Universidade Federal do Rio de Janeiro

DISSERTAÇÃO DE MESTRADO

DIEGO MURY GOMES DE LIMA

XMLINFERENCE: agregando inferência à consultas a documentos XML

Rio de Janeiro
2012

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações
e Pesquisas Computacionais

Diego Mury Gomes de Lima

XMLInference: agregando inferência à consultas a documentos XML

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática (PPGI) da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática

Orientadoras: Vanessa Braganholo, DSc
Carla Delgado, DSc

Rio de Janeiro
2012

Diego Mury Gomes de Lima

XMLInference: agregando inferência à consultas a documentos XML

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática

Aprovada em: Rio de Janeiro, 29 de fevereiro de 2012.

Profª. Vanessa Braganholo Murta, D.Sc.

Profª. Carla Amor Divino Moreira Delgado, D.Sc.

Profª. Adriana Santarosa Vivacqua, D.Sc.

Prof. João Carlos Pereira da Silva, D.Sc.

Profª. Marta Lima de Queirós Mattoso, D.Sc.

À Priscila Alencar dos Santos Mury.

Agradecimentos

A Deus, por me permitir mais essa conquista.

Aos meus pais, por estarem sempre ao meu lado.

À minha esposa Priscila, por todo o apoio, compreensão e por ter me ajudado nos momentos mais difíceis.

À professora Vanessa Braganholo, pela excelente orientação, por acreditar em mim e por muitas vezes me fazer acreditar que ia dar certo.

Ao professor Leonardo Murta, que me orientou extra-oficialmente, me ajudou em toda parte Prolog deste trabalho e, assim como a Vanessa, me incentivou até o final.

À professora Carla Delgado, que aceitou fazer parte do trabalho e contrubuiu com excelentes revisões e ideias novas.

Às professoras Adriana Vivacqua e Marta Mattoso, por terem aceitado participar desta banca.

Ao Professor João Carlos, que além de aceitar o convite de participar desta banca, contribuiu com o trabalho através de seus comentários durante os Seminários de Acompanhamento do PPGI.

Aos meus amigos e familiares, que compreenderam as minhas ausências.

Ao CNPq, pelo suporte financeiro.

Resumo

LIMA, Diego Mury Gomes de. **XMLInference:** agregando inferência à consultas a documentos XML. 2012. 110 f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

O expressivo aumento do volume de informações armazenadas em documentos XML estimula o aprimoramento de métodos capazes de obter respostas satisfatórias para consultas cada vez mais complexas. Enquanto as abordagens de consulta atuais analisam apenas as informações definidas explicitamente no documento, este trabalho apresenta uma abordagem capaz de processar conhecimento implícito através de inferências guiadas por regras derivadas automaticamente do esquema XML e por regras configuradas previamente pelo usuário. Tal fato pode proporcionar um aumento significativo nas possibilidades de consulta, permitindo fácil acesso a novas informações e possibilitando que consultas mais elaboradas obtenham os resultados esperados. Nossos experimentos mostram a efetividade de nossa abordagem. Nossa avaliação comparou execuções de consultas XQuery e de consultas equivalentes em Prolog. Surpreendentemente, nossa avaliação também mostra que algumas consultas são resolvidas de forma mais rápida quando executadas em Prolog.

Palavras-chaves: Consulta em documentos XML, Inferência, Prolog.

Abstract

LIMA, Diego Mury Gomes de. **XMLInference:** agregando inferência à consultas a documentos XML. 2012. 110 f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

The expressive growth in the volume of data stored as XML stimulates the improvement of methods for obtaining novel ways to explore this information. Methods for providing elaborated answers to increasingly more complex queries were recently investigated, but existing approaches mainly analyze information that is explicitly defined in the XML document. This work presents an approach that is capable of processing implicit knowledge through inference, guided by rules automatically obtained from the XML schema and by user defined rules. This can cause a significant increase in the query possibilities, allowing easy access to new information, serendipitous results and providing for more elaborate queries. Our experiments show the efectivity of our approach. Our evaluation compared query executions both in XQuery and in Prolog. Surprisingly, our evaluation also shows that some queries are executed faster in Prolog.

Keywords: Consulta em documentos XML, Inferência, Prolog.

Lista de Figuras

Lista de Tabelas

Lista de siglas e abreviaturas

API	*Application Programming Interface*
DTD	*Document Type Definition*
Prolog	*Programmation en Logique*
SQL	*Structured Query Language*
XML	*Extensible Markup Language*
XPath	*XML Path Language*
XSD	*XML Schema Definition*
YAP	*Yet Another Prolog*

Sumário

Capítulo 1. Introdução

1.1 Motivação

A linguagem XML consolidou-se rapidamente como um padrão universal para troca de informações na Web, resultando no surgimento de grandes coleções de documentos que armazenam um enorme volume de dados neste formato. Vem daí a necessidade de aperfeiçoar os métodos de consulta para que sejam capazes de manipular essas informações de forma eficiente (KOTSAKIS, 2002).

A maioria das linguagens de consulta para documentos XML disponíveis atualmente, como por exemplo, XPath (CLARK; DEROSE, 2010) e XQuery (BOAG *et al.*, 2010), obtêm suas respostas a partir do caminho percorrido na árvore XML. Essas abordagens navegam da raiz do documento ao elemento mais interno especificado na consulta, e aplicam os filtros definidos para processar as informações e obter o resultado. Nestes casos, a consulta é realizada apenas sobre os dados explícitos na estrutura do documento, ignorando quaisquer informações implícitas que possam vir a ser obtidas a partir desses dados ou sobre a própria estrutura do documento.

Observamos que os documentos XML podem possuir informações implícitas, quer sejam descritas em seus dados ou "escondidas" em sua estrutura. Assuma, por exemplo, um documento que contém informação sobre filiação (pares de elementos contendo nome do pai e nome do filho). Neste documento, existem elementos que definem que João é pai de Pedro e Pedro é pai de Paulo. Neste cenário, é simples construir uma consulta que retorna o nome do pai de Paulo. No entanto, não é trivial escrever uma consulta XQuery que retorne o nome do avô de Paulo, uma vez que essa informação não está explícita no documento. Assim, percebemos que obter estas informações através dos métodos de consultas tradicionais é possível, mas pode se tornar uma tarefa bastante trabalhosa.

Visando a elaboração de uma alternativa aos métodos de consulta que venha a simplificar o acesso às informações que podem estar implícitas nos documentos XML, o trabalho realizado nesta dissertação propõe uma reformulação da metodologia empregada nas consultas. A proposta utiliza os benefícios provenientes da programação

em lógica para guiar o processo de consulta, propiciando um aumento das possibilidades de respostas e o reaproveitamento de regras previamente definidas. Mais especificamente, a abordagem proposta neste trabalho utiliza máquinas de inferência com o objetivo de aumentar as possibilidades de consulta, viabilizando a obtenção de resultados mais elaborados. Retornando ao exemplo, o documento XML que define que João é pai de Pedro e Pedro é pai de Paulo, não possui informações suficientes para responder consultas sobre avô. Porém, uma vez que a regra que define avô como pai do pai (ou pai da mãe) é inserida pelo usuário, o sistema estaria apto a responder que João é avô de Paulo.

1.2 Caracterização do Problema

As linguagens tradicionais de consulta XML satisfazem à maioria dos requisitos estabelecidos como necessários para a realização de consultas a documentos XML (MELLO *et al.*, 2000). No entanto, essas linguagens não são voltadas para a consulta sobre informações implícitas no documento, deixando a responsabilidade dessa tarefa para o usuário.

As linguagens de consulta em documentos XML utilizadas atualmente recebem uma entrada (*input*), navegam na estrutura do documento a fim de obter o trecho que atende às condições da consulta e geram uma saída (*output*). Ao analisarmos este tipo de abordagem para consultas chegamos à conclusão de que obter informações implícitas desta maneira pode se tornar um processo com um alto grau de complexidade, demandar consultas grandes e, acima de tudo, exigir que o usuário interessado na consulta tenha grande conhecimento sobre a estrutura do documento consultado e da linguagem de consulta utilizada.

Como um exemplo, podemos mostrar que para utilizar recursão em uma consulta que deseja obter os descendentes de uma pessoa em XQuery é necessário criar uma consulta que contenha uma função recursiva e que faça uma chamada a esta, passando os parâmetros que vão ser processados durante a recursão. A parte superior da Figura 1.1 ilustra um documento XML que descreve pessoas através dos seus nomes e dos nomes dos seus filhos, enquanto a parte inferior mostra uma consulta XQuery que pretende obter os descendentes de uma pessoa. Nota-se que não se trata de uma consulta trivial, ainda que a estrutura do documento seja bem simples. A elaboração desta

consulta demanda um grande conhecimento da estrutura do documento a ser consultado e da linguagem de consulta utilizada. Vale dizer ainda que documentos mais complexos tendem a aumentar a dificuldade da elaboração da consulta, fazendo com que o esforço necessário para obter a informação desejada seja maior.

```
<pessoas>
   <pessoa>
      <nome>Jose</nome>
      <filho>João</filho>
   </ pessoa >
   < pessoa >
      <nome>João</nome>
      <filho>Pedro</filho>
      <filho>Lucas</filho>
   </ pessoa >
   < pessoa >
      <nome>Lucas</nome>
      <filho>Tiago</filho>
   </ pessoa >
</ pessoas >
```

```
declare function local:desc($desc as xs:string+,
      $ant as xs:string){
   for $d in doc("pessoas.xml")/pessoas/pessoa
      return
         if ($d/filho = $desc) then ($d/filho/text())
         else if ($d/nome = $desc) then
            (local:desc($d/filho, $d/nome))
         else ()
};

<resultado>
{for $c in doc("pessoas.xml")/pessoas/pessoa
   return <relacao>
      <ancestral>{$c/nome/text()}</ancestral>
      <descendente>{local:desc($c/filho, $c/nome)}</descendente>
   </relacao>}
</resultado>
```

Figura 1.1: Documento XML descrevendo pessoas e consulta XQuery, utilizando recursão, sobre descendentes.

Um outro problema analisado nos métodos tradicionais é a impossibilidade do reaproveitamento das consultas. Caso uma segunda consulta necessite utilizar algum trecho da consulta anteriormente elaborada na Figura 1.1, este trecho deverá ser integralmente reescrito para que possa ser utilizado, não havendo nenhuma forma de fazer uma chamada externa mesmo se tratando de uma função. É o caso do exemplo **local:desc,** que retorna os antecedentes de uma pessoa. Esta limitação tende a aumentar o tamanho das consultas, tornando-as mais complexas, e faz com que haja repetição do código da consulta caso mais de uma consulta tenha que utilizar um mesmo trecho de código.

1.3 Objetivos

Diante das dificuldades para obter as informações implícitas em documentos XML apresentadas, idealizamos este trabalho com o principal objetivo de contribuir com uma alternativa às linguagens de consulta existentes nesses documentos, permitindo que o resultado da consulta possa ser obtido através de inferências sobre os dados definidos nos documentos. Para isso, propomos um método que é capaz não apenas de efetuar consultas, mas também de processar as informações na base de conhecimento, realizar inferências e fazer com que estas se reflitam no resultado final da consulta.

Para possibilitar a realização de inferências, foram necessários dois passos: (i) adotar uma máquina de inferência capaz de processar os dados e guiar o processo de consulta; (ii) converter os elementos definidos na estrutura do documento XML para uma sintaxe compreensível pela máquina de inferência adotada para que, após a tradução das informações, essa máquina esteja apta a analisar as informações, realizar as possíveis inferências e fornecer os resultados esperados.

A avaliação experimental da nossa abordagem foi possível a partir de um protótipo da solução desenvolvido para realizar todo o trabalho proposto, desde a tradução das informações em uma linguagem passível de inferência, até a execução da consulta utilizando uma máquina de inferência. Desta forma, vários experimentos foram realizados com o objetivo de analisar o desempenho da metodologia proposta, e avaliar se a realização de inferências contribui realmente para facilitar o acesso às informações que não estão explícitas no documento XML.

1.4 Organização dos Capítulos

O restante desta dissertação descreve os estudos e as avaliações que foram realizados durante o desenvolvimento deste trabalho e detalha a metodologia que idealizamos para facilitar o acesso às informações implícitas no documento XML. O texto encontra-se organizado em capítulos da seguinte forma.

O Capítulo 2 apresenta as linguagens que foram estudadas e estão relacionadas com o trabalho desenvolvido. Inicialmente foram analisadas duas das linguagens de consultas já existentes em documentos XML, seguido da apresentação do padrão que

define a estrutura e o vocabulário utilizado nos documentos alvo da consulta. O capítulo encerra detalhando a linguagem lógica adotada para descrever as informações contidas nos documentos alvo das consultas e proporcionar a realização de inferência sobre as mesmas.

No Capítulo 3 são apresentados os trabalhos relacionados à nossa pesquisa, citando as suas características, seus objetivos, de que forma estão relacionados com a abordagem proposta nesta dissertação. Estes trabalhos contribuíram para auxiliar o estudo no sentido de aprimorar nossa ideia e esclarecer dúvidas que já haviam sido tratadas em suas abordagens.

O Capítulo 4 inicia-se dando uma ideia geral da nossa abordagem, ou seja, explica o macro processo do nosso projeto. Suas subseções seguem detalhando cada etapa com o objetivo de explicar minuciosamente todo o processo que compreende a ideia central da pesquisa desenvolvida. A tradução dos esquemas e documentos XML é analisada caso a caso. Ao final deste capítulo, espera-se que o leitor esteja familiarizado com a metodologia proposta nesta dissertação, e compreenda a relevância da mesma.

Durante a realização da pesquisa foi necessário desenvolver um protótipo do método de consulta proposto para que a nossa ideia fosse validada. Assim, no Capítulo 5 apresentamos os detalhes sobre a implementação deste protótipo, listando os seus requisitos e suas características.

Já no Capítulo 6, apresentamos a avaliação experimental que foi realizada a partir do protótipo implementado. Neste capítulo é discutido como o experimento foi elaborado, quais objetivos nortearam o processo, em que ambiente os experimentos foram realizados e que conclusões pode-se tirar dos resultados obtidos.

Finalmente, o Capítulo 7 traz as conclusões deste trabalho, expondo as considerações finais bem como algumas contribuições desta pesquisa, e ainda algumas possibilidades de trabalhos futuros.

Capítulo 2. Linguagens de Consulta XML e Prolog

2.1 Introdução

Obter as informações necessárias de forma fácil, correta e eficiente tornou-se um desafio essencial nos dias de hoje. Dadas as necessidades dos usuários, as abordagens de consulta foram se desenvolvendo no intuito de processar consultas complexas sobre bases de dados cada vez maiores. A popularização dos dados semi-estruturados aumentou a demanda por técnicas que analisassem não apenas os dados armazenados, mas também a estrutura do documento que descreve tais dados.

Esse capítulo analisa as principais linguagens de consultas em documentos XML, apontando suas características, sintaxe e diferenças. Este estudo será importante para a compreensão das comparações, que serão realizadas em capítulos futuros, entre consultas realizadas com essas linguagens e utilizando o método proposto nesta dissertação. Além disso, também será apresentada a linguagem que define os esquemas que validam os documentos XML, denominada *XML Schema Definition* (XSD) ou simplesmente *XML Schema*. O entendimento desta linguagem auxiliará a compreensão da abordagem proposta uma vez que os esquemas possuem um papel significativo nesta.

Ainda neste capítulo é apresentada a linguagem que segue o paradigma da Programação em Lógica Matemática denominada Prolog (***PRO****grammation en* ***LOG****ique*). Suas características são detalhadas para que sejam entendidos os benefícios provenientes da utilização da Programação em Lógica, que são fortemente explorados durante a realização das consultas através da abordagem desenvolvida.

2.2 Consultas em Documentos XML

A padronização do XML como linguagem de intercâmbio de dados na Internet destacou a necessidade da elaboração de métodos de consulta específicos, capazes de analisar tanto os dados descritos como também a estrutura do documento a fim de obter respostas satisfatórias e eficientes para consultas sobre documentos diversos. Segundo

Mello et al. (2000) os requisitos necessários para a realização de consultas sobre documentos XML foram estabelecidos, e várias linguagens surgiram com o objetivo de atendê-los da melhor forma possível. Com o passar do tempo, o W3C determinou um padrão considerado ideal para a realização de consultas em documentos XML, baseado nestas linguagens.

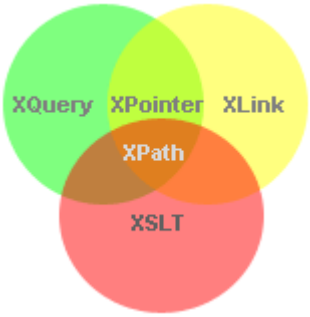

Figura 2.1: Relação da XPath com outras tecnologias XML – Fonte: (W3CSCHOOLS, 1999)

O avanço das pesquisas e o estabelecimento de novos paradigmas fizeram com que muitas linguagens de consulta caíssem em desuso (CHAMBERLIN, D. *et al.*, 2000; DEUTSCH *et al.*, 1999; ROBIE, JONATHAN *et al.*, 1998) e contribuíram para o desenvolvimento de linguagens mais elaboradas que conseguissem mesclar os benefícios de outras abordagens para a obtenção de resultados eficientes. Neste contexto, é importante ressaltar que as linguagens XPath (CLARK; DEROSE, 2010) e XQuery (BOAG *et al.*, 2010) foram as duas linguagens padronizadas pelo W3C, sendo destas a XPath um subconjunto da XQuery como pode ser visto na Figura 2.1.

Dito isso, vamos começar as próximas seções desse capítulo descrevendo cada uma dessas linguagens de consulta que se tornaram recomendações do W3C

2.2.1 XPath

A XPath é uma linguagem de consulta em documentos XML que tornou-se uma recomendação W3C no final de 1999. O termo XPath é a abreviação de XML Path Language. A linguagem possui este nome porque utiliza expressões de caminho para explorar a estrutura hierárquica do documento XML e obter informações que satisfaçam o caminho definido e os filtros aplicados. Estas expressões de caminho possuem sintaxe parecida com aquela utilizada para navegar em sistemas de arquivos. Porém, no contexto da XPath não se trata de diretórios e sim de elementos que formam a estrutura do documento XML.

```
<livros>
    <livro ano='2011'>
        <titulo>As Esganadas</titulo>
        <autor>
            <nome>Jô</nome>
            <sobrenome>Soares</sobrenome>
        </autor>
        <paginas>264</paginas>
        <editora>Companhia das Letras</editora>
    </livro>
    <livro ano='2011'>
        <titulo>Steve Jobs: A Biografia</titulo>
        <autor>
            <nome>Walter</nome>
            <sobrenome>Isaacson</sobrenome>
        </autor>
        <paginas>624</paginas>
        <editora>Companhia das Letras</editora>
    </livro>
    <livro ano='2002'>
        <titulo>Pai Rico Pai Pobre</titulo>
        <autor>
            <nome>Robert</nome>
            <sobrenome>Kiyosaki</sobrenome>
        </autor>
        <paginas>186</paginas>
        <editora>Campus</editora>
    </livro>
</livros>
```

Figura 2.2: Exemplo de documento XML descrevendo livros

Observando o documento XML ilustrado pela Figura 2.2, nota-se que os títulos dos livros são representados pelo elemento "titulo" que é filho do elemento "livro" que por sua vez é filho da raiz "livros". Portanto, seu valor pode ser obtido com o uso da linguagem XPath através da consulta **/livros/livro/titulo/text()**. É possível observar que o predicado *buint-in* "text()" retorna o conteúdo textual do elemento alcançado pelo caminho que foi definido. Se estivéssemos, por exemplo, explorando um sistema de arquivos, o caminho **/livros/livro/titulo** chegaria ao diretório "titulo", que é filho de "livro" que por sua vez é filho de "livros". A consulta anterior utilizou o caminho completo do elemento "titulo" para obter a resposta, porém, também é possível utilizar outros tipos de expressões para obter o mesmo resultado: **//titulo/text()**, **/livros//titulo/text(), /livros/*/titulo/text()**, **/*/livro/titulo/text()**, **/*/*/titulo/text()**. Considerando então que o primeiro elemento sempre é a raiz, "*" poderá representar qualquer elemento da expressão de caminho, por exemplo: em **/livros/*/titulo/text()**, o "*" substitui o elemento livro no caminho **/livros/livro/titulo/text()**. Da mesma forma que pode-se utilizar "//" para interpretar qualquer caminho entre a origem e o destino. Exemplificando, a consulta **/livros//paginas**, retornará todos os elementos "paginas" descendentes de "livros" em qualquer profundidade na árvore.

A linguagem XPath permite a utilização de filtros para aumentar as possibilidades de consulta e proporcionar melhores resultados. Sendo assim, é possível obter o título do livro que possui menos de 200 páginas utilizando a consulta **/livros/livro[paginas < 200]/titulo/text().** Na sintaxe das consultas os atributos são diferenciados dos elementos através do uso de "@", portanto, a consulta **//livro[@ano='2011')]/autor/nome/text()** retorna o nome dos autores cujo livro fora publicado no ano de 2011.

Existem várias funções XPath já definidas e que podem ser aplicadas, porém, o mais importante para a consulta é o caminho que será percorrido para chegar até o trecho que contém os elementos que farão parte da resposta ou que serão analisados pelos filtros para refinar o resultado. Podemos exemplificar tal aplicação das funções mostrando a utilização da função *"starts-with"* através da consulta: **//livro[@ano='2011' and starts-with(./titulo/text(), 'Steve Jobs')]/autor/nome/text()**, que retorna o primeiro nome do autor de um livro publicado em 2011 cujo título comece com "Steve Jobs" como resultado.

Outra característica da linguagem que pode ser observada é que ela não segue a sintaxe XML.

2.2.2 XQuery

Segundo Baru *et al.* (1998) no final dos anos 90 as linguagens de consultas para documentos XML estavam divididas entre dois paradigmas: o Paradigma de Banco de Dados compreendia linguagens que utilizavam sintaxe parecida com a da SQL para realizar as consultas. Já as abordagens que utilizavam expressões de caminho para explorar o documento XML seguiam o Paradigma de Programação Funcional. Esforços para mesclar essas abordagens e gerar uma perspectiva integrada foram responsáveis pelo desenvolvimento de uma linguagem de consulta denominada QUILT (CHAMBERLIN, D. *et al.*, 2000), que mais tarde deu origem à XQuery.

A XQuery foi desenvolvida pelo grupo de trabalho XML Query do W3C e em 2007 tornou-se uma recomendação deste consórcio. A linguagem XQuery utiliza tanto cláusulas bastante similares ao SQL como também expressões de caminho, e sua flexibilidade permite realizar consultas sobre apenas um documento, vários documentos ou coleções inteiras.

```
<titulos anoPublicacao = "2011">
{for $l in doc("cap2-Exemplo.xml")/livros/livro
where $l/@ano = '2011'
order by $l/titulo
return $l/titulo}
</titulos>
```

Figura 2.3: Exemplo de consulta XQuery utilizando a cláusula FOR

A estrutura da linguagem baseia-se em cinco cláusulas principais (relacionadas à vertente de banco de dados): FOR, LET, WHERE, ORDER BY e RETURN, e expressões de caminho XPath (vinculadas aos ideais da Programação Funcional). Outra característica importante da linguagem é a possibilidade de configurar uma nova estrutura para o resultado, permitindo a construção de um novo documento XML a partir das informações retornadas pela consulta.

```
<titulos anoPublicacao="2011">
    <titulo>As Esganadas</titulo>
    <titulo>Steve Jobs: A Biografia</titulo>
</titulos>
```

Figura 2.4: Resultado da consulta ilustrada na Figura 2.3

A Figura 2.3 mostra um exemplo de consulta XQuery executada sobre o documento XML ilustrado pela Figura 2.2. O resultado obtido é a listagem ordenada dos títulos dos livros publicados no ano de 2011, e é evidenciado na Figura 2.4. Pode-se observar que o elemento "títulos" e seu atributo "anoPublicação" não fazem parte do documento original, porém a linguagem XQuery permite a adição de conteúdos como estes para proporcionar a resposta desejada. A expressão FOR (assim como o LET que é apresentado mais adiante) utiliza expressões de caminho relativo (similares às utilizadas em XPath) para associar valores às variáveis (o nome das variáveis são definidos começando por "$"). Vale ressaltar que o FOR é equivalente ao termo "*from*" da linguagem SQL. Já o WHERE e o ORDER BY, assim como seus homônimos em SQL, aplicam os filtros necessários para a obtenção das respostas desejadas e realizam a ordenação do resultado, enquanto o termo RETURN define o conteúdo que será apresentado como resposta à consulta. Sua expressão equivalente em SQL seria o "*select*".

A Figura 2.5 ilustra a diferença entre os resultados obtidos com a utilização dos termos FOR e LET. Na parte superior estão as consultas e abaixo encontram-se seus respectivos resultados. Ambas as consultas irão obter como resultado a listagem de todos os títulos de livros presentes no documento ilustrado na Figura 2.2. É possível observar que o FOR cria um elemento "titulos", definido como estrutura da resposta,

para cada instância do elemento processado “titulo”, enquanto o LET engloba todo o resultado como filho de um único elemento “titulos”. Neste caso, o resultado obtido com o uso do LET é um documento XML bem-formado, enquanto a consulta através do FOR pode ser considerada apenas como o trecho de um documento devido ao fato da inexistência de uma raiz (para ser considerado um documento bem-formado, deve ser inserido um elemento raiz que engloba toda a resposta).

for $l in doc("cap2.xml")/livros/livro return <titulos> {$l/titulo} </titulos>	let $l := doc("cap2.xml")/livros/livro return <titulos> {$l/titulo} </titulos>
<titulos> <titulo>As Esganadas</titulo> </titulos> <titulos> <titulo>Steve Jobs: A Biografia</titulo> </titulos> <titulos> <titulo>Pai Rico Pai Pobre</titulo> </titulos>	<titulos> <titulo>As Esganadas</titulo> <titulo>Steve Jobs: A Biografia</titulo> <titulo>Pai Rico Pai Pobre</titulo> </titulos>

Figura 2.5: Diferença entre o uso de FOR e LET

Assim como a XPath, a XQuery não segue a sintaxe XML. Porém, sua padronização e seu poder de expressão foram responsáveis por sua disseminação no meio acadêmico. Uma extensão dessa linguagem, a *XQuery Update Facility*, tornou-se uma recomendação no início de 2011, dando a ela a capacidade de realizar alterações nos documentos XML.

2.3 *XML Schema*

Um *XML-Schema* é um documento escrito em linguagem XML, que descreve a estrutura que o documento XML deve possuir para ser válido em relação a este esquema. Dessa forma, um esquema especifica os elementos e atributos que o documento deve possuir, a ordem destes, a hierarquia que deve ser seguida e outras definições que determinam um modelo que deve ser satisfeito para que o documento seja válido.

A Figura 2.6 ilustra o esquema que valida o documento XML mostrado pela Figura 2.2. É possível notar que toda a estrutura do documento está definida, e que os elementos são classificados em dois tipos distintos: simples e complexos.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
           xmlns:xs=http://www.w3.org/2001/XMLSchema
           elementFormDefault="qualified">
  <xs:element name="livros" type="tLivros"/>
  <xs:complexType name="tLivros">
    <xs:sequence>
      <xs:element name="livro" type="tLivro"
           minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="tLivro">
    <xs:sequence>
      <xs:element name="titulo" type="xs:string"/>
      <xs:element name="autor" type="tAutor"/>
      <xs:element name="paginas" type="xs:integer"/>
      <xs:element name="editora" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="ano" type="xs:string"/>
  </xs:complexType>
    <xs:complexType name="tAutor">
    <xs:sequence>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="sobrenome" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figura 2.6: XML Schema que valida o documento ilustrado na Figura 2.2

Os elementos simples não possuem sub-elementos, ou seja, seu conteúdo é um tipo primitivo (string, integer, boolean ...). Como exemplo podemos utilizar os elementos “titulo”, “editora”, “nome” e “sobrenome” da Figura 2.6, pois são elementos simples do tipo *string*, cujo conteúdo será textual. Outro exemplo seria o elemento “paginas”, pois também é um elemento simples, porém, seu conteúdo deverá ser *integer* para que o documeto seja válido. Pode-se dizer que os elementos simples são as folhas da árvore que representa o documento XML.

Um elemento simples em um XSD apresenta-se da forma:

<xs:element name="nome_do_elemento" type="tipo_do_elemento"/>

Já os elementos complexos são elementos que possuem outros elementos como filhos, alterando a estrutura do documento com a criação de uma nova estrutura à partir dele. Sendo assim, a linguagem XSD utiliza delimitadores de grupo para determinar que forma terá esta nova estrutura que descende deste elemento complexo. Como exemplo podemos observar os elementos “livros”, “livro” e “autor” da Figura

2.6. Todos eles estão ligados a um tipo que foi definido por um *complexType*. Cada tipo define quais são os sub-elementos dos elementos daquele tipo, e também como estes elementos podem aparecer. Para isso, é necessário especificar delimitadores de grupo dentro dos tipos complexos. Pode-se citar três tipos de delimitadores de grupo:

Sequence: esse delimitador de grupo define que todos os filhos do elemento complexo devem estar presentes no documento XML, na mesma ordem em que estão definidos no esquema;

Choice: o elemento complexo delimitado por *choice* terá como filho apenas uma das opções descritas por este delimitador; e,

All: os filhos do elemento complexo podem ser apresentados em qualquer ordem e apenas uma vez cada.

A definição de um elemento complexo possui a seguinte forma:

```
<xs:element name=”nome_do_elemento” type=”tipo_complexo”/>
<xs:complexType name=” tipo_complexo “>
   <delimitador_de_grupo>
      <xs:element name=”nome_do_elemento_filho” type=”tipo_do_filho”/>
      <xs:element name=”nome_do_elemento_filho” type=”tipo_do_filho”/>
      (...)
      <xs:element name=”nome_do_elemento_filho” type=”tipo_do_filho”/>
   </delimitador_de_grupo >
</xs:complexType>
```

A cardinalidade do elemento, sendo ele simples ou complexo, é determinada pelos atributos *minOccurs* e *maxOccurs*, determinando respectivamente a ocorrência mínima e a máxima do elemento. A definição de *maxOccurs* como *unbounded* implica na não determinação de limite máximo para a ocorrência do elemento. Como exemplo temos o elemento “livro” da Figura 2.6, que pode aparecer várias vezes como filho de “livros”.

Vale dizer então que os esquemas definem o vocabulário que deve ser seguido pelos documentos XML. Nessa dissertação, os documentos que serão alvos de consulta podem estar ligados a um esquema. Neste caso, a informação presente no esquema é aproveitada por nossa abordagem para a geração de regras automáticas, com o objetivo de auxiliar o usuário no momento da consulta.

2.4 Prolog

A linguagem Prolog é uma linguagem de programação baseada nos conceitos da Programação em Lógica, que foi criada no início da década de 70 com base nas cláusulas de Horn (KEISLER, 1965). Trata-se de uma linguagem declarativa que descreve sua base de conhecimento através de fatos e regras.

Em Prolog, um fato é definido pelo seu nome e seus argumentos. A Figura 2.7 ilustra a ocorrência de alguns fatos Prolog.

```
pai(pedro , joao).
pai(paulo, pedro).
pai(ana, pedro).
homem(pedro).
mulher(ana).
```

Figura 2.7: Exemplos de fatos Prolog

Se fosse perguntado a essa base de conhecimento quem possui "pedro" como pai (?- pai(X, pedro).) a resposta seria: X = paulo e X = ana. Assume-se que todo o conhecimento não definido é falso (hipótese do mundo fechado), portanto, a resposta à consulta se "joao é homem" é falsa, uma vez que este fato não consta na base de conhecimento.

As regras Prolog são estruturas que podem ser divididas em **cabeça** e **corpo**, separados por ":-". Em uma regra, a cabeça é válida apenas se o corpo for válido. Por exemplo, "filho_homem(joao, pedro) :- filho(joao, pedro), homem(pedro)." define que "pedro é filho homem de joao se pedro é filho de joao e pedro é homem". Esta regra poderia ser generalizada trocando as constantes "joao" e "pedro" por duas variáveis. A Figura 2.8 ilustra a regra "ancestral" em Prolog.

```
ancestral(X, Y) :- pai(X, Y).
ancestral(X, Y) :- pai(X, Z), ancestral(Z, Y).
```

Figura 2.8: Exemplo de regras Prolog

O resultado da consulta sobre "de quem joao é ancestral" (?- ancestral(joao, X).) a partir da base de conhecimento formada pelos fatos daFigura 2.7 e regras da Figura 2.8 seria X = pedro; X = paulo; X = ana.

A interação com o usuário é feita através de consultas sobre as relações (fatos e regras, também chamados de Base de Conhecimento) descritas pelo programa Prolog previamente carregado. Tais consultas utilizam o conceito de unificação lógica (ROBINSON, 1965) para realizar as combinações necessárias para obter a resposta

esperada. Outros benefícios da programação em lógica tais como recursividade e *backtracking* são utilizados pela linguagem para proporcionar maior poder de consulta. Outra característica da linguagem, seu poder de inferência provém do princípio da resolução no qual o Prolog se baseia.

O restante desta seção explica cada um desses conceitos que envolvem a execução dos programas em Prolog.

Inferência

A inferência lógica é o processo de deduzir novas conclusões a partir das informações que são tidas como verdade. A combinação destas informações é capaz de gerar um novo conhecimento, por exemplo: a combinação dos fatos: "João é pai de Pedro" e "Pedro é pai de Paulo" com a regra "o avô de uma pessoa é o pai de seu pai" permite inferir que "João é avô de Paulo". Essa informação não está contida na base de conhecimento, porém, pode ser obtida através da máquina de inferência.

Backtracking

O algoritmo de *backtracking* determina pontos em um programa Prolog para onde a execução irá retornar em caso de alguma falha. Em Prolog, a base de conhecimento pode ser representada por uma árvore, e as consultas são realizadas através de uma busca em profundidade. Utilizando o algoritmo de *backtracking*, após a ocorrência de uma falha em determinado ramo desta árvore, a execução irá descartá-lo e automaticamente tentará buscar uma solução alternativa a partir do ponto de retorno estabelecido.

Recursividade

Em Prolog, a recursão ocorre através de uma regra que chama a si própria para obter uma informação. Um exemplo clássico é a regra que descreve os ancestrais/descendentes de uma pessoa, que utiliza recursão para caminhar na árvore genealógica: "Uma pessoa X é ancestral de uma outra pessoa Z se ((X é pai de Z) ou (X é pai de Y e Y é ancestral de Z))". Neste exemplo, se a primeira condição da regra (X é pai de Z) não for válida, a regra checa quem é o filho correspondente (o equivalente a dar um passo na árvore genealógica) e chama novamente a regra para ver se é suficiente ou deverá seguir para o próximo nível da árvore genealógica.

Unificação lógica

A unificação lógica é uma das principais ideias sobre as quais a linguagem Prolog se baseia. Baseado nesta ideia, o mecanismo de inferência Prolog utiliza estratégias de associação das informações contidas em sua base de conhecimento com a consulta submetida, com o objetivo de obter novas informações. Sua concepção representa a atribuição de valores às variáveis através de combinações, fazendo com que expressões diferentes tornem-se similares. Em Prolog, um átomo é uma sequência de letras, números e *underscore* ("_") que deve ser iniciada com uma letra minúscula, já uma variável pode conter os mesmos elementos que um átomo, porém, inicia-se obrigatoriamente com uma letra maiúscula ou por *underscore*.

A unificação assegura regras como:

- Se X = Y e Y = "ana" então X e Y são unificados pelo átomo "ana"
- Se f(X) = f(a) então X é unificado com "a"
- Se f(g(X)) = f(Y) então Y é unificado com g(X)

Como exemplo, as listas [1, [3, X, 7], X, Z] e [W, Y, [1, Z], 2] podem ser unificadas tendo como resultado: W = 1, X = [1, 2], Y = [3, [1, 2], 7] e Z = 2.

2.5 Considerações Finais

O estudo de todas estas tecnologias contribuiu para a aquisição do conhecimento básico e necessário para a continuação do desenvolvimento deste trabalho.

Este capítulo apresentou as tecnologias principais que foram utilizadas durante a elaboração desta pesquisa, possibilitando a definição do seu escopo inicial, e a determinação das primeiras metas. O estudo destas tecnologias será importante tanto para a compreensão dos exemplos utilizados nos demais capítulos, como para um melhor entendimento do método proposto e de sua implementação.

Concluindo, podemos dizer que para o desenvolvimento do projeto apresentado, foi necessária a busca por mais informações a respeito do problema identificado e da área de atuação desta pesquisa. Foi através dos trabalhos relacionados,

que serão apresentados no próximo capítulo, que complementamos o conhecimento necessário para a consolidação e finalização da ideia principal.

Capítulo 3. Trabalhos Relacionados

3.1 Introdução

Para melhor fundamentar esta pesquisa, foi necessária a busca por trabalhos que já tenham explorado a integração entre as linguagens XML e Prolog. Tais trabalhos poderiam contribuir com detalhes sobre os benefícios provenientes desta conbinação e com ideias de como dados originalmente em XML poderiam ser processados em Prolog. Constatamos que esta é uma área que já foi explorada e que trouxe como conseqüência a consolidação de diversos trabalhos com diferentes objetivos.

A ideia para o desenvolvimento do trabalho descrito nesta dissertação originou-se da possibilidade de explorar os benefícios da Programação em Lógica (Inferência, Recursividade, Reutilização de regras, etc.) (LLOYD, 1987) para obter respostas mais elaboradas em consultas realizadas sobre dados semi-estruturados. A partir de então, o estudo realizado revelou que a utilização de Programação em Lógica para facilitar a representação e manipulação de dados é uma prática adotada há bastante tempo (CERI *et al.*, 1990; CHANG; WALKER, 1986). Já em 1991 Niemi e Jarvelin (1991) propuseram a adoção de Prolog para representar, pela primeira vez, toda a base de conhecimento proveniente de um banco de dados relacional.

No final do século XX a disseminação da Internet, a globalização da informação e a padronização da representação de dados semi-estruturados em linguagem XML resultaram no surgimento de um volume significativo de dados descritos neste formato, incentivando esforços com a finalidade de melhorar a manipulação dos mesmos. Neste cenário, não demorou muito para o surgimento de estudos que integrassem o processamento de dados XML com a Programação em Lógica, a fim de demonstrar certa semelhança na representação dos dados e os benefícios provenientes desta combinação (ALMENDROS-JIMÉNEZ *et al.*, 2008; BAILEY *et al.*, 2005; SEIPEL, 2002).

Já em 1999, Calvanese et al. (CALVANESE *et al.*, 1999) propuseram uma técnica capaz de representar DTDs em Lógica Descritiva (DONINI *et al.*, 1996) com o objetivo de melhorar os tempos de verificação da conformidade dos documentos e da

análise de equivalência entre as DTDs. O trabalho deles consiste em representar o vocabulário dos documentos XML em lógica descritiva através de regras de tradução entre as sintaxes. Foi demonstrado que seria possível utilizar esta ideia para aumentar a eficiência das consultas sobre bases de documentos XML. O estudo então definiu que uma consulta a uma base de documentos com o objetivo de retornar os documentos que satisfaçam os requisitos dessa consulta pode ser vista como uma DTD, e seu resultado será o conjunto de documentos válidos para esta DTD.

A busca por trabalhos que pudessem contribuir com a nossa pesquisa resultou na identificação de diversas abordagens que utilizaram a integração entre as linguagens XML e Prolog com um determinado objetivo. A seguir, é apresentada cada uma destas abordagens, que foram analisadas e serviram para a consolidação do trabalho apresentado.

3.2 Abordagens que integram as linguagens XML e Prolog

A utilização de Programação em Lógica para manipular dados semi-estruturados, especificamente em linguagem XML, deve sempre levar em consideração uma tarefa essencial: a representação destes dados em uma linguagem lógica. A XML é uma linguagem auto-descritiva e com total flexibilidade quanto à sua estrutura, portanto, representá-la em uma linguagem Lógica pode não ser uma tarefa trivial. Por causa da importância dessa tarefa, foram realizados estudos para que fosse possível obter bons resultados. Assim, o restante desta seção vem mostrar o que cada um dos autores estudados como referência utilizou para solucionar essa questão.

3.2.1 XmlLog

Boley (2000) comprovou em seu trabalho que existe uma forte relação entre Programação em Lógica e XML, utilizando o que ele chamou de *ground structure*. A Figura 3.1 ilustra tal similaridade apontada por Boley. A parte superior da figura contém dois trechos em XML (um elemento simples e um complexo) e a parte inferior contém a representação destes em linguagem Prolog. A partir de então, Boley utilizou uma representação de termos Herbrand e cláusulas de Horn (LLOYD, 1987) em linguagem XML para demonstrar como seria sua representação em Prolog puro, denominada XmlLog.

```
<sentence> Onoffbook sold 12417 copies of XML4You online </sentence>


<triple>
  <subject> Onoffbook </subject>
  <predicate> sold online </predicate>
  <object> 12417 copies of XML4You </object>
</triple>
```

```
sentence("Onoffbook sold 12417 copies of XML4You online")


triple(
  subject("Onoffbook"),
  predicate("sold online"),
  object("12417 copies of XML4You")
)
```

Figura 3.1: Representação de estruturas XML em Prolog – Fonte: (BOLEY, 2000)

```
<struc>
  <constructor>service-tunnel</constructor>
  <struc>
    <constructor>undersea-connection</constructor>
    <ind>britain</ind>
    <struc>
      <constructor>surrounded-country</constructor>
      <ind>belgium</ind>
      <ind>luxembourg</ind>
      <ind>germany</ind>
      <ind>switzerland</ind>
      <ind>italy</ind>
      <ind>spain</ind>
    </struc>
  </struc>
</struc>
```

```
service-tunnel(
  undersea-connection(
    britain,
    surrounded-country
        (belgium,luxembourg,germany,switzerland,italy,spain)))
```

Figura 3.2: Documento XML em termos Herbrand e sua representação em Prolog – Fonte: (BOLEY, 2000)

A utilização da linguagem XML para definir as informações através de termos Herbrand é possível pelo uso dos elementos <ind>, <var> e <struct>, representando respectivamente constantes, variáveis e estruturas. A Figura 3.2 mostra um exemplo de um documento XML definido em termos Herbrand e sua representação em Prolog. Os elementos <struct> denotam estruturas em Prolog, os <constructor> o nome dos construtores e os <ind> referem-se às constantes. Este exemplo não contém elementos <var>, porém, é intuitivo que eles implicam variáveis Prolog.

Já a representação de Cláusulas de Horn em XML pode ser utilizada para representar fatos e regras, segundo Boley. Nesse caso, toda estrutura é englobada pelo elemento <hn>, o predicado por <relationship>, e sua cabeça por <relator>. Novamente o elemento <ind> define as constantes e <var> as variáveis. Nessa abordagem os fatos são representados por apenas um predicado (<relationship>), enquanto as regras possuem um predicado que representa sua cabeça seguido de pelo menos mais um que define seu corpo. A Figura 3.3 mostra um documento XML descrito em Cláusulas de Horn e a regra Prolog correspondente (de acordo com o estudo de Boley).

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

```
travel(Someone,channel-tunnel) :-
    carry(eurostar,Someone).
```

Figura 3.3: Documento XML em Cláusulas de Horn e sua representação em Prolog – Fonte: (BOLEY, 2000)

O autor mostrou ainda como realizar consultas sobre documentos XML (utilizando XQL) e sobre o XmlLog. A abordagem de Boley adotou a representação dos dados em Prolog de forma não granular, ou seja, utilizando predicados extensos referentes a um trecho significativo do documento XML, divergindo assim da forma granular que adotamos com o objetivo de facilitar a realização de consultas Prolog.

3.2.2 Field-Notation

Na abordagem proposta por Seipel (2002) é apresentado um modelo denominado *field-notation*, que utiliza lista de associação para representar objetos complexos provenientes de documentos XML em Prolog. Esse método gera uma lista com todo o conteúdo do documento XML em uma estrutura de predicados aninhados e na forma "atributo:valor", conforme ilustrado na Figura 3.4.

A partir desta representação, seu trabalho definiu ainda uma biblioteca Prolog chamada FNPath, capaz de selecionar elementos nas listas de associação, utilizando

expressões de caminho na árvore e considerando a estrutura aninhada das listas. A parte superior da Figura 3.5 mostra uma consulta a uma lista de associação onde a variável "O" é a base de conhecimento, e "X", "Y" e "Z" são consultas que visam obter os valores de suas expressões de caminho. Tais consultas utilizaram a biblioteca FNPath definida pela pesquisa de Seipel para selecionar os resultados desejados. No entanto se fosse realizada uma consulta em Prolog puro, a terceira consulta ("Z := O^chart@wkn") seria na forma:

O = _:_:X, member(chart:Y:_,X), member(wkn:Z, Y).

```
<stocks index="dax100">
  <chart wkn="200400">
    <entry date="14.12.2002" value="30"/>
  </chart>
  <chart wkn="600800">
    <entry date="14.12.2002" value="40"/>
    <entry date="15.12.2002" value="50"/>
  </chart>
</stocks>
```

```
stocks:[index:dax100]:[
  chart:[wkn:200400]:[
    entry:[date:'14.12.2002',
value:30]:[] ],
  chart:[wkn:600800]:[
    entry:[date:'14.12.2002',
value:40]:[],
    entry:[date:'15.12.2002',
value:50]:[] ] ]
```

Figura 3.4: Documento XML e sua representação em *Field-Notation* – Fonte: (SEIPEL, 2002)

```
?- O = stocks:[index:dax100]:[
   chart:[wkn:200400]:[
      entry:[date:'14.12.2002', value:30]:[]]],
   X := O^chart, Y := O@index, Z := O^chart@wkn
```

```
X = chart:[wkn:200400]:[
      entry:[date:'14.12.2002', value:30]:[]],
Y = dax100,
Z = 200400
```

Figura 3.5: Consulta à lista de associação utilizando expressões de caminho – Fonte: (SEIPEL, 2002)

A pesquisa de Seipel nos mostrou que o tipo de representação dos dados XML em linguagem Prolog iria interferir diretamente na forma que a consulta seria realizada e que existiriam diversas alternativas de representações, sendo uma delas o uso de listas de associação. Isso nos fez enxergar que a criação do nosso próprio método de tradução voltado à permitir consultas simples sobre a base de conhecimento gerada possibilitaria

a diminuição do trabalho de elaboração das consultas. Consideramos que essa seria então uma boa alternativa para satisfazer nossos objetivos. A possibilidade de tradução dos XML *Schemas* em regras Prolog e de inserção de novas regras contribuíram para reforçar a necessidade de um método de tradução próprio.

3.2.3 Inferência de tipos

O trabalho de Coelho e Florido (2003) propõe um *framework* capaz de realizar transformações em um documento XML através de inferências de tipos guiadas por DTDs. Este *framework* recebe duas DTDs (DTD_*input* e DTD_*output*) como entrada e é capaz de transformar um documento XML válido por DTD_*input* em outro válido por DTD_*output*.

O processo inicia-se com tradução do documento XML válido por DTD_*input* para termos Prolog, e das DTDs para tipos regulares (DART; ZOBEL, 1992) conforme ilustrado pela Figura 3.6. Após estas etapas, o *framework* realiza uma verificação de tipos das informações para enfim transformar os termos processados em Prolog para o documento XML válido pela DTD_*output*. Os tipos regulares derivados das DTDs realizam as combinações necessárias para auxiliar os termos Prolog a representar as informações do documento XML gerado como resultado da transformação.

Documento XML:

```
<teachers>
  <name>Jorge Coelho</name>
  <office>403</office>

<email>jcoelho@isep.ipp.pt</email>
  <name>Mario Florido</name>
  <office>202</office>
</teachers>
```

DTD:

```
<!ELEMENT teachers
(name,office,email?)*>
<!ELEMENT name #PCDATA>
<!ELEMENT office #PCDATA>
<!ELEMENT email #PCDATA>
```

Predicado Prolog:

```
teachers([
  (name(``Jorge Coelho''),
  office(``403''),
  email(``jcoelho@isep.ipp.pt'')),
  (name(``Mario Florido''),
  office(``202''))])
```

Tipos Regulares:

```
T1 -> {teachers(T2)}
T2 -> {nil,.(T3,T2)}
T3 -> {T4,T5}
T4 -> {(name(string),
   office(string),email(string))}
T5 -> {(name(string),
   office(string))}
```

Figura 3.6: Tradução de documento XML em Prolog e de DTD em tipos regulares – Fonte: (COELHO; FLORIDO, 2003)

Na abordagem proposta por Coelho e Florido a tradução das DTDs em tipos regulares gerou um conjunto de regras a respeito da estrutura do documento XML,

como pode ser visto na Figura 3.6, que foram combinadas com os termos Prolog para que o objetivo fosse alcançado.

A derivação de tipos regulares a partir das DTDs nos revelou que a tradução dos esquemas em regras automáticas seria importante para consolidar a nossa base de conhecimento, ainda que os nossos objetivos fossem diferentes. Observamos também que assim como o nosso trabalho, Coelho e Florido realizaram inferências com a ajuda da linguagem Prolog para obter informações que não estavam explicitadas nos documentos, contribuindo para nos fazer entender que o que almejávamos era viável.

Em relação à tradução adotada, mais uma vez foi utilizada uma tradução não granular, contrastando com o nosso objetivo de produzir um resultado de tradução capaz de facilitar tanto a criação de regras manuais quanto a realização das consultas.

Conforme será explicado no próximo capítulo, após alguns testes nós observamos que a utilização de predicados granulares facilitaria tanto o trabalho de configuração das regras manuais quanto de realização das consultas, uma vez que permitiria a utilização apenas dos predicados Prolog desejados, e possibilitaria diversas combinações entre estes predicados.

3.2.4 Implementação de XPath em Programação em Lógica

Dentre os trabalhos relacionados, nenhum possui características tão semelhantes ao nosso projeto quanto o proposto por Almendros-Jiménez et al. (2008). Sua pesquisa propõe a representação de documentos XML em linguagem Prolog e a implementação de uma máquina de consultas XPath através desta mesma linguagem lógica, proporcionando assim, um maior poder de consulta e obtenção de resultados que não poderiam ser obtidos através de consultas tradicionais em XPath. O processo de tradução desenvolvido por eles assemelha-se ao nosso pelo fato de traduzir esquemas em regras Prolog e documentos XML em fatos, e ainda por utilizar tradução granular. Essa tradução, porém, numera os atributos e elementos do documento XML para diferenciar a sua representação, relacioná-los aos seus pais e facilitar a consulta.

Desta forma, cada nó da árvore que representa o documento XML a ser processado é numerado recebendo um atributo *nodenumber* e outro *typenumber*. O *nodenumber* inicia-se com valor “1” (sendo este a raiz da árvore). Cada nó filho possui o *nodenumber* igual ao do pai acrescido de um ponto (para representar um novo nível) e

de uma ordem pela qual se apresenta na árvore (que é calculada da esquerda para a direita). Portanto, na árvore que representa o documento XML, a raiz terá *nodenumber* = 1, seu filho mais a esquerda terá *nodenumber* = 1.1, o segundo filho do seu terceiro filho terá *nodenumber* = 1.3.2 e assim por diante.

```
<books>
  <book year="2003">
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
    <title>Data on the Web</title>
    <review>A <em>fine</em> book.</review>
</book>
  <book year="2002">
    <author>Buneman</author>
    <title>XML in Scotland</title>
    <review>
      <em>
        The
        <em>best</em>
        ever!
</em></review></book></books>
```

Figura 3.7: Exemplo de documento XML – Fonte: (ALMENDROS-JIMÉNEZ *et al.*, 2008)

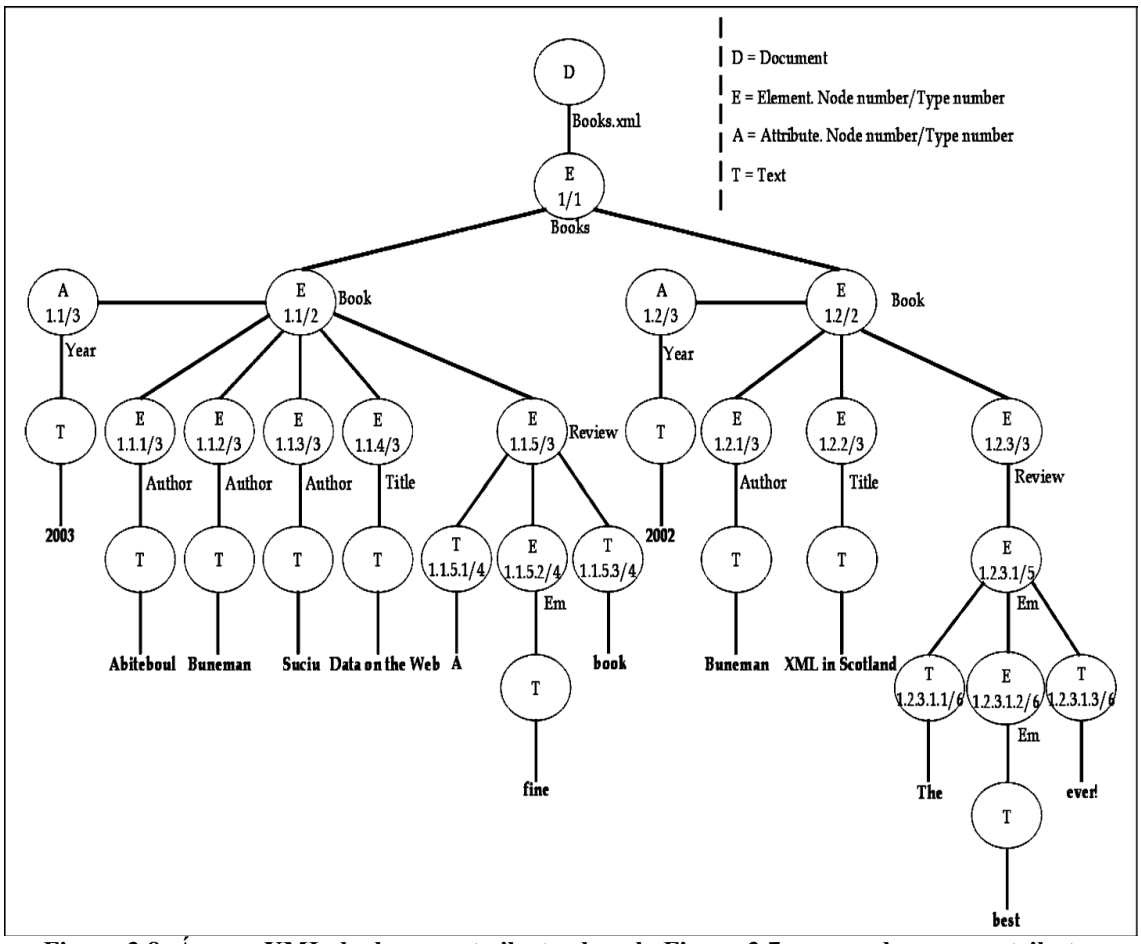

Figura 3.8: Árvore XML do documento ilustrado pela Figura 3.7 numerada com os atributos *nodenumber* e *typenumber* – Fonte: (ALMENDROS-JIMÉNEZ *et al.*, 2008)

Já o atributo *typenumber* é igual a $l + n + 1$, onde l é o *typenumber* do nó pai e n é o número de nós à esquerda na árvore XML que possuem o mesmo nome do nó analisado, porém, estrutura diferente (chamado por Almendros-Jiménez de fracamente distintos).

A Figura 3.7 ilustra um exemplo de documento XML e a Figura 3.8 mostra sua árvore XML com os nós já numerados. É possível assim observar que cada nó atributo (A) e elemento (E) possuem sua numeração da forma *nodenumber*/*typenumber*.

Regras (Schema):

```
books(bookstype(Books, []), NodeBooks,1) :-
   book(Books, [NodeBook|NodeBooks],2).
book(booktype(Author, Title, Review, [Year]), NodeBook ,2) :-
   author(Author, [NodeAuthor|NodeBook],3),
   title(Title, [NodeTitle|NodeBook],3),
   review(Review, [NodeReview|NodeBook],3),
   year(Year, NodeBook,3).
review(reviewtype(Unlabeled,Em,[]),NodeReview,3):-
   unlabeled(Unlabeled,[NodeUnlabeled|NodeReview],4),
   em(Em,[NodeEm|NodeReview],4).
review(reviewtype(Em,[]),NodeReview,3):-
   em(Em,[NodeEm|NodeReview],5).
em(emtype(Unlabeled,Em,[]),NodeEms,5) :-
   unlabeled(Unlabeled,[NodeUnlabeled|NodeEms],6),
   em(Em, [NodeEm|NodeEms],6).
```

Fatos (Documento XML):

```
year('2003', [1, 1], 3).
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2, 1, 1], 3).
author('Suciu', [3, 1, 1], 3).
title('Data on the Web', [4, 1, 1], 3).
unlabeled('A', [1, 5, 1, 1], 4).
em('fine', [2, 5, 1, 1], 4).
unlabeled('book.', [3, 5, 1, 1], 4).
year('2002', [2, 1], 3).
author('Buneman', [1, 2, 1], 3).
title('XML in Scotland', [2, 2, 1], 3).
unlabeled('The', [1, 1, 3, 2, 1], 6).
em('best', [2, 1, 3, 2, 1], 6).
unlabeled('ever!', [3, 1, 3, 2, 1], 6).
```

Figura 3.9: Regras e fatos traduzidos a partir do documento ilustrado na Figura 3.7 – Fonte: (ALMENDROS-JIMÉNEZ *et al.*, 2008)

Após a numeração dos nós da árvore XML inicia-se o processo de tradução. De acordo com o trabalho proposto por ALMENDROS-JIMÉNEZ *et al.*, (2008), para cada nó não terminal (nós cujos filhos não sejam apenas as folhas da árvore) do tipo:

<tag att1 = v1 ,... , attn = vn, nodenumber = i, typenumber = k>elem1, ..., elems</tag>

a tradução ocorre segundo a regra:

tag(tagtype(Tagi1, ..., Tagit, [Att1, ..., Attn]),NodeTag, k):-
tagi1(Tagi1, [NodeTagi1|NodeTag], r), ..., tagit(Tagit, [NodeTagit|NodeTag], r),
att1(Att1 ,NodeTag, r), ..., attn(Attn ,NodeTag, r).

onde:

- *Tagtype* é uma nova função usada pra construir um TR
- {tagij|$ij \in \{1, \ldots, s\}, 1 \leq j \leq t\}$ é o conjunto das tags dos elementos *elem*1,..., *elems*;
- *NodeTag* é a variável que representa o *nodenumber* do nó;
- r é o *tyoenumber* do elemento em *elem*1,..., *elems*.

Onde ainda são gerados fatos na forma: *attj(vj, i, k).*

Já no caso de nós terminais (nós cujos filhos são tipos básicos), cuja forma é:

<tag nodenumber = i, typenumber = k>value</tag>

são gerados fatos conforme a regra: *tag(value, i, k).*

A Figura 3.9 mostra o resultado da tradução do documento ilustrado na Figura 3.7 de acordo com as regras supracitadas. Há uma forte similaridade com a tradução que adotamos em nosso trabalho, pois além de fatos granulares, são derivadas diversas regras que podem auxiliar o processo de realização das consultas, tendo as abordagens como principal diferença a forma em que é feita a indexação dos fatos, o que implica diretamente na forma com que fatos e regras se relacionam entre si. Enquanto a abordagem estudada numerou os nós através de *nodenumber* e *typynumber* levando em consideração sua posição na árvore XML, nosso método gerou "ids" seqüenciais para identificar os fatos, e acrescentou a variável ID às regras para tornar possível as combinações necessárias para vinculá-las aos fatos.

A Figura 3.10 ilustra como fica o resultado da tradução do documento XML mostrado pela Figura 3.7 caso ele seja processado pela nossa abordagem. Uma comparação com os fatos descritos na Figura 3.9 revela que nossa tradução gera fatos também sobre os elementos não terminais, o que não é realizado por Almendros-Jiménez. Com isso, se desconsiderarmos tais fatos, teremos duas bases de conhecimento iguais ainda que com uma sintaxe um pouco diferente, como é notado na indexação e na

tradução dos elementos mistos, onde é adotado o fato "*unlabeled*", enquanto utilizamos "xml/elementoMisto" conforme explicado no Capítulo 4.

```
books(id1).
book(id1, id2).
year(id2, '2003').
author(id2, 'Abiteboul').
author(id2, 'Buneman').
author(id2, 'Suciu').
title(id2, 'Data on the Web').
review(id2, id3).
em(id3, id4).
xml/elementoMisto(id4, 'A').
em(id4, 'fine').
xml/elementoMisto(id4, 'book').
book(id1, id5).
year(id5, '2002').
author(id5, 'Buneman').
title(id5, 'XML in Scotland').
review(id5, id6).
em(id6, id7).
xml/elementoMisto(id7, 'The').
em(id7, 'best').
xml/elementoMisto(id4, 'ever!').
```

Figura 3.10: Tradução do documento XML ilustrado na Figura 3.7 de acordo com o método descrito nesta dissertação.

É importante ressaltar, no entanto, que os propósitos das duas abordagens são bastante distintos. Enquanto a abordagem de Almendros-Jiménez *et al.* (2008) propõe o uso de Prolog para processar consultas XPath, nossa abordagem propõe o uso de Prolog para obter respostas a consultas a dados implícitos nos documentos XML.

3.3 Considerações Finais

O conhecimento adquirido a partir da leitura dos trabalhos mencionados foi de fundamental importância para a formatação do projeto proposto. Foi através da obtenção de tais conceitos que pudemos aprimorar a nossa ideia inicial, nos certificando das escolhas feitas durante cada passo desenvolvido.

Vale dizer que, ainda que alguns dos trabalhos apresentados não tivessem o objetivo de realizar consultas, eles foram extremamente significativos para a compreensão da relação que pôde ser construída entre as linguagens Prolog e XML, tornando possível a elaboração de um método de tradução capaz de produzir informações de acordo com o desejado.

Ressaltamos então que foi através da utilização das contribuições explanadas que realizamos o desenvolvimento de nossa ideia principal, combinando o que foi acrescentado por estas com o aprendizado adquirido. Assim, passaremos ao próximo capítulo, onde será devidamente explorada a abordagem proposta nesta dissertação.

Capítulo 4. XMLInference

4.1 Introdução

Este capítulo apresenta a abordagem que desenvolvemos para possibilitar a realização de consultas em documentos XML com o auxílio de inferências e reaproveitamento de regras, explicando cada passo do processo e expondo exemplos para facilitar o seu entendimento. O método envolve a tradução de dados semi-estruturados, especificamente XML, para uma linguagem lógica capaz de definir além destes dados, regras de inferência que possibilitam a conclusão de novas informações e proporcionam respostas mais elaboradas a consultas que antes só analisavam as informações explícitas no documento XML.

Esse método pode ser aplicado a qualquer documento XML. Caso haja um esquema que valide o documento, suas informações também serão aproveitadas. Nossa abordagem inclui a tradução das informações a respeito da estrutura do documento contidas no esquema, para regras que poderão facilitar futuras consultas. Isso pode ser visto em detalhes na Seção 4.4.1.

O restante deste capítulo está organizado em quatro subseções. A Seção 4.2 apresenta uma visão geral da nossa abordagem, explicando as etapas existentes e a ordem em que as atividades são executadas. A tradução dos dados definidos em linguagem XML para a linguagem lógica escolhida é explicada na Seção 4.3. Na Seção 4.4 é discutido como ocorre a definição das regras que auxiliam o processo de inferência e a realização da consulta. Por fim, a Seção 4.5 encerra o capítulo apresentando as suas considerações finais.

4.2 Um Método de Consulta Capaz de Inferir Dados

A abordagem proposta neste trabalho processa os dados contidos na estrutura do documento XML com o objetivo de inferir novas informações que passam a fazer parte da base de conhecimento, possibilitando a obtenção de respostas complexas a partir de informações explícitas no documento ou deduzidas automaticamente através de

regras (informações implícitas). Para isso, são derivadas regras a partir dos esquemas que definem o vocabulário a ser seguido pelos documentos XML e ainda novas regras mais específicas são inseridas manualmente. Nosso objetivo é viabilizar a inferência sobre informações contidas na base de conhecimento, estendendo as possibilidades de consulta e facilitando a obtenção de resultados que demandariam consultas complexas e grandes, através da utilização dos métodos de consultas tradicionais.

O processo fundamental para o sucesso da abordagem é a tradução dos dados representados em linguagem XML para uma linguagem lógica que proporcione poder de inferência. Surgem então duas questões: "que linguagem escolher?" e "como traduzir os dados sem perder a semântica existente na estrutura do documento XML?". O parágrafo a seguir responde à primeira pergunta, enquanto a segunda será respondida nas seções seguintes, ainda neste capítulo.

Para realizar inferências, precisamos traduzir os dados contidos no documento XML para alguma linguagem que nos forneça tal capacidade. As linguagens RuleML (BOLEY, 2001), Datalog (GALLAIRE; MINKER, 1978), RIF (KIFER, 2008) e Prolog (COLMERAUER *et al.*, 1973) possibilitam inferências e foram estudadas neste trabalho.

A Datalog é uma linguagem de consulta fortemente baseada em Prolog, que está mais voltada para a área de banco de dados. A sintaxe adotada é similar à sintaxe do Prolog. A linguagem possui algumas vantagens sobre o SQL, porém, quando comparado ao Prolog algumas desvantagens são evidenciadas. O fato de Datalog não permitir termos complexos como argumento de um predicado, e possuir restrições no uso de negação e recursividade, tornou-a incompatível com a abordagem proposta neste trabalho.

A RuleML é o resultado de um esforço para fornecer um padrão de definição de regras na Internet. A linguagem descreve tanto as informações como os seus relacionamentos, tornando possível a realização de inferência. O fato da RuleML ser uma linguagem de marcação voltada para representar regras de inferências com foco em Ontologias na Web nos fez perceber que o uso de uma linguagem lógica se adequava melhor aos nossos objetivos. Os estudos sobre a linguagem RuleML indicaram que suas implementações atuais utilizam uma máquina Datalog como mecanismo de inferência, o que nos fez descartar seu uso em nosso trabalho.

A Rule Exchange Format ou simplesmente RIF (KIFER, 2008) por outro lado, é uma recomendação W3C para representar as regras através da internet. Sendo esta uma proposta muito recente, algumas pesquisas e tecnologias relacionadas encontram-se ainda em desenvolvimento.

Tivemos resultados posistivos em nossas tentativas de utilizar a linguagem Prolog e sua máquina de inferência para realizar a tarefa de interpretar dados implícitos e definir novas informações a partir de documentos XML. Isso somado ao fato do Prolog ser uma linguagem extremamente difundida no meio acadêmico e proporcionar os benefícios provenientes da programação em lógica, chegamos à conclusão de que seria a linguagem ideal para representar os dados e processar as inferências.

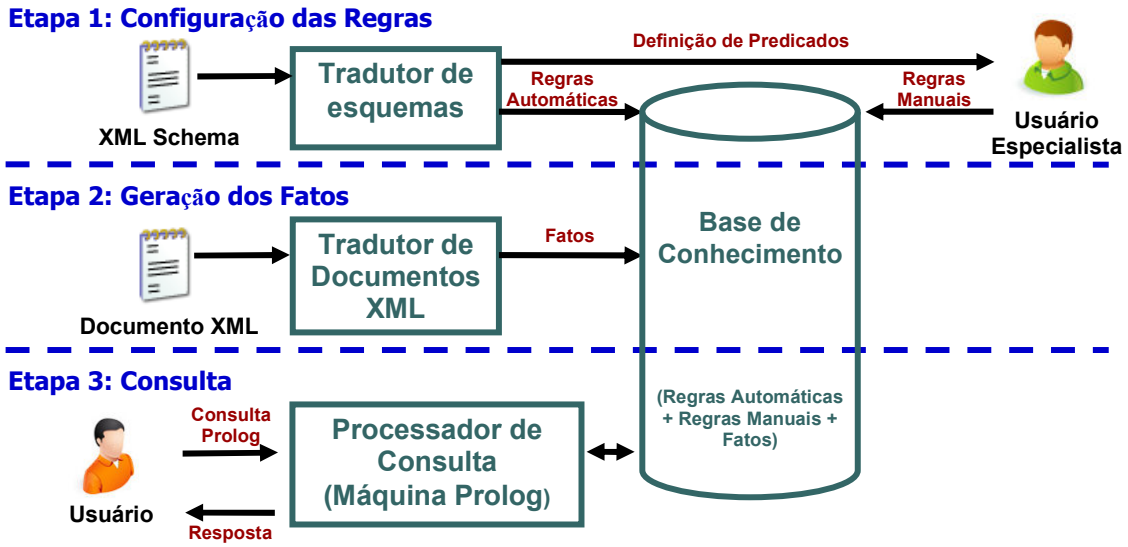

Figura 4.1: Etapas da abordagem proposta

O nosso método é dividido em três etapas: configuração das regras, geração dos fatos e consulta. A Figura 4.1 ilustra tais etapas e suas fases. Durante a etapa de configuração das regras, o usuário deve informar ao sistema os esquemas que validam os documentos XML que serão alvo das consultas. Estes esquemas são analisados por um tradutor de esquemas que gera regras Prolog (denominadas regras automáticas) a partir dos dados processados, além de possíveis definições de predicados. Utilizando estas definições, que são as assinaturas dos predicados Prolog, um usuário especialista (com conhecimento no domínio e em Prolog) pode gerar novas regras, manualmente. Estas regras estão relacionadas com o tipo de informação que se deseja obter. Note que normalmente estas regras definem formas de se obter informações que não podem ser obtidas a partir apenas das informações que estão disponíveis (isto é, explícitas) nos documentos XML. O objetivo desta inserção é aumentar as possibilidades de inferência

e consequentemente o potencial das respostas possíveis de serem alcançadas a partir das consultas realizadas.

No final da primeira etapa, a base de conhecimento possuirá regras Prolog que foram derivadas automaticamente pelo método aqui proposto e regras definidas manualmente, a critério do usuário especialista. Tal base de conhecimento será armazenada como um novo projeto e poderá ser carregada posteriormente quando for necessário. Isso significa que, uma vez realizada a etapa de configuração, o usuário pode utilizar a etapa de consulta sempre que desejar. Vale lembrar que a derivação das regras automáticas não é um processo obrigatório e não ocorrerá se não houver esquemas associados aos documentos. Porém sua ocorrência facilita o trabalho tanto do usuário especialista quanto do usuário final que realizará a consulta, pois tais regras poderão ser reutilizadas por ambos sem a necessidade de serem reescritas. Já a existência do usuário especialista é fundamental para aumentar o potencial de inferência das informações e diminuir o trabalho do usuário final. Sem ele, o trabalho deste usuário seria maior, tendo que configurar as regras necessárias para a obtenção dos resultados almejados.

Na segunda etapa o usuário deve carregar no sistema os documentos XML que serão passíveis de consulta. Um tradutor de documentos XML processa estes documentos e deriva fatos Prolog a partir da informação definida em linguagem XML. Uma explicação detalhada sobre esta tradução é apresentada na Seção 4.3. Após esta fase, todas as informações contidas no documento XML e nos esquemas estão definidas em linguagem Prolog (como fatos e regras, respectivamente), juntamente com as regras inseridas manualmente pelo usuário especialista. Essas informações constituem a base de conhecimento final do sistema e são passíveis de inferência.

Por fim, na terceira e última etapa o usuário está apto a realizar consultas, que são processadas pela máquina de inferência Prolog. As consultas são realizadas em linguagem Prolog e podem ser vistas como reutilização das regras previamente definidas quer seja automaticamente pelo tradutor de esquemas ou inseridas manualmente por um usuário especialista Os resultados são então apresentados ao usuário, que pode submeter mais consultas uma vez que o ambiente encontra-se todo preparado.

4.3 Tradução dos Documentos XML

A abordagem proposta neste trabalho utiliza um método de tradução dos dados armazenados nos documentos XML para os fatos correspondentes, descritos em Prolog, baseado na estrutura em forma de árvore do documento. O processo de tradução gera vários fatos a partir de um único documento XML, transformando os elementos em predicados e seus conteúdos em constantes. Para relacionar predicados distintos, um identificador (uma constante Prolog) é acrescentado como parâmetro, caracterizando que há correspondência entre esses dados no documento XML (por exemplo, relação de pai/filho).

A partir dos trabalhos relacionados, foram identificados dois tipos de tradução referentes ao tamanho dos termos gerados, sendo estes:

Tradução não granular: Grande parte dos trabalhos analisados durante a revisão da literatura (BOLEY, 2000; COELHO; FLORIDO, 2003; SEIPEL, 2002; WIELEMAKER, 2005) adotam ou citam um método de tradução que gera um único predicado que contém todo o documento XML de forma aninhada. Como exemplo, a parte inferior da Figura 4.2 ilustra um documento XML (parte superior) e o resultado de sua tradução para Prolog (parte inferior) segundo a abordagem proposta por Coelho e Florido (2003). No exemplo, o elemento complexo "teachers" e todos os seus filhos foram traduzidos em um único predicado Prolog, por isso chamamos de tradução não granular.

```
<teachers>
    <name>Jorge Coelho</name>
    <office>403</office>
    <email>jcoelho@isep.ipp.pt</email>
    <name>Mario Florido</name>
    <office>202</office>
</teachers>
```

```
teachers([(name("Jorge Coelho"),
    office("403"),
    email("jcoelho@isep.ipp.pt")),
    (name("Mario Florido"),
    office("202"))]).
```

Figura 4.2: Exemplo de Tradução XML – Prolog não granular

Tradução granular: No trabalho de Almendros-Jiménez et al. (2008) observamos que a utilização de predicados granulares poderia facilitar a realização das consultas, permitindo o uso apenas dos predicados que seriam realmente necessários. Este tipo de

tradução possibilita ainda a combinação entre quaisquer destes predicados a fim de obter novas informações. Após alguns testes chegamos à conclusão que a utilização de predicados não granulares dificultaria o processo de consulta, uma vez que o usuário deveria escrever todo o predicado para submeter uma consulta e obter os resultados. Foi a partir desta observação que definimos que utilizaríamos uma tradução granular em nossa abordagem.

Na nossa abordagem, todo o processo é guiado por cinco tipos de tradução que são listadas a seguir. Um exemplo do resultado da aplicação dessas traduções é ilustrado na Figura 4.3.

Documento XML		**Tradução Prolog**
<cliente>	1.	cliente(id1).
<histórico criacao=”13/07/1984”>	2.	historico(id1, id2).
<operacao>compra</operacao>	3.	criacao(id2, '13/07/1984').
realizada em	4.	operacao(id2, 'compra').
<data>23/02/1990</data>	5.	xml/elementoMisto(id2, 'realizada em').
<valor moeda=”R$”>586,00</valor>	6.	data(id2, '23/02/1990').
</historico>	7.	valor(id2, id3, '586,00').
</cliente>	8.	moeda(id3, 'R$').

Figura 4.3: Exemplo de tradução de um documento XML

Tradução da raiz do documento (Figura 4.3, linha 1). A raiz do documento XML deve ser tratada de forma especial, pois este é o único elemento do documento que não possui pai. Na grande maioria dos casos a raiz é um elemento composto (a não ser que se trate de um caso raro onde o documento contenha apenas um único elemento simples). Sendo assim, o resultado da tradução de uma raiz composta é um fato com o nome do elemento e argumento único igual a um identificador gerado pela aplicação com o objetivo de estabelecer vínculo com seus elementos filhos. Caso a raiz possua atributos, estes são traduzidos em fatos com o nome do atributo e argumentos iguais ao identificador da raiz (que é o pai do atributo) e valor do atributo.

Tradução de elementos simples sem atributo (Figura 4.3, linhas 4 e 6). Elementos simples sem atributos possuem o seu conteúdo textual como único filho na árvore XML, possibilitando que sua tradução possa ter como resultado um fato com nome igual ao nome do elemento e argumentos iguais ao identificador do elemento pai e o conteúdo do elemento corrente.

Tradução de elemento simples com atributo (Figura 4.3, linhas 7 e 8). Os atributos são considerados filhos do elemento que os contém. Um novo identificador é gerado para que os elementos filhos (atributos) possam referenciar seu pai. O resultado é

composto por um fato que representa o elemento, mais um fato para cada atributo existente: o primeiro possui o nome do elemento, e três argumentos: o identificador do seu elemento pai, um novo identificador gerado para representá-lo e o seu conteúdo textual. Os outros fatos possuem o nome do atributo e argumentos iguais ao identificador de seu pai (que representa o elemento que possui o atributo) e ao valor do atributo correspondente.

Tradução de elemento complexo (Figura 4.3, linhas 2 e 3). Elementos complexos possuem outros elementos como filhos. Portanto, um novo identificador deve ser gerado para relacionar os filhos ao pai. Consequentemente, o resultado da tradução é a geração de um fato com o nome do elemento composto e dois argumentos: o identificador do seu elemento pai e um novo identificador gerado para referenciá-lo. Seus filhos são traduzidos de acordo com a regra de tradução adequada às suas características (simples, composto, misto), utilizando o identificador referente ao elemento complexo (seu elemento pai) como identificador do elemento pai. Caso o elemento complexo possua atributos, estes serão traduzidos em fatos com o nome do atributo e argumentos iguais ao identificador do elemento complexo (que é o pai do atributo) e valor do atributo.

Tradução de elemento misto (Figura 4.3, linha 5). Elementos mistos são traduzidos de maneira semelhante aos elementos complexos, exceto pela criação de fatos "xml/elementoMisto" para os filhos texto do elemento (uma vez que eles não possuem nomes). Essa nomenclatura foi adotada levando em consideração o fato da especificação do XML não permitir que o nome dos elementos contenha o caractere "/", impedindo, consequentemente, que tal termo seja resultado de alguma tradução que não seja o caso de elemento misto. Portanto, seus filhos do tipo texto são representados por fatos "xml/elementoMisto" com argumentos iguais ao identificador do elemento pai (o elemento complexo) e o seu conteúdo textual. Já seus outros filhos são traduzidos de acordo com a regra de tradução adequada às suas características (simples, composto, misto).

Vale citar que em linguagem Prolog, as palavras que se iniciam por letras maiúsculas são interpretadas como variáveis, portanto, durante a tradução dos documentos XML em fatos Prolog, as letras maiúsculas são convertidas em minúsculas para evitar problemas nos resultados gerados. O Capítulo 7 detalha melhor esta limitação e propõe uma solução alternativa.

4.4 Definição das Regras

Para possibilitar a inferência de informações através do uso de Prolog, é necessário que haja regras relevantes capazes de guiar o processo. Regras Prolog são basicamente as relações entre os fatos existentes, por exemplo, a regra "avo(X, Z) :- pai(X, Y), pai(Y, Z)." cria um fato "avo(X, Z)" a partir da relação entre dois fatos "pai/2" (fatos pai com aridade 2). Conforme foi dito anteriormente as regras são obtidas de duas maneiras: automaticamente dos esquemas (se houver) e inseridas manualmente. As regras automáticas são derivadas diretamente dos esquemas para uma sintaxe compreensível pela Máquina de Inferência Prolog e definem a estrutura inicial que deve ser seguida pelos fatos, como será visto na Seção 4.4.1. Já as regras manuais são inseridas por um usuário especialista a fim de aumentar o poder de inferência da consulta, como será visto na Seção 4.4.2.

4.4.1 Regras Automáticas

As regras automáticas são regras provenientes dos esquemas, portanto, são regras relacionadas com a estrutura do documento XML. Analisando um documento como uma árvore, observamos que a definição de um elemento simples (apenas conteúdo textual e/ou atributo) não ramificaria a estrutura do documento. Pelo contrário, este elemento seria pai de uma folha (um nó terminal). Porém, a definição dos elementos complexos (elementos que possuem outros elementos como filhos) implicaria na criação de uma nova estrutura a partir daquele nó, o que proporcionaria a alteração da estrutura original. Concluímos que os elementos complexos possuem importância no que diz respeito à relação entre as informações definidas no documento XML, e que sendo assim, poderiam contribuir também para criar relacionamentos entre os fatos Prolog.

Analisando a definição de elementos complexos em um esquema observamos que são os delimitadores de grupo de XML Schema (W3C, 2001) os responsáveis por configurar a forma que a estrutura que descende destes elementos irá possuir. Portanto, na nossa abordagem são os delimitadores de grupo que estabelecem de que forma a tradução dos esquemas em regras Prolog deve ser conduzida. Considerando que existem três delimitadores de grupo distintos (*sequence, choice* e *all*) observamos seus

comportamentos e definimos a tradução referente a cada um deles. Devido à flexibilidade da estrutura da linguagem XML, podem ocorrer algumas variações. Nas subseções seguintes é apresentado como o método de tradução processa as definições encontradas nos esquemas e derivam as regras automáticas em linguagem Prolog.

4.4.1.1 Elementos Complexos com Filhos Elementos Simples

Os casos mais simples de regras automáticas são derivados de elementos complexos que possuem apenas elementos simples como filhos. Analisando a árvore do documento XML estes elementos constituiriam estruturas de apenas dois níveis de elementos (o elemento composto é o pai e no nível seguinte estão os filhos elementos simples). Neste caso são utilizadas as regras básicas de tradução para cada um dos delimitadores de grupo: *sequence*, *choice* e *all*.

Tradução do *sequence*: Dentre os delimitadores de grupo existentes o *sequence* é o mais utilizado (LAENDER *et al.*, 2009). Sua especificação sugere que o elemento complexo que o emprega deve seguir a estrutura definida no esquema, respeitando a ordem e cardinalidade. O método de tradução proposto produz uma regra que representa a estrutura estabelecida pelo *sequence* analisado. A Figura 4.4 (parte superior) mostra um trecho de um esquema XML que possui o delimitador de grupo *sequence*.

Esquema XML

```
<xs:element name="endereco" type="tEndereco"/>
<xs:complexType name="tEndereco">
  <xs:sequence>
    <xs:element name="logradouro" type="xs:string"/>
    <xs:element name="numero" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Tradução Prolog

```
endereco(ID, LOGRADOURO, NUMERO) :-
  logradouro(ID, LOGRADOURO),
  numero(ID, NUMERO).
```

Figura 4.4: Exemplo de ocorrência de *sequence* e sua tradução para Prolog.

A regra Prolog gerada neste caso possui sua cabeça referente ao elemento complexo (no exemplo da Figura 4.4, “endereco”) com os seguintes argumentos: uma variável Prolog representando o identificador da regra “ID”, responsável por vincular a regra a outros termos definidos em linguagem Prolog (fatos e regras), e uma variável para cada filho que possuir (“LOGRADOURO” e “NUMERO” no exemplo), com o objetivo de obter o valor destes através da associação do identificador da regra com os

fatos (ou regras) referentes a cada um destes filhos (vale lembrar que os fatos são derivados diretamente dos documentos XML). Já o corpo da regra possui um fato referente a cada filho, com argumentos iguais ao identificador da regra (para estabelecer relação com a cabeça da regra) e à variável correspondente. A Figura 4.4 (parte inferior) ilustra a regra automática obtida após a tradução do trecho do esquema mostrado na Figura 4.4 (parte superior).

Tradução do *choice*: A especificação do delimitador de grupo *choice* permite que o elemento complexo que o contém possua como filho apenas uma das opções definidas em sua estrutura. Sendo assim, o método de tradução cria uma regra para cada opção do *choice*, contemplando cada uma das possíveis estruturas válidas. A Figura 4.5 exibe um trecho de um esquema com ocorrência do delimitador de grupo *choice*.

Esquema XML

```
<xs:element name="unidade" type="tUnidade"/>
<xs:complexType name="tUnidade">
  <xs:choice>
    <xs:element name="area" type="xs:string"/>
    <xs:element name="volume" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

Tradução Prolog

```
unidade(ID, area, AREA) :-
  area(ID, AREA).
unidade(ID, volume, VOLUME) :-
  volume(ID, VOLUME).
```

Figura 4.5: Exemplo de ocorrência de *choice* e sua tradução para Prolog.

O resultado da tradução são regras Prolog referentes às possíveis opções de estrutura que o elemento complexo pode possuir, conforme ilustrado na Figura 4.5, parte inferior. A cabeça da regra é formada pelo elemento complexo "unidade", que possui como argumentos uma variável representando o identificador da regra "ID", uma constante Prolog que representa o identificador da opção (igual ao nome do elemento), responsável por indicar qual das opções foi adotada, e uma variável referente ao elemento correspondente. Na Figura 4.5, "area" e "volume" são respectivamente os identificadores de opção, assim como "AREA" e "VOLUME" são as variáveis. O corpo da regra possui o fato referente ao filho correspondente à regra com argumentos iguais ao identificador da regra (para estabelecer relação com a cabeça da regra) e à variável apropriada. O número de regras criadas é igual ao número de opções configuradas no *choice*.

Tradução do *all*: O delimitador de grupo *all* permite que o elemento complexo contenha qualquer combinação das opções definidas em sua estrutura (em qualquer ordem), possibilitando a ocorrência de vários elementos diferentes. Durante a tradução, o elemento que possuir o delimitador de grupo se torna a cabeça da regra, que será definida com os seguintes parâmetros: um identificador da regra "ID", um identificador da opção e uma variável que representa o filho correspondente à opção escolhida. Mais uma vez, a regra irá relacionar os possíveis filhos através do seu identificador. A Figura 4.6 (parte superior) mostra um trecho de um esquema que possui um elemento composto *all*, enquanto a Figura 4.6 (parte inferior) ilustra o resultado da tradução deste mesmo trecho utilizando o algoritmo proposto.

As especificações do delimitador de grupo *all* definem que ele só pode ser empregado como grupo mais externo de qualquer modelo de conteúdo, e ainda que seus filhos devam ser todos elementos, logo, não pode haver estruturas complexas contidas ou que contenham o delimitador de grupo *all*. Outra restrição impede que os elementos filhos de *all* tenham cardinalidade maior que 1, limitando para 0 ou 1 os valores permitidos para *minOccurs* e indicando que o valor de *maxOccurs* deve ser 1.

Esquema XML

```
<xs:element name="contato" type="tContato"/>
<xs:complexType name="tContato">
  <xs:all>
    <xs:element name="telefone" type="xs:string"/>
    <xs:element name="email" type="xs:string"/>
    <xs:element name="fax" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

Tradução Prolog

```
contato(ID, telefone, TELEFONE) :-
  telefone(ID, TELEFONE).
contato(ID, email, EMAIL) :-
  email(ID, EMAIL).
contato(ID, fax, FAX) :-
  fax(ID, FAX).
```

Figura 4.6: Exemplo de ocorrência de *all* e sua tradução para Prolog.

Na Figura 4.6 (parte inferior), o argumento "telefone" é o identificador de opção da primeira regra gerada, que se refere ao elemento de mesmo nome (filho do elemento composto delimitado pelo *all*), enquanto "TELEFONE" refere-se à variável que representa o valor deste. Nota-se aqui que os identificadores de opção se comportam como metadados de "contato", enquanto as variáveis são os dados

propriamente ditos, permitindo consequentemente consultas na forma "?- contato(ID_CONTATO, METADADO, DADO).".

```
<contato>
    <telefone>7777-7777</telefone>
    <fax>5555-5555</fax>
    <email>email@email.com</email>
</contato>
```

Figura 4.7: Exemplo de documento XML que segue o esquema da Figura 4.6

A partir do documento XML representado pela Figura 4.7, e assumindo que na tradução o elemento "contato" possua ID=7, a consulta "?- contato(7, METADADO, DADO)." gera os seguintes resultados:

METADADO = telefone; DADO = 7777-7777

METADADO = fax; DADO = 5555-5555

METADADO = email; DADO = email@email.com

Analisando a Figura 4.5 e a Figura 4.6 é possível observar que as traduções dos delimitadores de grupo *choice* e *all* geram regras semelhantes. Porém, o fato de partirmos do princípio que os documentos XML são válidos pelo esquema informado faz com que a estrutura dos elementos complexos referente a esses delimitadores esteja de acordo com suas restrições. Portanto, a similaridade ocorre apenas na sintaxe das regras, estando o comportamento das instâncias de acordo com o delimitador de grupo utilizado.

4.4.1.2 Elementos Complexos com Filhos Elementos Complexos

A flexibilidade da linguagem XML permite que existam infinitas possibilidades de estruturas aninhadas, o que faz com que um elemento complexo possa ser pai de muitos outros elementos complexos, fazendo com que a representação de sua estrutura em árvore contenha muitos níveis. O método de tradução de elementos complexos com filhos complexos utiliza os mesmos princípios das regras básicas, mencionadas anteriormente, para gerar as regras automáticas. Porém, o fato de os filhos não serem mais elementos simples torna necessário alguns cuidados para que as regras geradas representem fielmente as estruturas válidas possíveis existentes nos documentos XML.

Tradução de *sequence* com filho *choice*: A tradução do delimitador de grupo *sequence* que possui um filho *choice* utiliza regras definidas através da união da tradução de ambos os grupos. O trecho de esquema representado pela Figura 4.8 indica que o elemento composto "cliente" possuirá sempre dois filhos, sendo que o primeiro poderá ser "cpf" ou "cnpj" e o segundo será sempre "telefone". Na prática, seria como se houvesse dois *sequences* diferentes, um contendo os filhos "cpf" e "telefone" e outro "cnpj" e "telefone". Portanto, a tradução define duas regras distintas, uma para cada opção possível. Vale lembrar que a cabeça da regra irá conter também a opção que foi escolhida (um identificador de opção conforme ocorre na tradução do *choice* simples), para facilitar futuras consultas. A Figura 4.8 (parte inferior) ilustra o resultado da tradução do trecho do esquema contido na Figura 4.8 (parte superior).

Esquema XML

```
<xs:element name="cliente" type="tCliente"/>
<xs:complexType name=" tCliente ">
  <xs:sequence>
    <xs:choice>
      <xs:element name="cpf" type="xs:string"/>
      <xs:element name="cnpj" type="xs:string"/>
    </xs:choice>
    <xs:element name="telefone" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Tradução Prolog

```
cliente(ID, cpf, CPF, TELEFONE) :-
  cpf(ID, CPF),
  telefone(ID, TELEFONE).
cliente(ID, cnpj, CNPJ, TELEFONE) :-
  cnpj(ID, CNPJ),
  telefone(ID, TELEFONE).
```

Figura 4.8: *Sequence* com filho *choice* e sua tradução para Prolog.

Tradução de *choice* com filho *sequence*: Um elemento composto delimitado por *choice* pode possuir filhos *sequence*. Neste caso o método de tradução associa uma constante Prolog (s1, ..., sn) para cada filho *sequence*, com o objetivo de aplicar estas constantes como identificador de opção do *choice.* Este processo é necessário uma vez que não haverá elemento diretamente correspondente a este *sequence* para que seu nome seja utilizado como identificador de opção. O restante do processo é análogo à tradução do *choice* com filhos elementos simples. A Figura 4.9 (parte inferior) ilustra o resultado da tradução do trecho do esquema contido na Figura 4.9 (parte superior).

Esquema XML

```
<xs:element name="cliente" type="tCliente"/>
<xs:complexType name=" tCliente ">
  <xs:choice>
    <xs:sequence>
      <xs:element name="razaoSocial"
            type="xs:string"/>
      <xs:element name="cnpj" type="xs:string"/>
      <xs:element name="cpfProprietario"
            type="xs:string"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="cpf" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Tradução Prolog

```
cliente(ID, s1, RAZAOSOCIAL, CNPJ, CPFPROPRIETARIO) :-
  razaoSocial(ID, RAZAOSOCIAL), cnpj(ID,CNPJ, ),
  cpfProprietario(ID, CPFPROPIETARIO).
cliente(ID, s2, NOME, CPF) :-
  nome(ID, NOME), cpf(ID, CPF).
```

Figura 4.9: *Choice* com filho *sequence* e sua tradução para Prolog.

Existem ainda outras combinações de delimitadores de grupos possíveis, tais como *sequence* com filho *sequence* e *choice* com filho *choice*. Essas combinações são resolvidas com o auxílio da "propriedade do elemento neutro" que é explicada a seguir.

Propriedade do Elemento Neutro: Caso um elemento complexo possua o mesmo delimitador de grupo que seu pai (*sequence* com filho *sequence* ou *choice* com filho *choice*) podemos dizer que este é um elemento neutro para a tradução. Definindo C como o delimitador de grupo *choice*, S como o delimitador de grupo *sequence* e ainda X, Y e Z como os possíveis filhos que ambos os delimitadores podem possuir, as propriedades definidas são discutidas a seguir.

Sequence. A especificação do *sequence* define que seus filhos devem aparecer na ordem em que são definidos. Portanto, sempre que houver *sequences* aninhados o método de tradução irá considerar que se trata de um único *sequence*, o mais externo (pai), herdando o conteúdo do *sequence* filho*, re*spеitando a ordem definida no esquema. Formalmente, a seguinte transformação é realizada: **S(X, S(Y, Z)) = S(X, Y, Z).**

A Figura 4.10 ilustra em sua parte superior um trecho de um esquema com ocorrência de *sequences* aninhados enquanto a sua parte central mostra outro trecho

similar, porém utilizando apenas filhos *elements*. É possível notar que o *sequence* interno é ignorado e seu conteúdo pode ser visto como filho de seu elemento complexo pai.

O trecho da Figura 4.10 (parte superior) valida documentos XML cujo elemento complexo "cliente" possua os filhos "nome" e "cpf" seguidos de "telefone". Se o *sequence* interno for omitido o exemplo seria igual ao ilustrado na parte central da figura. Sendo assim, sempre que houver *sequences* aninhados o método de tradução irá considerar que trata-se de um único *sequence* contendo todos os elementos filhos, respeitando a ordem em que são definidos. A parte inferior da Figura 4.10 ilustra o resultado da tradução obtido.

Esquema XML: *sequences* aninhados

```
<xs:element name="cliente" type="tCliente"/>
<xs:complexType name=" tCliente ">
  <xs:sequence>
    <xs:sequence>
      <xs:element name="nome"
           type="xs:string"/>
      <xs:element name="cpf"
           type="xs:string"/>
    </xs:sequence>
    <xs:element name="telefone"
           type="xs:string"/>
  </xs: sequence >
</xs:complexType>
```

Esquema XML: apenas um *sequence*

```
<xs:element name="cliente" type="tCliente"/>
<xs:complexType name=" tCliente ">
  <xs:sequence>
    <xs:element name="nome"
           type="xs:string"/>
    <xs:element name="cpf" type="xs:string"/>
    <xs:element name="telefone"
           type="xs:string"/>
  </xs: sequence >
</xs:complexType>
```

Tradução Prolog:

```
cliente(ID, NOME, CPF, TELEFONE) :-
    nome(ID, NOME), cpf(ID, CPF), telefone(ID, TELEFONE).
```

Figura 4.10: Exemplo da tradução de *sequences* aninhados.

Choice. A especificação do *choice* indica que o elemento complexo possuirá uma das opções definidas como filho, logo, um caso onde um dos possíveis filhos de um *choice* seja outro *choice* pode ser interpretado pelo tradutor como um único *choice* que herda todas as opções possíveis. Formalmente, temos **C(X, C(Y, Z)) = C(X, Y, Z).**

Observando o lado esquerdo da propriedade, é possível perceber que caso ocorra a segunda opção "**C(Y, Z)**", novamente duas estruturas são possíveis. O lado direito da propriedade define apenas um *choice* validando todas as opções possíveis (seu filho "X" e as duas opções herdadas "Y" e "Z").

Esquema XML: *choices* aninhados

```
<xs:element name="unidade" type="tUnidade"/>
<xs:complexType name="tUnidade">
  <xs:choice>
    <xs:element name="area"
           type="xs:string"/>
    <xs:element name="comprimento"
           type="xs:string"/>
    <xs:choice>
      <xs:element name="massa"
           type="xs:string"/>
      <xs:element name="volume"
           type="xs:string"/>
    </xs:choice>
  </xs:choice>
</xs:complexType>
```

Esquema XML: apenas um *choice*

```
<xs:element name="unidade" type="tUnidade"/>
<xs:complexType name="tUnidade">
  <xs:choice>
    <xs:element name="area"
           type="xs:string"/>
    <xs:element name="comprimento"
           type="xs:string"/>
    <xs:element name="massa"
           type="xs:string"/>
    <xs:element name="volume"
           type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

Tradução Prolog:

```
unidade(ID, area, AREA) :- area(ID, AREA).
unidade(ID, comprimento, COMPRIMENTO) :-
    comprimento(ID, COMPRIMENTO).
unidade(ID, massa, MASSA) :- massa(ID, MASSA).
unidade(ID, volume, VOLUME) :- volume(ID, VOLUME).
```

Figura 4.11: Exemplo da tradução de *choices* aninhados.

A Figura 4.11 (parte superior) ilustra a ocorrência de delimitador de grupo *choice* com filho *choice*. Seguindo este exemplo, um documento XML válido deve possuir como filho de "unidade" o elemento "area" ou "comprimento", ou ainda "massa" ou "volume". Seria o equivalente ao trecho do esquema representado pela Figura 4.11 (parte central), onde os filhos do *choice* interno podem ser tratados

diretamente como filhos do externo. A tradução trata o trecho do esquema diretamente como um *choice* que possui somente filhos *elements* e obtém o resultado contido na parte inferior da Figura 4.11.

4.4.1.3 Cardinalidade dos Elementos

A cardinalidade dos elementos é representada no esquema através dos atributos *minOccurs* e *maxOccurs*, sendo estes responsáveis por definir a ocorrência mínima e máxima respectivamente do elemento no documento XML. Sendo assim, a definição em *XML Schema*:

<xs:element name="telefone" type="xs:string" minOccurs = "1" maxOccurs = "3"/>

indica que o documento válido por tal esquema deve possuir de um a três elementos "telefone" para que seja válido.

Vale dizer também que o atributo *minOccurs* com valor zero indica que pode não haver ocorrência do elemento e que *maxOccurs* igual a "*unbounded*" significa que não há limite máximo para tal ocorrência.

Neste contexto, foi possível identificar que as regras automáticas não estavam retornando os elementos complexos que possuíssem algum filho sem ocorrência no documento XML. Como as regras automáticas geradas através de tradução do *sequence* possuem a forma:

elementoComplexo(ID, Filho1, ..., FilhoN) :- filho1(ID, Filho1), ..., filhoN(ID, FilhoN).

Caso um dos filhos, digamos o terceiro filho, não ocorra no documento XML, tal regra não retornará seu pai, pelo fato da condição *filho3(ID, Filho3)* não ser satisfeita.

Com isso, resolvemos que a definição dos elementos nos esquemas que possuem o atributo *minOccurs* igual a zero não participa das regras automáticas, para possibilitar a obtenção de todos os elementos complexos, ainda que sem a presença de seus filhos opcionais.

Desta forma, podemos dizer que a cardinalidade interfere diretamente na geração das regras no caso de elementos opcionais (*minOccurs* igual a zero). Cabe lembrar que as regras automáticas atuam como um facilitador das consultas, não sendo estas essenciais para o processo. Ressaltamos ainda que a obtenção dessas informações

pode ser configurada tanto no cadastro das regras manuais quanto na realização das consultas, uma vez que elas estão presentes nos fatos contidos na base de conhecimento.

4.4.2 Inserção das Regras Manuais

As regras manuais são regras mais elaboradas, que não podem ser derivadas automaticamente a partir dos esquemas. A abordagem proposta permite que elas sejam inseridas por um usuário especialista durante a etapa de configuração das regras, com o objetivo de tornar possível a realização de inferências mais complexas ou obter respostas que não podem ser obtidas somente com as informações contidas no documento e esquema XML. A ideia é que sejam cadastradas regras voltadas ao tipo de respostas que se quer obter. Porém, o usuário especialista é livre para criar qualquer tipo de regra que julgar útil.

Esquema XML

```
<xs:element name="pessoa" type="tPessoa"/>
<xs:complexType name="tPessoa">
  <xs:sequence>
    <xs:choice>
      <xs:sequence>
        <xs:element name="razao_social" type="xs:string"/>
        <xs:element name="cnpj" type="xs:string"/>
        <xs:element name="cpfProprietario" type="xs:string"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="nome" type="xs:string"/>
        <xs:element name="cpf" type="xs:string"/>
        <xs:element name="filho" type="xs:string"/>
      </xs:sequence>
    </xs:choice></xs:sequence></xs:complexType>
```

Regra Manual Prolog

```
socio(Nome1, Nome2) :-
   nome(A, Nome1),
   cpf(A, B),
   cpfProprietario (C, B),
   cnpj(C, D),
   nome(E, Nome2),
   cpf(E, F),
   cpfProprietario (G, F),
   cnpj(G, D),
   Nome1 \= Nome2.
```

Figura 4.12: Exemplo de XML *Schema* e regra manual Prolog.

Como um exemplo, o esquema definido na Figura 4.12 não possui qualquer informação a respeito de sociedade. Porém, pode ser necessário obter tal relação entre as pessoas cadastradas. Analisando a definição dos predicados gerada a partir da tradução do esquema, um usuário especialista pode criar regras que permitam inferir

informações além das que foram explicitamente traduzidas do documento e esquema XML. A semântica por trás de um conjunto de elementos XML (por exemplo, o fato de que dois proprietários de uma mesma empresa são sócios) não é capturada automaticamente por um esquema de tradução, mas pode ser percebida por um usuário especialista. Ele pode avaliar se ela enriquece a base e decidir incluí-la na forma de uma regra. Vale lembrar que isso ocorre somente na etapa de configuração, e este usuário pode não ser o que realizará a consulta (ele apenas irá configurar o ambiente).

A Figura 4.12, parte inferior, ilustra as regras manuais que devem ser inseridas por um usuário especialista para tornar possível obter a relação de sociedade entre as pessoas durante a etapa de consulta. Durante esta etapa, o usuário obtém informações sobre a relação de sociedade entre pessoas através do termo de consulta: "?- socio(X, Y).". Através de XQuery, a mesma informação seria obtida através de uma consulta muito mais complexa (Figura 4.13).

```
for $pessoa1 in doc("teste2XML.xml")/pessoas/pessoa,
  $pessoa2 in doc("teste2XML.xml")/pessoas/pessoa,
  $empresa1 in doc("teste2XML.xml")/pessoas/pessoa,
  $empresa2 in doc("teste2XML.xml")/pessoas/pessoa
where $pessoa1/cpf = $empresa1/cpf_do_proprietario
  and $pessoa1/cpf != $pessoa2/cpf
  and $pessoa2/cpf = $empresa2/cpf_do_proprietario
  and $empresa1/cnpj = $empresa2/cnpj
Return <socio>
         {$pessoa1/nome/text()} e socio de {$pessoa2/nome/text()}
       </socio>
```

Figura 4.13: Consulta XQuery sobre sociedade entre pessoas

```
herdeiro(Herdeiro, Antecessor) :-
    filho(ID, Herdeiro), nome(ID, Antecessor).
herdeiro(Herdeiro, Antecessor) :- filho(ID, Herdeiro),
    nome(ID, Antecessor2), herdeiro(Antecessor2, Antecessor)
    herdeiro(Antecessor2, Antecessor1).
socioHerdeiro(Nome1, Nome2) :- socioHerdeiro2(Nome1, Nome2).
socioHerdeiro(Nome1, Nome2) :- socioHerdeiro2(Nome2, Nome1).
socioHerdeiro2(Nome1, Nome2) :- herdeiro(Nome1, Antecessor1),
    socio(Antecessor1,Nome2).
socioHerdeiro2(Nome1, Nome2) :- herdeiro(Nome1, Antecessor1),
     herdeiro(Nome2, Antecessor2), socio(Antecessor1, Antecessor2).
```

Figura 4.14: Exemplo de regra manual socioHerdeiro

O uso de Prolog proporciona a reutilização de regras com o objetivo de elaborar consultas mais complexas. A Figura 4.14 mostra seis regras Prolog que tornam possível a obtenção de respostas a respeito de sociedade proveniente de herança (sócios herdeiros) no exemplo tratado. Note que as regras "socioHerdeiro" reutilizam "socio" e

“herdeiro”, previamente definidas. Essa facilidade não é permitida em XQuery, tendo o usuário que definir todas as relações desejadas a cada consulta.

4.5 Considerações Finais

Este capítulo detalhou a abordagem proposta neste trabalho, explicando como cada etapa deve ser executada para possibilitar a realização de inferências durante a consulta em documentos XML, para assim facilitar a obtenção de respostas mais elaboradas. O entendimento deste capítulo é de extrema importância para a compreensão do projeto desenvolvido, pois nele temos o detalhamento da ideia principal, e a solidificação do contexto no qual se pretende atuar de forma mais elaborada.

Os capítulos seguintes discorrem sobre a implementação proposta neste trabalho e os experimentos que foram realizados para que fosse possível a comparação dos resultados obtidos com as abordagens de consultas em documentos XML tradicionais. Assim, podemos dizer que esse capítulo sustenta o conteúdo principal da pesquisa desenvolvida, possibilitando a assimilação da ideia central deste trabalho, e a continuação do desenvolvimento da mesma.

Capítulo 5. Implementação

5.1 Introdução

Para avaliarmos este trabalho e comprovar que o que estamos propondo seria viável, foi necessário desenvolver um sistema que implementasse o método proposto no capítulo anterior. Durante o desenvolvimento desta tarefa, novos requisitos (situações que não estávamos considerando) foram surgindo e passaram a fazer parte da abordagem, tendo que ser incluídos como definições do método. Algumas vezes, a implementação demonstrou que certas definições precisariam ser repensadas para que fosse possível alcançar o objetivo esperado. A implementação da ideia possui um destaque especial neste trabalho, uma vez que em alguns momentos ela validava o que estávamos propondo e em outros nos mostrava que o que estávamos pensando não valeira para todos os casos.

A próxima seção analisa os requisitos do sistema e algumas definições que foram concluídas no decorrer do desenvolvimento da abordagem, enquanto a Seção 5.3 demonstra o uso do sistema implementado, guiado através de exemplos. Por fim a Seção 5.4 encerra o capítulo com as considerações finais sobre a implementação da nossa abordagem.

5.2 Definições e Requisitos

Desde o início da pesquisa sabíamos da necessidade da implementação do nosso trabalho para que pudéssemos valida-lo. Esta tarefa, porém, só pôde ter início quando conseguimos fechar um escopo da abordagem. Dentre as definições básicas estavam a divisão do método em três etapas (configuração das regras, definição dos fatos e consulta) e a utilização da linguagem Java para desenvolver o sistema. Neste ponto já estava claro que a tarefa mais difícil seria a definição da tradução tanto dos esquemas quanto dos documentos XML para linguagem Prolog sem perder os dados definidos nem as informações que poderiam ser obtidas a partir da estrutura dos documentos XML. Tendo este desafio como meta, duas abordagens de tradução foram analisadas: granular e não granular. Conforme explicado no capítulo anterior, o fato de

reduzir o tamanho da consulta a ser submetida e de permitir a combinação das informações necessárias para obtenção da resposta desejada resultou na escolha de implementar a tradução granular.

Quanto à realização das consultas em Prolog, como havíamos definido a linguagem Java para implementar o sistema, decidimos utilizar a API tuProlog (DENTI *et al.*, 2001) para integrar a máquina de inferência Prolog ao sistema. Esta API fornece um subconjunto das instruções da máquina Prolog contendo as suas propriedades essenciais e proporciona um ambiente capaz de carregar bases de conhecimento, processar consultas Prolog e fornecer as respostas desejadas.

O penúltimo requisito estabelecido antes da implementação foi a possibilidade de carregar mais de um documento XML alvo da consulta, ou seja, vários documentos XML podem ser traduzidos para linguagem Prolog para que a consulta seja realizada sobre as informações definidas sobre quaisquer destes documentos. Esse requisito estava acompanhado de outro: a possibilidade de carregar mais de um documento torna possível que estruturas distintas sejam carregadas, portanto, o sistema deveria estar apto a carregar também múltiplos esquemas a fim de proporcionar regras automáticas para facilitar a configuração de regras manuais e a realização das consultas.

Por fim, a última definição foi a implementação da ideia de Projeto que acrescentou uma nova funcionalidade ao sistema, possibilitando que as configurações realizadas na etapa de configuração das regras (primeira etapa) fossem salvas com um nome atribuído pelo usuário e pudessem ser carregadas no futuro, em outra vez que o sistema fosse utilizado.

Dessa forma, os requisitos do protótipo a ser desenvolvido estavam definidos, o que possibilitou sua implementação.

5.3 Exemplo de Uso

Após as definições que ocorreram conforme explicitado na seção anterior, conseguimos implementar uma versão satisfatória que pudesse ser testada e comparada com as outras abordagens de consulta em documentos XML. Esta seção apresenta a versão final do sistema desenvolvido. A avaliação experimental será abordada no próximo capítulo.

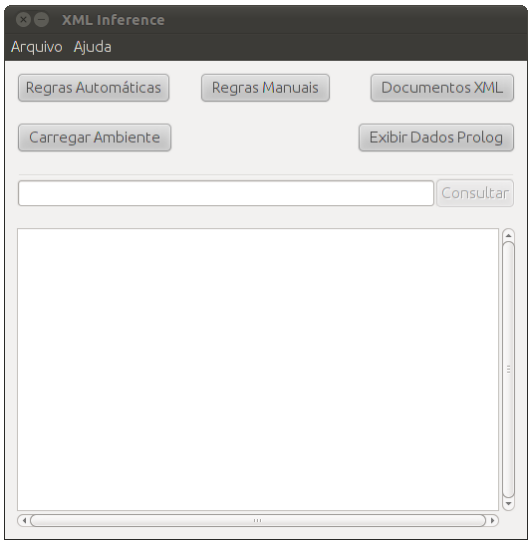

Figura 5.1: Tela principal do sistema implementado

A tela principal do sistema implementado está ilustrada na Figura 5.1. Cinco ações são possíveis: carregar as regras automáticas, inserir as regras manuais, carregar os documentos XML (fatos), carregar uma configuração salva anteriormente (a ação "Consultar" encontra-se desabilitada) e exibir os dados em linguagem Prolog. Supondo que o sistema nunca foi executado antes (não há nenhuma pré-configuração salva) e que se deseja executar todas as etapas do sistema, a opção "Regras Automáticas" deve ser escolhida para que sejam carregados os esquemas referentes aos documentos que serão posteriormente carregados. Como resposta, uma outra janela do sistema é aberta fornecendo uma interface para que documentos XML Schema sejam carregados. Vários esquemas podem ser escolhidos simultaneamente.

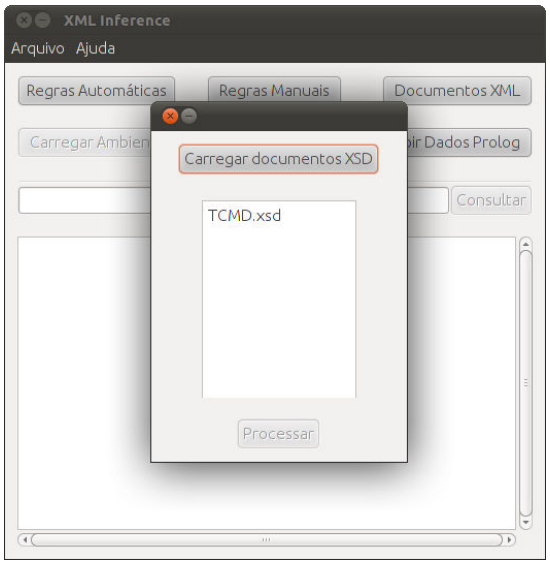

Figura 5.2: Tela de carga de XML *Schemas*

Na Figura 5.2 apenas o XML Schema denominado “TCMD.xsd” foi escolhido. Após selecionar as opções necessárias o botão “Processar” irá carregar os esquemas e invocará o tradutor de esquemas que derivará regras Prolog a partir das informações neles contidas (as denominadas regras automáticas).

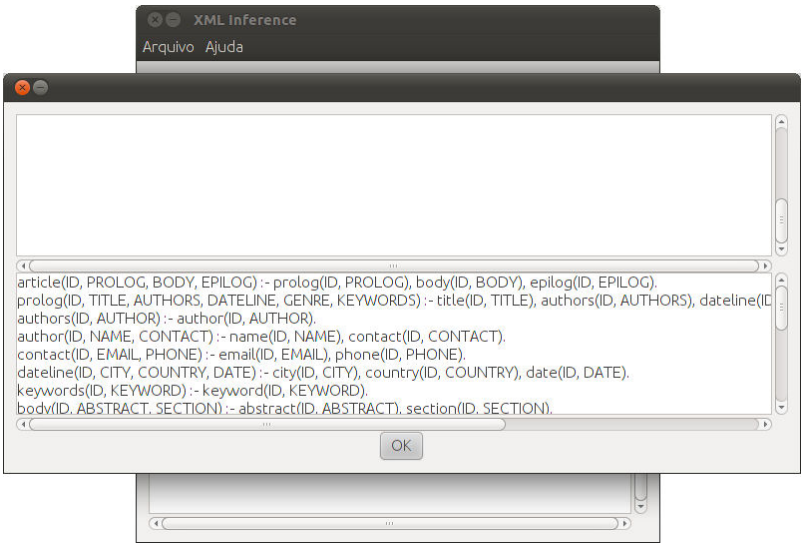

Figura 5.3: Tela de inserção das regras manuais

Após a tradução dos esquemas em regras, a tela principal é exibida novamente e a escolha da opção “Regras Manuais” dá sequência à execução padrão do método. Estamos chamando de execução padrão a que executa todos os processos possíveis (carga dos esquemas, inserção das regras manuais, carga dos documentos XML e consulta), uma vez que os procedimentos de carregar os esquemas e inserir as regras manuais não são obrigatórios e podem ser omitidos durante o processo de consulta.

Ao escolher a opção “Regras Manuais” na tela principal, o sistema exibe uma nova tela conforme ilustra a Figura 5.3, expondo as regras que já se encontram armazenadas na base de conhecimento (parte inferior da figura) para facilitar a inserção de novas regras manualmente na *TextArea* da parte superior da tela exposta na figura. Esta configuração deve ser feita por um usuário especialista em Prolog, uma vez que se trata de inserir regras em Prolog puro no sistema, e isso pode ser uma tarefa difícil para um usuário sem conhecimento da linguagem. Ao término do cadastro de regras o botão “OK” deve ser acionado. O sistema então solicita um nome para projeto que será criado. Este projeto irá conter as configurações que foram realizadas até o momento, ou seja, o sistema irá armazenar as regras automáticas (provenientes dos esquemas selecionados) e as regras manuais dando a este conjunto um nome. Esta configuração contempla os

dados provenientes da primeira etapa do sistema e poderá ser recarregada posteriormente quando o sistema for iniciado novamente.

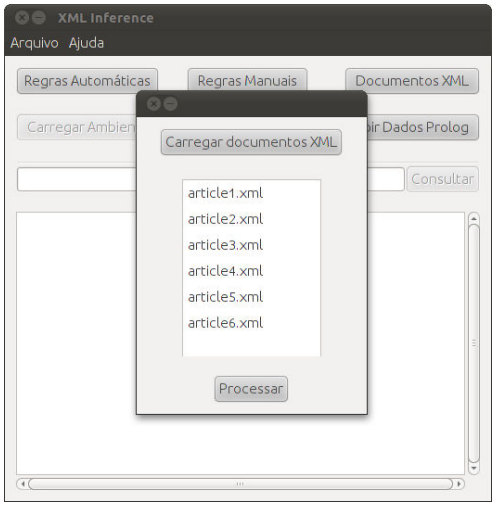

Figura 5.4: Tela de carga dos documentos XML

Com o fim da configuração das regras inicia-se a segunda etapa do sistema, denominada "geração dos fatos". O sistema retorna à sua tela principal e a opção "Documentos XML" deve ser escolhida para dar sequencia ao processo e habilitar os campos de consulta ao final desta etapa. Uma nova tela é exibida fornecendo uma interface para que os documentos XML sejam carregados (o processo é análogo à carga dos esquemas). Após o usuário escolher quais documentos XML deseja converter para Prolog, o botão "Processar" invoca o tradutor de XML que é responsável por gerar os fatos Prolog a partir dos documentos XML carregados. Estes fatos também são acrescentados ao projeto criado na etapa anterior, e poderão ser recarregados futuramente se esse for invocado. A Figura 5.4 ilustra a tela de carga com seis documentos XML selecionados para serem processados.

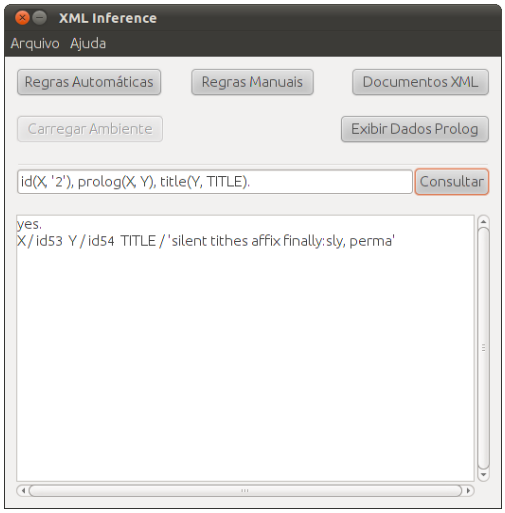

Figura 5.5: Tela contendo uma consulta e o seu resultado

Após a carga dos documentos XML a terceira etapa inicia-se e o sistema dispõe de todas as informações necessárias para realizar as consultas e processar inferências se necessário. Mais uma vez a tela principal é exibida e uma nova opção é habilitada: “Consultar” (este botão fica desabilitado até que os documentos XML sejam carregados). O usuário formula sua consulta no *TextField* e o botão “Consultar” invoca a API tuProlog, responsável por carregar as regras e os fatos Prolog, processar a consulta a partir do termo submetido pelo usuário através do *TextField* e exibir o resultado obtido na *TextArea*. A Figura 5.5 exibe uma consulta que pretende obter o título do artigo cujo “ID” é igual a 2 (“id(X, '2'), prolog(X, Y), title(Y, TITLE).”) e o seu resultado impresso na *TextArea*.

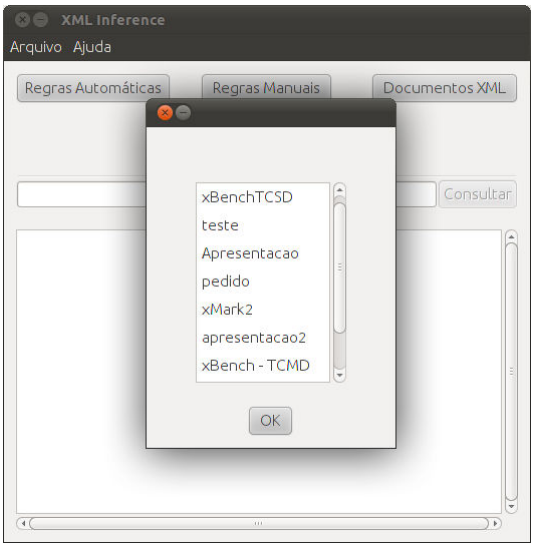

Figura 5.6: Tela de carga de um Projeto configurado anteriormente

Conforme mencionado anteriormente, o resultado da primeira etapa do sistema (configuração das regras) é armazenado e possui um nome escolhido pelo usuário que cadastrou as regras manuais. Toda vez que o sistema é iniciado é possível carregar as configurações realizadas durante a tradução das regras automáticas e inserção das regras manuais. A opção “Carregar Ambiente” da tela principal do sistema exibe uma tela com o nome de todos os projetos pré-cadastrados. Para carregar um basta selecioná-lo e clicar no botão “Ok”. O sistema retornará à tela principal, porém, a base de conhecimento irá conter todas as regras e fatos que possuía no momento em que o projeto foi salvo, e ainda será possível inserir mais regras manuais, caso seja necessário, e alterar o projeto (dando a ele o mesmo nome) ou criar um novo. A Figura 5.6 mostra a tela de seleção de projetos com várias opções previamente cadastradas. Os projetos irão persistir mesmo que o sistema seja encerrado.

A qualquer momento, através do botão “Exibir Dados Prolog” o usuário pode visualizar todas as regras e fatos que foram configurados. O conhecimento destes dados auxilia tanto na inserção das regras manuais quanto na realização das consultas. A Figura 5.7 ilustra a tela do sistema no momento em que as informações Prolog estão sendo exibidas para o usuário.

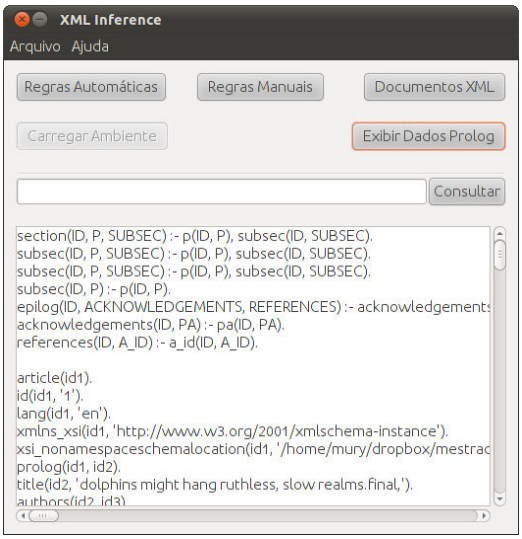

Figura 5.7: Tela exibindo as informações em linguagem Prolog

5.4 Considerações Finais

Este capítulo descreveu a elaboração do protótipo que foi implementado para comprovar a eficácia da ideia proposta, identificando os requisitos necessários para o seu desenvolvimento.

O objetivo deste capítulo é explicar como funciona o protótipo de consulta. O entedimento do funcionamento do protótipo é importante para a leitura do próximo capítulo, onde teremos então, a apresentação dos resultados das consultas a partir da utilização deste sistema, e a comparação desses resultados com os que foram obtidos através de outras abordagens de consulta.

Capítulo 6. Avaliação Experimental

6.1 Introdução

A avaliação experimental da abordagem proposta foi possível graças ao protótipo apresentado no Capítulo 5. Foram feitos alguns testes com o protótipo para comparar o desempenho das consultas realizadas utilizando a abordagem proposta com o de consultas utilizando a linguagem XQuery. Além do desempenho, avaliamos também o tamanho das consultas em cada caso, permitindo uma avaliação mais completa a respeito de seus resultados. O principal objetivo desta avaliação foi constatar se a realização de inferências seria capaz de reduzir o tempo de resposta e o tamanho das consultas necessárias para a obtenção dos dados implícitos nos documentos XML.

A Seção 6.2 apresenta como os experimentos foram planejados, quais seus objetivos e em qual ambiente ocorreram. Na Seção 6.3 fazemos uma análise dos resultados obtidos. As ameaças à validade dos experimentos são introduzidas na Seção 6.4. Por fim, na Seção 6.5 encerramos o capítulo com as considerações finais a respeito dos experimentos.

6.2 Elaboração dos Experimentos

Para que os resultados dos experimentos fossem precisos e se tornassem relevantes, foi necessário a elaboração de um plano de execução onde todas as etapas do processo estivessem bem definidas e organizadas. Além disso, foi de suma importância a definição dos objetivos pretendidos, e ainda a escolha do ambiente e das configurações sob as quais os experimentos ocorreriam. Foi imprescindível a definição da base de conhecimento que seria utilizada como alvo para as consultas, e das consultas propriamente ditas.

6.2.1 A Geração da Base de Conhecimento

Antes de realizar a avaliação experimental foi necessário definir quais consultas seriam utilizadas e ainda, sobre qual base de conhecimento elas seriam

processadas. Após análise de métodos e padrões, o *benchmark* XML denominado XBench (YAO *et al.*, 2004) foi o escolhido para suprir essas necessidades. Através do seu gerador de banco de dados foi possível criar a base de conhecimento com as informações que seriam passíveis de consulta. Durante este processo foram identificadas as necessidades de algumas configurações específicas para o que se estava propondo.

Em relação à estrutura que define as informações, o gerador de banco de dados do XBench permite quatro combinações, sendo elas: *text-centric/single-document* (TC/SD), *text-centric/multiple documents* (TC/MD), *data-centric/single-document* (DC/SD) e *data-centric/multiple-documents)* DC/MD. A compreensão de cada uma das características envolvidas nessas combinações foi fundamental para a escolha final do que foi utilizada no processo de geração. As possíveis características para tais combinações são:

Text-centric: define que a base de dados será gerada sobre uma perspectiva centrada em texto. Isso significa que os elementos XML possuem muito texto como conteúdo;

D*ata-centric:* ao contrário da *text-centric*, indica que a base de dados gerada será centrada nos dados. Neste caso, o conteúdo dos elementos XML é menor;

Single-document: define que toda a base de dados será representada em um único documento XML;

Multiple-documents: determina que a base de dados gerada será descentralizada, sendo representada por diversos documentos XML.

O tamanho da base de dados é um outro parâmetro que deve ser configurado durante sua geração. O gerador de banco de dados do XBench possibilita a escolha entre quatro opções possíveis: *small, normal, large* e *huge*. A definição desse fator implica diretamente no tamanho do documento XML que representa a base de dados (caso seja uma abordagem *single*-document), ou na quantidade de documentos que a descrevem (tratando-se de uma perspectiva *multiple-documents*). Como exemplo, podemos citar a criação de uma base de dados como TC/SD que será representada por um único documento com 10 Mb aproximadamente caso seu tamanho seja configurado como *small*, ou aproximadamente 100 Mb quando definida com o tamanho *normal*. Já uma

base de conhecimento TC/MD será descrita por 26 documentos XML no caso de ter seu tamanho *small,* e por 266 quando sendo *normal*.

Além do gerador da base de dados, o XBench também fornece consultas específicas para cada uma das configurações escolhidas durante o processo de geração. Estas consultas estão escritas em linguagem XQuery e têm como objetivo a exploração de informações distintas, ou seja, são consultas com objetivos diferentes, que exploram a obtenção de formas de resultados diversificados, utilizando assim, várias facilidades fornecidas pela linguagem para obtenção do resultado desejado.

Na nossa avaliação, escolhemos a abordagem TC/MD com tamanho *small*, pois todas as informações necessárias para a formação dessa base estavam representadas em diversos documentos XML. Esta configuração forneceu uma base de dados composta por 26 documentos XML que descrevem informações sobre artigos científicos. O Anexo A ilustra o esquema que define o vocabulário de todos os documentos gerados.

Sobre esta base de dados, foi aplicada a abordagem proposta nesta dissertação para gerar uma base de conhecimento com fatos representando o conteúdo dos 26 documentos XML e regras representando o esquema da base.

6.2.2 Objetivos

Para que os experimentos fossem relevantes e contribuíssem para a pesquisa de forma clara, foram definidos dois objetivos:

Objetivo 1: Comparação do tempo necessário para que a consulta seja processada e o resultado obtido, para então concluir qual dos métodos de consulta seria mais rápido, a abordagem proposta ou a linguagem XQuery.

Objetivo 2: Avaliação de como o método proposto interfere no tamanho das consultas almejadas, quando comparado à linguagem XQuery. Ou seja, analisar se o uso da programação em lógica contribui para a otimização do tamanho das consultas. Esta análise está ligada diretamente à simplicidade da consulta, assumindo a premissa de que quanto menor uma consulta, mais simples ela é, e portanto mais fácil de ser escrita pelo usuário.

6.2.3 Cenário Utilizado

Durante a realização da avaliação experimental algumas decisões foram tomadas para guiar o processo de forma a garantir uma comparação justa entre as abordagens. Esta seção explica de forma detalhada cada uma das configurações definidas nesta etapa do trabalho.

Consulta 1	Retornar o título do artigo que possua atributo id com valor "1".
Consulta 2	Achar o título do artigo cujo nome do autor seja "Jocob Goss".
Consulta 3	Agrupar os artigos por data e calcular o número de artigos em cada grupo.
Consulta 4	Achar o título da seção seguinte à seção entitulada "*Introduction*" no artigo cujo atributo id possua valor "8".
Consulta 5	Retornar o título da primeira seção do artigo com atributo id igual a "9".
Consulta 6	Achar os títulos dos artigos onde as palavras "*the*" e "*hockey*" são mencionadas no mesmo parágrafo do "abstract".
Consulta 7	Achar os títulos dos artigos onde a palavra "*hockey*" é mencionada em todos os parágrafos do *abstract*.
Consulta 8	Retornar os nomes de todos os autores do artigo cujo atributo id possua valor "2" (levando em consideração que não se sabe o nome de um elemento no caminho).
Consulta 9	Retornar todos os nomes dos autores do artigo cujo atributo id possua valor "3" (levando em consideração que não se sabe o nome de vários elementos consecutivos).
Consulta 10	Listar os títulos dos artigos ordenados por país.
Consulta 11	Listar os títulos dos artigos cujo elemento "*country*" possua o valor igual a "Kenya", e ordenar por data.
Consulta 12	Recuperar o "*body*" do artigo cujo atributo id possua valor "4".
Consulta 13	Construir um resumo do artigo cujo atributo id seja "5", incluindo o título, o nome do primeiro autor, a data e o *abstract*.
Consulta 14	Listar o título do artigo que não possui o elemento "*genre*".
Consulta 15	Listar o nome dos autores cujo elemento "*contact*" seja vazio.
Consulta 16	Retornar o artigo cujo atributo id possua valor "6".
Consulta 17	Retornar os títulos dos artigos que contém a palavra "*hockey*".
Consulta 18	Listar os títulos e o *abstract* dos artigos que contém a frase "*the hockey*".
Consulta 19	Listar os nomes dos artigos citados pelo artigo cujo atributo id possua valor igual a "7".
Consulta Extra	Listar os ids dos artigos que estejam vinculados através de referências, com o artigo cujo id seja igual a "15"

Figura 6.1: Consultas Realizadas

Sobre a base de conhecimento gerada (TC/MD), o XBench disponibiliza 19 consultas XQuery, que decidimos utilizar em nossos experimentos. No entanto, decidimos pela criação de mais uma consulta, voltada para a exploração da possibilidade da realização de inferência proporcionada pelo nosso método, para que assim, pudéssemos fazer uma análise mais eficaz sobre os benefícios da abordagem proposta. A Figura 6.1 mostra o que as 20 consultas realizadas pretenderam obter, sendo as 19 primeiras fornecidas pelo XBench e a consulta extra criada por nós.

A avaliação compara as consultas realizadas em XQuery com consultas em linguagem Prolog utilizando a abordagem já definida. Com o objetivo de enriquecer a avaliação, cada consulta foi executada sobre cinco cenários distintos. Nos dois primeiros as consultas foram submetidas em linguagem XQuery, já nos outros três as consultas foram realizadas em Prolog. Para isso, foram escritas consultas equivalentes em Prolog, utilizando a base de conhecimento gerada pela abordagem.

Cenário 1 - as consultas XQuery foram executadas diretamente sobre os documentos XML gerados, utilizando o processador de consulta Galax (FERNANDEZ; SIMÉON, 2007). O processador Galax funciona via linha de comando e consulta documentos XML armazenados no sistema de arquivos;

Cenário 2 - os documentos XML foram armazenados no banco de dados XML nativo eXist (MEIER, 2011), e as consultas foram processadas sobre esse mesmo banco de dados, utilizando o processador de consultas XQuery fornecido pelo SGBD;

Cenário 3 – as consultas foram realizadas utilizando o protótipo apresentado no Capítulo 5, utilizando a máquina Prolog fornecida pela API tuProlog;

Os próximos cenários utilizaram duas máquinas Prolog extras, para que pudéssemos fazer a comparação dos resultados obtidos. Para a realização desse processo as etapas de tradução (etapas 1 e 2) do protótipo desenvolvido foram executadas gerar a base de conhecimento Prolog. A partir desse ponto, essas bases foram desacopladas do protótipo e puderam ser carregadas pelas máquinas de inferência utilizadas.

Cenário 4 – A base de conhecimento já traduzida para Prolog foi carregada pela máquina de inferência SWI-Prolog (WIELEMAKER *et al.*, 2010), e as consultas executadas. A escolha desta máquina se deu pela sua estabilidade e disseminação no meio acadêmico;

Cenário 5 – Por fim, a mesma base de conhecimento carregada no cenário anterior foi utilizada pela máquina de inferência YAP (COSTA, V. S. *et al.*, 2000), por se tratar de uma implementação de máquina Prolog conhecida por ter um ótimo desempenho em termos de tempo de resposta das consultas.

Com o intuito de complementar os cenários apresentados, foram utilizadas metodologias que fizeram com que o processo de avaliação fosse finalizado. Assim, o primeiro passo foi zerar a memória *cache* do computador usado no início de cada uma das 20 consultas, que tiveram suas execuções repetidas onze vezes cada. Para evitar a influência da chamada "partida a frio", decidimos descartar o primeiro resultado de cada uma das consultas, aproveitando assim os dez resultados restantes. Foi feita então a média do tempo de processamento das 10 execuções restantes. Todo o processo foi repetido em todos os cinco cenários descritos (Galax, eXist, tuProlog, SWI-Prolog e YAP).

Vale citar que o ambiente utilizado estava instalado em uma máquina com as seguintes configurações: segunda geração do processador Intel Core i5-2410 M (3M Cache, 2.30 GHz), 6 Gb DDR3 1333MHhz de memória, sistema operacional Ubuntu 11.04 64-bits (*Natty Narwhal*), Galax versão 1.0, eXist versão 1.4.1, tuProlog versão 2.4.0, SWI-Prolog versão 5.10.3 (64 bits), YAP versão 6.2.1.

6.3 Análise dos Resultados

Após a realização dos experimentos, os resultados foram registrados e comparados para que pudéssemos obter uma conclusão a respeito da nossa ideia. A proposta de utilizar o mesmo o ambiente, base de conhecimento e consulta para os cinco cenários utilizados tinha o objetivo de reproduzir uma comparação justa entre as abordagens, reduzindo os fatores que poderiam alterar os resultados. A seguir são apresentadas as conclusões obtidas quanto aos tempos de resposta e tamanhos das consultas submetidas a cada um dos cinco cenários.

6.3.1 Tempo de Resposta

Para facilitar a análise e compreensão dos tempos de resposta encontrados, a média e o desvio padrão das dez execuções de cada consulta foram calculados. Os

resultados obtidos para os cenários que utilizaram a linguagem XQuery para a execução das consultas podem ser apreciados na Tabela 6.1. Os valores estão mensurados em milissegundos.

	Galax		eXist	
Consulta	**Média**	**Desvio Padrão**	**Média**	**Desvio Padrão**
1	472,1000	1,9119	765,6383	23,5066
2	469,6000	2,5033	776,3266	21,8873
3	532,5000	4,9944	954,0784	32,7289
4	513,0000	4,0551	747,2201	31,4532
5	475,6000	2,8362	773,5963	38,9980
6	474,1000	3,9846	812,6468	35,8818
7	470,6000	2,5905	811,4922	23,6656
8	472,8000	2,5298	867,9216	37,5924
9	482,3000	3,5292	1004,5140	66,4565
10	479,8000	3,5527	905,1833	43,2889
11	465,9000	2,8460	772,6403	19,8351
12	885,3000	35,1000	1228,5078	45,9342
13	489,9000	11,6375	773,6038	22,4869
14	464,4000	6,6030	833,3844	33,7954
15	470,1000	3,8715	4182,7856	153,8509
16	748,5000	30,8877	1116,2180	41,9132
17	2968,1000	27,3392	17662,7771	935,9134
18	3209,1000	85,4471	3440,7675	107,5424
19	515,9000	3,6953	873,7925	31,7206
extra	572,6000	6,3979	724,4286	29,2638

Tabela 6.1: Tempo em milissegundos de execução das consultas para cenários utilizando XQuery (Cenários 1 e 2)

Da mesma forma, a Tabela 6.2 ilustra os resultados obtidos para os cenários que utilizaram a linguagem Prolog para a realização das mesmas consultas.

	tuProlog		SWI-Prolog		YAP	
Consulta	**Média**	**DP**	**Média**	**DP**	**Média**	**DP**
1	1009,02646	61,74612	0,05752	0,00981	-	-
2	1017,90505	77,94667	0,48661	0,00283	-	-
3	1005,77041	85,84864	0,36251	0,00774	-	-
4	979,73357	62,27453	0,07710	0,01201	-	-
5	993,55451	71,06585	0,01061	0,00025	-	-
6	1657,58597	72,22715	9,11455	0,09663	2,00000	0
7	2201,34602	89,21575	15,81332	1,60823	3,00000	0
8	976,71163	71,04941	0,08592	0,00197	-	-
9	961,89376	54,85471	0,11844	0,00259	-	-
10	1084,82150	78,59166	0,17275	0,00352	-	-
11	1057,68810	64,99266	0,11215	0,00597	-	-
12	4191,95528	203,80556	41,91641	0,58214	1,00000	0
13	972,04812	71,98909	0,03392	0,00319	-	-
14	957,95286	59,06064	0,10326	0,00327	-	-
15	975,26408	78,03873	0,74329	0,00899	-	-
16	8208,78695	893,44362	10,49194	0,59383	-	-
17	-	-	1336,59949	6,99779	375,10000	2,80674
18	-	-	4374,64833	23,57895	2381,60000	10,53249
19	1041,11376	54,54049	0,17054	0,02959	-	-
extra	6441,76514	84,03353	138,90898	1,64445	55,50000	2,71825

Tabela 6.2: Tempo em milissegundos de execução das consultas para cenários utilizando Prolog (Cenários 3, 4 e 5)

Com base nos valores das médias mostrados nas tabelas acima, foi possível construir o gráfico ilustrado pela Figura 6.2, onde é possível comparar melhor os tempos de execução das consultas submetidas nos diferentes cenários apresentados.

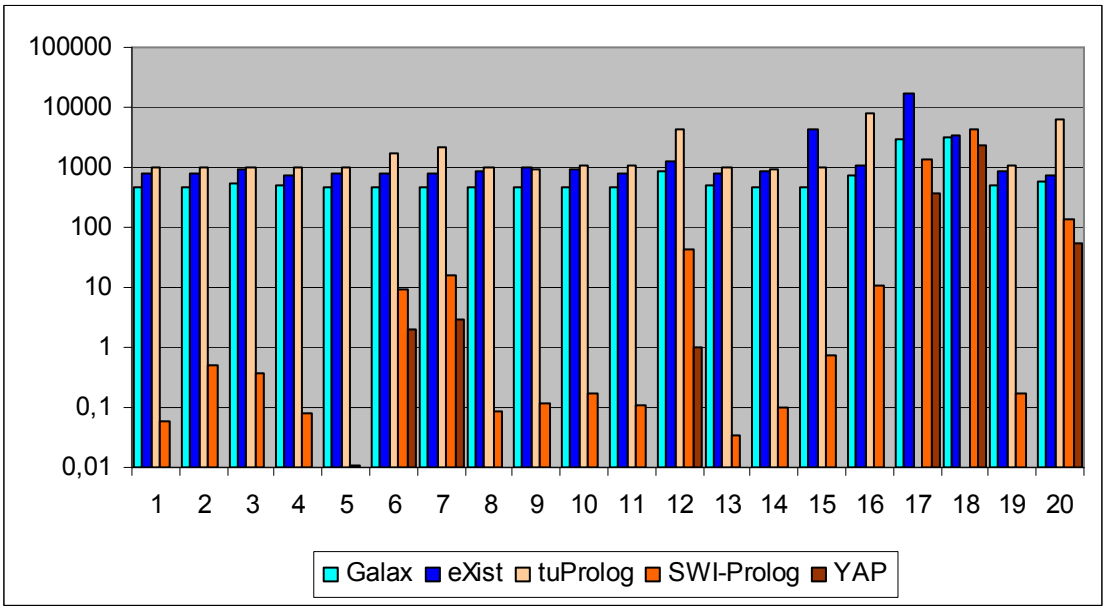

Figura 6.2: Gráfico comparativo dos tempos de execução

Analisando o gráfico acima é possível observar que a média dos tempos de execução das consultas realizadas pelo Galax (cenário 1) é ligeiramente inferior à média obtida pelas consultas submetidas ao eXist (cenário 2) e que esta diferença é quase constante. As exceções ocorrem nas consultas 15 e 17, onde a diferença das médias se acentua, e nas consultas 18 e extra, onde os resultados se aproximam. Este resultado por si só é bastante intrigante, pois, teoricamente, o eXist, que é um SGBD nativo e utiliza indexação dos documentos XML, deveria ter tido desempenho muito superior ao desempenho do Galax, que tem que, a cada consulta, ler os documentos XML em disco para depois processar as consultas. De fato, o Galax é tido como um processador de consulta extremamente eficiente no meio acadêmico.

Em relação ao tuProlog (cenário 3), é notável que a Tabela 6.2 não possui os valores referentes às consultas 17 e 18. Optamos por não colocá-los, pois tais consultas paravam de executar antes de imprimir todo o resultado, devido ao tamanho da resposta. Mesmo assim calculamos o tempo de execução necessário para apresentar o que foi impresso: a consulta 17 obteve média 11602,86065 e desvio padrão 403,09387, e a 18 valores 22592,08908 e 520,39683 respectivamente. Conclui-se que suas médias são ainda superiores às obtidas no segundo cenário. Essa observação nos fez questionar se o fato de se tratar de uma API Java que estava sendo utilizada pela nossa aplicação para

proporcionar uma máquina de inferência, contribuiu para o aumento dos tempos de execução, ou se realmente as consultas realizadas em Prolog seriam mais demoradas.

Para sanar esta dúvida, optamos por utilizar outras máquinas de inferência Prolog. Esta análise foi feita pelos cenários 4 e 5. O quarto cenário proporcionou médias bem inferiores em relação aos anteriores, chegando a tempos 10 mil vezes melhores aos obtidos pelo cenário 1 (consulta 5). Dessa forma começamos a ter a comprovação de que a abordagem proposta poderia proporcionar melhores tempos de resposta do que os métodos de consultas tradicionais. Esta constatação foi bastante surpreendente, pois, ao idealizar o método, esperávamos ganhar apenas em simplicidade, nunca em desempenho. Ainda analisando o gráfico da Figura 6.2, é possível observar que com exceção da décima oitava, todas as consultas processadas pela máquina de inferência SWI-Prolog obtiveram suas médias de resposta inferiores aos observados nos três cenários anteriores.

Em relação ao quinto cenário, observamos o alcance dos menores tempos de resposta. No entanto, a regra para obtenção dos tempos em YAP não é capaz de fornecer respostas inferiores à um milissegundo, retornando zero como resultado. Como pode ser observado na Tabela 6.2, apenas seis resultados foram computados (consultas 6, 7, 12, 17, 18 e extra) de forma precisa, sendo estes os melhores dentre os cinco cenários avaliados. Todas as outras consultas demoravam menos do que 1 milisegundo. Concluímos assim, que o YAP foi capaz de gerar resultados melhores do que os cenários que utilizaram a linguagem XQuery (cenários 1 e 2), uma vez que todos os seus resultados (além dos que foram apresentados), foram inferiores à um milissegundo (motivo pelo qual não puderam ser devidamente medidos).

6.3.2 Tamanho das Consultas

Quanto à avaliação do tamanho das consultas necessárias para obter o mesmo resultado em linguagem XQuery e Prolog foi observado que a nossa abordagem possibilitou a confecção de consultas mais simples, fazendo com que o trabalho do usuário final fosse reduzido. Como exemplo pode-se observar a Figura 6.3 que contém em sua parte superior a consulta 1 em linguagem XQuery, e em linguagem Prolog em sua parte inferior. É possível perceber que utilizando a nossa abordagem a resposta pôde ser obtida através de uma consulta mais simples. Porém, o usuário especialista pode

facilitá-la ainda mais através da regra: “consulta1(TITLE) :- findall(C, (article(A), id(A,'1'), prolog(A,B), title(B,C)), TITLE).”, fazendo com que o usuário final apenas use-a na forma: “consulta1(TITLE).”.

Consulta 1: Retornar o título do artigo que possua atributo id com valor “1”.

```
for $art in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="1"]
return
    $art/prolog/title
```

```
findall(C, (article(A), id(A,'1'), prolog(A,B), title(B,C)), TITLE).
```

Figura 6.3: Consulta 1 – descrição, representação em linguagem XQuery e representação em Prolog

Consulta 10: Listar os títulos dos artigos ordenados por país.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
       /article/prolog
       order by $a/dateline/country
       return
           <Output>
               {$a/title}
               {$a/dateline/country}
    </Output>
```

```
orderCountry(R) :- findall((PAIS, TITLE), (article(A), prolog(A, B),
    dateline(B, G), title(B, TITLE), country(G, PAIS)), L),
    sort(L, R).

orderCountry(X).
```

Figura 6.4: Consulta 10 – descrição, representação em linguagem XQuery e representação em Prolog

A Figura 6.4 ilustra a comparação da consulta 10 entre as linguagens XQuery e Prolog. Neste caso a regra Prolog “orderCountry” teve que ser inserida manualmente pois não pôde ser derivada do esquema. Novamente há duas opções: o usuário especialista cadastra-a durante o processo de inserção de regras manuais e o usuário final utiliza-a conforme mostrado na figura (“orderCountry(X).”), ou o próprio usuário

final realiza a consulta diretamente. Neste último caso ele terá que informar o corpo da regra para isso.

Consulta 16: Retornar o artigo cujo atributo id possua valor "6".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="6"]
return
    $a
```

```
dumpArticle(ID, [ID, DUMP_PROLOG, DUMP_BODY, DUMP_EPILOG]) :-
   article(ID, PROLOG, BODY, EPILOG),
   dumpProlog(PROLOG, DUMP_PROLOG), dumpBody(BODY, DUMP_BODY),
   dumpEpilog(EPILOG, DUMP_EPILOG).
dumpProlog(ID, [ID, TITLE, DUMP_AUTHORS, DUMP_DATELINE,
   DUMP_KEYWORDS]) :- prolog(ID, TITLE, AUTHORS, DATELINE, KEYWORDS),
   findall(X, dumpAuthors(AUTHORS, X), DUMP_AUTHORS),
   dumpDateline(DATELINE, DUMP_DATELINE),
   dumpKeywords(KEYWORDS, DUMP_KEYWORDS).
dumpAuthors(ID, [ID, DUMP_AUTHOR]) :- authors(ID, AUTHOR),
   dumpAuthor(AUTHOR, DUMP_AUTHOR).
dumpAuthor(ID, [ID, NOME, DUMP_CONTACT]) :- author(ID, NOME, CONTACT),
   dumpContact(CONTACT, DUMP_CONTACT).
dumpContact(ID, [ID, EMAIL, PHONE]) :-
   findall(X, email(ID, X), EMAIL), findall(Y, phone(ID, Y), PHONE).
dumpDateline(ID, [ID, CITY, COUNTRY, DATE]) :-
   findall(X, city(ID, X), CITY),
   findall(Y, country(ID, Y), COUNTRY), findall(Z, date(ID, Z), DATE).
dumpKeywords(ID, [ID, KEYWORD]) :-
   findall(X, keyword(ID, X), KEYWORD).
dumpBody(ID, [ID, DUMP_ABST, DUMP_SECT]) :- abstract(ID, ABSTRACT),
   dumpAbstract(ABSTRACT, DUMP_ABST),
   findall(X, (section(ID, SECTION),
   dumpSection(SECTION, X)), DUMP_SECT).
dumpAbstract(ID, [ID, P]) :- findall(X, p(ID, X), P).
dumpSection(ID, [ID, P, DUMP_SUBSEC]) :- subsec(ID, SUBSEC),
   findall(X, dumpSubsec(SUBSEC, X), DUMP_SUBSEC).
dumpSubsec(ID, [ID, P, DUMP_SUBSEC]) :- subsec(ID, SUBSEC),
   findall(X, dumpSubsec(SUBSEC, X), DUMP_SUBSEC).
dumpSubsec(ID, [ID, P]) :- findall(X, p(ID, X), P).
dumpEpilog(ID, [ID, DUMP_REF]) :- epilog(ID, REF),
   dumpReferences(REF, DUMP_REF).
dumpReferences(ID, [ID, DUMP]) :- findall(X, a_id(ID, X), DUMP).

id(A, '6'), article(A, PROLOG, BODY, EPILOG),
   findall(DUMP_PROLOG, dumpProlog(PROLOG, DUMP_PROLOG), Prolog),
   findall(DUMP_BODY, dumpBody(BODY, DUMP_BODY), Body),
   findall(DUMP_EPILOG, dumpEpilog(EPILOG, DUMP_EPILOG), Epilog).
```

Figura 6.5: Consulta 16 – descrição, representação em linguagem XQuery e representação em Prolog

Consulta Extra: Listar os ids dos artigos que estejam vinculados através de referências, com o artigo cujo id seja igual a "15"

```
declare function local:vinculoArtigo($id as xs:string+) {
   for $ref in (doc("article1.xml"), doc("article2.xml"),
   doc("article3.xml"), doc("article4.xml"), doc("article5.xml"),
   doc("article6.xml"), doc("article7.xml"), doc("article8.xml"),
   doc("article9.xml"), doc("article10.xml"), doc("article11.xml"),
   doc("article12.xml"), doc("article13.xml"), doc("article14.xml"),
   doc("article15.xml"), doc("article16.xml"), doc("article17.xml"),
   doc("article18.xml"), doc("article19.xml"), doc("article20.xml"),
   doc("article21.xml"), doc("article22.xml"), doc("article23.xml"),
   doc("article24.xml"), doc("article25.xml"), doc("article26.xml"))
      /article[@id=$id]/epilog/references
   return
      <vinculo>
         <refs_diretas>{$ref/a_id}</refs_diretas>
         <refs_indiretas>
            {local:vinculoArtigo($ref/a_id/text())}
         </refs_indiretas>
      </vinculo>
};

for $r in ( doc("article1.xml"), doc("article2.xml"),
   doc("article3.xml"), doc("article4.xml"), doc("article5.xml"),
   doc("article6.xml"), doc("article7.xml"), doc("article8.xml"),
   doc("article9.xml"), doc("article10.xml"), doc("article11.xml"),
   doc("article12.xml"), doc("article13.xml"), doc("article14.xml"),
   doc("article15.xml"), doc("article16.xml"), doc("article17.xml"),
   doc("article18.xml"), doc("article19.xml"), doc("article20.xml"),
   doc("article21.xml"), doc("article22.xml"), doc("article23.xml"),
   doc("article24.xml"), doc("article25.xml"), doc("article26.xml"))
      /article[@id="15"]/epilog/references
return
   <resposta>
      <refs_diretas>{$r/a_id}</refs_diretas>
      <refs_indiretas>
         {local:vinculoArtigo($r/a_id/text())}
      </refs_indiretas>
   </resposta>
```

```
referencia(A1, A2) :- article(A), id(A, A1), epilog(A, C),
   references(C, D), a_id(D, X), a_id(D, A2).
vinculoArtigo(A1, A2) :- referencia(A1, A2).
vinculoArtigo(A1, A2) :- referencia(A1, A3), vinculoArtigo(A3, A2).


setof(Y, vinculoArtigo('15', Y), L).
```

Figura 6.6: Consulta Extra – descrição, representação em linguagem XQuery e representação em Prolog

Durante os experimentos foi possível observar que ainda que as informações implícitas pudessem ser obtidas mais facilmente através da nossa abordagem, a linguagem XQuery simplificava a manipulação da estrutura do documento através das expressões de caminho. A Figura 6.5 mostra a comparação entre as linguagens referente à consulta 16. Nesta consulta deseja-se obter todas as informações do artigo cujo atributo "id" possua valor igual a 6. Em linguagem XQuery foi possível criar uma

variável “a” e atribuir o valor do elemento raiz “*article*” cujo atributo “id” seja igual a 6 a ela. Para obter o resultado bastou retornar o valor desta variável. Já em Prolog, não existe esta estrutura a ser percorrida, pois não se trata de uma linguagem voltada para manipular dados desta forma. Então, para a realização da consulta foi necessário criar regras capazes de explorar todas as informações acerca do artigo. Como pode ser visto na parte inferior da Figura 6.5, foram inseridas manualmente regras “dump” com este objetivo, e o usuário final teve apenas que utilizá-las para obter a resposta necessária.

No entanto, foi a consulta extra que melhor demonstrou os benefícios da utilização de uma linguagem lógica para permitir a realização de inferências. O objetivo desta consulta era listar os atributos “ids” dos artigos que possuíam um vínculo em qualquer nível com o artigo cujo mesmo atributo possuía valor igual a “15”. Logo, seria necessária uma função recursiva que processasse cada atributo “id” acessível a partir de todas as referências do artigo com valor de “id” igual a “15”. A Figura 6.6 ilustra a comparação desta consulta, onde é possível observar que além de sua simplicidade em linguagem Prolog, mais uma vez a necessidade de conhecer a estrutura do documento e o trabalho de fornecer as regras de recursão ficam sob a responsabilidade do usuário especialista, sobrando para o usuário final apenas a tarefa de utilizar as regras criadas.

O Anexo B contém a comparação de todas as consultas em linguagem XQuery e Prolog.

6.4 Ameaças à Validade

Existem questões que interferem nos resultados dos experimentos afetando a sua validade. Estas questões denominam-se ameaças à validade (WOHLIN *et al.*, 2000). Durante a avaliação experimental, muito cuidado foi tomado para que tais ameaças fossem evitadas e os resultados obtidos fossem justos. No entanto, não podemos garantir que tais resultados não tenham sido afetados de alguma forma por essas ameaças. Esta seção identifica os possíveis fatores que podem ter influenciado o resultado obtido e se tornaram uma ameaça à validade dos experimentos, categorizando-os em validade interna e validade externa.

Ameaças à validade interna são questões que interferem na relação causa e efeito sem o conhecimento do pesquisador e fazem com que as conclusões sobre os experimentos não sejam verdadeiras. Como exemplo, pode-se concluir erroneamente

que A implica B, sem o conhecimento de um fator X que seria o responsável pelo resultado B. No experimento executado, a seguinte ameaça à validade interna foi identificada:

- O tempo de execução das consultas foi obtido de formas diferentes para cada uma das cinco abordagens avaliadas. No primeiro cenário o tempo foi obtido através do comando "time" fornecido pelo Linux. No caso do eXist, um programa em linguagem Java utilizou o método "nanoTime()" da classe "System" para capturar os tempos imediatamente antes e após a chamada do método de execução da consulta. O resultado veio da subtração do tempo final pelo inicial. O terceiro cenário também utilizou a abordagem anterior. Os resultados da máquina SWI-Prolog foram obtidos pela subtração dos tempos obtidos pela regra *built-in* "get_time(X)" executada antes e depois da consulta. Por fim, o cálculo do tempo de execução das consultas realizadas no quinto cenário foi possível através da regra "time(X)", que calcula o tempo de execução de "X".

As ameaças à validade externa são os fatores que impedem a generalização dos experimentos, fazendo com que o resultado observado seja limitado a certos ambientes. Foram identificadas as seguintes ameaças à validade externa nos experimentos realizados:

- Os experimentos foram todos executados sobre a base de conhecimento fornecida pelo XBench, fazendo com que todas as consultas fossem realizadas sobre uma única coleção de documentos XML.

Cabe ressaltar que esta base de conhecimento possuía uma estrutura complexa o suficiente para que os resultados da avaliação fossem satisfatórios.

- Dezenove das vinte consultas utilizadas foram fornecidas pelo XBench. O fato de termos criado mais uma consulta objetivou mostrar como o resultado se comportaria para consultas que seriam mais facilmente resolvidas com a ajuda de inferências.

Cabe ressaltar que as dezenove consultas disponibilizadas pelo *benchmark* possuíam interesses e formas de obter a informação de formas diversas, o que tende a proporcionar a generalização dos resultados obtidos.

6.5 Considerações Finais

Este capítulo apresentou de forma minuciosa a avaliação experimental realizada para comparar a abordagem descrita nesta dissertação com os métodos de consultas tradicionais em linguagens XML.

Foi possível observar que os resultados obtidos foram favoráveis à ideia proposta, apresentando menores tempos de execução e desonerando o usuário final do trabalho de criação de consultas complexas. Algumas vezes, o processo é facilitado através da criação de regras por parte de um usuário especialista, para que auxiliem o processo e permitam a obtenção de respostas mais elaboradas.

Ainda que a avaliação experimental tenha comprovado a eficácia do método de consulta desenvolvido, foi possível perceber que as consultas que exploram a estrutura do documento XML e são facilitadas pelas expressões de caminho em linguagem XQuery necessitaram de mais esforço por parte do usuário especialista, demandando mais regras capazes de explorar as informações necessárias em linguagem Prolog.

Analisando os resultados dos experimentos referentes ao tamanho das consultas é possível perceber que na maioria dos casos o método de consulta proposto nesta dissertação possibilitou consultas menores e mais elegantes, porém, mesmo assim não podemos afirmar com certeza que nosso método foi capaz de simplificar a consulta. Esta afirmação requer uma análise de usabilidade e deve levar em conta o conhecimento de Prolog por conta do usuário, o que não foi feito neste trabalho. Portanto o ganho na utilização do método está na eficiência e na diminuição do tamanho da consulta necesária para a obtenção dos resultados.

Assim, podemos concluir que a utilização de uma linguagem lógica e a possibilidade de realização de inferências pode ser bastante útil no processo de consulta à informações implícitas em documentos XML. Porém, o fato da linguagem Prolog não estar voltada a processar dados em linguagem XML, algumas vezes faz com que regras extras tenham que ser elaboradas para obter as informações desejadas, conforme foi visto na consulta 16.

Capítulo 7. Conclusão e Trabalhos Futuros

7.1 Contribuições

Esta dissertação apresentou um método de consulta em documentos XML que utiliza inferência para facilitar a obtenção de respostas mais elaboradas. A partir do vocabulário imposto pelos esquemas que definem as estruturas dos documentos a serem consultados, identificamos que seria possível extrair regras a respeito das instâncias presentes nos documentos XML, e que essas regras poderiam contribuir para o enriquecimento da base de conhecimento, favorecendo a realização de inferências e facilitando assim a realização das consultas.

No decorrer da pesquisa, o protótipo desenvolvido e os experimentos realizados sobre este comprovaram que o nosso objetivo foi atingido, demonstrando que as consultas utilizando o nosso método foram realmente menores quando comparadas às consultas que utilizam linguagem XQuery, e que o tempo de processamento da consulta pelas máquinas de inferência SWI-Prolog e YAP foram inferiores aos tempos de processamento das consultas originais em XQuery. Dessa forma, podemos destacar algumas contribuições proporcionadas por este trabalho:

- Uma metodologia de consulta em documentos XML voltada a facilitar o acesso às informações implícitas através de inferências. A abordagem utiliza um método de tradução dos dados originalmente representados pelos documentos XML para uma linguagem de Programação em Lógica capaz de permitir a realização das inferências. Além disso, é possível que um usuário especializado em Prolog cadastre novas regras (que chamamos de regras manuais) visando criar novos relacionamentos, permitindo assim o acesso às novas informações (antes implícitas). Com isso, utilizando uma base de conhecimento mais sólida, as consultas submetidas pelo usuário tendem a ser mais simples e capazes de retornar respostas mais elaboradas. Cabe resaltar que, ainda que os resultados obtidos nos experimentos sejam satisfatórios, a intenção do método de consulta proposto não é substituir a linguagem XQuery, mas sim servir como uma opção em caso de consultas a informações implícitas, visando diminuir o tamanho da consulta e diminuir o tempo de execução. Não podemos deixar de reconhecer que a

XQuery é uma linguagem muito poderosa e que possui vantagens em determinados cenários de consultas que não foram abordados nesta dissetação.

- Um método de tradução do vocabulário imposto pelos esquemas para regras em linguagem Prolog. Essa tradução foi minuciosamente estudada com o objetivo de gerar um resultado que fosse relevante e contribuísse com informações extras a respeito da estrutura dos documentos XML que foram consultados. Nesse caso, o processo é composto por regras que analisam os delimitadores de grupo dos elementos complexos definidos no esquema para produzirem as regras Prolog correspondentes, levando em conta toda a estrutura interna deste elemento complexo. Foram definidas três regras principais, sendo uma para cada delimitador de grupo, e a partir destas algumas variações surgiram para suprir a possibilidade de existência de estruturas complexas internas a um elemento complexo. Ainda que não possamos garantir com certeza que o método de tradução esteja correto, vale citar que durante a realização dos experimentos a tradução do esquema ocorreu conforme esperado, o que nos mostrou que para aquele cenário as regras de tradução estavam corretas.

- Cinco regras capazes de traduzir toda a informação descrita em documentos XML para fatos Prolog. Tais regras foram elaboradas com o objetivo de derivar fatos granulares que contenham toda a semântica existente nos documentos XML, ou seja, nenhuma informação é perdida durante a tradução. Isso foi possível, pois conseguimos através de estudos classificar os elementos contidos em um documento XML em cinco tipos diferentes (elemento raiz, elemento simples sem atributo, elemento simples com atributo, elemento complexo e elemento misto). A partir de então criamos uma regra de tradução para cada tipo de elemento possível. A intenção de trabalhar com fatos granulares é proposital, pois visa facilitar a combinação de fatos e regras Prolog durante a criação de novas regras (as chamadas manuais), ou até mesmo na realização das consultas. Assim como o método de tradução dos esquemas em regras Prolog, as regras de tradução dos fatos obtiveram excelentes resultados durante a realização dos experimentos, demonstrando que também seriam suficientes para traduzir os documentos fornecidos pelo xBench de forma correta.

- Implementação de protótipo da ideia proposta, onde é possível executar todas as etapas da metodologia de consulta em documentos XML idealizada por esta pesquisa. Para isso, foi utilizada a linguagem de programação Java integrada à API tuProlog para desenvolver o sistema e permitir que as informações fossem processadas

por uma máquina de inferência. As etapas da abordagem encontram-se bem separadas, permitindo que a base de conhecimento traduzida seja desacoplada do protótipo e integrada a uma outra máquina de inferência, conforme foi feito durante os experimentos com a SWI-Prolog e a YAP.

A abordagem foi avaliada através de experimentos que analisaram o tamanho das consultas e o seu tempo de processamento. As consultas foram feitas diretamente sobre os documentos XML, utilizando a linguagem XQuery, e também sobre a base de conhecimento Prolog. Para as consultas em linguagem XML, utilizamos o processador Galax e o banco de dados nativo XML eXist. Já as consultas em Prolog utilizaram a metodologia desenvolvida de tradução de dados XML para Prolog. Tais consultas foram testadas em três máquinas de inferência diferentes: API Java tuProlog, e máquinas Prolog SWI-Prolog e YAP. Assim, o resultado obtido expõe o desempenho de todas as consultas realizadas em um mesmo ambiente, tornando-se uma comparação valiosa não somente quando se tratar da nossa pesquisa, mas também quando compararmos as linguagens utilizadas entre si.

Vale dizer que no decorrer desta pesquisa obtivemos a publicação de um artigo no XXVI Simpósio Brasileiro de Banco de Dados (LIMA *et al.*, 2011). Este artigo foi convidado para ser publicado no *Journal of Information and Data Management*, e encontra-se em fase de avaliação pelo corpo editorial da revista (LIMA *et al.*, 2012).

7.2 Limitações

Apesar de termos desenvolvido a nossa ideia para conseguirmos chegar a uma versão aprimorada, existem algumas limitações que precisam ser abordadas em trabalhos futuros. Esta seção apresenta tais limitações e um esboço de solução que adotamos para tentar superá-las.

Uma das limitações da abordagem é o fato da linguagem Prolog interpretar palavras começando por letras maiúsculas como variáveis. Assim, o nosso mecanismo de tradução pode não operar de forma correta quando o documento XML possuir nomes de elementos (ou até mesmo de atributos) que comecem por uma letra maiúscula. A solução adotada no protótipo implementado foi converter todas as letras maiúsculas para minúsculas durante o processo de tradução. Uma outra solução estudada foi a utilização de um caractere no início das palavras traduzidas para evitar que os termos se

iniciassem com letras maiúsculas (como por exemplo: "_" poderia ser adotado por ser reconhecido em linguagem Prolog). O problema desta solução é a geração de palavras diferentes das originais que estão presentes nos documentos XML, o que poderia confundir o usuário na hora da realização da consulta.

A possibilidade do uso de pontuação (".", "-", ":" ...) e acentos para definir nomes de elementos em XML contrasta com a sintaxe Prolog, que permite apenas letras, números, *underscore* ("_") e "/". Portanto, a nossa abordagem se limita a documentos XML que contenham apenas os caracteres permitidos em linguagem Prolog, produzindo traduções com erro de sintaxe para os demais casos. Para este caso, uma possível solução é a substituição do caractere em desacordo por um aceito em Prolog, ou seja, remover os acentos (trocar "á" por "a"), substituir caracteres de pontuação por um caractere especial, como o *underscore,* e assim adequar a tradução para um resultado aceito em linguagem Prolog. Essa geração de palavras diferentes das originais presentes nos documentos XML torna-se, novamente, um problema para os usuários que realizarão a consulta.

Devemos também analisar o fato de não extrairmos regras automáticas de DTDs, pois torna-se uma limitação para documentos que são validados por estas ao invés de esquemas. Ainda que não sejam obrigatórias, essas regras diminuem o esforço do usuário que irá inserir regras manuais e que poderão ainda ser reutilizadas durante o processo de consulta. Assim, apesar de a tradução de DTDs não fazer parte do escopo desta dissertação, esta tarefa pode ser incluída como um possível trabalho de aprimoramento futuro.

Portanto, ainda que existam limitações para a nossa abordagem, tivemos o cuidado de desenvolver a nossa ideia para que ela se tornasse capaz de satisfazer a maioria dos casos. No entanto, sabemos da existência de pequenos ajustes a serem feitos para que o nosso método consiga se adaptar à toda flexibilidade existente na linguagem XML. A abordagem proposta é genérica o suficiente para um primeiro estudo, e permitiu a obtenção de bons resultados experimentais, onde toda a base de conhecimento pôde ser traduzida e as consultas realizadas. A base fornecida por este trabalho pode agora ser aprimorada para uso efetivo em sistemas que necessitam de consultas a dados implícitos.

7.3 Trabalhos Futuros

Ainda que tenhamos conseguido fechar um escopo e desenvolver a nossa pesquisa focada no objetivo que havíamos proposto, o assunto que tratamos é bastante complexo, o que faz com que a nossa abordagem ainda possa evoluir futuramente.

Durante a realização da pesquisa, conseguimos vislumbrar algumas ideias interessantes e julgamos que seria válido citá-las nesta dissertação como trabalhos futuros. Dessa forma o restante desta seção apresenta ideias que podem contribuir para a evolução da abordagem proposta.

Embora as informações iniciais estejam representadas em documentos XML, os usuários finais devem realizar as consultas em linguagem Prolog, uma vez que todos os dados serão traduzidos para linguagem lógica com o objetivo de possibilitar a realização de inferências. Ainda que grande parte dos usuários interessados em submeter consultas a documentos XML sejam de áreas tecnológicas e possivelmente estejam familiarizados com Prolog, reconhecemos a dificuldade do uso desta linguagem para usuários que não possuem experiência com a mesma. Porém, esse empecilho pode ser solucionado com a adoção de uma interface gráfica que gere consultas Prolog a partir de uma seleção de opções em alto nível (COSTA; BRAGANHOLO, 2010). Com isso, o usuário poderia criar a sua consulta através de uma interface mais amigável, sem ter que conhecer a sintaxe Prolog.

Outra melhoria que pode ser citada como um trabalho futuro é a substituição do usuário especialista por um processo de *Data Mining*, fazendo com que a inserção de regras manuais se torne uma tarefa automatizada. Esta alteração elimina a necessidade da existência de um especialista Prolog e generaliza as regras produzidas uma vez que tende a inserir mais regras referentes às informações contidas na base de conhecimento, tornando possível a obtenção de novas respostas.

Durante a apresentação de um artigo no SBBD 2011 (LIMA *et al.*, 2011), nos foi sugerido uma generalização da nossa ideia para que fosse integrada a outras linguagens de consulta. Na ocasião, durante a comparação da nossa abordagem com a consulta XQuery em um exemplo que utilizava recursão, foi sugerido uma adaptação do nosso método para representar consultas em linguagem SQL, visando facilitar a elaboração de consultas recursivas e diminuir o tamanho de algumas consultas

complexas. Ainda que essa ideia fuja do escopo da nossa pesquisa, imaginamos ser interessante listá-la como trabalho futuro para validarmos se a nossa abordagem também é aplicável a linguagens de consulta sobre dados estruturados, como nos fora apontado em tal encontro.

Conforme citado na seção anterior, um possível trabalho futuro seria a tradução do vocabulário imposto por uma DTD para regras Prolog, possibilitando assim a extração de regras automáticas caso os documentos alvo da consulta sejam validados por DTD ao invés de esquemas.

Finalizando, também podemos citar a realização dos experimentos com usuários com níveis de experiência diferentes (tanto em Prolog quanto em XQuery) para avaliar a adequação da proposta frente a esses diferentes níveis de conhecimento. Dessa forma, conseguiremos analisar se a possibilidade da realização de inferências traz realmente benefícios ao usuário interessado em informações implícitas, e se a nossa abordagem exige algum conhecimento específico para que os resultados sejam satisfatórios a todos os usuários.

Referências Bibliográficas

ALMENDROS-JIMÉNEZ, J. M.; BECERRA-TERÓN, A.; ENCISO-BANOS, F. J. Querying XML documents in logic programming. **Journal of Theory and Practice of Logic Programming**, v. 8, n. 3, p. 323–361, 2008.

BAILEY, J.; BRY, F.; FURCHE, T.; SCHAFFERT, S. **Web and semantic web query languages: A survey**. Proc. of Reasoning Web, First International Summer School, LNCS 3564. Heidelberg, Germany. **Anais**... p. 35-133, 2005.

BARU, C.; LUDAESCHER, B.; PAPAKONSTANTINOU, Y.; VELIKHOV, P.; VIANU, V. Features and requirements for an xml view definition language: Lessons from xml information mediation. **W3C Workshop on Query Languages**, 1998.

BOAG, S.; CHAMBERLIN, DON; FERNÁNDEZ, M. F. *et al.* **XQuery 1.0: An XML Query Language (Second Edition)**. W3C Recommendation, 2010. Disponível em: http://www.w3.org/TR/xquery/. Acesso em: 7 abr. 2010.

BOLEY, H. **Relationships between logic programming and XML**. Proc. of the Workshop on Logic Programming, Berlin, Germany. **Anais**... p. 19-34 , 2000.

BOLEY, H. **The rule markup language: RDF-XML data model, XML schema hierarchy, and XSL transformations**. International Conference on Web Knowledge Management and Decision Support, Tokyo, Japan. **Anais**... p. 5-22 , 2001.

CALVANESE, D.; GIACOMO, G. DE; LENZERINI, M. Representing and reasoning on XML documents: A description logic approach. **Journal of Logic and Computation**, v. 9, n. 3, p. 295, 1999.

CERI, S.; GOTTLOB, G.; TANCA, L. Logic programming and databases. **Berlin: Springer-Verlag**, 1990.

CHAMBERLIN, D.; ROBIE, J.; FLORESCU, D. **Quilt: An XML query language for heterogeneous data sources**. Lecture Notes in Computer Science, Springer-Verlag. **Anais**... p.53-62, Disponível em: http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf., 2000.

CHANG, C. L.; WALKER, A. **PROSQL: a Prolog programming interface with SQL/DS**. Proceedings from the First International Workshop on Expert Database Systems. Redwood City, CA, USA. **Anais**... 1986. Disponível em: http://dl.acm.org/citation.cfm?id=21004.21019. Acesso em: 19 nov. 2011.

CLARK, J.; DEROSE, S. **XML Path Language (XPath) Version 1.0**. W3C Recommendation, 1999. Disponível em: http://www.w3.org/TR/xpath/. Acesso em: 7 abr. 2010.

COELHO, J.; FLORIDO, M. Type-based XML processing in logic programming. **International Symposium on Practical Aspects of Declarative Languages**, New Orleans, USA, p. 273–285, 2003.

COLMERAUER, A.; KANOUI, H.; PASERO, R.; ROUSSEL, P. Un systeme de communication homme-machine en francais. **Groupe de Recherche en Intelligence Artificielle, Université d'Aix- Marseilles, France**, v. 3, p. 99, 1973.

COSTA, P. V. C.; BRAGANHOLO, V. **Uma nova abordagem para consulta a dados de proveniência**. Workshop de Teses e Dissertações em Banco de Dados (WTDBD). Belo Horizonte, MG. **Anais**... p. 1-6 , 2010.

COSTA, V. S.; DAMAS, L.; REIS, R.; AZEVEDO, R. **YAP: Yet Another Prolog**. Disponível em: http://www.dcc.fc.up.pt/~vsc/Yap/documentation.html. Acesso em: 2 nov. 2011.

DART, P. W.; ZOBEL, J. A regular type language for logic programs. **Types in logic programming**, p. 157–187, 1992.

DENTI, E.; OMICINI, A.; RICCI, A. tuProlog: A light-weight Prolog for Internet applications and infrastructures. **Practical Aspects of Declarative Languages**, p. 184–198, 2001.

DEUTSCH, A.; FERNANDEZ, M.; FLORESCU, D.; LEVY, A.; SUCIU, D. A query language for XML. **Computer networks**, v. 31, n. 11-16, p. 1155–1169, maio 1999.

DONINI, F. M.; LENZERINI, M.; NARDI, D.; SCHAERF, A. Reasoning in description logics. **Reasoning in description logics. In Gerhard Brewka, editor, Principles of Knowledge Representation, Studies in Logic, Language and Information**, p. 191–236, 1996.

FERNANDEZ, M.; SIMÉON, J. **Galax**. Disponível em: http://galax.sourceforge.net/. Acesso em: 2 nov. 2011.

GALLAIRE, H.; MINKER, J. **Logic and data bases**. Advances in Data Base Theory, Plenum Press, New York: ed. H. Gallaire and J. Minker, Plenum Press, 1978. Disponível em: http://dblp.uni-trier.de/db/indices/a-tree/g/Gallaire:Herv=eacute=.html.

KEISLER, H. J. Reduced products and Horn classes. **Transactions of the American Mathematical Society**, v. 117, p. 307–328, 1965.

KIFER, M. Rule interchange format: The framework. **Web Reasoning and Rule Systems**, p. 1–11, 2008.

KOTSAKIS, E. **Structured information retrieval in XML documents**. ACM Symposium on Applied Computing. Madrid, Spain .**Anais**... p. 663-667, 2002.

LAENDER, A. H. .; MORO, M. M.; NASCIMENTO, C.; MARTINS, P. An X-ray on Web-available XML Schemas. **ACM SIGMOD Record**, v. 38, n. 1, p. 37–42, 2009.

LIMA, D. M. G.; DELGADO, C.; MURTA, L.; BRAGANHOLO, V. **Consultando documentos XML utilizando inferência**. Simpósio Brasileiro de Banco de Dados. Florianópolis, SC, Brasil, out 2011.

LIMA, D. M. G.; DELGADO, C.; MURTA, L.; BRAGANHOLO, V. Towards Querying Implicit Knowledge in XML Documents. **Journal of Information and Data Management**, submetido. 2012.

LLOYD, J. W. **Foundations of logic programming**. Nova York, Springer-Verlag, 1987.

MEIER, W. **eXist**. Disponível em: http://exist.sourceforge.net/. Acesso em: 2 nov. 2011.

MELLO, R.; DORNELES, C.; KADE, A.; BRAGANHOLO, V.; HEUSER, C. A. Dados Semi-Estruturados. **Simpósio Brasileiro de Banco de Dados (SBBD) -**

Tutorial, 2000, João Pessoa. XV Simpósio de Banco de Dados: Minicursos e Tutoriais, p. 475–511, 2000.

NIEMI, T.; JARVELIN, K. Prolog-based meta-rules for relational database representation and manipulation. **Software Engineering, IEEE Transactions on**, v. 17, n. 8, p. 762–788, 1991.

ROBIE, JONATHAN; LAPP, J.; SCHACH, D. **XML Query Language (XQL)**. Disponível em: http://www.w3.org/TandS/QL/QL98/pp/xql.html. Acesso em: 7 abr. 2010.

ROBINSON, J. A. A machine-oriented logic based on the resolution principle. **Journal of the ACM (JACM)**, v. 12, n. 1, p. 23–44, 1965.

SEIPEL, D. **Processing XML-documents in Prolog**. Proc. Workshop Logische Programmierung.(2002). Technische Universität Dresden, Dresden, Germany. **Anais**... p. 1-15 , 2002.

W3C. **XML Schema**. Disponível em: . Acesso em: http://www.w3.org/XML/Schema. 17 jan. 2012.

W3CSCHOOLS. **W3CSchools - XPath**. Disponível em: http://www.w3schools.com/xpath/xpath_intro.asp.Último acesso em: 16 jan. 2011.

WIELEMAKER, J. SWI-Prolog SGML/XML Parser, Version 2.0.5, Tech. rep., Human Computer-Studies (HCS)**, University of Amsterdam,** v. 15, p. 1018, 2005.

WIELEMAKER, J.; SCHRIJVERS, T.; TRISKA, M.; LAGER, T. Swi-prolog. **Theory and Practice of Logic Programming (TPLP)**, 2010.

WOHLIN, C.; RUNESON, P.; HÖST, M. *et al.* **Experimentation in Software Engineering: An Introduction**. 1. ed., Norwell, USA, Kluwer Academic Publishers, 2000.

YAO, B. B.; OZSU, M. T.; KHANDELWAL, N. **XBench benchmark and performance testing of XML DBMSs**. International Conference on Data Engineering. Boston, MA, USA. **Anais**... p. 621-632 , 2004.

Anexo A. Esquema que valida os documentos XML alvo da consulta

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
      elementFormDefault="qualified">
   <xs:element name="article" type="Tarticle"/>
   <xs:complexType name="Tarticle">
      <xs:sequence>
         <xs:element name="prolog" type="Tprolog"/>
         <xs:element name="body" type="Tbody"/>
         <xs:element name="epilog" type="Tepilog"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:byte" use="required"/>
      <xs:attribute name="lang" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tprolog">
      <xs:sequence>
         <xs:element name="title" type="xs:string"/>
         <xs:element name="authors" type="Tauthors" minOccurs="0"/>
         <xs:element name="dateline"  type="Tdateline"
               minOccurs="0"/>
         <xs:element name="genre" type="xs:string" minOccurs="0"/>
         <xs:element name="keywords"  type="Tkeywords"
               minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tauthors">
      <xs:sequence>
         <xs:element name="author" type="Tauthor" maxOccurs="48"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tauthor">
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="contact" type="Tcontact"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tcontact">
      <xs:sequence>
         <xs:element name="email" type="xs:string" minOccurs="0"/>
         <xs:element name="phone" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tdateline">
      <xs:sequence>
         <xs:element name="city" type="xs:string"/>
         <xs:element name="country" type="xs:string"/>
         <xs:element name="date" type="xs:date"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tkeywords">
      <xs:sequence>
         <xs:element name="keyword" type="xs:string"
```

```
            maxOccurs="19"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tbody">
      <xs:sequence>
         <xs:element name="abstract" type="Tabstract"
               minOccurs="0"/>
         <xs:element name="section" type="Tsection" maxOccurs="15"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tabstract">
      <xs:sequence>
         <xs:element name="p" type="xs:string" maxOccurs="7"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tsection">
      <xs:sequence>
         <xs:element name="p" type="xs:string" minOccurs="0"
               maxOccurs="29"/>
         <xs:element name="subsec" type="Tsubsec" minOccurs="0"
               maxOccurs="23"/>
      </xs:sequence>
      <xs:attribute name="heading" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tsubsec">
      <xs:sequence>
         <xs:element name="p" type="xs:string" minOccurs="0"
               maxOccurs="46"/>
         <xs:element name="subsec" type="Tsubsec2" minOccurs="0"
               maxOccurs="31"/>
      </xs:sequence>
      <xs:attribute name="heading" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tsubsec2" >
      <xs:sequence>
         <xs:element name="p" type="xs:string" minOccurs="0"
               maxOccurs="11"/>
         <xs:element name="subsec" type="Tsubsec3" minOccurs="0"
               maxOccurs="8"/>
      </xs:sequence>
      <xs:attribute name="heading" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tsubsec3">
      <xs:sequence>
         <xs:element name="p" type="xs:string" maxOccurs="12"/>
      </xs:sequence>
      <xs:attribute name="heading" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tepilog">
      <xs:sequence>
         <xs:element name="acknowledgements"
               type="Tacknowledgements" minOccurs="0"/>
         <xs:element name="references" type="Treferences"
               minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tacknowledgements">
      <xs:sequence>
         <xs:element name="pa" type="xs:string" maxOccurs="3"/>
      </xs:sequence>
   </xs:complexType>
```

```
   <xs:complexType name="Treferences">
      <xs:sequence>
         <xs:element name="a_id" type="xs:string"
               maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Anexo B. Comparação entre as consultas em linguagem XQuery e Prolog

Consulta 1: Retornar o título do artigo que possua atributo id com valor "1".

```
for $art in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="1"]
return
  $art/prolog/title
```

```
findall(C, (article(A), id(A,'1'), prolog(A,B), title(B,C)), TITLE).
```

Consulta 2: Achar o título do artigo cujo nome do autor seja "Jocob Goss".

```
for $prolog in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog
where
  $prolog/authors/author/name="Jocob Goss"
return
  $prolog/title
```

```
findall(E, (article(A), prolog(A, B), authors(B, C), author(C, D),
  nome(D, 'jocob goss'), title(B, E)), TITLE).
```

Consulta 3: Agrupar os artigos por data e calcular o número de artigos em cada grupo.

```
for $a in distinct-values((
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
```

```
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog/dateline/date)
let $b := (doc("article1.xml"), doc("article2.xml"),
doc("article3.xml"), doc("article4.xml"), doc("article5.xml"),
doc("article6.xml"), doc("article7.xml"), doc("article8.xml"),
doc("article9.xml"), doc("article10.xml"), doc("article11.xml"),
doc("article12.xml"), doc("article13.xml"), doc("article14.xml"),
doc("article15.xml"), doc("article16.xml"), doc("article17.xml"),
doc("article18.xml"), doc("article19.xml"), doc("article20.xml"),
doc("article21.xml"), doc("article22.xml"), doc("article23.xml"),
doc("article24.xml"), doc("article25.xml"), doc("article26.xml"))
/article/prolog/dateline[date=$a]
return
  <Output>
    <Date>{$a}</Date>
    <NumberOfArticles>{count($b)}</NumberOfArticles>
  </Output>
```

```
countDate(DATE, L) :- bagof(ID, (article(A), prolog(A, B),
  dateline(B, ID), date(ID, DATE)), IDS), length(IDS, L).

findall(output(date(X), numberOfArticles(Y)), countDate(X, Y), L).
```

Consulta 4: Achar o título da seção seguinte à seção entitulada "*Introduction*" no artigo cujo atributo id possua valor "8".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="8"]/body/section[@heading="a"],
$p in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="8"]/body/section[. >> $a][1]
return
  <HeadingOfSection>
    {$p/@heading}
  </HeadingOfSection>
```

```
nextSection([X, Y|Z], X, Y).
nextSection([A|B], X, Y) :- nextSection(B, X, Y).

findall(D, (article(A), id(A, '8'), body(A, B), section(B, C),
  heading(C, D)), E), nextSection(E, 'a', Y).
```

Consulta 5: Retornar o título da primeira seção do artigo com atributo id igual a "9".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="9"]
return
  <HeadingOfSection>
    {$a/body/section[1]/@heading}
  </HeadingOfSection>
```

```
article(A), id(A, '9'), body(A, B), section(B, C), heading(C, X), !.
```

Consulta 6: Achar os títulos dos artigos onde as palavras "*the*" e "*hockey*" são mencionadas no mesmo parágrafo do "abstract".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article
where some $b in $a/body/abstract/p satisfies
  (contains($b, "the") and contains($b, "hockey"))
return
  $a/prolog/title
```

```
/*Consulta tuProlog*/

contains(A, B) :- atom_chars(A, AA), atom_chars(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

findall(F, (article(A), body(A, B), abstract(B, C), p(C, D),
  contains(D, 'the'), contains(D, 'hockey'), prolog(A, E),
  title(E, F)), TITLE).
```

```
/*Consulta SWI-Prolog e YAP*/

contains(A, B) :- name(A, AA), name(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
```

```
  contain(L1, [], R, [], R).

findall(F, (article(A), body(A, B), abstract(B, C), p(C, D),
  contains(D, 'the'), contains(D, 'hockey'), prolog(A, E),
  title(E, F)), TITLE).
```

Consulta 7: Achar os títulos dos artigos onde a palavra "*hockey*" é mencionada em todos os parágrafos do *abstract*.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article
where every $b in $a/body/abstract/p satisfies
  contains($b, "hockey")
return
  $a/prolog/title
```

```
/*Consulta tuProlog*/

listaContem([], X) :- fail.
listaContem([X|[]], Z) :- contains(X, Z).
listaContem([X|Y], Z) :- contains(X, Z), listaContem(Y, Z), Y\=[].
contains(A, B) :- atom_chars(A, AA), atom_chars(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

findall(F, (article(A), findall(D, (body(A, B), abstract(B, C),
  p(C, D)), L), listaContem(L, 'hockey'), prolog(A, E),
  title(E, F)), TITLE).
```

```
/*Consulta SWI-Prolog e YAP*/

listaContem([], X) :- fail.
listaContem([X|[]], Z) :- contains(X, Z).
listaContem([X|Y], Z) :- contains(X, Z), listaContem(Y, Z), Y\=[].
contains(A, B) :- name(A, AA), name(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

findall(F, (article(A), findall(D, (body(A, B), abstract(B, C),
  p(C, D)), L), listaContem(L, 'hockey'), prolog(A, E),
  title(E, F)), TITLE).
```

Consulta 8: Retornar os nomes de todos os autores do artigo cujo atributo id possua valor "2" (levando em consideração que não se sabe o nome de um elemento no caminho).

```
for $art in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="2"]
return
  $art/prolog/*/author/name
```

```
findall(X, (article(A), id(A, '2'), prolog(A, B), authors(B, C),
  author(C, D), nome(D, X)), NAME).
```

Consulta 9: Retornar todos os nomes dos autores do artigo cujo atributo id possua valor "3" (levando em consideração que não se sabe o nome de vários elementos consecutivos).

```
for $art in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="3"]
return
  $art//author/name
```

```
findall(X, (article(A), id(A, '3'), prolog(A, B), authors(B, C),
  author(C, D), nome(D, X)), NAME).
```

Consulta 10: Listar os títulos dos artigos ordenados por país.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog
order by $a/dateline/country
return
  <Output>
    {$a/title}
    {$a/dateline/country}
```

```
  </Output>
```

```
orderCountry(R) :- findall((PAIS, TITLE), (article(A), prolog(A, B),
  dateline(B, G), title(B, TITLE), country(G, PAIS)), L),
  sort(L, R).

OrderCountry(X).
```

Consulta 11: Listar os títulos dos artigos cujo elemento "*country*" possua o valor igual a "Kenya", e ordenar por data.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog
where $a/dateline/country="Kenya"
order by $a/dateline/date
return
  <Output>
    {$a/title}
    {$a/dateline/date}
  </Output>
```

```
orderTitlebyDate(R, PAIS) :- findall((DATE, TITLE), (article(A),
  prolog(A, B), dateline(B, G), title(B, TITLE), date(G, DATE),
  country(G, PAIS)), L), sort(L, R).

orderTitlebyDate(X, 'kenya').
```

Consulta 12: Recuperar o "*body*" do artigo cujo atributo id possua valor "4".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="4"]
return
  <Article>
    {$a/body}
  </Article>
```

```
dumpBody(ID, [ID, DUMP_ABST, DUMP_SECT]) :-
  body(ID, ABSTRACT, SECTION), dumpAbstract(ABSTRACT, DUMP_ABST),
  dumpSection(SECTION, DUMP_SECT).
dumpAbstract(ID, [ID, P]) :- p(ID, P).
dumpSection(ID, [ID, P, DUMP_SUBSEC]) :- section(ID, P, SUBSEC),
  dumpSubsec(SUBSEC, DUMP_SUBSEC).
```

```
dumpSubsec(ID, [ID, P, DUMP_SUBSEC]) :- subsec(ID, P, SUBSEC),
  dumpSubsec(SUBSEC, DUMP_SUBSEC).
dumpSubsec(ID, [ID, DUMP]) :- p(ID, P).

findall((B, C), (article(A), id(A, '4'), body(A, B),
  dumpBody(B, C)), BODY).
```

Consulta 13: Construir um resumo do artigo cujo atributo id seja "5", incluindo o título, o nome do primeiro autor, a data e o *abstract*.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="5"]
return
  <Output>
    {$a/prolog/title}
    {$a/prolog/authors/author[1]/name}
    {$a/prolog/dateline/date}
    {$a/body/abstract}
  </Output>
```

```
article(A), id(A, '5'), prolog(A, B), title(B, C), authors(B, D),
  author(D, E), nome(E, F), !, dateline(B, G), date(G, H),
  body(A, I), abstract(I, J), p(J, K).
```

Consulta 14: Listar o título do artigo que não possui o elemento "*genre*".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog
where empty ($a/genre)
return
  <NoGenre>
    {$a/title}
  </NoGenre>
```

```
findall(C, (article(A), prolog(A, B), title(B, C),
  not(genre(B, X))), TITLE).
```

Consulta 15: Listar o nome dos autores cujo elemento "*contact*" seja vazio.

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article/prolog/authors/author
where empty($a/contact/text())
return
  <NoContact>
    {$a/name}
  </NoContact>
```

```
findall(X, (article(A), prolog(A, B), authors(B, C), author(C, D),
  nome(D, X), contact(D, '')), NAME).
```

Consulta 16: Retornar o artigo cujo atributo id possua valor "6".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="6"]
return
  $a
```

```
dumpArticle(ID, [ID, DUMP_PROLOG, DUMP_BODY, DUMP_EPILOG]) :-
  article(ID, PROLOG, BODY, EPILOG),
  dumpProlog(PROLOG, DUMP_PROLOG), dumpBody(BODY, DUMP_BODY),
  dumpEpilog(EPILOG, DUMP_EPILOG).
dumpProlog(ID, [ID, TITLE, DUMP_AUTHORS, DUMP_DATELINE,
  DUMP_KEYWORDS]) :- prolog(ID, TITLE, AUTHORS, DATELINE, KEYWORDS),
  findall(X, dumpAuthors(AUTHORS, X), DUMP_AUTHORS),
  dumpDateline(DATELINE, DUMP_DATELINE),
  dumpKeywords(KEYWORDS, DUMP_KEYWORDS).
dumpAuthors(ID, [ID, DUMP_AUTHOR]) :- authors(ID, AUTHOR),
  dumpAuthor(AUTHOR, DUMP_AUTHOR).
dumpAuthor(ID, [ID, NOME, DUMP_CONTACT]) :-
  author(ID, NOME, CONTACT), dumpContact(CONTACT, DUMP_CONTACT).
dumpContact(ID, [ID, EMAIL, PHONE]) :-
  findall(X, email(ID, X), EMAIL), findall(Y, phone(ID, Y), PHONE).
dumpDateline(ID, [ID, CITY, COUNTRY, DATE]) :-
  findall(X, city(ID, X), CITY),
  findall(Y, country(ID, Y), COUNTRY),
  findall(Z, date(ID, Z), DATE).
dumpKeywords(ID, [ID, KEYWORD]) :-
  findall(X, keyword(ID, X), KEYWORD).
dumpBody(ID, [ID, DUMP_ABST, DUMP_SECT]) :- abstract(ID, ABSTRACT),
```

```
  dumpAbstract(ABSTRACT, DUMP_ABST),
  findall(X, (section(ID, SECTION),
  dumpSection(SECTION, X)), DUMP_SECT).
dumpAbstract(ID, [ID, P]) :- findall(X, p(ID, X), P).
dumpSection(ID, [ID, P, DUMP_SUBSEC]) :- subsec(ID, SUBSEC),
  findall(X, dumpSubsec(SUBSEC, X), DUMP_SUBSEC).
dumpSubsec(ID, [ID, P, DUMP_SUBSEC]) :- subsec(ID, SUBSEC),
  findall(X, dumpSubsec(SUBSEC, X), DUMP_SUBSEC).
dumpSubsec(ID, [ID, P]) :- findall(X, p(ID, X), P).
dumpEpilog(ID, [ID, DUMP_REF]) :- epilog(ID, REF),
  dumpReferences(REF, DUMP_REF).
dumpReferences(ID, [ID, DUMP]) :- findall(X, a_id(ID, X), DUMP).

id(A, '6'), article(A, PROLOG, BODY, EPILOG),
  findall(DUMP_PROLOG, dumpProlog(PROLOG, DUMP_PROLOG), Prolog),
  findall(DUMP_BODY, dumpBody(BODY, DUMP_BODY), Body),
  findall(DUMP_EPILOG, dumpEpilog(EPILOG, DUMP_EPILOG), Epilog).
```

Consulta 17: Retornar os títulos dos artigos que contém a palavra "*hockey*".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article
where contains ($a//p, "hockey")
return
  $a/prolog/title
```

```
/*Consulta tuProlog*/

contains(A, B) :- atom_chars(A, AA), atom_chars(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

(findall(D1, (article(A1), body(A1, B1), prolog(A1, C1),
  title(C1, D1), abstract(B1, E1), p(E1, F1),
  contains(F1, 'hockey')), TITLE1),
  findall(D2, (article(A2), body(A2, B2), prolog(A2, C2),
  title(C2, D2), section(B2, E2), p(E2, F2),
  contains(F2, 'hockey')), TITLE2),
  findall(D3, (article(A3), body(A3, B3), prolog(A3, C3),
  title(C3, D3), section(B3, E3), subsec(E3, F3), p(F3, G3),
  contains(G3, 'hockey')), TITLE3),
  findall(D4, (article(A4), body(A4, B4), prolog(A4, C4),
  title(C4, D4), section(B4, E4), subsec(E4, F4), subsec(F4, G4),
  p(G4, H4), contains(H4, 'hockey')), TITLE4),
  findall(D5, (article(A5), body(A5, B5), prolog(A5, C5),
  title(C5, D5), section(B5, E5), subsec(E5, F5), subsec(F5, G5),
  subsec(G5, H5), p(H5, I5), contains(I5, 'hockey')), TITLE5)).
```

```
/*Consulta SWI-Prolog e YAP*/

contains(A, B) :- name(A, AA), name(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

(findall(D1, (article(A1), body(A1, B1), prolog(A1, C1),
  title(C1, D1), abstract(B1, E1), p(E1, F1),
  contains(F1, 'hockey')), TITLE1),
  findall(D2, (article(A2), body(A2, B2), prolog(A2, C2),
  title(C2, D2), section(B2, E2), p(E2, F2),
  contains(F2, 'hockey')), TITLE2),
  findall(D3, (article(A3), body(A3, B3), prolog(A3, C3),
  title(C3, D3), section(B3, E3), subsec(E3, F3), p(F3, G3),
  contains(G3, 'hockey')), TITLE3),
  findall(D4, (article(A4), body(A4, B4), prolog(A4, C4),
  title(C4, D4), section(B4, E4), subsec(E4, F4), subsec(F4, G4),
  p(G4, H4), contains(H4, 'hockey')), TITLE4),
  findall(D5, (article(A5), body(A5, B5), prolog(A5, C5),
  title(C5, D5), section(B5, E5), subsec(E5, F5), subsec(F5, G5),
  subsec(G5, H5), p(H5, I5), contains(I5, 'hockey')), TITLE5)).
```

Consulta 18: Listar os títulos e o *abstract* dos artigos que contém a frase "*the hockey*".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article
where contains ($a//p, "the hockey")
return
  <Output>
    {$a/prolog/title}
    {$a/body/abstract}
  </Output>
```

```
/*Consulta tuProlog*/

contains(A, B) :- atom_chars(A, AA), atom_chars(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

(findall((D1, F1), (article(A1), body(A1, B1), prolog(A1, C1),
  title(C1, D1), abstract(B1, E1), p(E1, F1),
  contains(F1, 'the hockey')), TITLE_ABSTRACT1),
  findall((D2, H2), (article(A2), body(A2, B2), prolog(A2, C2),
```

```
  title(C2, D2), section(B2, E2), p(E2, F2),
  contains(F2, 'the hockey'), abstract(B2, G2),
  p(G2, H2)), TITLE_ABSTRACT2),
  findall((D3, I3), (article(A3), body(A3, B3), prolog(A3, C3),
  title(C3, D3), section(B3, E3), subsec(E3, F3),
  p(F3, G3), contains(G3, 'the hockey'), abstract(B1, H1),
  p(H1, I1)), TITLE_ABSTRACT3),
  findall((D4, J4), (article(A4), body(A4, B4), prolog(A4, C4),
  title(C4, D4), section(B4, E4), subsec(E4, F4), subsec(F4, G4),
  p(G4, H4), contains(H4, 'the hockey'), abstract(B1, I1),
  p(I1, J1)), TITLE_ABSTRACT4),
  findall((I5, K5), (article(A5), body(A5, B5), prolog(A5, C5),
  title(C5, D5), section(B5, E5), subsec(E5, F5), subsec(F5, G5),
  subsec(G5, H5), p(H5, I5), contains(I5, 'the hockey'),
  abstract(B1, J1), p(J1, K1)), TITLE_ABSTRACT5)).
```

```
/*Consulta SWI-Prolog e YAP*/

contains(A, B) :- name(A, AA), name(B, BB),
  contain(AA, [], BB,[], BB).
contain(_, X, [], X, R) :- not(X=[]).
contain([X|L1], Y, [X|L2], Z, R) :-
  contain(L1, [X|Y], L2, [X|Z], R).
contain([W|L1], X, [Y|L2], Z, R) :- W \= Y,
  contain(L1, [], R, [], R).

(findall((D1, F1), (article(A1), body(A1, B1), prolog(A1, C1),
  title(C1, D1), abstract(B1, E1), p(E1, F1),
  contains(F1, 'the hockey')), TITLE_ABSTRACT1),
  findall((D2, H2), (article(A2), body(A2, B2), prolog(A2, C2),
  title(C2, D2), section(B2, E2), p(E2, F2),
  contains(F2, 'the hockey'), abstract(B2, G2),
  p(G2, H2)), TITLE_ABSTRACT2),
  findall((D3, I3), (article(A3), body(A3, B3), prolog(A3, C3),
  title(C3, D3), section(B3, E3), subsec(E3, F3),
  p(F3, G3), contains(G3, 'the hockey'), abstract(B1, H1),
  p(H1, I1)), TITLE_ABSTRACT3),
  findall((D4, J4), (article(A4), body(A4, B4), prolog(A4, C4),
  title(C4, D4), section(B4, E4), subsec(E4, F4), subsec(F4, G4),
  p(G4, H4), contains(H4, 'the hockey'), abstract(B1, I1),
  p(I1, J1)), TITLE_ABSTRACT4),
  findall((I5, K5), (article(A5), body(A5, B5), prolog(A5, C5),
  title(C5, D5), section(B5, E5), subsec(E5, F5), subsec(F5, G5),
  subsec(G5, H5), p(H5, I5), contains(I5, 'the hockey'),
  abstract(B1, J1), p(J1, K1)), TITLE_ABSTRACT5)).
```

Consulta 19: Listar os nomes dos artigos citados pelo artigo cujo atributo id possua valor igual a "7".

```
for $a in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id='7']/epilog/references/a_id,
```

```
  $b in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article
where $a = $b/@id
return
  <Output>
    {$b/prolog/title}
  </Output>
```

```
findall(G, (article(A), id(A, '7'), epilog(A, B), references(B, C),
  a_id(C, D), id(E, D), prolog(E, F), title(F, G)), TITLE).
```

Consulta Extra: Listar os ids dos artigos que estejam vinculados através de referências, com o artigo cujo id seja igual a "15".

```
declare function local:vinculoArtigo($id as xs:string+) {
  for $ref in (
    doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
    doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
    doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
    doc("article10.xml"), doc("article11.xml"),
    doc("article12.xml"), doc("article13.xml"),
    doc("article14.xml"), doc("article15.xml"),
    doc("article16.xml"), doc("article17.xml"),
    doc("article18.xml"), doc("article19.xml"),
    doc("article20.xml"), doc("article21.xml"),
    doc("article22.xml"), doc("article23.xml"),
    doc("article24.xml"), doc("article25.xml"),
    doc("article26.xml"))/article[@id=$id]/epilog/references
  return
    <vinculo>
      <refs_diretas>{$ref/a_id}</refs_diretas>
      <refs_indiretas>
        {local:vinculoArtigo($ref/a_id/text())}
      </refs_indiretas>
    </vinculo>
};

for $r in (
doc("article1.xml"), doc("article2.xml"), doc("article3.xml"),
doc("article4.xml"), doc("article5.xml"), doc("article6.xml"),
doc("article7.xml"), doc("article8.xml"), doc("article9.xml"),
doc("article10.xml"), doc("article11.xml"), doc("article12.xml"),
doc("article13.xml"), doc("article14.xml"), doc("article15.xml"),
doc("article16.xml"), doc("article17.xml"), doc("article18.xml"),
doc("article19.xml"), doc("article20.xml"), doc("article21.xml"),
doc("article22.xml"), doc("article23.xml"), doc("article24.xml"),
doc("article25.xml"), doc("article26.xml"))
/article[@id="15"]/epilog/references
return
  <resposta>
    <refs_diretas>{$r/a_id}</refs_diretas>
```

```
    <refs_indiretas>
      {local:vinculoArtigo($r/a_id/text())}</refs_indiretas>
 </resposta>
```

```
referencia(A1, A2) :- article(A), id(A, A1), epilog(A, C),
  references(C, D), a_id(D, X), a_id(D, A2).
vinculoArtigo(A1, A2) :- referencia(A1, A2).
vinculoArtigo(A1, A2) :- referencia(A1, A3), vinculoArtigo(A3, A2).

setof(Y, vinculoArtigo('15', Y), L).
```

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CCMN - Bloco C - Cidade Universitária - Ilha do Fundão
Rio de Janeiro - RJ CEP: 21941-916
www.ppgi.ufrj.br