Capítulo 5

Desmistificando XML: da Pesquisa à Prática Industrial

Mirella M. Moro, Vanessa Braganholo

Abstract

XML is a language for specifying semi or completely structured data. It has been widely explored by both research and industry communities. Eleven years after its proposal, XML has solved some really important problems and it did so beautifully. Moreover, XML is adopted as a standard language by many industries and research communities for exchanging data, varying from retail to healthcare, and including current applications such as Web Science and M-Government. This chapter is intended to people who would like to do research on XML or simply to study XML from the research or the industry point-of-view. It summarizes the main current uses of XML, the research problems it has helped to solve, and the issues that are still open. Note that this chapter is not intended to cover the whole vast literature on XML. It intends to be a starting point for any person who decided to know this wonderful, versatile, powerful language called XML.

Resumo

XML é uma linguagem para especificação de dados semi- ou complemente estruturados. Ela tem sido explorada tanto pela indústria quanto pela comunidade acadêmica. Onze anos depois de ter surgido, XML resolveu vários problemas importantes, de forma muito elegante. Além disso, XML foi adotada como recomendação para troca de dados por indústrias e comunidades científicas de muitas áreas, variando desde a indústria de vendas até a de saúde, incluindo aplicações extremamente atuais como Web Science e M-Government. Este capítulo é direcionado a pessoas que querem trabalhar com pesquisa em XML, ou que simplesmente querem estudar XML com foco industrial ou acadêmico. Ele resume os principais usos de XML nos dias atuais, os problemas de pesquisa que XML ajudou a resolver e os que ainda estão abertos. É importante ressaltar que este capítulo não pretende cobrir toda a vasta literatura sobre XML. Ao invés disso, ele pretende ser um ponto de partida para qualquer pessoa que decida conhecer esta linguagem versátil, extremamente útil e poderosa chamada XML.

5.1. Introdução

XML (*Extensible Markup Language*) é uma recomendação para publicação, combinação e intercâmbio de documentos multimídia, desenvolvido pelo consórcio W3C (*World Wide Web Consortium*). Assim como HTML, XML é uma linguagem de marcação, o que significa que os dados são envoltos por marcas

(*tags*). A diferença fundamental entre XML e HTML é que, em HTML, o conjunto de marcas que se pode usar é fixo (*table*, *body*, *b*, *p*, entre vários outros). Já em XML, pode-se usar quaisquer marcas que se queira. Isso fornece ao desenvolvedor uma enorme flexibilidade de representação, o que contribuiu em muito para o sucesso de XML.

A flexibilidade da especificação dos dados permite que XML seja utilizada no desenvolvimento de aplicações em diversos contextos. Primeiro, XML é consistentemente utilizada em aplicações de serviços Web (*Web services*) como o formato padrão de *feeds* RSS. Uma estrutura de dados comum em XML pode melhorar a integração entre participantes de transações de comércio eletrônico, permitindo a troca de mensagens sobre compra e venda de produtos eficientemente. Segundo, o encapsulamento de dados XML é propício para o processamento de características avançadas (tais como privacidade) quando se acessa dados através da Web ou via aplicações desktop. Terceiro, XML também contribui para estabelecer padrões para formatos de arquivos, facilitando a troca de dados entre múltiplas aplicações em diferentes plataformas. Por exemplo, padrões industriais especificados em XML incluem: ACORD[1] (seguros), FpML[2] (aplicações financeiras) e HL7[3] (saúde). Finalmente, métodos de busca eficientes em documentos podem ser diretamente aplicados em dados estruturados com XML.

Desde seu surgimento, XML vem sendo explorada pelas comunidades acadêmica e industrial. Hoje, 11 anos depois, XML ajudou a resolver, de forma muito simples, inúmeros problemas que se encontravam em aberto há bastante tempo. Além disso, XML vem sendo usada cada vez mais pela indústria nos mais diversos segmentos. Por esses e outros motivos, XML é a tecnologia ubíqua de maior sucesso para Web, juntamente com as tecnologias básicas URI, HTTP e HTML [Wilde and Glushko 2008].

Nesse contexto, o objetivo deste capítulo é proporcionar aos alunos de graduação e pós, bem como profissionais, em Computação e Informática um panorama geral da situação de XML nos dias atuais. O momento para esse panorama não poderia ser mais propício, visto a explosão da utilização de XML em trabalhos de pesquisa para a Web (e.g.: *Web Sciences* [Getov 2008]) bem como em tecnologia móvel para acesso universal à informação (e.g.: *M-Government*[4]).

Este capítulo é dividido em três partes principais. A primeira parte, *XML Básico*, apresenta a linguagem XML e as principais linguagens relacionadas. Esse não é o foco principal deste capítulo, mas é necessário para nivelar o conhecimento dos leitores, já que a grande maioria das universidades não in-

[1] `http://www.acord.org`

[2] `http://www.fpml.org`

[3] `http://www.hl7.org`

[4] `http://www.mgovernment.org/`

clui XML entre o conteúdo abordado em suas disciplinas. A segunda parte, *Pesquisa*, mostra um panorama geral das pesquisas em XML, focando nos problemas resolvidos e nos que ainda se encontram em aberto. Finalmente, a terceira parte, *Indústria*, discute como XML vem sendo utilizada na prática nas mais diversas indústrias. Os tópicos abordados neste capítulo mostram a versatilidade da linguagem XML e discutem aspectos que vão além dos temas abordados em (alguns poucos) cursos de graduação em Computação e Informática.

5.2. XML Básico

Antes de falar de XML propriamente dita, é necessário fornecer algumas explicações sobre o órgão W3C.

W3C. Todas as regras relativas à sintaxe dos documentos XML são especificadas por documentos elaborados pelo *World Wide Web Consortium* (W3C)[5], órgão responsável por várias iniciativas ligadas à Web. O W3C é também responsável pela especificação de outras iniciativas relacionadas à XML, como linguagens de consulta (XPath e XQuery), de transformação (XSLT), API para manipulação (DOM), entre outros. O site do W3C reúne todas as especificações dessas linguagens e iniciativas (além de vários outros recursos relacionados) e usa uma terminologia para classificar o nível de maturidade de cada uma delas. São três os níveis possíveis: “recomendação” (*recommendation*), “candidato a recomendação” (*candidate recommendation*), e “rascunho” (*working draft*). O processo de recomendação é bem definido [Jacobs 2005]. Resumidamente, ele começa quando surge interesse da comunidade em algum tópico (por exemplo, Serviços Web). A partir desse interesse, um grupo de trabalho é definido, o qual fica responsável por estudar o tópico e fazer uma proposta de padronização. Essa proposta começa sob o status de "rascunho" e passa por vários ciclos de revisão. Quando atinge um certo nível de estabilidade, passa para "candidato a recomendação". Sob esse status, ainda pode sofrer modificações. Finalmente, passa para "recomendação". Nesse ponto, a proposta é considerada um padrão Web. Cabe observar que para passar para o status de recomendação, é necessário que exista ferramental de apoio e verificação desenvolvido para avaliar a especificação.

O entendimento desse processo e sua terminologia é importante para medir o risco de basear-se em propostas ainda em “rascunho”. Um exemplo é a XQuery Scripting Extension [Chamberlin et al. 2008a], que pretende transformar a XQuery em uma linguagem de *scripts*. A versão atual dessa proposta foi publicada em maio de 2008 e ainda encontra-se em estudo (rascunho), podendo sofrer diversas modificações.

Com esse ponto esclarecido, podemos passar agora ao entendimento da linguagem XML.

[5] `http://www.w3.org`

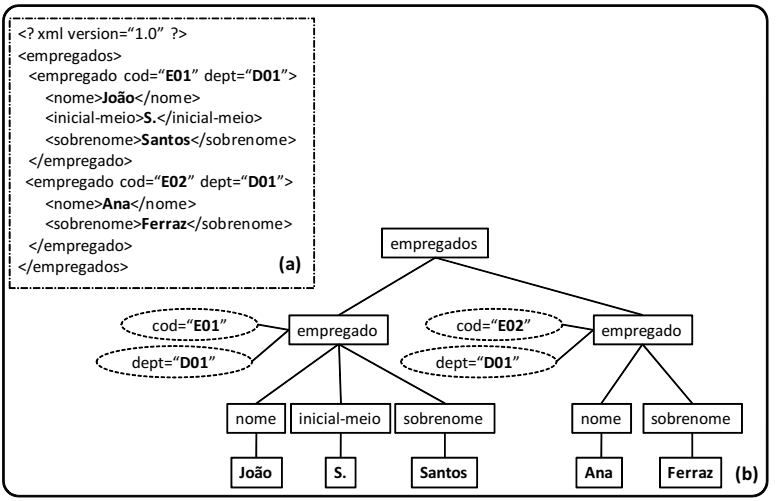

Figura 5.1. Documento XML (emps.xml) e sua representação em formato de árvore

XML. Um documento XML é formado por uma sequência de elementos que englobam valores texto e outros elementos, além de atributos. Elementos são representados com marcas (do inglês *tags*) (exemplo: `<empregado>`). Um elemento possui uma marca inicial e uma marca final, e tudo o que aparece entre essas duas marcas é o conteúdo do elemento (exemplo: `<nome>João</nome>`). Atributos aparecem dentro da marca inicial de um elemento e possuem um valor informado entre aspas (exemplo: cod="E01").

Um documento XML é tipicamente representado por uma estrutura em árvore, onde os nodos correspondem a elementos, atributos ou valores texto, e as arestas representam relações de elemento/subelemento ou elemento/valor. Um banco de dados XML é geralmente modelado como uma floresta de árvores com nodos nomeados, com uma árvore por documento. Ao contrário de bancos de dados relacionais, dados XML não precisam possuir esquema e são autodescritivos, já que possuem uma estrutura hierárquica definida por elementos e atributos. Por exemplo, a Figura 5.1 apresenta um documento XML e a sua representação no formato em árvore.

Por poder ser representado como uma árvore, um documento XML precisa respeitar algumas regras:

(i) Possuir uma única raiz. No documento da Figura 5.1, a raiz é *empregados*.

(ii) Todas as marcas são fechadas.

(iii) Os elementos são bem aninhados (as marcas fecham em ordem inversa à que foram abertas). Por exemplo, o elemento *empregado* no documento a seguir:

```
<empregado>
   <nome>João</empregado>
</nome>
```

não é bem aninhado, porque sua marca final aparece antes da marca final de seu subelemento *nome*. Todos os elementos que aparecem na Figura 5.1 são bem aninhados.

(iv) Atributos não se repetem em um mesmo elemento. Por exemplo, no elemento *empregado* da Figura 5.1 existem dois atributos: *cod* e *dept*. Não é permitido adicionar mais um atributo *cod* ou *dept* no mesmo elemento.

Documentos que seguem essas regras são considerados *documentos bem-formados*. Através dessas regras, verifica-se que XML pode ser entendida como "uma sintaxe para árvores" [Wilde and Glushko 2008]. De fato, praticamente toda a pesquisa em XML assume essa estrutura de dados como ponto de partida para estudos mais aprofundados sobre a linguagem.

É possível que o conteúdo de um elemento contenha texto e também subelementos, como no próximo exemplo. Nesse caso, o elemento *obs* é um elemento com conteúdo "misto".

```
<obs>Esta é a observação número <num>obs1</num></obs>
```

Um elemento pode ser vazio, ou seja, pode não ter conteúdo. Nesse caso, pode-se representá-lo por uma marca de abertura seguida de uma marca de fechamento (`<título></título>`), ou então usar uma abreviação, com apenas uma marca e o sinal de fechamento no final da marca (`<título/>`).

Outro ponto bastante importante diz respeito à ordem. Elementos em documentos XML são ordenados por definição. Isso significa que, se trocarmos os dois elementos *empregado* de ordem no documento da Figura 5.1 e compararmos esse novo documento com o documento original (da Figura 5.1), os dois documentos serão considerados documentos diferentes, apesar de terem o mesmo conteúdo textual. Já os atributos não são ordenados. Isso significa que se trocarmos a ordem dos atributos *cod* e *dept* no primeiro elemento empregado, e compararmos esse novo documento com o documento original, os dois documentos ainda serão considerados documentos iguais.

Namespaces. O fato de ser possível definir as marcas que farão parte de um documento XML poderia ser um impedimento para a interoperabilidade. Se um dos principais propósitos de XML é troca de informações entre aplicações, como a aplicação conhece a semântica de cada marca e consegue processar os dados adequadamente? Como um exemplo, considere um fornecedor que aceita pedidos de seus clientes no formato XML. Os documentos XML são

recebidos pelo servidor do fornecedor e processados automaticamente. Dias depois, o cliente recebe os produtos solicitados em seu endereço. Mas, se o cliente c_1 usa a marca *quant* para se referir à quantidade de um determinado produto, e o cliente c_2 usa a marca *q*, como o servidor pode saber processar documentos tão diversos? Para resolver esse problema, é possível definir um vocabulário usando uma linguagem de esquema para XML (DTD ou XML Schema). Um vocabulário define o conjunto de marcas que podem aparecer em um documento XML, e como elas podem aparecer (em que ordem, quem pode ser filho de quem, etc.). Detalhes serão vistos mais adiante. O importante é que o fornecedor pode estabelecer esse vocabulário e que os clientes c_1 e c_2 precisam usá-lo nos documentos que enviam para o fornecedor. Assim, o fornecedor sabe, por exemplo, que a marca *quantidade* determina a quantidade pedida de um determinado produto.

É bastante comum que vocabulários já definidos (em documentos XML) sejam aproveitados na definição de novos vocabulários. Nesse caso, surge um problema, pois podem haver conflitos se dois vocabulários (que serão usados juntos) possuírem marcas iguais, mas com contextos (conteúdos) diferentes. Por exemplo, suponha um vocabulário sobre matemática que define, entre outras coisas, uma marca *conjunto* e uma marca *elemento* (para representar elementos de um conjunto). Suponha também um vocabulário sobre química que define uma marca *elemento* para representar elementos químicos (por exemplo, Fe e Ca). Suponha agora que se deseja criar um vocabulário sobre os conceitos do ensino médio. Nesse vocabulário, serão utilizados os vocabulários da química e da matemática. No entanto, como distinguir um *elemento* da matemática de um da química? A solução para isso está no conceito de *namespaces*. Um namespace é associado a um vocabulário e identificado por uma URI. Por exemplo, o namespace da matemática poderia se chamar *http://www.matematica.com*, e o da química *http://www.quimica.com*. URIs são usadas por serem identificadores únicos, portanto não é necessário que o endereço realmente exista na Web. Nos documentos sobre ensino, os dois namespaces são declarados (usando *xmlns*), e pode-se associar um prefixo a cada um deles, para simplificar. Por exemplo, *m* para matemática e *q* para química. Assim, cada elemento deve ser usado junto com o prefixo, como no exemplo a seguir.

```
<ensino xmlns:m="http://www.matematica.com"
        xmlns:q="http://www.quimica.com">
  <m:conjunto>
     <m:elemento>1</m:elemento>
     <m:elemento>3</m:elemento>
  </m:conjunto>
  <q:elemento>Ca</q:elemento>
</ensino>
```

Desse modo, pode-se identificar qual dos elementos é um elemento químico, e qual é um elemento de um conjunto matemático.

O conceito de namespaces é muito importante, e faz parte do coração da recomendação XML. Esse conceito é usado em várias das recomendações relacionadas a XML que serão vistas nas próximas seções.

As subseções seguintes detalham as principais recomendações relacionadas a XML: as alternativas para representação de esquemas em XML (DTD e XML Schema); as APIs para XML (SAX e DOM); as linguagens de consulta (XPath e XQuery) e transformação de documentos XML (XSLT). É importante ressaltar que o conteúdo das subseções a seguir não pretende focar em sintaxe, e sim nos conceitos que fundamentam cada uma das recomendações apresentadas e sua expressividade.

5.2.1. Esquemas para Documentos XML

Na seção anterior mencionamos a possibilidade de se criar vocabulários específicos. Esses vocabulários são definidos em um esquema que é associado ao documento XML. Um documento XML que, além de ser bem-formado, segue as regras do esquema a que está associado, é chamado *documento válido*. Existem duas alternativas para representação de esquemas para documentos XML: DTD [Bray et al. 2006] e XML Schema [Fallside and Walmsley 2004].

DTD. A DTD (*Document Type Definition*) define regras de formação dos elementos. Por exemplo, `<!ELEMENT empregados (empregado+)>` define que existe um elemento *empregados* e que o conteúdo permitido para ele é um ou mais subelementos *empregado*. A cardinalidade (nesse caso, um ou mais) é dada pelo símbolo "+". Outros símbolos de cardinalidade possíveis são "?" (zero ou um) e "*" (zero ou mais). Sempre que um elemento não tem símbolo de cardinalidade associado, ele é obrigatório (ou seja: mínimo um, máximo um). Como exemplo, a regra `<!ELEMENT empregado (nome, inicial-meio?, sobrenome)>` define que o elemento empregado tem 3 subelementos (*nome*, *inicial-meio* e *sobrenome*), sendo que *inicial-meio* é opcional.

Além disso, a vírgula que separa os três subelementos também tem um significado importante. Ela determina a ordem em que os subelementos devem aparecer. Além da vírgula, existe também o "|" que denota uma escolha. Sempre que uma regra usa o "|", apenas um dos subelementos listados pode aparecer. Por exemplo, se a declaração de empregado fosse `<!ELEMENT empregado (nome | inicial-meio | sobrenome)>`, cada empregado poderia ter apenas um dos três subelementos.

Quando um elemento tem conteúdo textual, ele é declarado como *#PCDATA*, como no exemplo `<!ELEMENT nome (#PCDATA)>`. *#PCDATA* significa *Parsable Character Data*, o que implica que o conteúdo será analisado pelo parser (veja seção 5.2.2).

A seguir está a DTD completa que valida o documento da Figura 5.1. Note que existe também uma declaração de atributos, feita com *ATTLIST*.

```
<!ELEMENT empregados (empregado+)>
<!ELEMENT empregado (nome, inicial-meio?, sobrenome)>
<!ATTLIST empregado
            cod CDATA #REQUIRED
            dept CDATA #REQUIRED
>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT inicial-meio (#PCDATA)>
<!ELEMENT sobrenome (#PCDATA)>
```

Finalmente, para ligar um documento XML à sua DTD, adiciona-se uma declaração *DOCTYPE* antes do elemento raiz. A declaração *DOCTYPE* deve dizer quem é o elemento raiz e também o caminho para o arquivo que contém a DTD (ou o identificador para o caso de DTDs públicas). Desse modo, o documento da Figura 5.1 ficaria como segue. Mais detalhes podem ser consultados em [Bray et al. 2006].

```
<?xml version='1.0'?>
<!DOCTYPE empregados SYSTEM 'emps.dtd'>
<empregados>
   <empregado cod='E01' dept='D01'>
       <nome>João</nome>
       <inicial-meio>S.</inicial-meio>
       <sobrenome>Santos</sobrenome>
   </empregado>
  <empregado cod='E02' dept='D01'>
       <nome>Ana</nome>
       <sobrenome>Ferraz</sobrenome>
   </empregado>
</empregados>
```

Pelos exemplos anteriores, pode-se notar algumas das deficiências das DTDs. A principal delas reside na impossibilidade de especificar os tipos dos elementos atômicos, tais como inteiro, data, entre outros. O único tipo possível para um elemento atômico é *PCDATA*, ou seja, todos os elementos atômicos são tratados como *string*s. A cardinalidade também é uma limitação. Como representar uma cardinalidade mínima de 100 e máxima de 1.000 sem repetir o nome do elemento 1.000 vezes na DTD? Para suprir essas e outras limitações, a XML Schema foi proposta.

XML Schema. XML Schema, ao contrário da DTD, é baseada na própria XML. Ou seja, um esquema em XML Schema é também um documento XML. Todas as declarações de tipos e de elementos são feitas usando sintaxe XML. Cada elemento é associado a um tipo. Esse tipo pode ser simples (*simpleType*) ou complexo (*complexType*). Existem tipos simples prédefinidos (*integer*, *boolean*, *date*, entre outros), mas outros tipos podem ser criados através de restrições a tipos existentes. Por exemplo, um tipo *CEP* é uma máscara de 5 dígitos, um

traço e mais 3 dígitos. Um exemplo de declaração de elemento do tipo simples é mostrado a seguir.

```
<xs:element name='nome' type='xs:string' />
```

O elemento declarado é o elemento *nome*, e seu tipo é *xs:string*, onde o prefixo *xs* se refere ao namespace do XML Schema. Na prática, isso significa que o tipo *xs:string* é um dos tipos prédefinidos de XML Schema.

No entanto, as declarações de elemento normalmente aparecem dentro da declaração do tipo complexo que está associado ao elemento pai. Como um exemplo, considere a seguinte declaração.

```
<xs:complexType name="tEmpregado">
   <xs:sequence>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="inicial-meio" type="xs:string"
                  minOccurs="0"/>
      <xs:element name="sobrenome" type="xs:string"/>
   </xs:sequence>
   <xs:attribute name="cod" type="xs:string"/>
   <xs:attribute name="dept" type="xs:string" />
</xs:complexType>
```

Essa declaração mostra o tipo *tEmpregado*, que será associado ao elemento *empregado* mais adiante. Note que os subelementos de *empregado* estão todos declarados dentro do tipo complexo. Além disso, o operador *sequence* determina a ordem em que os subelementos devem aparecer. Ele é equivalente à vírgula na DTD. A declaração dos atributos aparece fora do operador *sequence*. Isso faz sentido, já que em XML atributos não são ordenados.

A cardinalidade em XML Schema é definida explicitamente através de *minOccurs* e *maxOccurs*. Quando omitidas, a cardinalidade mínima é 1 e a máxima é 1, de acordo com a especificação de XML Schema. O seguinte exemplo mostra a definição do tipo complexo *tEmpregados*, que será associado ao elemento *empregados*. A cardinalidade máxima de seu subelemento *empregado* é *unbounded*, o que corresponde à cardinalidade n. Além disso, a definição desse tipo complexo mostra a associação do elemento *empregado* a seu tipo *tEmpregado* (definido no exemplo anterior).

```
<xs:complexType name='tEmpregados'>
   <xs:sequence>
      <xs:element name='empregado' type='tEmpregado'
                  minOccurs='1' maxOccurs='unbounded' />
   </xs:sequence>
</xs:complexType>
```

Todas as declarações de tipos são colocadas dentro da raiz *schema*. Assim, garante-se a propriedade de raiz única. Na raiz, também é necessário fazer a declaração do namespace do XML Schema. A seguir está o esquema completo, escrito em XML Schema, para o documento da Figura 5.1.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:element name="empregados" type="tEmpregados"/>

  <xs:complexType name="tEmpregados">
    <xs:sequence>
      <xs:element name="empregado" type="tEmpregado"
            minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tEmpregado">
    <xs:sequence>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="inicial-meio" type="xs:string"
            minOccurs="0"/>
      <xs:element name="sobrenome" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="cod" type="xs:string"/>
    <xs:attribute name="dept" type="xs:string" />
  </xs:complexType>
</xs:schema>
```

Finalmente, é necessário associar o documento XML ao seu esquema. Isso é feito na raiz do documento XML, como mostrado a seguir. Nesse caso, usou-se *noNamespaceSchemaLocation* para fazer a associação já que o esquema não definiu um namespace próprio. Nos casos em que o esquema define um namespace, usa-se *schemaLocation* e associa-se o namespace à localização do esquema. Mais detalhes podem ser encontrados em [Fallside and Walmsley 2004].

```
<empregados xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="empregados.xsd">
```

Uma outra característica importante de XML Schema é a derivação de tipos. É possível criar novos tipos com base em tipos já existentes. Tipos complexos podem ser derivados por extensão ou por restrição. A derivação por extensão funciona como uma herança em Orientação a Objetos. O tipo novo herda os elementos do tipo base. Já na derivação por restrição, é possível fazer restrições de cardinalidade a um tipo existente.

Tipos simples podem ser derivados apenas por restrição. Um tipo simples é usado como base e sobre ele são aplicadas facetas ou expressões regulares, como o tipo CEP citado no início dessa seção. Também é possível definir um tipo simples como uma lista ou união de tipos existentes.

XML Schema também permite definir unicidade, chaves e referências a chaves, além da definição de tipos abstratos (que não podem ser instanciados). Uma comparação interessante de XML Schema com UML pode ser en-

contrada em [Braganholo and Heuser 2001]. Apesar desse trabalho usar a sintaxe de uma versão antiga de XML Schema, os principais conceitos usados na comparação permanecem válidos.

5.2.2. APIs para Manipulação de Documentos XML

Desenvolvedores de aplicações que manipulam documentos XML não precisam escrever código para interpretar e fazer *parser* das marcações XML. A maioria das linguagens de programação tem a capacidade de incorporar bibliotecas de software que podem fazer esse trabalho. As bibliotecas que são capazes de manipular documentos XML possuem um *processador XML*, que é responsável por disponibilizar o conteúdo do documento para a aplicação. O processador XML também é capaz de detectar problemas na marcação (documentos que não são bem-formados ou que não são válidos, que apontam para URLs de recursos que não existem, entre outros) [Bradley 2000].

Existem basicamente dois tipos de processador XML: os que fornecem à aplicação a árvore do documento XML, e os que disparam eventos para a aplicação. Em ambos os casos, a aplicação deve se comunicar com o processador através de uma API. As duas APIs principais para manipulação de documentos XML via linguagem de programação são DOM [Hors et al. 2004] e SAX [XML-DEV Community 2002]. A API DOM disponibiliza para a aplicação métodos para manipular a árvore XML em memória, e a API SAX funciona baseada em eventos. A seguir, discutimos brevemente o funcionamento de ambas.

DOM. Processadores baseados na API DOM leem o documento XML e montam na memória a estrutura em árvore correspondente. O programador pode então percorrer a árvore da forma que desejar, usando para isso os métodos fornecidos pela API (*getChildNodes()*, *getNodeValue()*, entre outros). A estrutura do documento em memória é semelhante à mostrada na Figura 5.1. Note que o conteúdo textual de um elemento é também um nodo da árvore, e seu pai é o nodo que representa o elemento ao qual ele pertence.

SAX. Já a API SAX é menos flexível, porém mais eficiente. Processadores baseados nessa API funcionam disparando eventos para a aplicação a cada vez que encontram algo significativo no documento (*startElement*, *endElement*, *characters*, e assim por diante). A Figura 5.2 mostra os eventos que são disparados quando o documento da Figura 5.1 é processado. A aplicação processa esses eventos da maneira que desejar. Para facilitar o entendimento, pode-se fazer uma analogia com eventos de interface que são disparados quando o usuário clica em um botão, por exemplo. Nesse momento, o método associado ao evento “clique de botão” é executado. Esse modo de funcionamento da API SAX faz com que a ordem em que o documento é processado seja fixa, ou seja, ele sempre é processado da raiz até o fim, e cada elemento é processado uma única vez. Não é permitido "voltar" o processamento, ao contrário de quando se usa a API DOM, quando se pode fazer o caminhamento

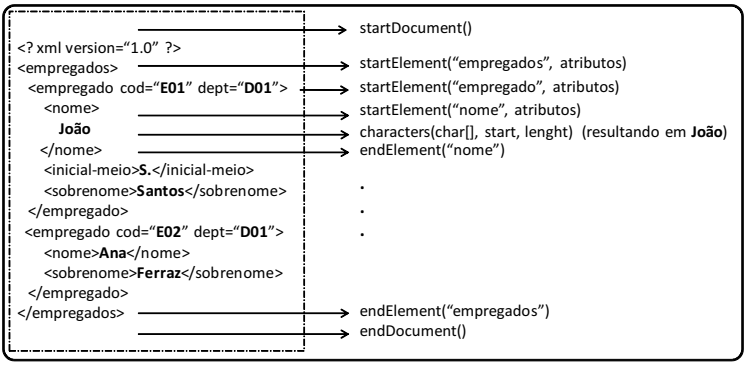

Figura 5.2. Documento XML e os eventos disparados pelo processador SAX

como e quantas vezes se queira, já que a estrutura do documento está em memória.

Discussão. Tantas diferenças geram dúvidas sobre qual das duas APIs utilizar. Existem algumas recomendações que devem ser observadas na hora de escolher a API mais adequada:

(i) Tamanho do documento: se o documento a ser processado é muito grande, o ideal é usar SAX para evitar problemas relacionados à memória. Lembre-se que com DOM o documento é colocado inteiro na memória, e dependendo do tamanho, pode faltar espaço.

(ii) Disponibilidade: O documento pode não estar disponível em sua totalidade quando do início do processamento. Isso ocorre nos casos em que ele está sendo gerado sob demanda, ou recebido via *stream* de dados. Nesses casos, usar DOM não é uma alternativa.

(iii) Desempenho: se desempenho é crucial, opte por SAX ao invés de DOM. A API SAX dispara os eventos no instante em que encontra algo significativo no documento. Isso faz com que o documento seja processado de forma bem mais rápida. Não é necessário esperar a leitura do documento todo para que o processamento a ser feito pela aplicação comece. Já em DOM, primeiro a árvore na memória tem que ser construída. Só depois é que a aplicação recebe um ponteiro para a árvore e assim pode começar a processá-la.

(iv) Facilidade: se o mais importante é facilidade, então opte por DOM. Caminhar em árvores é mais simples do que tratar eventos, principalmente porque o uso de SAX exige que a aplicação guarde informações de contexto. Por exemplo, ao tratar o evento *characters* (que é disparado

quando o parser encontra conteúdo textual no documento), a aplicação precisa saber a qual elemento esse texto pertence. Essa informação não é dada pelo parser. A aplicação precisa guardar essa informação (normalmente usando uma pilha) quando um evento *startElement* é disparado. O texto pertence ao último elemento que disparou o evento *startElement*.

Em termos práticos, várias empresas desenvolvedoras de software (IBM, Oracle, Microsoft, entre outras) disponibilizam processadores XML baseados em ambas as APIs, para diversas linguagens de programação. Como um exemplo, atualmente a linguagem Java inclui processadores para as duas APIs no pacote *javax.xml.parsers*.

5.2.3. Linguagens de Consulta para XML

XML possui duas linguagens de consulta principais recomendadas pelo W3C: a XPath [Clark and DeRose 1999] e a XQuery [Boag et al. 2007].

XPath. XPath é um subconjunto de XQuery e é baseada em expressões de caminho. Usando XPath é possível selecionar um conjunto de nodos no documento que está sendo consultado. Para isso, utiliza-se o operador "/", para dar um "passo" na árvore XML (percorrer um relacionamento pai-filho) ou "//", para dar vários "passos" de uma vez (relacionamento de ascendente-descendente). Para ilustrar, os exemplos dessa seção são consultas sobre o documento XML da Figura 5.1. A consulta `/empregados/empregado` seleciona os dois elementos *empregado* do documento. Cada "/" muda o contexto atual da consulta. O primeiro "/" coloca o contexto na raiz do documento e caminha para ela (*empregados*). O segundo caminha para os filhos *empregado* do contexto anterior (*empregados*). Do mesmo modo que na consulta anterior, a consulta `//empregado` seleciona os dois elementos *empregado* do documento. Nesse caso, a semântica da consulta é "retorne os elementos empregado que estejam em qualquer profundidade no documento XML". Além de elementos, é possível acessar atributos com a adição de "@" na frente de seu nome. Por exemplo, a expressão `//empregado/@cod` seleciona o atributo *cod* dos empregados.

Além desses, XPath fornece outros operadores. O operador "." referencia o elemento corrente. Por exemplo, a consulta `/empregados` é equivalente à consulta `/empregados/.` – ambas retornam o elemento *empregados*. O operador ".." seleciona o pai do elemento contexto atual. Por exemplo, `//empregados/..` seleciona o pai dos elementos *empregado* (no caso, *empregados*). Esses operadores são usados principalmente dentro de funções. Falaremos sobre funções em XPath mais adiante. O operador "*" é usado como um coringa e substitui o nome de um elemento em uma expressão de caminho. Por exemplo, `//empregado/*` seleciona os filhos dos elementos empregado (*nome*, *inicial-meio* e *sobrenome*).

Além dos operadores, é possível utilizar filtros nas consultas para restringir o conjunto de nodos que aparece no resultado da consulta. Um fil-

tro é uma expressão booleana colocada entre colchetes. Por exemplo, a expressão `//empregado[@cod='E01']` retorna os empregados cujo atributo *cod* é igual a E01. O contexto de um filtro é sempre o último passo do caminho percorrido antes do filtro. É importante ressaltar que um filtro não altera o contexto atual, ou seja, depois dele a expressão pode continuar do ponto onde havia parado antes do filtro. Por exemplo, a expressão `//empregado[@cod="E01"]/nome` retorna o nome do empregado cujo código é E01. Note que o contexto do passo *nome* é *empregado*, e não *@cod*. Outro ponto importante é que o conjunto de nodos a ser retornado é sempre o último passo do caminho (sem levar em consideração os filtros). Desse modo, a expressão `//empregado[@dept='D01']` retorna nodos *empregado*, e não os atributos *dept*.

Um filtro bastante utilizado é o filtro de posição. Por exemplo, a seguinte expressão seleciona o primeiro empregado: `//empregado[position()=1]`. No entanto, como essa é uma operação bastante comum, criou-se uma abreviatura para ela: `//empregado[1]`.

É possível também utilizar os operadores lógicos OR, AND e NOT dentro dos filtros. Como um exemplo, a expressão `//empregado[@dept='D01' AND nome='João']` retorna os empregados que trabalham no departamento D01 e cujo nome é João.

Funções também podem ser usadas nas consultas XPath. Grande parte delas são funções de manipulação de strings, como por exemplo *starts-with(contexto,string)*. Essa função retorna verdadeiro se o texto dos nodos *contexto* começam com a *string* especificada. Por exemplo, a expressão `//empregado[starts-with(nome,'J')]` retorna os nodos *empregado* cujo *nome* começa com 'J'. Uma lista completa das funções de XPath pode ser encontrada em [Clark and DeRose 1999].

A linguagem XPath é capaz de caminhar no documento XML selecionando conjuntos de nodos. Porém, ela foi concebida para ser utilizada em conjunto com outras linguagens (como a XQuery). Isso faz com que XPath em si seja um tanto limitada. Seu poder está no uso conjunto com outras linguagens.

XQuery. A linguagem XQuery é bastante complexa e poderosa. Ela é capaz de gerar respostas com estrutura completamente diferente da do documento consultado. Além disso, é capaz de gerar texto puro, e fragmentos de documentos XML. Expressões XPath podem ser usadas dentro de consultas XQuery. Além disso, a XQuery possui construtores de elementos, além de operadores mais complexos como o FLWOR. A seguinte consulta mostra o uso de construtores XML. A consulta é realizada sobre o documento da Figura 5.1, o qual é referenciado na consulta por *emps.xml*.

```
<emp-dept>
   {for $e in doc('emps.xml')//empregado
    return $e/nome }
</emp-dept>
```

Nessa consulta, existem dois construtores de elemento. O primeiro cria no resultado uma marca *emp-dept* que não existe no documento original. Para entender como o segundo construtor de elemento funciona (*$e/nome*), é necessário entender como a consulta é processada. Depois de colocar a marca *emp-dept* no resultado, a chave ("{") indica que o próximo trecho da consulta precisa ser processado. A expressão *for* usa a variável *$e* para iterar sobre (ou percorrer) todos os empregados do documento (um por vez). Para cada empregado, a expressão *return* é executada e *$e/nome* constrói no resultado um elemento com nome e o conteúdo do elemento *nome* do documento consultado. O resultado da consulta é apresentado a seguir.

```
<emp-dept>
   <nome>João</nome>
   <nome>Ana</nome>
</emp-dept>
```

É importante notar que cláusulas *for* podem ser mais complexas. Especificamente, elas podem ter predicados de seleção (cláusula *where*) e ordenação (cláusula *order by*). No exemplo a seguir, a cláusula *where* filtra os empregados sobre os quais a variável *$e* itera. Somente empregados que satisfazem ao critério de seleção disparam a cláusula *return*. Além disso, a cláusula *order by* faz com que os elementos *empregado*s, antes de serem iterados no *for*, sejam ordenados (no exemplo a seguir, a ordenação é feita pelo conteúdo do elemento *nome*).

```
<emp-dept>
   {
     for $e in doc('emps.xml')//empregado
     where $e/@dept='D01'
     order by $e/nome
     return
       $e/nome
   }
</emp-dept>
```

O resultado produzido pela consulta é o mostrado a seguir. Compare-o com o do exemplo anterior. A ordem do resultado foi alterada devido à presença da cláusula *order by*.

```
<emp-dept>
   <nome>Ana</nome>
   <nome>João</nome>
</emp-dept>
```

Se for necessário iterar apenas sobre valores distintos, pode-se usar a função *distinct-values*. O exemplo a seguir ilustra seu uso.

```
<departamentos>
   {for $d in distinct-values(doc('emps.xml')//empregado/@dept)
    return
    <departamento>
       <codigo>{$d}</codigo>
       <empregados>
          {for $e in doc('emps.xml')//empregado
           where $e/@dept=$d
           return
           <empregado>
              {$e/nome}
              {$e/sobrenome}
           </empregado>
          }
       </empregados>
    </departamento>
   }
</departamentos>
```

Essa consulta ilustra vários aspectos, além do uso de *distinct-values*. Primeiramente, a variável *$d* itera sobre os valores distintos de atributo *dept* existentes no documento. No caso do documento exemplo, existe apenas um valor distinto (D01). Portanto, a cláusula *return* será executada apenas uma vez. Dentro da cláusula *return*, existe uma consulta aninhada, com uma outra cláusula *for*. Essa consulta seleciona empregados relacionados ao departamento da primeira consulta (portanto, trata-se de uma consulta aninhada correlacionada). O resultado da consulta é mostrado a seguir.

```
<departamentos>
   <departamento>
      <codigo>D01</codigo>
      <empregados>
         <empregado>
            <nome>João</nome>
            <sobrenome>Santos</sobrenome>
         </empregado>
         <empregado>
            <nome>Ana</nome>
            <sobrenome>Ferraz</sobrenome>
         </empregado>
      </empregados>
   </departamento>
</departamentos>
```

Para obter o efeito de junção, os operadores *for* não precisam necessariamente ser usados em consultas aninhadas. Pode-se ligar mais de uma variável na mesma cláusula *for*. Isso produz um produto cartesiano. O exemplo a seguir faz um produto cartesiano de elementos de dois documentos. O documento *dept.xml* é mostrado na Figura 5.3.

```
<? xml version="1.0" ?>
<departamentos>
  <departamento cod="D01">
     <nome>Vendas</nome>
     <local>3º. andar</local>
  </departamento>
  <departamento cod="D02">
     <nome>Financeiro</nome>
     <local>4º. andar</local>
  </departamento>
</departamentos>
```

Figura 5.3. Documento XML de departamentos (dept.xml)

```
<resultado>
   {for $d in doc('dept.xml')//departamento),
        $e in doc('emps.xml')//empregado
    return
    <dep-emp>
       <departamento>{$d/nome/text()}</departamento>
       <empregado>{$e/nome/text()}</empregado>
    </dep-emp>
    }
</resultado>
```

O resultado da execução dessa consulta é mostrado a seguir. Note que a expressão `$d/nome/text()` retorna apenas o conteúdo textual do elemento *nome*.

```
<resultado>
    <dep-emp>
       <departamento>Vendas</departamento>
       <empregado>João</empregado>
    </dep-emp>
    <dep-emp>
       <departamento>Vendas</departamento>
       <empregado>Ana</empregado>
    </dep-emp>
    <dep-emp>
       <departamento>Financeiro</departamento>
       <empregado>João</empregado>
    </dep-emp>
    <dep-emp>
       <departamento>Financeiro</departamento>
       <empregado>Ana</empregado>
    </dep-emp>
</resultado>
```

Se adicionarmos algum critério na cláusula *where*, obtemos uma operação de junção. A seguinte consulta ilustra essa operação.

```
<resultado>
   {for $d in doc('dept.xml')//departamento),
        $e in doc('emps.xml')//empregado
    where $d/cod=$e/dept
    return
    <dep-emp>
       <departamento>{$d/nome/text()}</departamento>
       <empregado>{$e/nome/text()}</empregado>
    </dep-emp>
    }
</resultado>
```

O resultado da consulta é mostrado a seguir. Veja que agora não temos mais o produto cartesiano, e sim a junção de empregados e departamentos. O departamento de Finanças não aparece no resultado, pois não existe nenhum empregado que trabalhe nesse departamento.

```
<resultado>
    <dep-emp>
       <departamento>Vendas</departamento>
       <empregado>João</empregado>
    </dep-emp>
    <dep-emp>
       <departamento>Vendas</departamento>
       <empregado>Ana</empregado>
    </dep-emp>
</resultado>
```

A função *doc* é usada para referenciar o documento a ser consultado. Além dessa da função *doc*, existe outra função de entrada, a função *collection*. Através dela é possível consultar vários documentos XML de uma só vez. Esse recurso é bastante utilizado em Bancos de Dados XML Nativos (veja seção 5.4.2.2), onde documentos XML são armazenados em coleções. Para exemplificar, considere a consulta abaixo. A consulta assume a existência de uma coleção chamada *empregados* na qual estão armazenados um ou mais documentos XML que possuem um elemento *empregado*. A consulta retorna o subelemento *nome* de cada elemento *empregado* encontrado na coleção.

```
<emp-dept>
   {for $e in collection('empregados')//empregado
    return
    $e/nome
   }
</emp-dept>
```

XQuery também é capaz de realizar operações de agregação. No exemplo a seguir, uma função de agregação é usada para contar o número de empregados do documento.

```
<num-emp>
   {let $e := doc('emps.xml')//empregado
    return
    count($e)
   }
</num-emp>
```

Nessa consulta, ao contrário da anterior, a variável *$e* não itera sobre os empregados, se ligando a um de cada vez. Ao contrário, ela se liga ao conjunto de empregados. Isso acontece porque nessa consulta usamos o operador *let* para ligar a variável, ao invés do *for* das consultas anteriores. O resultado da consulta é o seguinte:

```
<num-emp>2</num-emp>
```

XQuery disponibiliza cinco funções de agregação: *count*, *sum*, *avg*, *max* e *min* [Bray et al. 2007]. Todas elas devem ser utilizadas com variáveis que foram ligadas através da expressão *let*.

As expressões *let* e *for* são parte das expressões FLWOR (*for*, *let*, *where*, *order by*, *return*) e podem ser usadas em conjunto. Além disso, XQuery possui expressões condicionais (*if then else*), quantificadores existencial e universal (*some* e *every*), *cast* de tipos, entre outras possibilidades.

Tudo isso aumenta o poder de expressão de XQuery, que é bastante superior ao da XPath. No entanto, nenhuma das duas permite atualizações. Existe uma proposta de extensão da XQuery para tratar atualizações, chamada *XQuery Update Facility*, que atualmente é candidata a recomendação do W3C [Chamberlin et al. 2008b]. Ela adiciona novos tipos de expressão à linguagem, que são: *insert*, *delete*, *rename*, *replace*, *rename* e *copy modify return*, expressão usada para transformar o documento.

Com essa extensão, para inserir um novo empregado no documento da figura 5.1, bastaria executar a seguinte expressão:

```
insert node
   <empregado cod='E03' dept='E02'>
      <nome>Alice</nome>
      <inicial-meio>P</inicial-meio>
      <sobrenome>Braga</sobrenome>
   </empregado>
after doc('emps.xml')//empregado[1]
```

Note que o empregado será inserido depois do primeiro empregado, ou seja, entre João e Ana. Do mesmo modo, para modificar o valor do sobrenome do segundo empregado para *Fernandes*, basta executar:

```
replace value of node doc('emps.xml')//empregado[2]/sobrenome
with 'Fernandes'
```

Apesar de ainda não ser uma recomendação, essa extensão vem suprir uma importante lacuna das linguagens de manipulação de dados para XML.

5.2.4. Transformação de Documentos XML

Uma das linguagens relacionadas a XML é a XSLT (*XSL Transformations*) [Clark 1999]. Como o próprio nome indica, XSLT faz parte da XSL (*Extensible Stylesheet Language Family*). A XSL é uma família de recomendações do W3C que define normas para fazer apresentação e transformação de documentos XML [Quin 2009]. Fazem parte dessa família três especificações:

- XPath – linguagem para acesso a partes dos documentos XML, cujos detalhes podem ser encontrados na Seção 5.2.3.
- XSLT – linguagem para transformação de documentos XML.
- XSL:FO – linguagem para apresentação de documentos XML.

A XSL:FO define regras de formatação para documentos XML. Pode-se gerar documentos PDF, por exemplo, com margens, tamanho de fonte, cor da fonte, entre várias outras opções, todas definidas através de um documento XSL:FO.

A XSLT especifica regras de transformação a serem aplicadas sobre um documento XML. Ao ser executada por um processador XSLT, a transformação é realizada e um resultado é gerado. Isso permite transformar um documento XML em qualquer outro formato baseado em texto. Desse modo, podemos transformar um documento XML em um documento HTML, em um relatório TXT, em outro documento XML com estrutura diferente, entre outros.

Por restrições de espaço, os detalhes da linguagem não são tratados aqui. Mais informações podem ser obtidas em [Clark 1999]. O principal ponto a ser entendido sobre a linguagem é sua grande importância para a XML. XSLT é uma das principais responsáveis pela interoperabilidade da recomendação XML. Na próxima seção, discutimos um cenário que evidenciará esse ponto.

5.2.5. Cenário de Uso

Esta seção exemplifica o uso das recomendações apresentadas nas seções anteriores em uma aplicação de comércio eletrônico. O objetivo desse exemplo é fornecer ao leitor um panorama real da interação entre as recomendações estudadas. O cenário é composto de um fornecedor F e dois clientes C_1 e C_2 que desejam fazer comércio eletrônico com o fornecedor F. Todo o comércio é feito através de trocas de mensagens (documentos) XML. Nesse caso, o fornecedor define o esquema XML (em XML Schema) que os clientes devem utilizar para enviar pedidos. No entanto, os clientes possuem seu formato próprio, que usam internamente em suas aplicações.

À primeira vista, esse cenário poderia sugerir retrabalho para os clientes. Ou eles deveriam reimplementar suas aplicações para que elas trabalhem internamente com o formato definido pelo fornecedor, ou, pelo menos, implementar um novo módulo (usando SAX ou DOM) para gerar pedidos no formato exigido pelo fornecedor. No entanto, nada disso é necessário. A aplicação de comércio eletrônico do cliente C_1 é capaz de gerar pedidos em um formato interno próprio, definido por uma DTD. A partir da necessidade de fazer comércio

com um novo fornecedor (no caso F), o programador (ou outra pessoa com conhecimento de XML) elabora um documento XSLT que transforma o pedido no formato interno de C_1 para um pedido que siga o XML Schema fornecido por F. O pedido então é enviado sem maiores problemas. O mesmo acontece com o cliente C_2.

O importante nesse cenário é que o cliente não precisa modificar seu esquema (e consequentemente suas aplicações) para poder fazer comércio com o fornecedor. A linguagem XSLT pode ser utilizada para transformar o pedido do cliente em um pedido que siga o esquema determinado pelo fornecedor.

Ao receber um pedido, o fornecedor F o processa através de uma aplicação que utiliza internamente a API SAX. Os dados são repassados ao setor financeiro, para realizar a cobrança, e ao setor de almoxarifado, para despacho das mercadorias.

5.3. Pesquisa

Esta seção apresenta os desafios de pesquisa que foram solucionados com XML, os desafios atuais e os novos desafios que surgem no horizonte da comunidade científica.

5.3.1. XML como Solução de Pesquisa

Nesta seção são discutidos tópicos de pesquisa que ficaram em aberto por muito tempo na comunidade de Banco de Dados e cujas soluções exploram as vantagens oferecidas pela XML: integração de dados, evolução de esquemas e consulta a dados dinâmicos.

5.3.1.1. Integração de Dados

É muito comum que organizações e empresas utilizem ambientes de trabalho distribuídos, onde seus usuários precisam trocar informação utilizando um modelo comum. XML é altamente utilizada para facilitar tal troca. A versatilidade da linguagem XML permite definir modelos genéricos para integrar diferentes fontes de dados. Por exemplo, é extremamente comum utilizar ferramentas para importar e exportar dados a partir ou para o formato XML. Porém, o grande desafio é como integrar dados de fontes diferentes.

Integração é um problema clássico na área de Banco de Dados [Wiederhold 1992]. Esse problema consiste em prover uma visão integrada de dados que se encontram em várias bases distintas (possivelmente heterogêneas), de modo que usuários possam consultá-los. A integração envolve um conjunto de operações para a manipulação de metadados, ou seja, de esquemas. Esse conjunto inclui as seguintes operações: casamento (*match*) de esquemas, mapeamento composto, diferenciação de esquemas, mapeamento invertido, junção de esquemas, tradução de esquemas e mapeamento entre linguagens de definição de esquemas.

Esta seção apresenta algumas soluções existentes na literatura que usam XML para resolver algumas dessas operações. Começamos com a solução

clássica para integração de dados que utiliza uma arquitetura de mediadores e *wrappers* [Wiederhold 1992]. Um mediador atua como uma camada global que recebe consultas e as redireciona para as bases de dados que possam respondê-las. A heterogeneidade é resolvida pelos *wrappers* acoplados a cada base de dados. Esses *wrappers* são responsáveis por "traduzir" os dados de uma fonte local para dados num formato comum ao sistema. XML serve muito bem a esse propósito e tem sido utilizada como "língua franca" em sistemas de integração de dados [Figueiredo et al. 2007].

Existem várias outras pesquisas que focam em operações específicas para a integração, como por exemplo o casamento de esquemas (*schema matching*), o qual envolve mapear um esquema para outro (*schema mapping*). Além de integração de dados, o casamento de esquemas é uma solução básica para várias aplicações de banco de dados, tais como comércio eletrônico, *data warehousing* e processamento semântico de consultas. Especificamente, o objetivo do casamento é identificar correspondências semânticas entre dois esquemas, sejam eles esquemas relacionais, XML, ontologias ou outros. Geralmente, o casamento de esquemas pode ser feito manualmente, o que exige grande esforço e tempo de um profissional especializado. Porém, algumas soluções semiautomáticas são mais recentes e robustas. Nesse caso, o casamento pode ser realizado em diferentes níveis como de esquema e de instância, de elemento e de estrutura, e os mecanismos podem ser baseados em linguagem ou em restrições. O trabalho de Rahm e Bernstein (2001) categoriza e apresenta várias soluções para o casamento.

XML surgiu como uma grande ajuda aos mecanismos de casamento. O seu alto poder de expressão e a grande versatilidade da sua linguagem para definir esquemas são suas maiores vantagens. Porém, o seu suporte para sistemas distribuídos e *namespaces* gera novos problemas, principalmente quando se considera grandes esquemas de dados. Uma solução viável é fragmentar o esquema em partes menores e reusar as partes comuns [Rahm et al. 2004].

5.3.1.2. Evolução de Esquemas

Aplicações de banco de dados estão em constante evolução, pois existe uma demanda crescente para acomodar novos requisitos. Esse caráter evolutivo das aplicações é ainda mais visível quando se trata de aplicações Web, nas quais novas funcionalidades e novos tipos de dados precisam ser gerenciados constantemente. Novos requisitos geralmente exigem alterações no esquema do banco de dados. Alterações no esquema também são necessárias para corrigir problemas (por exemplo de desempenho) ou para migrar para uma nova plataforma. Esses cenários são tão reais e comuns que evolução e versionamento de esquemas têm sido tópicos de pesquisa bastante ativos nos últimos 20 anos [Roddick 1995].

Evolução de esquemas é um processo complexo e envolve várias questões. Primeiro, qualquer alteração no esquema do dado precisa ser propagada

para as suas instâncias (considerando que o banco esteja populado), bem como visões e aplicações, e qualquer outro artefato que dependa da especificação dos dados. Tal propagação precisa ser realizada de maneira correta, garantindo a integridade dos dados, e eficiente, garantindo que o sistema não sofra qualquer prejuízo de desempenho. Por exemplo, imagine que uma nova coluna precisa ser adicionada a uma tabela relacional. É necessário decidir como essa propagação será feita para os dados armazenados (duas opções comuns são adicionar valores *NULL* para a coluna ou adicionar um valor *default*). Segundo, o processo idealmente não requer qualquer intervenção humana, devendo ser o mais automatizado possível. Terceiro, todas as operações precisam ser executadas com tempo mínimo de indisponibilidade do sistema. A versatilidade da definição de esquemas em XML foi de extrema importância na busca de soluções para esses problemas [Sedlar 2005].

É importante notar que as soluções para integração de dados podem ajudar muito na evolução de esquemas. Por exemplo, os métodos de mapeamento de esquemas podem auxiliar na geração e na adaptação dos esquemas envolvidos. Além disso, existem duas frentes diferentes de trabalho em evolução de esquemas com XML: (*i*) os trabalhos que utilizam XML para gerenciar a evolução de esquemas de dados que *não* são originalmente representados com XML; e (*ii*) os trabalhos que propõem técnicas para gerenciar a evolução de dados representados *em* XML.

XML e Evolução de Outros Dados. Nessa frente o foco está na utilização da linguagem XML para gerenciar a evolução de esquemas para quaisquer outros tipos de dados (relacionais, orientado a objetos). A dificuldade de gerenciar evolução de esquemas pode ser justificada pela falta de um mecanismo que gerencie a evolução dos próprios dados [Wang and Zaniolo 2003], que poderia ser realizada através de extensões para dados temporais. Nesse caso, o histórico dos dados relacionais pode ser visto como um documento XML. A solução proposta por Wang e Zaniolo (2003) é simples e permite gerenciar e realizar consultas sobre dados que evoluem juntamente com seus esquemas. Em resumo, dois rótulos temporais são acrescentados à cada informação relacional: *tstart* e *tend*. Esses rótulos representam o tempo de validade da informação relacional. Quando os dados relacionais são transformados em XML, esses rótulos são representados como atributos de cada elemento. Por exemplo, os dados do nome de um empregado (armazenados na coluna *nome* da tabela *Empregado*) tornam-se os seguintes elementos em XML:

```
<empregado tstart="2007-01-01" tend="2010-12-31">
  <nome tstart="2007-01-01" tend="2010-12-31">
        José Brasil
  </nome>
</empregado>
```

Considerando que esse esquema evolui para apresentar nome e sobrenome como colunas diferentes no início de 2009, a nova representação em

XML fica como segue.

```
<empregado tstart="2007-01-01" tend="2010-12-31">
  <nome tstart="2007-01-01" tend="2008-12-31">
        José Brasil</nome>
  <nome tstart="2009-01-01" tend="2010-12-31">
        José</nome>
  <sobrenome tstart="2009-01-01" tend="2010-12-31">
        Brasil</sobrenome>
</empregado>
```

Evolução de Dados XML. Nessa frente, o foco está no tratamento da evolução de esquemas para dados XML. O problema de atualizar as instâncias dos dados continua. Nesse caso, precisa-se garantir que os documentos sejam válidos de acordo com o novo esquema. O impacto das alterações na validade dos documentos é discutido em [Guerrini et al. 2005]. A natureza flexível da XML favorece as soluções para a evolução de seus esquemas. Por exemplo, para um mesmo banco de dados (representado como um documento ou conjunto de documentos XML), é possível ter mais de um esquema especificado. Além disso, a mesma solução apresentada por Wang e Zaniolo (2003) pode ser facilmente adaptada para esse caso.

Como exemplo de ferramenta para evolução de esquemas XML, nós citamos a *X-Evolution* [Guerrini and Mesiti 2008]. O seu protótipo é implementado com interface Web. As alterações no esquema podem ser definidas através de uma interface gráfica ou de uma linguagem de alteração proprietária. O protótipo ainda permite que seja realizada a revalidação do esquema, bem como a adaptação das instâncias do esquema original para o novo esquema.

Finalmente, o tutorial de Beyer et al. (2005) apresenta como a evolução de esquemas pode ser gerenciada através de um SGBD de grande porte, nesse caso o IBM DB2. Outras questões envolvidas com evolução de esquemas incluem: evolução de ontologias, workflows, modelos conceituais e interfaces de software [Rahm and Bernstein 2006].

5.3.1.3. Consulta a Dados Dinâmicos

Documentos XML Ativos (AXML) [Abiteboul et al. 2008] são documentos XML que possuem chamadas a Serviços Web embutidas. A ideia principal de documentos AXML é adicionar dinamicidade ao conteúdo Web através da XML. Por exemplo, um documento XML que contenha dados sobre cidades pode incluir chamadas a um serviço de previsão do tempo. Desse modo, cada chamada trará a previsão do tempo para uma cidade específica da seguinte forma.

```
<cidades>
   <cidade>
```

```
      <nome>Rio de Janeiro</nome>
      <estado>RJ</estado>
      <sc service="forecast@weather.com"/>
   </cidade>
</cidades>
```

A chamada de serviço encontra-se no elemento *sc*. Ao ser ativado, o serviço traz como resposta um valor ou uma subárvore XML. No caso do exemplo, a resposta seria como segue.

```
<previsao>
   <temperatura min="25" max="32"/>
   <condicao>nublado</condicao>
</previsao>
```

Essa subárvore é então adicionada ao documento XML, num processo chamado de *materialização*. Após a materialização, a chamada ao serviço pode continuar no documento para que seja acionada outras vezes, fazendo com que os dados estejam sempre atualizados. Nesse caso, o documento é definido como:

```
<cidades>
   <cidade>
      <nome>Rio de Janeiro</nome>
      <estado>RJ</estado>
      <sc service="forecast@weather.com"/>
      <previsao>
         <temperatura min="25" max="32"/>
         <condicao>nublado</condicao>
      </previsao>
   </cidade>
</cidades>
```

A ativação de uma chamada de serviço pode ocorrer em um tempo predeterminado (por exemplo, de 10 em 10 segundos), ou em tempo de consulta. Nesse último caso, cada vez que uma consulta é executada sobre o documento AXML, as chamadas de serviço devem ser ativadas e seus resultados materializados. Como uma chamada de serviço pode ser um processo demorado (o servidor pode estar sobrecarregado ou mesmo fora do ar), é fundamental que as consultas sejam otimizadas.

Um exemplo de otimização poderia ser: chame um serviço somente se a subárvore que o contém fizer parte do resultado da consulta. Essa estratégia de otimização, no entanto, não funciona em todos os casos. Por exemplo, uma consulta que peça todas as cidades cuja temperatura máxima prevista é maior do que 30 graus não se beneficia desse tipo de otimização, pois para responder à consulta é necessário chamar todos os serviços de previsão do tempo para todas as cidades do documento. O trabalho de Abiteboul et al. (2004) apresenta um algoritmo dinâmico que identifica o conjunto de serviços

que devem ser chamados para materializar a resposta de uma consulta. O algoritmo determina o conjunto de serviços que devem ser chamados e elimina serviços da lista de serviços a serem chamados com base em suas definições WSDL (o qual é um formato XML para descrever serviços Web através de operações com mensagens contendo informação orientada a documentos ou procedimentos [Booth and Liu 2007]).

Outro problema bastante interessante relacionado à otimização é a ordem em que os serviços devem ser chamados. Uma chamada de serviço pode incluir uma outra chamada embutida em seu resultado. Essa, por sua vez, precisa também ser ativada para que o resultado possa ser obtido. O trabalho de Ruberg e Mattoso (2008) investiga essa questão e define a ordem em que os serviços devem ser chamados, calculando um plano de execução (também chamado de *plano de materialização*). Isso é importante não somente porque as chamadas de serviço podem ser aninhadas, mas também porque pode existir uma restrição *followedBy* entre dois serviços, obrigando que um serviço sempre seja acionado depois que um determinado serviço foi chamado. O cálculo do plano de materialização baseia-se no grafo de dependências das chamadas de serviço, obtido através de uma análise do documento e do WSDL dos serviços [Pereira et al. 2006]. Tais otimizações possibilitam consultas eficientes a documentos AXML.

5.3.2. Pesquisa Recente com XML

Esta seção apresenta tópicos de pesquisa recentes na comunidade de XML: processamento de consultas XML, sistemas de disseminação de conteúdo, modelagem de dados híbridos XML-Relacional, fragmentação de bases de dados e integração de instâncias. Para cada tópico, são apresentados justificativa e motivação, problemas e soluções iniciais.

5.3.2.1. Processamento de Consultas XML

Com o advento de XML como base para muitas aplicações centradas em dados, questões relacionadas à consulta de dados XML tornaram-se primordiais. XML representa uma evolução considerável na maneira como dados são armazenados, gerenciados e consultados. Especificamente, o armazenamento de dados e a avaliação de consultas XML apresentam vários desafios em relação ao processamento de dados relacionais. Primeiramente, dados XML possuem estruturas mais complexas em função da organização hierárquica intrínseca dos documentos XML. Assim, os dados seguem uma organização no formato de árvore, na qual tarefas básicas (tais como indexação e consulta) que são realizadas eficientemente em estruturas planas (como listas e vetores) se tornam mais complexas e demoradas. Adicionalmente, dados XML não requerem um esquema rígido e fixo, tornando a organização dos dados ainda mais difícil. Finalmente, documentos XML são textuais e descritivos, nos quais a estrutura dos dados é definida para cada parte da informação, resultando na

repetição de marcações XML. Tal repetição contribui para aumentar o tamanho das coleções de dados quando especificadas como documentos XML.

Trabalhos sobre processamento de consulta XML evoluíram de técnicas para processar pares de elementos a técnicas para processar consultas do tipo árvore ou grafo. Não é nosso objetivo apresentar todos os inúmeros algoritmos que existem para avaliar consultas XML. Em vez disso, nós discutimos dois algoritmos clássicos para processamento de consultas no qual a maioria dos demais se baseiam ou referenciam.

O primeiro deles introduz o algoritmo para o processamento de pares ascendente//descendente, ou junção estrutural *StackTree* [Al-Khalifa et. al 2002]. Esse algoritmo é considerado o estado-da-arte para junção estrutural porque define o problema para documentos XML e introduz uma solução inicial simples e prática. Em resumo, o algoritmo recebe duas listas de elementos (uma com os ascendentes e outra com os descendentes) nas quais cada elemento é identificado por um intervalo numérico (o qual será explicado a seguir). O algoritmo percorre ambas as listas, armazenando em pilhas os elementos de mesmo caminho. Esse esquema de pilhas depois foi estendido para processar consultas mais complexas (veja a seguir na descrição do algoritmo *TwigStack*).

Nas listas de elementos que servem de entrada para o algoritmo, cada elemento possui um intervalo numérico associado. Esse intervalo numérico é calculado com uma travessia em pré-ordem da árvore, na qual o primeiro número do intervalo é definido quando o elemento é acessado pela primeira vez, e o segundo número é definido quando todos os seus descendentes foram processados.

Após o *StackTree*, outros algoritmos foram propostos com base nele, melhorando-o através da inclusão de índices e técnicas de particionamento. Porém, é importante notar que essa é a forma mais simples de estruturar uma consulta XML. Atualmente, a maioria das pesquisas sobre o tema focam em consultas mais complexas, como explicado a seguir.

O segundo algoritmo clássico para processamento de consultas XML é o *TwigStack* [Bruno et al. 2002]. Esse algoritmo é o estado-da-arte para o processamento de consultas no estilo de árvore, ou seja, além de pares ascendente//descendente e pai/filho, esse algoritmo também processa ramificações definidas por filtros. Por exemplo, a consulta $A[B]/C//D$ pode ser representada por uma pequena árvore (chamada *twig*) onde a raiz é o elemento A, o qual possui dois filhos B e C, onde C ainda possui um descendente D.

Especificamente, a Figura 5.4a mostra a estrutura de um documento XML com elementos $Raiz$, A, B e C. Os elementos e seus intervalos definem um conjunto de "listas de elementos", no qual existe uma lista para cada elemento, conforme ilustrado na Figura 5.4b. A entrada do algoritmo *TwigStack* é formada pelas listas de elementos expressos na consulta, conforme apresentada em 5.4c. Nesse caso, as listas dos elementos $Raiz$, A, B e C; caso o documento apresentasse outra lista qualquer, a mesma não seria considerada pelo algoritmo. Conforme os elementos vão sendo lidos, o algoritmo utiliza um sistema

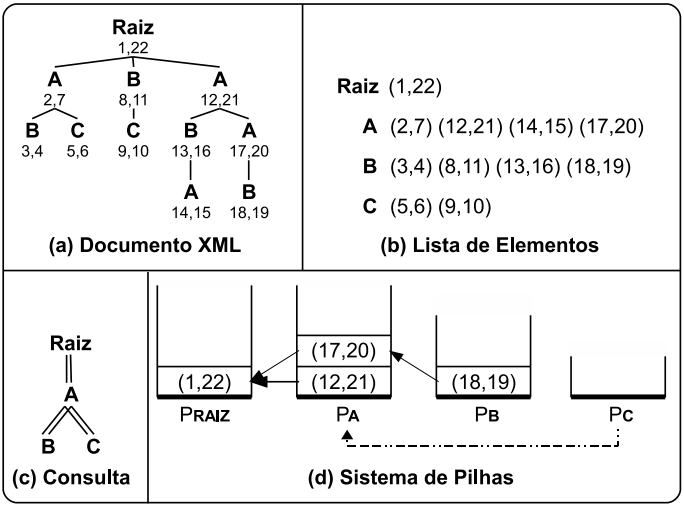

Figura 5.4. Exemplo da execução do algoritmo *TwigStack*.

de pilhas encadeadas para armazenar resultados parciais. Existe uma pilha auxiliar para cada elemento (por exemplo P_{Raiz} para o elemento $Raiz$) a fim de manter os ascendentes do caminho atual. Por exemplo, quando o algoritmo estiver processando o elemento $B(18,19)$, os elementos $A(12,21)$, $A(17,20)$ e $Raiz(1,22)$ estarão nas pilhas (5.4d).

Note que cada elemento na pilha aponta para a instância do ascendente mais próximo na pilha anterior. Desse modo, esse algoritmo garante que cada lista de elementos é lida uma única vez. Com o auxílio das pilhas, ele também garante que os resultados sejam impressos em ordem. A sua vantagem principal é que resultados intermediários (cujo número pode ser considerável) são mantidos com o mínimo de recursos possíveis. Nesse caso, o número de elementos armazenados nas pilhas é no máximo a altura da árvore do documento. A descrição completa do algoritmo está no artigo original [Bruno et al. 2002].

As pesquisas atuais em processamento de consultas consideram otimizações tanto para o tempo de consulta quanto para o espaço de armazenamento necessário. Além disso, novas semânticas de consulta também estão sendo pesquisadas, como por exemplo consultas no estilo busca por palavras e consultas sobre *streams* de dados XML. Recomendamos o artigo [Gou and Chirkova 2007] como uma apresentação completa sobre métodos de consultas XML. Além disso, a próxima subseção apresenta uma discussão sobre consultas sobre *streams* de dados XML.

5.3.2.2. Sistemas de Disseminação de Conteúdo XML

Em SGBDs XML, um usuário envia uma consulta ao sistema, que por sua vez a avalia sobre seus dados armazenados e retorna os resultados de tal avaliação ao usuário. Essa interação na qual o usuário envia sua consulta para ser executada por um servidor, que imediatamente retorna os resultados, é conhecida como *request/reply*. Com a evolução da Internet e de sistemas distribuídos, um novo paradigma agrega o conceito de disseminação de conteúdo aos sistemas de consulta [Fiege 2005]. Além de responder às consultas considerando os dados armazenados, as consultas também são armazenadas no sistema, e a avaliação continua sendo realizada à medida que novos dados são adicionados ao SGBD. Nesse caso, os resultados são disseminados aos usuários a *posteriori*. Especificamente, usuários (consumidores) cadastram seus interesses através da especificação de consultas, produtores enviam mensagens com dados, e o sistema identifica quais mensagens devem ser disseminadas para quais consumidores (baseado em seus interesses).

Existe uma evolução nos tipos de mensagens e consultas considerados nesses sistemas. Inicialmente, os sistemas eram simples, e os usuários registravam seus interesses diretamente com uma empresa. Com a popularização da Web e o advento de novos recursos para integração de dados, mensagens e consultas hoje suportam dados e linguagem XML. Pesquisas recentes sobre disseminação de conteúdo XML têm investigado problemas relacionados a diferentes partes da arquitetura do sistema. Os aspectos mais relevantes incluem: a indexação e a agregação de perfis (consultas), a codificação das mensagens na rede, a distribuição de consultas e a tarefa de filtrar mensagens [Diao et al. 2004, Li et al. 2007, Moro et al. 2007a, Vagena et al. 2007].

Especificamente, considere a tarefa de filtrar mensagens, a qual identifica quais documentos satisfazem a quais consultas. Para essa tarefa, a grande barreira para utilizar algoritmos para dados armazenados é que os dados são processados em fluxo contínuo (*streams*). Desse modo, o documento XML precisa ser processado em ordem, ou seja, começando pela raiz, passando pelos elementos e atributos seguintes conforme aparecem no documento. Entre as soluções existentes, algoritmos baseados em autômatos têm estado entre as mais populares [Diao et al. 2004, Vagena et al. 2007]. A justificativa para tal popularidade é a grande simplicidade desse mecanismo. Por exemplo, considerando as seis consultas na Figura 5.5a, o respectivo autômato *simplificado* para o processamento das mesmas é apresentado na Figura 5.5b. Nessa ilustração, os estados do autômato são representados por círculos identificados com um número, as transições são arestas contendo o seu respectivo símbolo, os estados finais possuem linha mais grossa e fundo cinza. Outras transições são omitidas para melhorar a clareza da imagem.

O objetivo da maioria dos trabalhos é escabilidade em relação ao número de perfis avaliados, o que é conseguido através de métodos de processamento de consultas múltiplas (*multi-query processing*)

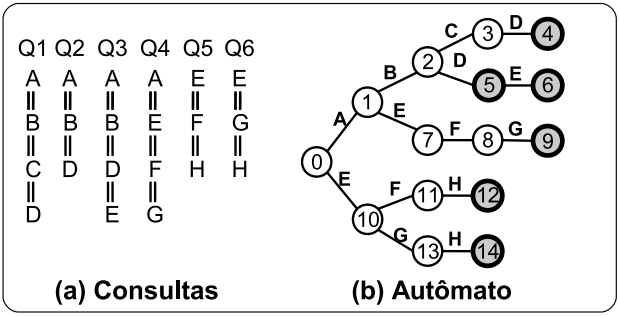

Figura 5.5. Autômato para o processamento de consultas XML

[Diao et al. 2004, Vagena et al. 2007] e poda de consultas irrelevantes (*early pruning*) [Moro et al. 2007a]. Outras soluções abordam escabilidade em relação ao número de mensagens e consideram algoritmos que operam em grupos (*batches*) [Vagena et al. 2007]. Nós referenciamos a [Moro et al. 2009b] para uma pesquisa mais atual e completa sobre esse tema.

5.3.2.3. Modelagem de Dados Híbridos XML-Relacional

Não é mais uma simples conjectura que dados XML e dados relacionais irão sempre coexistir e complementar um ao outro em gerenciamento de dados corporativos. Documentos e mensagens XML predominam em aplicações empresariais tal que formatos de dados XML têm sido padronizados para armazenamento e troca de dados entre várias indústrias. Enquanto dados críticos ainda estão no formato relacional, na prática, vários projetistas têm migrado para armazenar dados que não se adequam ao modelo relacional, em XML.

Nesse contexto, um SGBD puramente relacional ou puramente XML não resolve todas as necessidades e requisitos das bases de dados modernas. Enquanto um SGBD XML facilita o armazenamento de determinadas informações semiestruturadas, outras necessidades ainda são melhor supridas por SGBDs relacionais. Na indústria de saúde, por exemplo, XML é amplamente empregada para compartilhar metadados de registros médicos em repositórios permanentes. Em um cenário real, o esquema para os metadados contém mais de 200 variações para integrar os diversos tipos de documentos médicos a serem armazenados e consultados [Moro et al. 2007b]. Esses 200 tipos possuem uma seção comum compartilhada e extensões individuais específicas. Persistir tais metadados no formato relacional resulta em um grande número de tabelas e desempenho pobre. Além disso, adicionar um novo tipo de documento requer no mínimo duas semanas de re-engenharia do esquema relacional para acomodar o novo tipo.

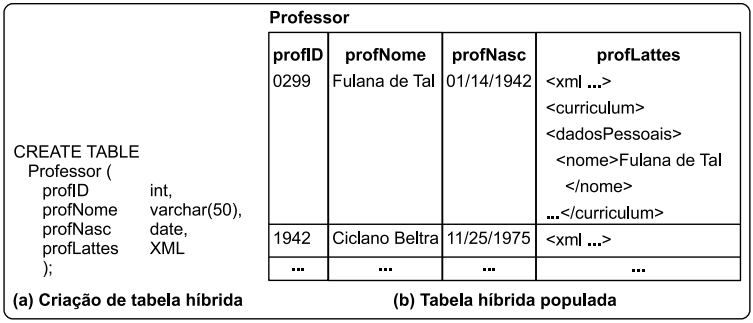

Figura 5.6. Exemplo de tabela híbrida XML/Relacional

Considerando esse caso real bem como outros tantos existentes, um sistema *híbrido* para dados relacionais *e* dados XML é a melhor solução para modelar, persistir, gerenciar dados relacionais e XML de uma maneira unificada [Moro et al. 2007b]. Por exemplo, a Figura 5.6 ilustra como uma tabela com dados híbridos pode ser criada. Nesse caso, são criadas colunas com tipos tradicionais (inteiro, caractere e data) bem como uma coluna do tipo XML. Embora os SGBDs forneçam suporte aos dados híbridos, não existe uma metodologia para a modelagem desses dados. Sem um modelo de dados compatível, várias opções para melhorar o desempenho do armazenamento e da consulta aos dados são perdidas; especificamente, todas aquelas que se utilizam de informações do esquema, pois um esquema que não seja adequado é praticamente inútil.

Porém, o problema de definir um esquema híbrido XML-Relacional para uma aplicação de dados é particularmente complexo por várias razões. Primeiro, enquanto a maioria dos fabricantes de SGBDs tem suporte para XML dentro dos seus produtos relacionais, pouquíssimos estudos industriais e pesquisas acadêmicas têm se aventurado a sugerir metodologias para esse novo ambiente de projeto. Segundo, os parâmetros e os requisitos do problema são muitos e diversos. Dado um cenário de uso, existem muitas maneiras de modelar as necessidades da informação a ser gerenciada e vários modos para escrever tal modelo. De mesmo modo, os usuários (projetistas de BD) podem também ter prioridades diferentes para o esquema. Em alguns casos, o importante é prover flexibilidade, em outros casos o importante é o desempenho e o tempo de consulta. Terceiro, em geral, as soluções não são únicas porque diferentes esquemas XML-Relacional podem satisfazer o mesmo conjunto de parâmetros e requisitos.

O trabalho de Moro et al. (2007b) propõe uma solução inicial para o problema de como decidir quais colunas serão modeladas como relacionais e quais como XML. Note que essa é apenas *uma* das questões envolvidas na

modelagem híbrida. Outras questões incluem: como definir um modelo elegante para representar ambos os tipos de dados; como adicionar a semântica de objetos de negócios a esse modelo; como modelar atributos esparsos e valores opcionais; como evoluir dados, esquema e consultas sobre o modelo híbrido; entre outros. Referenciamos Moro et al. (2009a) para uma lista mais completa de desafios da modelagem híbrida XML/relacional.

5.3.2.4. Fragmentação de Bases de Dados

Dependendo do domínio de aplicação em que se esteja trabalhando, documentos XML podem ser muito grandes (ocupando espaço na ordem de GB). Processar consultas de forma eficiente sobre documentos grandes é um desafio. Em bancos de dados relacionais, uma solução para aumentar o desempenho de processamento de consultas é o uso de paralelismo. Nesse caso, os dados são fragmentados e distribuídos em diferentes nós de uma rede [Özsu and Valduriez 1999]. As consultas sobre a base dados distribuída são automaticamente decompostas e executadas em paralelo, diminuindo o tempo total de processamento (nos casos em que a consulta se beneficia do esquema de fragmentação utilizado). A mesma ideia pode também ser aplicada a bases de documentos XML. Para isso, muito se tem pesquisado sobre técnicas de fragmentação de dados XML [Abiteboul et al. 2003, Bremer and Gertz 2003, Andrade et al. 2006, Amer-Yahia and Kotidis 2004, Bonifati et al. 2004, Bose et al. 2003, Ma and Schewe 2003].

As primeiras abordagens para fragmentação de bases XML foram propostas por Ma e Schewe (2003) e Bremer e Gertz (2003). No primeiro artigo [Ma and Schewe 2003] são propostos três tipos de fragmentação: *horizontal*, que agrupa elementos de um único documento XML de acordo com um critério de seleção especificado; *vertical*, que reestrutura o documento desaninhando alguns elementos; e um tipo especial de fragmento chamado *split*, que quebra o documento XML em um conjunto de novos documentos. Já no segundo artigo [Bremer and Gertz 2003], é proposto um tipo de fragmento que mistura fragmentação vertical com horizontal. O foco do trabalho está na definição e também na alocação dos fragmentos. As duas propostas possuem uma grave deficiência: funcionam para um único documento XML. Quando se tem uma base com vários documentos XML, os fragmentos precisam ser definidos para cada um dos documentos envolvidos.

Diferentes definições de fragmentos para XML têm sido utilizadas nas áreas de processamento de *streams* de dados [Bose et al. 2003], ambientes ponto a ponto [Abiteboul et al. 2003, Bonifati et al. 2004] e cenários baseados em serviços Web [Amer-Yahia and Kotidis 2004]. No entanto, eles ou não apresentam alternativas para fragmentação de múltiplos documentos, ou não distinguem entre os diferentes tipos de fragmentos, o que torna a verificação da correção do projeto de fragmentação uma tarefa bastante complexa.

Mais recentemente, a proposta de Andrade et al. (2006), chamada Par-

tiX, supre esses problemas. Ela permite que fragmentos sejam definidos sobre um único documento ou sobre uma coleção de documentos XML. Além disso, existe uma distinção clara entre os diferentes tipos de fragmentos *horizontal*, *vertical* e *híbrido*. Os fragmentos são definidos através de operações da álgebra TLC (*Tree Logical Class*) [Paparizos et al. 2004]. Um fragmento horizontal é definido através de uma operação de seleção sobre os documentos XML. Diante disso, os fragmentos resultantes possuem a mesma estrutura dos documentos originais. A operação de seleção apenas separa os documentos que obedecem ao critério de seleção em um mesmo fragmento. Como um exemplo, suponha que o documento da Figura 5.1 é apenas um de uma série de doĉumentos de uma coleção. Suponha também que cada documento da coleção contém um atributo *grupo* na raiz do documento (*empregados*) que identifica o grupo de empresas em que aqueles empregados trabalham (por exemplo, se grupo for *Globo*, os empregados trabalham para uma das empresas do grupo Globo), além de um atributo *empresa* com o nome da empresa em que os empregados trabalham. Um fragmento horizontal F_1h poderia ser definido como $\sigma_{/empregados/@grupo='Globo'}$. Supondo que a coleção de documentos se chama *CEmpregados*, a definição completa do fragmento F_1h ficaria como segue:

F_1h := <*CEmpregados*, $\sigma_{/empregados/@grupo='Globo'}$>

Fragmentos devem ser projetados de forma a cobrir toda a base de dados, de modo que qualquer dado apareça em pelo menos um fragmento (completude). Do mesmo modo, um dado não pode aparecer em mais de um fragmento (disjunção). Assim, precisamos também definir um fragmento para todos os outros documentos que não sejam de grupo Globo, como descrito a seguir.

F_2h := <*CEmpregados*, $\sigma_{/empregados/@grupo\neq'Globo'}$>

A coleção original pode ser reconstruída fazendo-se uma união dos fragmentos horizontais F_1h e F_2h.

Um fragmento vertical é definido através de uma operação de projeção. O operador de projeção para XML tem uma semântica bastante sofisticada. Com ele é possível especificar projeções que excluem subárvores cuja raiz está localizada em qualquer nível do documento XML. Como um exemplo, suponha que se deseje definir um fragmento vertical que não contém o elemento *inicial-meio*. Esse fragmento pode ser definido da seguinte forma:

F_1v := <*CEmpregados*, $\pi_{/empregados,\{/empregados/empregado/inicial-meio\}}$ >

Nesse exemplo, a subárvore `/empregados/empregado/inicial-meio` é podada do fragmento. Novamente, é necessário definir um outro fragmento que contém o elemento *inicial-meio*.

F_2v := <*CEmpregados*, $\pi_{/empregados/empregado/inicial-meio,\{\}}$>

O sistema PartiX se encarrega de inserir identificadores artificiais para que seja possível reconstruir a coleção original através da junção dos fragmentos verticais F_1v e F_2v. Da mesma forma, é possível definir fragmentos híbridos que misturam fragmentação vertical e horizontal (ou vice-versa).

Uma vez definidos os fragmentos, consultas podem ser processadas de forma mais eficiente. Uma metodologia para execução de consultas sobre bases XML fragmentadas de acordo com Andrade et al. (2006) foi proposta por Figueiredo (2007) [Figueiredo et al. 2007]. No entanto, embora existam várias técnicas de fragmentação diferentes, e que experimentos demonstrem que elas são efetivas, ainda não existe uma metodologia de projeto de fragmentação para XML. Uma metodologia é fundamental para que as técnicas de fragmentação possam ser utilizadas na prática.

5.3.2.5. Integração de Instâncias

O problema de integração envolve na verdade dois sub-problemas distintos: integração de esquemas e integração de instâncias. Na integração de esquemas, conforme discutido na seção 5.3.1.1, é necessário fazer mapeamentos entre construções (elementos) de cada esquema, de modo que consultas possam ser processadas automaticamente. No entanto, isso não resolve o problema de integração de instâncias. Ocorre que uma mesma entidade do mundo real (por exemplo, uma pessoa), pode aparecer em várias bases de dados distintas. Ao fazer uma consulta sobre essas várias bases de dados, seria interessante que se pudesse identificar as instâncias que correspondem à mesma entidade (pessoa), e que tais resultados fossem combinados em uma única resposta para o usuário.

A identificação de instâncias duplicadas não é um problema exclusivo de XML. Em bancos de dados relacionais o problema também ocorre e é crítico. Como um exemplo, em uma grande empresa, dependendo das restrições e gatilhos implementados no banco de dados, podem existir vários cadastros para o mesmo cliente. Uma revisão da literatura cobrindo métodos para solucionar tais problemas é apresentada em [Elmagarmid et al. 2007].

Em XML, esse problema vem sendo tratado com o uso de funções de similaridade que levam em consideração não só o esquema, mas também o conteúdo dos elementos XML. O objetivo é tentar identificar instâncias duplicadas mesmo que elas tenham esquema e conteúdo distintos, como no seguinte exemplo.

```
<empregado>
   <nome>João Silva Santos</nome>
   <cpf>123.456.789-00</cpf>
</empregado>
```

```
<pessoa>
   <primeiro-nome>João</primeiro-nome>
   <sobrenome>S. Santos</sobrenome>
   <cic>12345678900</cic>
</pessoa>
```

Uma das funções de similaridade mais conhecidas e utilizadas para esse propósito é a *tree edit distance* [Shasha and Zhang 1997]. No entanto, esse algoritmo dá mais peso à topologia da árvore do que à semântica das marcas XML. Esse fato causa inúmeros problemas, resultando invariavelmente em casamentos errados. A *structure aware XML distance* [Milano et al. 2006], por sua vez, tenta identificar estruturas comuns em documentos XML, mas exige que os caminhos dos nodos que casam sejam exatamente iguais, o que limita bastante o uso dessa abordagem. O trabalho de Weis e Naumann (2005) compara elementos com base em seus valores e também com base na similaridade de seus pais e irmãos na árvore XML. Uma outra alternativa é a técnica híbrida que dá pesos diferentes para similaridade de conteúdo e similaridade estrutural proposta por Kade e Heuser (2008).

Apesar de tantas propostas existentes, o problema de integração de instâncias está longe de ser resolvido completamente, sendo necessárias técnicas mais eficientes e que produzam melhores resultados.

5.3.3. Novos Desafios

O número de cenários de aplicações que utilizam dados estruturados, semi- e não-estruturados vem crescendo cada vez mais. Além disso, existem grandes coleções heterogêneas com dados estruturados relacionados a dados não-estruturados, tais como repositórios de documentos e emails. Na Web, dados estruturados também se multiplicam devido a três fontes principais: milhões de base de dados escondidas em formulários (a *deep web*); centenas de milhões de dados de alta qualidade em páginas com tabelas HTML e *mashups* que fornecem visões dinâmicas de dados estruturados; e dados provenientes da Web 2.0, tais como fotos e vídeos, serviços de anotações colaborativas e repositórios online de dados estruturados. Esse conjunto de repositórios espalhados pela Web tem recebido o nome de *dataspaces*.

Esses cenários motivaram a comunidade internacional a estabelecer novos desafios de pesquisa a serem abordados na área de Banco de Dados [Agrawal et al. 2008]. Especificamente, nota-se um período de transição da pesquisa em bases de dados estruturadas (como aquelas que seguem o modelo relacional) para o gerenciamento de coleções de dados ricas em dados estruturados, semi-estruturados e não-estruturados que estão espalhadas em diversos repositórios de negócios e na Web. Esses dados são gerados a partir da extração de informação de textos e páginas Web, incluindo coletas em blogs e comunidades online, logs de aplicações, sensores e da *deep* Web.

Outro fator importante é que o volume de dados a ser gerenciado aumentou consideravelmente. Desse modo, várias comunidades têm trabalhado com

grandes volumes de dados. Entre essas podemos citar: *e-Science*, processamento de linguagens naturais e redes sociais. Também é importante notar que existe uma diferença considerável entre os tipos de aplicações em cada uma dessas comunidades. Por exemplo, na comunidade de redes sociais, além de identificar dados relevantes, também é necessário *visualizar* tais dados de maneira adequada. Desse modo, novas soluções especializadas de Banco de Dados, provavelmente formadas por componentes mais simples, tornam-se necessárias. XML certamente será uma das tecnologias determinantes na busca de soluções para esses desafios.

Alguns novos desafios inspirados em Agrawal et al. (2008) incluem: processamento paralelo de documentos e consultas XML; identificação do contexto de dados textuais e respectiva representação em XML; extração de estrutura de dados textuais; busca de palavras em fontes XML heterogêneas sem qualquer representação semântica (ou seja, sem conhecimento de ontologias e domínios por exemplo); sistemas para dados XML auto-gerenciáveis; processamento de dados XML em dispositivos móveis (pouco consumo de energia e configuração de hardware limitada) e sistemas baseados em localização. Finalmente, a título de curiosidade, Agrawal et al. (2008) apresenta uma tabela com os tópicos recorrentes nas discussões dos desafios de banco de dados. O mais citado, que aparece em todas as seis edições anteriores do encontro, é SGBDs heterogêneos (com interoperabilidade e troca de dados entre seus subassuntos), seguido de workflows e dados incertos e probabilísticos.

5.4. Indústria

Esta seção apresenta alguns padrões de indústria especificados em XML, discute como questões de armazenamento são resolvidas na prática e algumas ferramentas que podem ser utilizadas no gerenciamento de documentos XML.

5.4.1. Padrões em XML

O sucesso de XML na indústria pode ser medido pela quantidade de formatos baseados em XML existentes. O site XML Cover Pages[6] apresenta uma lista de vários formatos baseados em XML. Essa lista é vasta e inclui iniciativas nos mais diversos domínios. Como exemplos, podemos citar os já mencionados na Introdução (ACORD, FPMl e HL7) e os seguintes:

- ARTS[7], formato para lojas de comércio a varejo.
- EML (Election Markup Language)[8], um formato definido pela OASIS para troca de dados em eleições (públicas ou privadas).
- MISMO[9], formato para empréstimos e hipotecas bancárias.

[6] `http://xml.coverpages.org/xml.html\#applications`

[7] `http://www.nrf-arts.org`

[8] `http://docs.oasis-open.org/election/eml/v5.0/os/EML-Schema-Descriptions-v5.0.html`

- RSS[10], um formato para notificação de novos conteúdos na Web.
- WSDL (Web Services Description Language), um formato recomendado pela W3C para descrição de serviços Web.
- XDMF[11], um formato para representar dados científicos em aplicações de alto desempenho (HPC).
- XMI[12] (XML Metadata Interchange), um formato para representar modelos e instâncias, definido pela OMG. Diversas ferramentas usam XML para salvar diagramas UML.
- XML RPC[13], uma especificação que permite chamadas remotas a procedimentos via Internet.

Todas esses iniciativas definiram um esquema XML (usando DTD ou XML Schema) e o disponibilizaram para a comunidade usuária. Alguns são formatos padronizados, como por exemplo o WSDL, recomendado pela W3C. Outros são adotados na prática por uma determinada comunidade, e acabam ganhando força, como é o caso do XDMF. De todo modo, essa pequena lista de formatos retirada de uma lista muito maior, mostra o quanto XML ganhou força no mercado e na comunidade científica. Definitivamente, é uma linguagem que veio para ficar.

5.4.2. Armazenamento

Empresas de grande porte investem altos recursos para adquirir licenças de Sistemas Gerenciadores de Bancos de Dados Objeto-Relacionais, e em manter seus DBAs (administradores de banco de dados) treinados e atualizados. Diante desse cenário, é normal que tais empresas relutem em adotar outros SGBDs para armazenar seus documentos XML. Pensando nisso, os principais fornecedores de SBGDs evoluíram para fornecer a esses clientes a capacidade de armazenar documentos XML.

Existem basicamente três maneiras de armazenar documentos XML em um banco de dados. A primeira, e mais simples, é armazenar um documento XML como uma sequência de caracteres, por exemplo um CLOB (*character large object*). Essa é a solução mais simples de implementar. Porém, é também a mais complexa para manipular os documentos XML. Por exemplo, existe o problema de realizar uma consulta XML em um documento que contém texto puro, sem qualquer distinção entre os elementos e os seus valores. Nesse caso, para cada consulta, o documento XML precisa ser lido a fim de encontrar as marcações de elementos (“<” e “>”). Esse processo é extremamente custoso, principalmente considerando documentos grandes.

[9] `http://www.mismo.org`

[10] `http://www.rssboard.org/rss-specification`

[11] `http://www.xdmf.org/index.php/Main_Page`

[12] `http://www.omg.org/technology/xml`

[13] `http://www.xmlrpc.com`

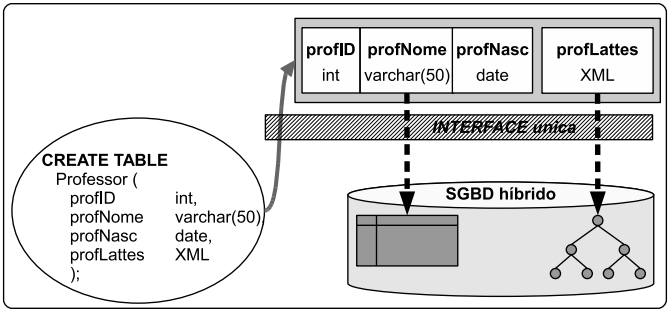

Figura 5.7. Armazenamento de dados XML em SGBD híbrido

A segunda maneira é mapear os elementos XML para relações (em SGBDs relacionais) e objetos (em SGBDs orientados a objeto e objeto-relacionais). Nesse caso, o problema anterior (do processamento de consulta) é relativamente simples de resolver, pois basta avaliar os elementos agora mapeados para relações ou objetos. Porém, ainda existem dois problemas com tal solução: (i) existem diferentes algoritmos para realizar o mapeamento, e cada aplicação industrial pode utilizar o seu próprio; e (ii) uma consulta XML pode ter seus dados distribuídos em diferentes relações e objetos, sendo necessário realizar junções entre relações/objetos diferentes. A operação de junção é uma das mais custosas em um SGBD. Desse modo, cada operação de consulta ainda teria seu desempenho prejudicado.

A terceira maneira de armazenar documentos XML é utilizar um armazenamento nativo. Nesse caso, os dados XML são armazenados considerando suas características próprias, como a hierarquia de elementos que forma uma estrutura de árvore. O armazenamento nativo pode ser implementado como uma parte individual do SGBD relacional ou XML. Nesse caso, o SGBD é considerado como um *banco de dados híbrido*. Além disso, pode-se definir um SGBD específico para documentos XML, desconsiderando a possibilidade de armazenar qualquer outro tipo de dado. Nesta seção, detalhamos essas duas formas de armazenar documentos XML no formato nativo.

5.4.2.1. Bancos de Dados Híbridos

As três grandes empresas desenvolvedoras de SGBDs são a Microsoft (SQL Server), a IBM (DB2) e a Oracle. Além desses ainda temos iniciativas livres para desenvolver SGBDs como o MySQL e o PostgreSQL.

Como mencionado na seção 5.3.2.3, dados relacionais (ou objeto-relacionais) podem existir em uma mesma tabela em harmonia com dados XML. Considerando o exemplo apresentado na Figura 5.6, os SGBDs SQL Server, DB2 e Oracle permitem que essa tabela seja criada utilizando um tipo

próprio XML. Nesse caso, embora para o usuário os dados sejam tratados de maneira uniforme através de uma interface única, o sistema gerenciador distingue o tipo de dados XML e o trata com operações próprias. Podemos imaginar que o SGBD possui duas partes individuais: uma com as operações para os dados de tipos tradicionais e outra com as operações para dados de tipo XML, conforme ilustrado na Figura 5.7.

Além disso, esses SGBDs fornecem ao usuário funções do padrão SQL/XML [Eisenberg and Melton 2002, Eisenberg and Melton 2004]. XML/SQL que permite que dados XML sejam gerados a partir de dados relacionais, aumentando as opções de quem precisa lidar com ambos os modelos de dados.

É importante notar que os SGBDs continuam oferecendo todas as suas funções (controle de transação e segurança por exemplo) tanto para dados SQL quanto XML. Além disso, oferecem facilidades específicas para dados XML como índices apropriados para tratarem da estrutura hierárquica dos elementos. Por exemplo, o código a seguir ilustra como índices podem ser criados nos três SGBDs comerciais considerando o caminho até o cpf do professor ou todos os caminhos. Note que linhas que começam com "−−" são comentários.

```
-- IBM DB2: cria índice com a restrição de unicidade
 CREATE UNIQUE INDEX lattesidx on Professor(profLattes)
  GENERATE KEY USING XMLPATTERN '/lattes/dpessoais/@cpf'
  AS SQL DOUBLE;

-- Oracle 10g: cria índice em cpf
 CREATE INDEX lattesidx ON Professor p
  (EXTRACTVALUE(VALUE(p),'/lattes/dadospessoais/@cpf'));

-- Microsoft SQL Server: cria índice em todos caminhos
 CREATE XML INDEX lattesidx on Professor(profLattes)
  USING XML INDEX xml_lattesidx FOR PATH;
```

Para armazenar dados XML no SGBD livre MySQL é necessário utilizar os tipos básicos, tais como *BLOB* e *LONGTEXT*. O MySQL não oferece armazenamento nativo XML até o momento. Nesse caso, é necessário implementar as funções para dados XML na própria aplicação sobre o banco de dados.

Para gerenciar dados XML no SGBD livre PostgreSQL é necessário utilizar o tipo básico *text* e um módulo de extensão (geralmente armazenado no diretório *contrib/xml*). Essa extensão acrescenta duas facilidades básicas aos PostgreSQL: (i) a validação de documentos XML com base em uma DTD (função *pgxml_parse*); e (ii) a avaliação de consultas XPath nos documentos armazenados (função *pgxml_xpath*).

5.4.2.2. Bancos de Dados XML Nativos

Em cenários onde não existe a preocupação de armazenar dados (objeto-) relacionais e XML em um repositório único, os bancos de dados nativos são

uma boa solução. Os bancos de dados XML nativos utilizam o modelo XML internamente, ou seja, não existe a necessidade de se realizar um mapeamento entre o modelo relacional ou objeto-relacional para XML [Bourret 2005]. A maioria desses SGBDs utiliza o modelo DOM para armazenar os documentos XML.

É importante notar que nesses bancos de dados não existe o conceito de "tabela" e "tupla". O que existem são "coleções" e "documentos". Uma coleção pode armazenar vários documentos XML. Documentos em uma coleção podem estar associados a um esquema, e o banco de dados é capaz de verificar, no momento da inserção, se o documento é ou não válido de acordo com o esquema associado. No entanto, o uso de esquemas não é obrigatório. Uma coleção pode conter documentos com estruturas completamente diferentes.

Os SGBDs XML nativos oferecem suporte a consultas XPath e XQuery, e, para atualização, possuem diferentes abordagens, já que a linguagem para atualização de documentos XML ainda não foi recomendada pela W3C. Em resumo, um Banco de Dados XML Nativo apresenta as seguintes características: define um modelo lógico para documentos XML, armazenando e recuperando documentos de acordo com tal modelo, e tem o documento XML como unidade lógica de armazenamento (assim como a unidade lógica do modelo relacional é uma linha em uma tabela). Note que o modelo XML deve incluir elementos, atributos, PCDATA e ordem. Exemplos desse modelo são os modelos XPath e XML Infoset [Cowan and Tobin 2004], bem como os modelos inferidos a partir das APIs DOM e SAX.

Entre os SGBDs XML nativos podemos citar:

- *eXist-DB*[14] é uma iniciativa aberta para criação de um SGBD XML nativo escrito em Java. Armazena documentos de acordo com o modelo XML e provê processamento de consultas XQuery de maneira eficiente utilizando índices. Tem suporte a XQuery 1.0, XPath 2.0, XSLT 1.0 (usando Apache Xalan), XSLT 2.0, WebDAV, SOAP, XMLRPC, entre outros.
- *MonetDB/XQuery*[15], desenvolvido pelo centro de pesquisa CWI (Holanda), provê suporte praticamente completo à linguagem XQuery, incluindo características como suporte a funções definidas pelo usuário e cache para consulta. O sistema escala para grandes coleções XML.
- *Ozone Database Project*[16] é uma iniciativa aberta para criação de um SGBD orientado a objetos com código aberto em Java. Inclui uma implementação do DOM para armazenar XML. Ainda oferece suporte para Apache Xerces-J e Xalan-J.
- *SEDNA*[17], implementado em C/C++, é um SGBD XML nativo não comercial que fornece vários serviços, entre eles: armazenamento, transações

[14] `http://exist.sourceforge.net/`

[15] `http://monetdb.cwi.nl/XQuery`

[16] `http://ozone-db.org`

ACID, segurança, índices, suporte a XQuery com função de busca e uma linguagem de atualização no nível de elementos.

- *Tamino*[18] associa o gerenciamento nativo a um servidor Web. O servidor acessa dados tanto de um repositório XML quanto de um relacional. Utiliza a linguagem de consulta XQL e tem suporte às APIs DOM, JDOM e SAX, serviços SOAP e WebDAB.
- *Timber*[19], desenvolvido pela Universidade de Michigan, fornece várias características importantes, tais como suporte a uma álbebra XML, indexação de dados e otimização de consultas. Outras informações podem ser encontradas no seu website e em [Jagadish et al. 2002].

Uma lista completa de outros SGBDs nativos XML pode ser encontrada no seguinte website: http://www.rpbourret.com/xml/XMLDatabaseProds.htm#native (acesso em 01 de fevereiro de 2009).

5.4.3. Ferramentas

Existem diversas ferramentas para trabalhar com XML. Algumas de código aberto, outras proprietárias. Esta seção apresenta um apanhado das principais ferramentas para validar, transformar, consultar e trabalhar com XML em forma de objetos em memória. As ferramentas são apresentadas por categoria, dependendo da funcionalidade que disponibilizam.

Editores. O principal editor disponível no mercado é sem dúvida o Altova XML Spy[20]. Por muitos anos, essa foi uma das principais ferramentas para se trabalhar com XML. Isso porque trata-se de uma ferramenta bastante completa, que apresenta opções para validação contra DTD e XML Schema, transformações XSLT, consultas XPaths e XQuery, entre outras. No entanto, ela é uma ferramenta proprietária e baseada em Windows. Ao longo dos anos, desde o surgimento da XML Spy, quando comparadas com ele as ferramentas livres eram sempre menos completas. Isso vem mudando recentemente. Hoje existem ferramentas livres bem mais robustas. Um exemplo é a Exchanger XML Lite[21]. Ela possui todas as funcionalidades citadas anteriormente e várias outras. É baseada em Java e a princípio independente de plataforma. Outra boa opção é o XML Copy Editor[22]. Esse também é livre e possui versões para Windows e Linux. Provê as funcionalidades de validação contra DTD e XML Schema, consultas XPath e transformações XSLT.

[17] `http://www.modis.ispras.ru/sedna`

[18] `http://www.softwareag.com/tamino`

[19] `http://www.eecs.umich.edu/db/timber`

[20] `http://www.altova.com/products/xmlspy/xml_editor.html`

[21] `http://www.freexmleditor.com/`

[22] `http://xml-copy-editor.sourceforge.net/`

Transformação. Todos os editores apontados no item anterior também são capazes de realizar transformações XML. Além disso, os principais navegadores Web (IE Explorer, Firefox, etc.) possuem um processador XSLT embutido. Desse modo, sempre que um arquivo XML que tem associado um arquivo XSLT for aberto no navegador, a transformação será executada, e o usuário verá apenas o resultado final da transformação. O arquivo original pode ser visto, no entanto, escolhendo a opção "visualizar código fonte" que os navegadores oferecem. Além dessas opções, existe uma outra bastante poderosa. A Altova (fabricante do XML Spy) tem uma ferramenta específica para transformações chamada Altova Map Force[23]. Essa ferramenta é capaz de gerar o arquivo com as regras de transformação de forma automática.

XML e Orientação a Objetos. Além das APIs discutidas na Seção 5.2.2, existem vários *frameworks* para manipulação de arquivos XML via linguagem de programação. O principal propósito desses *frameworks* é deixar transparente para a aplicação a existência de arquivos XML. Eles fazem a leitura do arquivo XML e transformam os elementos existentes no arquivo em objetos na memória. Assim, um elemento cliente se transforma em um objeto cliente na memória, e a manipulação dos dados passa a ser totalmente orientada a objetos. Exemplos de tais frameworks são Castor[24] e Apache XML Beans[25].

Além disso, várias ferramentas atualmente usam XML como formato de seus arquivos. Exemplos são aplicativos da família Microsoft Office, Open Office, Argo UML, entre outras.

5.5. Conclusão

O sucesso de XML reside em seu formato aberto e de fácil adoção. No entanto, XML por si só não teria obtido o alcance que tem hoje se não fossem suas linguagens relacionados. As APIs, linguagens de consulta, esquemas e demais tecnologias contribuíram para que XML fosse adotada tão intensamente pela indústria. Seu formato aberto também estimulou os pesquisadores a considerarem XML como apoio fundamental na resolução de vários problemas clássicos de pesquisa. Além disso, XML vem sendo usada cada vez mais pela indústria nos mais diversos segmentos. Por esses e outros motivos, XML é a tecnologia ubíqua de maior sucesso para Web, juntamente com as tecnologias básicas URI, HTTP e HTML [Wilde and Glushko 2008].

Especificamente na comunidade de Banco de Dados, nota-se um período de transição da pesquisa em bases de dados estruturadas (como aquelas que seguem o modelo relacional) para o gerenciamento de coleções de

[23] `http://www.altova.com/products/mapforce/data_mapping.html`

[24] `http://www.castor.org/xml-framework.html`

[25] `http://xmlbeans.apache.org/`

dados extremamente ricas em dados estruturados, semiestruturados e não-estruturados que estão espalhadas em diversos repositórios de negócios e na Web [Agrawal et al. 2008]. XML certamente será uma das tecnologias determinantes na busca de soluções para esses novos desafios que são extremamente relevantes. O objetivo deste capítulo é proporcionar aos alunos de graduação e pós, bem como profissionais, em Computação e Informática um panorama geral da situação de XML nos dias atuais. O momento para esse panorama não poderia ser mais propício, visto a explosão da utilização de XML em trabalhos de pesquisa para a Web (e.g.: *Web Sciences*) bem como em tecnologia móvel para acesso universal à informação (e.g.: *M-Government*).

Finalmente, este capítulo apresentou conhecimentos básicos de XML, uma noção geral dos problemas de pesquisa que XML ajudou e está ajudando a solucionar bem como um panorama da influência de XML nos padrões industriais. Espera-se que os seus leitores estejam motivados a estudar mais a fundo XML, contribuindo assim para o avanço científico e tecnológico do país.

Agradecimentos

Gostaríamos de agradecer a todos os colaboradores que trabalharam conosco nos últimos 8 anos em tópicos relacionados a XML, entre eles: Carina F. Dorneles (UPF), Renata M. Galante (UFRGS), Carlos A. Heuser (UFRGS), Adrovane Kade (UFRGS), Vassilis J. Tsotras (UCR, USA), Zografoula Vagena (Microsoft, UK), Susan Davidson (UPenn, USA), Marta Mattoso (COPPE-UFRJ), Guilherme Figueiredo, Cláudio Ferraz, André Vargas, Fernanda Baião (UNIRIO), Ronaldo S. Mello (UFSC), Dênio Duarte (UnoChapecó), Alberto H. F. Laender (UFMG) e José Palazzo M. de Oliveira (UFRGS). Este trabalho foi parcialmente financiado por CNPq, CAPES, FAPEMIG e FAPERJ.

Referências bibliográficas

[Abiteboul et al. 2004] Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., and Preda, N. (2004). Lazy Query Evaluation for Active XML. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 227 – 238.

[Abiteboul et al. 2008] Abiteboul, S., Benjelloun, O., and Milo, T. (2008). The Active XML project: an overview. *VLDB Journal*, 17(5):1019–1040.

[Abiteboul et al. 2003] Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I., and Milo, T. (2003). Dynamic XML Documents with Distribution and Replication. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 527–538.

[Agrawal et al. 2008] Agrawal, R., Ailamaki, A., Bernstein, P. A., Brewer, E. A., Carey, M. J., Chaudhuri, S., Doan, A., Florescu, D., Franklin, M. J., Garcia-Molina, H., Gehrke, J., Gruenwald, L., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Korth, H. F., Kossmann, D., Madden, S., Magoulas, R., Ooi, B. C., O'Reilly, T., Ramakrishnan, R., Sarawagi, S., Stonebraker, M.,

Szalay, A. S., and Weikum, G. (2008). The Claremont Report on Database Research. *SIGMOD Record*, 37(3):9–19.

[Al-Khalifa et. al 2002] Al-Khalifa et. al, S. (2002). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Procs. of Intl. Conf. on Data Engineering - ICDE*, pages 141–152.

[Amer-Yahia and Kotidis 2004] Amer-Yahia, S. and Kotidis, Y. (2004). A Web-Services Architecture for Efficient XML Data Exchange. In *Procs. of Intl. Conf. on Data Engineering - ICDE*, pages 523–534.

[Andrade et al. 2006] Andrade, A., Ruberg, G., Baião, F. A., Braganholo, V. P., and Mattoso, M. (2006). Efficiently Processing XML Queries over Fragmented Repositories with PartiX. In *Procs. of Intl. Conf. on Extending Database Technology - EDBT, Workshops*, pages 150–163.

[Beyer et al. 2005] Beyer, K. S., Özcan, F., Saiprasad, S., and der Linden, B. V. (2005). DB2/XML: Designing for Evolution. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 948–952.

[Boag et al. 2007] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. (2007). XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007. *http://www.w3.org/TR/xquery/*.

[Bonifati et al. 2004] Bonifati, A., Matrangolo, U., Cuzzocrea, A., and Jain, M. (2004). XPath lookup queries in P2P networks. In *Procs. of ACM Intl. Work. on Web Information and Data Management - WIDM*, pages 48–55.

[Booth and Liu 2007] Booth, D. and Liu, C. K. (2007). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C Recommendation 26 June 2007. *http://www.w3.org/TR/wsdl20-primer/*.

[Bose et al. 2003] Bose, S., Fegaras, L., Levine, D., and Chaluvadi, V. (2003). A Query Algebra for Fragmented XML Stream Data. In *Procs. of Intl. Work. on Database Programming Languages - DBPL*, pages 195–215.

[Bourret 2005] Bourret, R. (2005). Native XML Databases in the Real World. In *Procs. of XML Conference & Exposition*.

[Bradley 2000] Bradley, N. (2000). *The XML Companion, Second Edition*. Addison-Wesley.

[Braganholo and Heuser 2001] Braganholo, V. and Heuser, C. (2001). XML Schema, RDF(S) e UML: uma comparação. In *Iberoamerican Workshop on Requirements Engineering and Software Environments*, pages 78–90.

[Bray et al. 2006] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2006). Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation 16 August 2006, edited in place 29 September 2006. *http://www.w3.org/TR/2006/REC-xml-20060816/*.

[Bray et al. 2007] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2007). XQuery 1.0 and XPath 2.0 Functions and Operators.

W3C Recommendation 23 January 2007. *http://www.w3.org/TR/2007/REC-xpath-functions-20070123/*.

[Bremer and Gertz 2003] Bremer, J.-M. and Gertz, M. (2003). On Distributing XML Repositories. In *Procs. of Intl. Work. on the Web and Databases - WebDB*.

[Bruno et al. 2002] Bruno, N., Koudas, N., and Srivastava, D. (2002). Holistic Twig Joins: Optimal XML Pattern Matching. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 310–321.

[Chamberlin et al. 2008a] Chamberlin, D., Engovatov, D., Florescu, D., Ghelli, G., Melton, J., Simeon, J., and Snelson, J. (2008a). XQuery Scripting Extension 1.0. W3C Working Draft 3 December 2008. *http://www.w3.org/TR/xquery-sx-10/*.

[Chamberlin et al. 2008b] Chamberlin, D., Florescu, D., Melton, J., Robie, J., and Simï¿½on, J. (2008b). XQuery Update Facility 1.0. W3C Candidate Recommendation 01 August 2008. *http://www.w3.org/TR/xquery-update-10/*.

[Clark 1999] Clark, J. (1999). XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November 1999. *http://www.w3.org/TR/xslt*.

[Clark and DeRose 1999] Clark, J. and DeRose, S. (1999). XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. *http://www.w3.org/TR/xpath*.

[Cowan and Tobin 2004] Cowan, J. and Tobin, R. (2004). XML Information Set (Second Edition). W3C Recommendation 4 February 2004. *http://www.w3.org/TR/xml-infoset*.

[Diao et al. 2004] Diao, Y., Rizvi, S., and Franklin, M. J. (2004). Towards an Internet-Scale XML Dissemination Service. In *Procs. of Intl. Conf. on Very Large Data Bases - VLDB*, pages 612–623.

[Eisenberg and Melton 2002] Eisenberg, A. and Melton, J. (2002). SQL/XML is making good progress. *SIGMOD Record*, 31(2):101-108.

[Eisenberg and Melton 2004] Eisenberg, A. and Melton, J. (2004). Advancements in SQL/XML. *SIGMOD Record*, 33(3):79-86.

[Elmagarmid et al. 2007] Elmagarmid, A. K., Ipirotis, P. G., and Verykios, V. S. (2007). Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge And Data Engineering*, 19(1): 1-16.

[Fallside and Walmsley 2004] Fallside, D. C. and Walmsley, P. (2004). XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004. *http://www.w3.org/TR/xmlschema-0/*.

[Fiege 2005] Fiege, L. (2005). Data Dissemination. In *Encyclopedia of Database Technologies and Applications*, pages 105–109.

[Figueiredo 2007] Figueiredo, G. (2007). Consultas sobre visões XML de Bases Relacionais Distribu ídas. Dissertação de Mestrado, Programa de Enge-

nharia de Sistemas e Computação/UFRJ, Rio de Janeiro, RJ.

[Figueiredo et al. 2007] Figueiredo, G., Braganholo, V., and Mattoso, M. (2007). Um Mediador para o Processamento de Consultas sobre Bases XML Distribuídas. In *Anais do Simpósio Brasileiro de Banco de Dados - SBBD, Sessão de Demos*, pages 21–26.

[Getov 2008] Getov, V. (2008). E-Science: the added value for modern discovery. *IEEE Computer*, 41(11):30–31.

[Gou and Chirkova 2007] Gou, G. and Chirkova, R. (2007). Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering - TKDE*, 19(10):1381–1403.

[Guerrini and Mesiti 2008] Guerrini, G. and Mesiti, M. (2008). X-Evolution: A Comprehensive Approach for XML Schema Evolution. In *Procs. of Intl. Conf. on Database and Expert Systems Applications - DEXA ,Workshops*, pages 251–255.

[Guerrini et al. 2005] Guerrini, G., Mesiti, M., and Rossi, D. (2005). Impact of XML Schema Evolution on Valid Documents. In *Procs. of ACM Intl. Work. on Web Information and Data Management - WIDM*, pages 39–44.

[Hors et al. 2004] Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. (2004). Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation 07 April 2004. *http://www.w3.org/TR/DOM-Level-3-Core*.

[Jacobs 2005] Jacobs, I. (2005). World wide web consortium process document. *http://www.w3.org/2005/10/Process-20051014/*.

[Jagadish et al. 2002] Jagadish, H. V., Al-khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., and Yu, C. (2002). TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291.

[Kade and Heuser 2008] Kade, A. and Heuser, C. (2008). Matching XML Documents in Highly Dynamic Applications. In *Procs. of ACM Symp. on Document Engineering - DocEng*, pages 191–198.

[Li et al. 2007] Li, G., Hou, S., and Jacobsen, H.-A. (2007). XML Routing in Data Dissemination Networks. In *Procs. of Intl. Conf. on Data Engineering - ICDE*, pages 1400–1404.

[Ma and Schewe 2003] Ma, H. and Schewe, K.-D. (2003). Fragmentation of XML Documents. In *Anais do Simpósio Brasileiro de Banco de Dados - SBBD*.

[Milano et al. 2006] Milano, D., Scannapieco, M., and Catarci, T. (2006). Structure-Aware XML Object Identification. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 29(2):67–76.

[Moro et al. 2007a] Moro, M. M., Bakalov, P., and Tsotras, V. J. (2007a). Early

Profile Pruning on XML-aware Publish/Subscribe Systems. In *Procs. of Intl. Conf. on Very Large Data Bases - VLDB*, pages 866–877.

[Moro et al. 2007b] Moro, M. M., Lim, L., and Chang, Y.-C. (2007b). Schema Advisor for Hybrid Relational-XML DBMS. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 959–970.

[Moro et al. 2009a] Moro, M. M., Lim, L., and Chang, Y.-C. (2009a). Challenges on Modeling Hybrid XML-Relational Databases. In Pardede, E., editor, *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies*, pages 28–45. IGI Global.

[Moro et al. 2009b] Moro, M. M., Vagena, Z., and Tsotras, V. J. (2009b). Recent Advances and Challenges in XML Document Routing. In Pardede, E., editor, *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies*, pages 136–150. IGI Global.

[Özsu and Valduriez 1999] Özsu, M. T. and Valduriez, P. (1999). *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall.

[Paparizos et al. 2004] Paparizos, S., Wu, Y., Lakshmanan, L., and Jagadish, H. (2004). Tree Logical Classes for Efficient Evaluation of XQuery. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 71–82.

[Pereira et al. 2006] Pereira, D., Ruberg, G., and Mattoso, M. (2006). Geração Eficiente de Planos de Materialização para Documentos XML Ativos. In *Anais do Simpósio Brasileiro de Banco de Dados - SBBD*, pages 236–250.

[Quin 2009] Quin, L. (2009). The Extensible Stylesheet Language Family (XSL). *http://www.w3.org/Style/XSL/*.

[Rahm and Bernstein 2001] Rahm, E. and Bernstein, P. A. (2001). A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350.

[Rahm and Bernstein 2006] Rahm, E. and Bernstein, P. A. (2006). An Online Bibliography on Schema Evolution. *SIGMOD Record*, 35(4):30–31.

[Rahm et al. 2004] Rahm, E., Do, H.-H., and Maßmann, S. (2004). Matching Large XML Schemas. *SIGMOD Record*, 33(4):26–31.

[Roddick 1995] Roddick, J. F. (1995). A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37:383–393.

[Ruberg and Mattoso 2008] Ruberg, G. and Mattoso, M. (2008). XCraft: Boosting the Performance of Active XML Materialization. In *Procs. of Intl. Conf. on Extending Database Technology - EDBT*, pages 299–310.

[Sedlar 2005] Sedlar, E. (2005). Managing Structure in Bits & Pieces: the Killer Use Case for XML. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 818–821.

[Shasha and Zhang 1997] Shasha, D. and Zhang, K. (1997). Approximate Tree Pattern Matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press.

[Vagena et al. 2007] Vagena, Z., Moro, M. M., and Tsotras, V. J. (2007). RoX-Sum: Leveraging Data Aggregation and Batch Processing for XML Routing. In *Procs. of Intl. Conf. on Data Engineering - ICDE*, pages 1466–1470.

[Wang and Zaniolo 2003] Wang, F. and Zaniolo, C. (2003). Representing and Querying the Evolution of Databases and their Schemas in XML. In *Procs. of Intl. Conf. on Software Engineering & Knowledge Engineering - SEKE*, pages 33–38.

[Weis and Naumann 2005] Weis, M. and Naumann, F. (2005). DogmatiX Tracks Down Duplicates in XML. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442.

[Wiederhold 1992] Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49.

[Wilde and Glushko 2008] Wilde, E. and Glushko, R. J. (2008). XML Fever. *Communications of the ACM*, 51(7):40–46.

[XML-DEV Community 2002] XML-DEV Community (2002). SAX 2.0.1. *http://www.saxproject.org/*.