz do documento (d1 in DocumentImpls)-->
                <xsl:value-of select="$cursor"/>
                <xsl:value-of select="$contadorAtual"/>
                <xsl:text> in DocumentImpls </xsl:text>
            </xsl:when>
            <xsl:otherwise>
                <!--Declaracao dos cursores dos elementos-->
                <!--Virgula para separar as declaracoes dos cursores-->
                <xsl:text>, </xsl:text>
                <xsl:value-of select="$cursor"/>
                <xsl:value-of select="$contadorAtual"/>
                <xsl:text> in </xsl:text>
                <xsl:choose>
                    <xsl:when test="$contadorAtual=1">
```

```
                    <!--Declaracao do primeiro cursor(c1 in d1.childrenNode)-->
                    <xsl:value-of select="$cursorDocumento"/>
                    <xsl:value-of select="$contadorAtual"/>
                    <xsl:text>.childrenNode </xsl:text>
                </xsl:when>
                <xsl:otherwise>
                    <!--Declaracao dos cursores seguintes-->
                    <xsl:value-of select="$cursorElemento"/>
                    <xsl:value-of select="$contadorAnterior"/>
                    <xsl:text>.childrenNode </xsl:text>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<!--Template para a construcao das condicoes definidas implicitamente na clausula FROM(.name) e
explicitamente na clausula WHERE(.value)-->
<xsl:template name="condicoes">
    <!--Declaracao dos parametros-->
    <xsl:param name="cursor"/>
    <xsl:param name="contador"/>
    <xsl:param name="nome"/>
    <xsl:param name="atributo"/>
    <xsl:param name="operador"/>
    <xsl:param name="tipo"/>
    <!--Corpo do template-->
    <xsl:if test="$tipo = 'real'">
        <xsl:text> ( </xsl:text>
        <xsl:value-of select="$cursor"/>
        <xsl:value-of select="$contador"/>
        <xsl:text>.</xsl:text>
        <xsl:value-of select="$atributo"/>
        <xsl:value-of select="$operador"/>
        <xsl:text>"</xsl:text>
        <xsl:value-of select="$nome"/>
        <xsl:text>"</xsl:text>
        <xsl:text> ) </xsl:text>
    </xsl:if>
</xsl:template>

<!--Template para a construcao da projecao sem utilizar o "//"-->
<xsl:template name="projecoesReais">
    <!--Declaracao dos parametros-->
    <xsl:param name="contador"/>
    <!--Declaracao das variaveis-->
    <xsl:variable name="cursorElemento">c</xsl:variable>
    <!--Corpo do template-->
    <xsl:text>result:</xsl:text>
    <xsl:value-of select="$cursorElemento"/>
    <xsl:value-of select="$contador"/>
    <xsl:text>.value </xsl:text>
</xsl:template>

<!--Template para a construcao da projecao ao utilizar o "//"-->
<xsl:template name="projecoesVirtuais">
    <!--Declaracao dos parametros-->
    <xsl:param name="contador"/>
    <xsl:param name="tipoElementoAnterior"/>
    <!--Declaracao das variaveis-->
    <xsl:variable name="cursorElemento">c</xsl:variable>
    <!--Corpo do template-->
    <xsl:if test="$tipoElementoAnterior = 'virtual'">
        <xsl:text>, </xsl:text>
    </xsl:if>
    <xsl:text>result</xsl:text>
    <xsl:value-of select="$contador"/>
    <xsl:text>: </xsl:text>
    <xsl:value-of select="$cursorElemento"/>
    <xsl:value-of select="$contador"/>
    <xsl:text>.value</xsl:text>
```

```
</xsl:template>

    <!--Template para a insercao dos conectivos (and, or, ...)-->
    <xsl:template name="conectivos">
        <xsl:param name="conectivo"/>
        <xsl:value-of select="$conectivo"/>
        <!--Espaco em branco-->
        <xsl:text> </xsl:text>
    </xsl:template>

    <!--Template para a abertura dos parenteses necessarios-->
    <xsl:template name="abreParenteses">
        <xsl:text>(</xsl:text>
    </xsl:template>

    <!--Template para o fechamento dos parenteses necessarios-->
    <xsl:template name="fechaParenteses">
        <xsl:text>)</xsl:text>
    </xsl:template>

    <!--Template para a insercao dos conectivos-->
    <xsl:template name="insereConectivos">
        <xsl:param name="conectivo"/>
        <xsl:choose>
            <xsl:when test="position()!=last()">
                <xsl:value-of select="$conectivo"/>
            </xsl:when>
        </xsl:choose>
    </xsl:template>

    <!--Fim da declaracao dos templates-->

    <!--Inicio da transformacao-->
    <xsl:template match="/">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="XQUERY">
        <!--A ordem e importante pois o RETURN construira a clausula SELECT, o FOR sera responsavel pela
clausula FROM e por uma parte da WHERE, e a clausula WHERE da XQuery sera transformada na clausula
WHERE da SQL3-->
        <xsl:apply-templates select="RETURN"/>
        <xsl:apply-templates select="FOR"/>
        <xsl:apply-templates select="WHERE"/>
    </xsl:template>

    <!--Inicio da transformacao da clausula FOR-->
    <xsl:template match="FOR">
        <xsl:text> FROM </xsl:text>
        <xsl:apply-templates select="PATH-F/DOCUMENTS-F"/>
        <xsl:apply-templates select="ancestor::XQUERY/PATHS"/>
    </xsl:template>

    <!--Declaracao do cursor da raiz do documento (d1 in DocumentImpls)-->
    <xsl:template match="DOCUMENTS-F">
        <xsl:for-each select="$documents-f/DOCUMENT-F">
            <xsl:call-template name="cursores">
                <xsl:with-param name="cursor">d</xsl:with-param>
                <xsl:with-param name="contadorAtual" select="position()"/>
            </xsl:call-template>
        </xsl:for-each>
    </xsl:template>

    <!--Declaracao dos cursores dos elementos-->
    <xsl:template match="PATHS">
        <xsl:for-each select="$paths/ELEMENT">
            <xsl:call-template name="cursores">
                <xsl:with-param name="cursor">c</xsl:with-param>
                <xsl:with-param name="contadorAtual" select="position()"/>
                <xsl:with-param name="contadorAnterior" select="position()-1"/>
            </xsl:call-template>
```

```
            <!--Declaracao de um cursor a mais para acessarmos o valor do elemento-->
            <!--Somente e declarado caso o ultimo "elemento" nao seja um atributo-->
            <xsl:if test="position()=last()">
                <xsl:variable name="atributo" select="@ATTR"/>
                <xsl:if test="$atributo != 'yes'">
                    <xsl:call-template name="cursores">
                        <xsl:with-param name="cursor">c</xsl:with-param>
                        <xsl:with-param name="contadorAtual" select="position()+1"/>
                        <xsl:with-param name="contadorAnterior" select="position()"/>
                    </xsl:call-template>
                </xsl:if>
            </xsl:if>
        </xsl:for-each>
        <xsl:text> WHERE </xsl:text>
<!--Declaracao das condicoes obrigatorias referentes a clausula FROM-->
            <xsl:for-each select="$documents-f/DOCUMENT-F">
                <!--Declaracao do nome do documento-->
                <xsl:call-template name="condicoes">
                    <xsl:with-param name="cursor">d</xsl:with-param>
                    <xsl:with-param name="contador" select="position()"/>
                    <xsl:with-param name="atributo">name</xsl:with-param>
                    <xsl:with-param name="operador">=</xsl:with-param>
                    <xsl:with-param name="nome" select="."/>
                    <xsl:with-param name="tipo">real</xsl:with-param>
                </xsl:call-template>
                <xsl:call-template name="conectivos">
                    <xsl:with-param name="conectivo">and</xsl:with-param>
                </xsl:call-template>
            </xsl:for-each>
            <xsl:for-each select="$paths-aux/PATH-AUX">
                <xsl:call-template name="abreParenteses"/>
                <xsl:for-each select="ELEMENT">
                    <xsl:variable name="tipo" select="@TYPE"/>
                    <!--Declaracao dos nomes dos elementos-->
                    <xsl:call-template name="condicoes">
                        <xsl:with-param name="cursor">c</xsl:with-param>
                        <xsl:with-param name="contador" select="position()"/>
                        <xsl:with-param name="atributo">name</xsl:with-param>
                        <xsl:with-param name="operador">=</xsl:with-param>
                        <xsl:with-param name="nome" select="."/>
                        <xsl:with-param name="tipo" select="$tipo"/>
                    </xsl:call-template>
                    <xsl:if test="$tipo = 'real'">
                        <xsl:call-template name="insereConectivos">
                            <xsl:with-param name="conectivo"> and </xsl:with-param>
                        </xsl:call-template>
                    </xsl:if>
                </xsl:for-each>
                <xsl:call-template name="fechaParenteses"></xsl:call-template>
                <xsl:call-template name="insereConectivos">
                    <xsl:with-param name="conectivo"> or </xsl:with-param>
                </xsl:call-template>
            </xsl:for-each>
            <!-- Somente inserimos o conectivo "and" se existir alguma coisa na cláusula WHERE da XQuery-->
            <xsl:if test="$contPaths-w > 0">
                <xsl:call-template name="conectivos">
                    <xsl:with-param name="conectivo"> and </xsl:with-param>
                </xsl:call-template>
            </xsl:if>
    </xsl:template>
    <!--Fim da transformacao da clausula FOR-->

    <!--Inicio da transformacao da clausula LET-->
    <xsl:template match="LET"></xsl:template>
    <!--Fim da transformacao da clausula LET-->

    <!--Inicio da transformacao da clausula WHERE-->
    <xsl:template match="WHERE">
        <xsl:apply-templates select="PATHS-W"/>
    </xsl:template>
```

```
    <xsl:template match="CONECTIVOS"></xsl:template>

    <xsl:template match="CURSOR-W"></xsl:template>

    <xsl:template match="ELEMENTS-W"></xsl:template>

    <xsl:template match="PATHS-W">
        <xsl:for-each select="$paths-w/PATH-W">
            <xsl:variable name="posicao" select="position()"/>
            <xsl:variable name="valor" select="VALUE-W"/>
            <xsl:variable name="operador" select="VALUE-W/@OPERATOR-W"/>
            <xsl:for-each select="ELEMENTS-W/ELEMENT-W">
                <!--Vai percorrer os elementos ate chegar ao ultimo da expressao de caminho, que e o
correspondente ao valor contido no elemento VALUES-W-->
                <xsl:if test="position()=last()">
                    <xsl:call-template name="condicoes">
                        <xsl:with-param name="cursor">c</xsl:with-param>
                        <xsl:with-param name="contador" select="$contElements-f+position()+1">
        </xsl:with-param>
                        <xsl:with-param name="atributo">value</xsl:with-param>
                        <xsl:with-param name="operador" select="$operador"/>
                        <xsl:with-param name="nome" select="$valor"/>
                        <xsl:with-param name="tipo">real</xsl:with-param>
                    </xsl:call-template>
                    <!--Percorre os conectivos ate chegar ao correspondente. Este controle e feito atraves da
posicao do elemento PATH-W. Apos o primeiro PATH-W, e inserido o primeiro conectivo. Apos o segundo, o
segundo conectivo e assim sucessivamente-->
                    <xsl:for-each select="$conectivos/CONECTIVO">
                        <xsl:if test="position()=$posicao">
                            <xsl:variable name="conectivo" select="."/>
                            <xsl:call-template name="conectivos">
                                <xsl:with-param name="conectivo" select="$conectivo"/>
                            </xsl:call-template>
                        </xsl:if>
                    </xsl:for-each>
                </xsl:if>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:template>
    <!--Fim da transformacao da clausula WHERE-->

    <!--Inicio da transformacao da clausula RETURN-->
    <xsl:template match="RETURN">
        <xsl:text>SELECT resultado(</xsl:text>
         <xsl:apply-templates/>
         <xsl:text>)</xsl:text>
    </xsl:template>

    <xsl:template match="PATH-R">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="CURSOR-R">
        <!--Este fragmento de codigo so serve para as consultas que retornarem apenas o cursor, por exemplo,
RETURN $n. -->
        <xsl:if test="$contElements-r=0">
            <xsl:choose>
                <!--Caso o operador "//" nao seja utilizado, o template projecoesReais sera utilizado. Este
controle e feito atraves da variavel contAtributosVirtuais-r, que conta o numero de elementos com tipo igual a
virtual-->
                <xsl:when test="$contAtributosVirtuais-r=0">
                    <xsl:for-each select="$paths/ELEMENT">
                        <xsl:if test="position()=last()">
                            <!--Se for atributo, acessa o valor no mesmo nivel da arvore -->
                            <xsl:variable name="atributo" select="@ATTR"/>
                            <xsl:choose>
                                <xsl:when test="$atributo = 'yes'">
                                    <xsl:call-template name="projecoesReais">
                                        <xsl:with-param name="contador" select="position()"/>
                                    </xsl:call-template>
                                </xsl:when>
```

```
                                <!--Se for elemento, acessa o valor em um nivel abaixo da arvore -->
                                <xsl:otherwise>
                                  <xsl:call-template name="projecoesReais">
                                    <xsl:with-param name="contador" select="position()+1"/>
                                  </xsl:call-template>
                                </xsl:otherwise>
                              </xsl:choose>
                            </xsl:if>
                        </xsl:for-each>
                    </xsl:when>
                    <xsl:otherwise>
                        <!--Caso o operador "//" seja utilizado, o template projecoesVirtuais sera chamado-->
                        <xsl:for-each select="$paths/ELEMENT">
                            <xsl:variable name="tipo" select="@TYPE"/>
                            <xsl:variable name="tipoElementoAnterior" select="preceding-
sibling::*[position()=1]/@TYPE"/>
                            <xsl:if test="$tipo='virtual'">
                                <xsl:call-template name="projecoesVirtuais">
                                    <xsl:with-param name="contador" select="position()+1"/>
                                    <xsl:with-param name="tipoElementoAnterior" select="$tipoElementoAnterior">
                    </xsl:with-param>
                                </xsl:call-template>
                            </xsl:if>
                        </xsl:for-each>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:if>
        </xsl:template>

        <xsl:template match="ELEMENTS-R">
            <!--Caso o retorno da consulta seja uma expressao de caminho, por exemplo, RETURN $n/data/dia, o
fragmento de codigo a seguir sera utilizado-->
            <xsl:for-each select="$elements-r/ELEMENT-R">
                <xsl:if test="position()=last()">
                    <xsl:call-template name="projecoesReais">
                        <xsl:with-param name="contador" select="$contElements-f+$contElements-r+1">
         </xsl:with-param>
                    </xsl:call-template>
                </xsl:if>
            </xsl:for-each>
        </xsl:template>
        <!--Fim da transformacao da clausula RETURN-->

        <!--Fim da transformacao-->
</xsl:stylesheet>
```