Armazenamento e Publicação de Documentos XML utilizando Sistemas de Bancos de Dados Relacionais

Vanessa Godoy Kinoshita

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Armazenamento e Publicação de Documentos XML utilizando Sistemas de Bancos de Dados Relacionais

Vanessa Godoy Kinoshita[1]

Agosto - 2004

Banca Examinadora:

- Prof. Dr. Célio Cardoso Guimarães (Orientador)
- Prof. Dr. Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP
- Prof.ª Dr.ª Claudia Maria Bauzer Medeiros
  Instituto de Computação - UNICAMP
- Prof. Dr. Geovane Cayres Magalhães
  Instituto de Computação - UNICAMP

[1]**Este trabalho foi realizado com suporte financeiro do CNPq.**

Armazenamento e Publicação de Documentos XML utilizando Sistemas de Bancos de Dados Relacionais

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Vanessa Godoy Kinoshita e aprovada pela Banca Examinadora.

Campinas, 13 Agosto de 2004.

Prof. Dr. Célio Cardoso Guimarães
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Resumo

XML tornou-se uma das tecnologias mais adotadas para a representação e intercâmbio de dados na *Web*. Documentos formatados neste padrão se propagaram, gerando problemas em relação ao seu gerenciamento. Várias soluções vêm sendo estudadas, desde o desenvolvimento de sistemas específicos até o uso da tecnologia orientada a objetos. Devido à confiabilidade, escalabilidade, ferramentas e desempenho associados aos sistemas de banco de dados relacionais, eles se tornaram uma opção bastante pesquisada nos últimos anos. Desde 1999, houve uma grande quantidade de artigos publicados na área, envolvendo diferentes técnicas. Pode-se agrupá-las em três áreas: métodos para a criação de esquemas relacionais para armazenamento de documentos XML; algoritmos para tradução de consultas em linguagem XML para SQL; e métodos para a construção de documentos XML a partir de um banco de dados relacional. O objetivo desta dissertação é fazer uma compilação dos resultados, destacando as vantagens e limitações dos métodos. Outras contribuições incluem: a classificação dos métodos de armazenamento, criação de um documento XML simples utilizado como exemplo em todo o trabalho, análise comparativa e discussão de algumas das questões em aberto.

Abstract

XML became one of the most adopted technologies to format and interchange data on the Web. The number of documents in XML increased in the last few years and problems related to management of these documents have appeared. Several proposals have been studied, including the development of native XML systems and adoption of object oriented technology. Due to the maturity level of relational systems, they became a popular solution. Beginning in 1999, many papers have been published in this area, addressing diferent techniques. They can be grouped in three areas: methods to extract relational schemata to store XML documents; algoritms to translate XML queries in SQL queries; and methods to publish XML documents from relational data. The purpose of this dissertation is to survey these results, emphasizing the advantages and limitations of each one. Other contributions include: a classification of the methods to store XML documents, definition of a simple XML document used as an example throughout this work, a comparative analysis, and discussion of some open questions in the area.

Agradecimentos

Gostaria de agradecer primeiramente aos meus pais, Inês e Luiz, que sempre me ajudaram a conquistar todos os meus sonhos. Apesar de nos vermos pouco, saibam que estão comigo em todos os momentos da minha vida, de alegria ou tristeza. Também não posso esquecer das minhas queridas irmãs, Tâ e Tati, valeu todo o apoio.

Um agradecimento especial ao meu orientador, Célio Cardoso Guimarães, pela paciência e dedicação. Por acreditar em meu trabalho e pela disponibilidade em me atender quando necessário.

Gostaria de agradecer também ao meu noivo, Renato, por ter suportado uma pessoa chata escrevendo uma dissertação. Acho que agora nós já podemos nos casar.

E ao Instituto de Computação da UNICAMP, pelo ambiente de trabalho que proporcionou a conclusão desta dissertação.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

XML (*Extensible Markup Language*) é uma das tecnologias computacionais de aplicabilidade maior do que a imaginada em sua concepção. Quando o *XML Working Group*, formado sob os auspícios da W3C (*World Wide Web Consortium*) em 1996, começou a desenvolver o XML, a idéia principal era a criação de um padrão mais potente que o HTML (*HyperText Markup Language*) para a representação de dados na *Internet*. O desacoplamento da definição dos dados da sua forma de apresentação possibilita que estes mesmos dados sejam utilizados em uma grande quantidade de aplicações.

Nos últimos anos, houve um interesse crescente na utilização de documentos XML para solucionar alguns problemas da área de gerenciamento de dados. Estuda-se o uso de XML para intercâmbio de dados na *Internet* ou entre empresas e para integração de dados, entre outros. Com tantas informações formatadas em XML, a questão do gerenciamento desses documentos torna-se evidente. Ou seja, como armazenar, recuperar e atualizar os dados de forma eficiente?

Há pelo menos quatro abordagens para esse fim [31]. A primeira abordagem consiste na utilização de arquivos textos. Os dados são armazenados utilizando-se pouco espaço físico; entretanto, essa abordagem dificulta a realização de consultas. Para uma aplicação que necessite manipular dados com freqüência, não é uma boa opção.

A segunda abordagem engloba a construção de sistemas de bancos de dados específicos, que armazenam documentos XML em sua forma "original". Exemplos de protótipos desta categoria são Lore [59], Tamino [67], Timber [42], Natix [46] e Xyleme [1]. Esses sistemas são especificamente projetados para armazenar e recuperar dados semi-estruturados, utilizando para isso estruturas e índices especiais e técnicas específicas de otimização de consultas. Teoricamente, deveriam ser a melhor alternativa. Entretanto, ainda deve levar um tempo relativamente longo para que tais sistemas amadureçam e atinjam uma boa escalabilidade para grandes volumes de dados.

A terceira abordagem consiste na utilização de sistemas de bancos de dados que suportam

a tecnologia orientada a objetos. Nesta abordagem, explora-se a ampla capacidade de modelagem de dados deste tipo de sistema. XORator [61] e o trabalho de Klettke et al. [48] são exemplos de sistemas que usam essa abordagem. Apesar do modelo orientado a objetos ser muito semelhante à estrutura de um documento XML, os SGBDs (Sistemas Gerenciadores de Banco de Dados) atuais não suportam adequadamente a tecnologia orientada a objetos.

A quarta abordagem consiste na utilização de sistemas de banco de dados relacionais. Em contrapartida às outras abordagens, sistemas de bancos de dados relacionais apresentam boa escalabilidade e possibilitam a coexistência de dados XML e dados relacionais, facilitando a construção de aplicações que utilizam ambos os tipos de dados com pouco esforço extra. Além disso, pode-se aproveitar as funcionalidades e recursos oriundos do estágio da maturidade já atingido pelos sistemas relacionais, como otimização de consultas SQL, tolerância a falhas, transações, controle de concorrência, etc.

A integração do XML com os sistemas de bancos de dados relacionais para possibilitar armazenamento e recuperação de documentos XML tornou-se uma área bastante pesquisada nos últimos anos. Estudos concentram-se no desenvolvimento de algoritmos para extrair esquemas relacionais dos documentos XML, na criação de técnicas para traduzir consultas escritas em linguagem de consulta para XML em consultas SQL e, por fim, no desenvolvimento de algoritmos para a construção de um documento XML a partir de dados armazenados em um banco de dados relacional.

Desde 1999, uma grande quantidade de artigos foram publicados nesta área, envolvendo diferentes métodos. Com tanta diversidade, é difícil fazer uma comparação direta entre eles, saber como os resultados podem ser agrupados, ou mesmo determinar quais são as questões em aberto. O objetivo desta dissertação é compilar os resultados recentemente publicados, apontando as vantagens e limitações de cada um.

As principais contribuições desta dissertação são:

- levantamento do estado da arte na área de integração de documentos XML com banco de dados relacionais;
- criação de um documento XML simples que foi utilizado para exemplificar a maioria dos mapeamentos apresentados neste trabalho;
- classificação dos métodos de armazenamento de documentos XML;
- análise comparativa dos algoritmos de mapeamento; e
- discussão das questões em aberto na área.

O restante desta dissertação está organizado da seguinte forma: capítulo 2 define os conceitos básicos para a compreensão do restante do texto. O capítulo 3 apresenta o estado da arte para os métodos de armazenamento de documentos XML. A seguir, no capítulo 4,

descrevem-se os métodos para recuperação de dados armazenados em um banco de dados relacional formatados em XML. Os capítulos 3 e 4 também incluem uma análise comparativa entre os métodos e discussão das questões em aberto. Finalmente, o capítulo 5 apresenta as conclusões deste trabalho.

Capítulo 2

Conceitos Básicos sobre XML e Padrões Correlatos

Este capítulo tem a finalidade de apresentar os conceitos básicos sobre XML e alguns padrões correlatos, como DTD, XML Schema, DOM e XSLT. Além disso, pretende-se apresentar uma visão geral de algumas linguagens de consulta XML (XPath, XML-QL e XQuery), para facilitar a compreensão dos próximos capítulos desta dissertação.

2.1 XML - Extensible Markup Language

A quantidade de pessoas que acessam a *Internet* cresce a cada dia. Proporcionalmente, também cresce a quantidade de informações publicadas na *Web*. A necessidade de representar, manipular e recuperar tais informações tem incentivado o estudo de novas linguagens nos últimos anos.

A linguagem-mãe é a SGML (*Standard Generalized Markup Language*) [77]. Seu desenvolvimento iniciou-se no final dos anos 60 por pesquisadores da IBM que pretendiam solucionar problemas relativos à profusão de informações em diferentes formatos. Decidiram que haveria uma padronização nos documentos através do uso de marcações para identificar as estruturas do conteúdo. As informações sobre o formato da apresentação ficariam em outros arquivos, chamados "folhas de estilo" (*style sheets*), que poderiam ser utilizados pelos computadores para gerar a versão final de um documento. Para que se pudesse reconhecer e rejeitar documentos inválidos (com falta ou abundância de informações), a estrutura de cada tipo de documento deveria ser rigidamente definida por um DTD (*Document Type Definition*). A separação da apresentação e validação forneceu grande flexibilidade à linguagem, porque os DTDs e as folhas de estilo poderiam ser facilmente modificados sem afetar diretamente o conteúdo dos documentos. Em 1986, SGML tornou-se um padrão internacional sancionado pela ISO (*International Organization for Standards*) [41] e passou a ser utilizada

para armazenamento e intercâmbio de informações por todo o mundo.

Com a criação da *World Wide Web*, Tim Berners-Lee do CERN (Laboratório Europeu para Partículas Físicas) começou, em 1989, a desenvolver uma tecnologia para o compartilhamento de informações através do uso de hipertextos (*hyperlinked text documents*). Ele baseou sua nova linguagem na SGML e a chamou de HTML (*HyperText Markup Language*) [60].

Com o desenvolvimento acelerado da *Web*, as limitações da HTML se tornaram aparentes. Sua falta de extensibilidade e sua definição ambígüa frustraram os desenvolvedores e documentos HTML errados se propagaram. Alguns comandos específicos de plataforma foram criados como extensões de HTML pelos vendedores de navegadores a fim de ganhar mercado. Em resposta a esta ameaça à interoperabilidade e escalabilidade da *Web*, a W3C (*World Wide Web Consortium*) [84] criou uma folha de estilo para HTML, a CSS (*Cascading Style Sheets*) [55], que poderia ser utilizada no lugar das marcações proprietárias. A W3C também adicionou uma extensibilidade limitada à HTML. De qualquer modo, foram apenas soluções provisórias. A necessidade de uma nova linguagem que fosse padronizada, totalmente extensível e estruturalmente rígida estava aparente.

Como resultado, XML (*Extensible Markup Language*) [11] foi criado em 1996 por um grupo denominado *XML Working Group* e supervisionado pela W3C. O grupo tinha como base a SGML, uma linguagem volumosa, poderosa e complexa. XML é uma versão simplificada da SGML especialmente projetada para aplicações *Web*. Mantêm-se as funcionalidades necessárias para que possa ser utilizada no ambiente da *Web* e omitem-se as características que tornam SGML tão complexa. As duas principais diferenças entre elas são [20, 35]:

1. fácil implementação - XML é de fácil compreensão, possibilitando o desenvolvimento de um número maior de aplicações e facilitando sua disseminação pela *Web*;

2. DTD não é necessário - apesar de ser altamente recomendado, o DTD pode ser omitido a fim de facilitar e acelerar o processamento dos documentos XML na *Web*, principalmente a parte de transmissão de dados.

Tanto SGML como XML são, na verdade, meta-linguagens, ou seja, linguagens que servem para definir novas linguagens com diferentes aplicações. Por isso, não se pode dizer que XML é uma "melhora" da HTML, pois HTML é uma instância da SGML criada para publicar dados na Web. Trata-se de uma mudança de conceito. Por isso, XML supera alguns limites apresentados pela HTML, tais como [58]:

- não ser extensível, ou seja, não se pode construir novas marcações;

- não comportar dados estruturados;

- tornar difícil a recuperação dos dados, demandando procedimentos complexos implementados por especialistas em linguagens de programação, e os resultados nem sempre são confiáveis;

- não ser um bom formato para o intercâmbio de informações;
- não poder ser usado para o processamento de dados complexos;
- não dizer nada sobre os dados, porque a semântica do conteúdo não é explícita.

Em 1998, XML tornou-se uma especificação formal da W3C. Desde então, vem sendo amplamente utilizado por desenvolvedores que desejam ganhar funcionalidade e interoperabilidade em suas aplicações *Web*.

Seu uso em banco de dados tem crescido bastante, já que a natureza estruturada e não formatada de um documento XML possibilita sua manipulação por aplicações de banco de dados. No futuro, com o contínuo crescimento da *Web*, a tendência é que XML se torne a linguagem universal de representação de dados.

2.1.1 Vantagens do XML

Independência de dados, ou seja, a separação do conteúdo de sua apresentação, é uma das características essenciais do XML. Outras vantagens são apresentadas [51, 63]:

- **Simplicidade**: XML é baseado em texto, para que qualquer pessoa possa criar novos documentos, até nas mais primitivas ferramentas de processamento de texto. XML também fornece um ambiente amigável para os desenvolvedores, pelo menos para os padrões da computação. O rígido conjunto de regras do XML permite a construção de documentos compreensíveis tanto pelos humanos como pelas máquinas. Sua especificação é bastante sucinta, resumindo-se à cerca de trinta páginas, possibilitando aos desenvolvedores o aprendizado rápido da linguagem. Os *parsers*[1] XML também são relativamente fáceis de serem implementados, especialmente os que apenas verificam documentos bem formados (definição na seção 2.1.4).
- **Internacionalização**: Alguns formatos não suportam facilmente a internacionalização, ou seja, tem dificuldade de representar informações contidas em um alfabeto Unicode (*Unicode WorldWide Character Standard*) [81]. XML permite a inclusão de qualquer caractere que siga o padrão Unicode. Dessa maneira, pode-se utilizar caracteres de diversos idiomas na criação de documentos XML.
- **Extensibilidade**: XML pode ser extensível de duas maneiras. Primeiro, os usuários podem criar documentos XML livremente, definindo suas próprias marcações. Segundo, o próprio padrão XML está sendo estendido com padrões adicionais que agregam estilos, *linking* e habilidades de referenciar ao conjunto de funcionalidades do XML.

[1]Um *parser* lê um documento XML, verifica sua sintaxe, comunica quaisquer erros ocorridos e permite o acesso ao conteúdo. O processo todo é denominado *parsing*.

- **Interoperabilidade**: XML pode ser utilizado em diversas plataformas e interpretado por uma grande variedade de ferramentas. Como as estruturas do documento se comportam consistentemente, os *parsers* que as interpretam podem ser construídos a um custo relativamente baixo em várias linguagens.

- **Abertura**: XML é completamente aberto e disponível gratuitamente na *Web*. A W3C regularmente libera relatórios que possibilitam o acompanhamento dos trabalhos em progresso. Além disso, muitas linguagens e padrões relacionados ao XML também estão sendo desenvolvidos abertamente.

- **Experiência**: Apesar do XML ser um padrão novo, há um grupo de "especialistas" aptos para o desenvolvimento de novas ferramentas e novas aplicações. Essas pessoas são, na verdade, desenvolvedores de SGML com vasta experiência e que acompanharam o surgimento do XML. Felizmente, suas habilidades podem ser transferidas para XML, facilitando a disseminação da linguagem.

- **Suporte a buscas complexas**: XML permite a introdução de informações extras aos dados, melhorando o armazenamento. Isso possibilita que usuários recebam resultados mais precisos em suas buscas.

2.1.2 Estrutura dos Documentos XML

Um documento XML é composto por marcações e textos [58]: marcações representam a estrutura lógica do documento, enquanto textos representam o conteúdo.

Em XML, tudo que se encontra entre os sinais de menor (<) e maior (>) é considerado uma marcação, denominada *tag*. Ao contrário da HTML, que é uma linguagem pré-definida, XML permite que novas *tags* sejam criadas. Enquanto as *tags* pré-definidas da HTML servem para especificar os aspectos visuais do documento, em XML elas são utilizadas para estruturar o documento, definir o conteúdo e descrever os dados.

Esta seção faz uma introdução informal às partes que compõem um documento XML. Para uma introdução mais detalhada e completa, ver [78].

Elementos

Os elementos são as marcações mais comuns de um documento XML [85]. Um elemento é composto por uma *tag* inicial e uma *tag* final, na forma `<nome-do-elemento> conteudo-do-elemento </nome-do-elemento>`. O `nome-do-elemento` normalmente identifica a natureza do seu conteúdo. Por exemplo, `<curso> Banco de Dados </curso>`. Além disso, XML é "*case-sensitive*", ou seja, `<nome>`, `<Nome>` e `<NOME>` são *tags* diferentes e não serão tratadas de forma equivalente por nenhum *parser* XML.

Alguns elementos podem ser vazios e neste caso não contêm conteúdo. Se um elemento não é vazio, o conteúdo é delimitado pela *tag* inicial e pela *tag* final, como visto anteriormente. No caso de um elemento vazio, uma forma concisa pode ser utilizada, a "/>" no final da *tag* inicial (<nome-do-elemento/>). O final "/>" na sintaxe modificada indica a um *parser* XML que o elemento é vazio e uma *tag* final não deve ser procurada.

Um elemento não vazio pode possuir subelementos ou caracteres como conteúdo. Pode ocorrer a situação onde um elemento contém subelementos e caracteres simultaneamente. A definição é recursiva, formando uma estrutura de árvore. Essa hierarquia em árvore é rígida, ou seja, um elemento pode conter outros elementos, mas um elemento não pode estar parcialmente contido em outro. Em outras palavras, a seqüência

```
<elemento1>
    <elemento2>
</elemento1>
    </elemento2>
```

é inválida porque os elementos 1 e 2 estão sobrepostos. Além desse aspecto, XML também exige a presença de um **elemento raiz**, ou seja, um elemento que conterá todos os outros do documento.

Atributos

Um elemento pode conter vários atributos, usados para especificar melhor o conteúdo. Atributos [85] são pares do tipo `nome-valor` que ocorrem dentro das *tags* iniciais, logo após o nome do elemento, na forma:

```
<nome-do-elemento nome-do-atributo = "valor-do-atributo">.
```

No exemplo <curso tipo = "integral">, o elemento `curso` possui um atributo denominado `tipo` com valor `integral`.

Em XML, todos os valores de atributos devem estar entre aspas simples ou dupla.

Comentários

Comentários [85] começam com a seqüência "<!- - " e terminam com a seqüência "- ->". Entre as seqüências pode existir qualquer dado, exceto os caracteres "- -".

Os comentários podem ser colocados em qualquer lugar de um documento XML, contanto que não seja dentro de outras *tags*. Eles não fazem parte do conteúdo textual de um documento XML e, por isso, um *parser* XML não precisa repassá-los para a aplicação durante o processamento.

Instruções de Processamento

Um dos objetivos do uso de XML é tentar remover qualquer informação específica sobre o processamento do documento do texto do documento em questão. Apesar disso, ocasionalmente é muito conveniente incluir tais informações. Um exemplo é a necessidade de informar ao processador onde uma nova página começa. Esse tipo de decisão normalmente é deixada para o processador, mas sempre haverá ocasiões onde seja necessário interferir. Uma instrução de processamento inserida no documento é uma maneira simples e efetiva de fazer isso sem interferir em outros aspectos das marcações.

Uma instrução de processamento [78] começa com a seqüência "<?" e termina com a seqüência "?>". Entre as seqüências há duas palavras separadas por espaço: pela convenção, a primeira é o nome de algum processador e a segunda é um dado que será usado pelo processador. A única restrição imposta pelo XML é que a primeira palavra deve ser um nome XML válido. A outra pode ser uma seqüência arbitrária de caracteres, que não inclua a seqüência de término "?>". Exemplo: <?`tex \newpage?>`. O processador que utilizará a informação será o `tex` e a instrução será para iniciar uma nova página (`newpage`).

Instruções de processamento que iniciam com a palavra `xml` são reservadas e servem para a padronização do XML, como em <?`xml version = "1.0"?>`.

Namespaces

Como XML permite que os usuários criem suas próprias *tags*, podem ocorrer "colisões de nomes", isto é, dois elementos semanticamente diferentes com o mesmo nome. Por exemplo, pode-se utilizar o elemento `local` para especificar uma cidade ou onde um quadro se localiza em um museu:

```
<local>Campinas</local>
<local>corredor D</local>
```

Namespaces [20] foram criados para eliminar tais ambigüidades. Os nomes são diferenciados através de prefixos que definem melhor o significado dos elementos ou atributos. Assim, o exemplo acima ficaria da seguinte forma:

```
<endereco:local>Campinas</endereco:local>
<museu:local>corredor D</museu:local>
```

O usuário pode criar seus próprios prefixos. Qualquer seqüência de caracteres pode ser utilizada como prefixo, exceto a palavra **xml**. Para criar um namespace, deve-se associar cada prefixo a uma URI (*Uniform Resource Identifier*), ou seja, uma seqüência de caracteres que seja única.

Exemplo:

```
<elemento xmlns:endereco = "info-endereco"  xmlns:museu = "info-obra">
```

A palavra `xmlns` cria os dois prefixos: `endereco` e `museu`. Os textos `info-endereco` e `info-obra` foram utilizados como URIs. Uma prática comum é o uso de URLs (*Universal Resource Locators*) como URIs, porque os nomes dos domínios usados nas URLs são garantidamente únicos. Por exemplo:

```
<elemento xmlns:endereco = "http://www.unicamp.br"
          xmlns:museu = "http://www.ufmg.br">
```

Essas URLs não são visitadas pelo *parser* XML, apenas representam uma série de caracteres para diferenciar nomes, nada mais.

2.1.3 Exemplo de Documento XML

A figura 2.1 apresenta um documento XML representando uma bibliografia. Os espaços e a indentação foram colocados apenas para melhorar a visualização.

```
<?xml version = "1.0"?>
<bibliografia>
      <livro>
            <titulo>Informatica</titulo>
            <autor id = "1">
                  <nome>Fernando</nome>
                  <sobrenome>Meirelles</sobrenome>
                  <endereco>Sao Paulo</endereco>
            </autor>
            <editor nomecompleto = ``Milton Mira''/>
      </livro>
      <artigo>
            <titulo>Updating XML</titulo>
            <autor id = "2">
                  <nome>Igor</nome>
                  <sobrenome>Tatarinov</sobrenome>
                  <email>igor@cs.washington.edu</email>
            </autor>
            <autor id = ``3''>
                  <sobrenome>Weld</sobrenome>
                  <email>weld@cs.washington.edu</email>
            </autor>
            <autorcontato IDautor = "2"/>
            <ano>2001</ano>
      </artigo>
</bibliografia>
```

Figura 2.1: Exemplo de um documento XML

Na primeira linha há uma instrução de processamento informando que a versão da linguagem utilizada é 1.0. Na linha seguinte aparece o elemento raiz, <bibliografia>. O elemento raiz possui dois subelementos: <livro> e <artigo>. O elemento <livro> inclui os subelementos <titulo>, <autor> e <editor>, este último com o atributo `nomecompleto`. Além de possuir três subelementos (<nome>, <sobrenome> e <endereco>), o elemento <autor> possui um atributo denominado `id`. Já o elemento <artigo> possui

o subelemento <titulo>, dois subelementos <autor>, o subelemento <autorcontato> e um subelemento <ano>. Autores de artigos diferem dos autores de livros pela presença do subelemento <email> ao invés de <endereco>. E o elemento <autorcontato> possui o atributo IDautor, que é uma referência ao autor com id igual a 2 (Igor Tatarinov).

2.1.4 Validade dos Documentos XML

Há duas categorias de documentos XML: os bem formados e os válidos. Quando um *parser* lê o documento XML e verifica que está em conformidade com a estrutura definida para ele (DTD ou XML Schema), o documento é dito válido. Neste caso, os aninhamentos estão corretos, os atributos requisitados são fornecidos, os valores são do tipo correto; ou seja, o documento segue todas as exigências da estrutura.

Se o documento XML falha na conformidade com sua estrutura, mas é sintaticamente correto, ele é bem formado, mas não é válido. Um documento que inclui seqüências de caracteres ou marcações que não podem ser processadas, não é considerado um documento XML. Além disso, um documento XML bem formado deve atender a várias condições, dentre elas:

- deve existir um elemento raiz, que irá incluir todos os outros elementos do documento;
- um elemento deve estar completamente contido no elemento raiz ou em outro elemento, ou seja, não pode haver sobreposição de elementos;
- as *tags* que delimitam os elementos sempre devem estar presentes, na forma normal (<elemento></elemento>) ou concisa (<elemento/>);
- nenhum atributo pode aparecer mais de uma vez na mesma *tag* inicial;
- os valores dos atributos não podem conter referências a entidades externas, como outros tipos de arquivos.

Por definição, um documento válido é também bem formado.

2.2 DTD - Document Type Definition

Uma das vantagens do XML é permitir ao usuário a criação de suas próprias *tags*. Entretanto, para aplicações específicas, as *tags* não podem se encontrar em uma ordem arbitrária. Para definir a estrutura de um documento XML e dar significado ao seu conteúdo, criou-se o DTD (*Document Type Definition*) [78].

DTD é um conjunto de declarações que determinam quais os elementos e atributos são permitidos, os aninhamentos e ordens possíveis para as *tags*, valores de atributos, seus tipos

e padrões, nomes de arquivos externos referenciados e outras meta-informações a respeito do conteúdo de um documento XML. As declarações seguem a notação EBNF (*Extended Backus-Naur Form*) [68].

Há quatro tipos de declarações no DTD, entretanto apenas as duas mais importantes serão detalhadas nas próximas seções: declarações de elementos e declarações de atributos. Para uma explicação mais completa, ver [85].

2.2.1 Declarações de Elementos

Declarações de elementos identificam os nomes dos elementos e a natureza do seu conteúdo. Uma declaração de elementos é da forma:

<!ELEMENT nome-do-elemento (modelo-do-conteudo)>.

A primeira parte da declaração, o `nome-do-elemento`, pode conter caracteres alfanuméricos, hífens, *underscore* ou pontos, mas deve iniciar com uma letra. A segunda parte, o `modelo-do-conteudo`, define o que o elemento pode conter: grupo de elementos, PCDATA (*Parsed Character DATA*), EMPTY ou ANY.

Grupo de elementos

O modelo do conteúdo pode conter mais de um componente. Utilizam-se "conectores" para especificar a ordem em que esses elementos aparecem. Há dois possíveis conectores: as vírgulas, significando uma seqüência ("and"), e as barras verticais (|), significando escolha ("ou").

Por exemplo: <!ELEMENT A (B, C,(D | E))>

Essa declaração identifica o elemento A, que deve conter um elemento B, seguido de um elemento C, seguido ou de um elemento D ou de um elemento E.

Além disso, pode-se determinar a quantidade de vezes que cada subelemento deve aparecer, através da inclusão de "indicadores de ocorrência" no final do elemento. Se um subelemento não possui indicador, significa que ele deve aparecer exatamente uma vez. Os indicadores são:

- opcional (?)

  <!ELEMENT A (B, C?)>

  O elemento A deve possuir um subelemento B, enquanto a presença do elemento C é opcional. Caso C exista, ele deve aparecer apenas uma vez.

- uma ou mais vezes (+)

  <!ELEMENT A (B+, C)>

O elemento *B* pode ocorrer uma ou mais vezes dentro do elemento *A*, seguido de um elemento *C*.

- zero ou mais vezes (*)

  `<!ELEMENT A (B*)>`

  O elemento *A* pode ser vazio ou pode conter uma ou mais vezes o elemento *B*.

  Os indicadores também podem ser utilizados para especificar ocorrência de grupos de subelementos, como o exemplo:

  `<!ELEMENT A (B, C)*>`

  O grupo *(B,C)* pode ocorrer zero ou mais vezes dentro do elemento A.

Modelos de conteúdo complexos podem ser construídos usando conectores e indicadores de ocorrência. Por exemplo:

```
<!ELEMENT fazenda (fazendeiro+, (cachorro* | gato?), porco*, (cabra | vaca)?, (galinha+ | pato*))>
```

PCDATA

Os elementos que podem conter apenas caracteres são declarados usando o símbolo especial #PCDATA.

Exemplo: `<!ELEMENT A (#PCDATA)>`

Elementos que possuem subelementos e PCDATA em sua declaração são chamados de elementos com "conteúdo misto". Por exemplo, o elemento *A* pode conter caracteres ou os subelementos *B* ou *C*: `<!ELEMENT A (#PCDATA|B|C)*>`.

Todos os modelos de conteúdo que incluam PCDATA devem estar na forma: PCDATA aparece primeiro, os subelementos devem estar separados por barras verticais (|), e o grupo todo deve ser opcional (zero ou mais vezes).

EMPTY

EMPTY indica que o elemento não tem nenhum conteúdo, ou seja, nem caracteres nem subelementos. Mas, pode possuir atributos dentro da *tag* inicial.

Forma da declaração: `<!ELEMENT nome-elemento EMPTY>`

ANY

ANY indica que qualquer conteúdo é permitido para aquele elemento.

Forma da declaração: `<!ELEMENT nome-elemento ANY>`

O modelo de conteúdo ANY é algumas vezes útil durante a conversão de documentos, mas deve ser evitado ao máximo porque desabilita toda a verificação do conteúdo desse elemento.

2.2.2 Declarações de Atributos

As declarações de atributos identificam quais os elementos que podem ter atributos, quantos e quais são os atributos, os valores possíveis para eles e o valor padrão de cada um.

Uma declaração de atributos na forma geral:

```
<!ATTLIST nome-elemento atributo1 TIPO VALOR
                        atributo2 TIPO VALOR>
```

Depois de se identificar o elemento através do `nome-elemento`, seguem-se as declarações para cada atributo, formadas por três partes: um nome, um tipo e um valor padrão.

Apesar de existirem pequenas restrições, o usuário está livre para escolher um nome para o atributo. Os nomes só não podem se repetir para um mesmo elemento.

Depois do nome, aparece o tipo do elemento na declaração. Os tipos mais utilizados são:

1. CDATA

   Atributos do tipo CDATA podem receber qualquer texto. Exemplo de uma declaração de atributo do tipo CDATA:

   ```
   <!ATTLIST editor nomecompleto CDATA #REQUIRED>
   ```

2. ID

   O valor de um atributo do tipo ID deve ser um nome que servirá para identificar o elemento. Assim, os valores usados para atributos do tipo ID devem ser sempre diferentes um do outro em um mesmo documento XML. Além disso, cada elemento pode ter somente um atributo do tipo ID.

   Exemplo de uma declaração de atributo do tipo ID:

   ```
   <!ATTLIST autor id ID #REQUIRED>
   ```

3. IDREF ou IDREFS

   O valor de um atributo do tipo IDREF servirá como apontador para algum outro elemento do documento XML. Então, seu valor será o valor do atributo ID do elemento referenciado.

   Exemplo de uma declaração de atributo do tipo IDREF:

   ```
   <!ATTLIST autorcontato IDautor IDREF #IMPLIED>
   ```

   O valor de um atributo do tipo IDREFS pode conter múltiplos valores IDREF separados por espaços. Por exemplo, a declaração

   ```
   <!ATTLIST autorescontato ids IDREFS #IMPLIED>
   ```

   permite que mais de um elemento seja referenciado no documento XML:

   ```
   <autorescontato ids = "1 2"/>
   ```

4. Lista de nomes

   Quando se quer restringir a gama de valores para um atributo, pode-se definir uma lista com os possíveis nomes.

   Por exemplo, o elemento `sobremesa` possui o atributo `fruta` que pode adquirir os valores `abacaxi`, `laranja` ou `morango`.

   `<!ATTLIST sobremesa fruta (abacaxi | laranja | morango) #REQUIRED>`

Além da especificação do tipo, pode-se determinar se os atributos são opcionais ou obrigatórios, e definir um valor padrão para eles. Há quatro possibilidades:

1. #REQUIRED

   O atributo deve possuir um valor explicitamente especificado em cada ocorrência do elemento XML.

2. #IMPLIED

   O atributo não é obrigatório e nenhum valor padrão precisa ser definido. Se o valor não for especificado, o *parser* XML deve prosseguir sem esse atributo.

3. "valor"

   O valor do atributo não é obrigatório para cada elemento do documento XML, mas se não estiver presente, receberá o valor padrão (*default*) especificado.

   Por exemplo, o valor padrão para o atributo `fruta` será `abacaxi`.

   `<!ATTLIST sobremesa fruta (abacaxi | laranja | morango) "abacaxi">`

4. #FIXED "valor"

   Uma declaração de atributo pode especificar que um atributo tenha um valor fixo. Neste caso, o atributo não é requerido, mas se ocorrer, deve ter o valor especificado.

   Exemplo:

   `<!ATTLIST endereco cep #FIXED "13500000">`

2.2.3 DOCTYPE - Document Type Declaration

DTDs podem ser declarados interna ou externamente através do DOCTYPE. Um DTD interno inclui todas as informações necessárias no próprio documento XML, antes do primeiro elemento. A forma geral é: `<!DOCTYPE elemento-raiz [ DTD ]>`.

Exemplo:

```
<?xml version = "1.0"?>
<!-- DTD -->
<!DOCTYPE autor [
<!ELEMENT autor (nome?, sobrenome)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT sobrenome (#PCDATA)>
]>

<!-- XML -->
<autor>
      <nome>Donald</nome>
      <sobrenome>Knuth</sobrenome>
</autor>
```

Um DTD externo corresponde a um arquivo separado que é anexado ao documento XML. As informações contidas em ambos são as mesmas. A *tag* de declaração do DTD externo possui a seguinte forma: `<!DOCTYPE elemento-raiz SYSTEM "nome-arquivo.dtd">`.

No exemplo, o arquivo *definicao.dtd* contém as regras do DTD.

```
<?xml version = "1.0"?>
<!-- DTD -->
<!DOCTYPE autor SYSTEM "definicao.dtd">

<!-- XML -->
<autor>
      <nome>Donald</nome>
      <sobrenome>Knuth</sobrenome>
</autor>
```

2.2.4 Exemplo de DTD

O DTD da figura 2.2 é um dos possíveis DTDs que podem ser construídos para o documento XML da figura 2.1.

O elemento `bibliografia` possui vários elementos `livro` e `artigo`. Ambos possuem um elemento `titulo` e pelo menos um elemento `autor`. O elemento `livro` também possui um elemento `editor`, que por sua vez possui vários elementos `livro`, gerando uma recursão. Já o elemento `artigo` possui um elemento `autorcontato`, que é vazio, e um elemento `ano`. O elemento `autor` possui um elemento `nome` opcional, um elemento `sobrenome` e um elemento `endereco` ou `email`.

2.3 XML Schema

Assim como o DTD, XML Schema [27] descreve o que pode ou não ser incluído em uma classe de documentos XML: quais são os elementos e seus atributos, a hierarquia desses elementos, seus conteúdos, enfim, a estrutura que os documentos devem seguir para serem considerados válidos.

```
<!ELEMENT bibliografia (livro*, artigo*)>
<!ELEMENT livro (titulo, autor+, editor)>
<!ELEMENT artigo (titulo, autor+, autorcontato, ano)>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT autor (nome?, sobrenome, endereco | email)>
<!ATTLIST autor id ID #REQUIRED>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT sobrenome (#PCDATA)>
<!ELEMENT endereco ANY>
<!ELEMENT email (#PCDATA)>
<!ELEMENT editor (livro*)>
<!ATTLIST editor nomecompleto CDATA #REQUIRED>
<!ELEMENT autorcontato EMPTY>
<!ATTLIST autorcontato IDautor IDREF #IMPLIED>
<!ELEMENT ano (#PCDATA)>
```

Figura 2.2: Exemplo de um DTD para o documento XML da figura 2.1

O desenvolvimento do XML Schema começou em 1998 na W3C com o objetivo de estender as funcionalidades do DTD. Tornou-se especificação formal em 2001 e já é considerado, por alguns autores, como o sucessor do DTD. Há pelo menos duas diferenças entre eles [12]:

- XML Schema suporta tipos de dados; e
- XML Schema é escrito em XML, enquanto DTD utiliza a notação EBNF.

Esta seção faz uma breve introdução ao XML Schema. Para uma descrição mais detalhada e completa, ver [18, 82].

2.3.1 Declaração de Elementos

Como XML Schema é um documento XML, a primeira linha contém uma instrução de processamento indicando a versão utilizada: <?`xml version = "1.0"`?>.

A linha seguinte deve indicar que o objetivo do documento é definir um novo XML Schema. Isso é feito através da declaração de um namespace:

<`xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema`">.

O elemento <`xsd:schema`> é a raiz do documento. Após todas as declarações, o XML Schema deve terminar com </`xsd:schema`>. A palavra *xsd* é uma abreviação para "*XML Schema Definition*".

Há dois tipos básicos de elementos: os simples e os complexos.

Elementos Simples

Os elementos `titulo`, `nome`, `sobrenome`, `email` e `ano` no DTD da figura 2.2 são simples porque não contêm subelementos e atributos. Tais elementos são declarados da seguinte maneira em XML Schema:

```
<xsd:element name = "titulo"  type = "xsd:string"/>
<xsd:element name = "nome"  type = "xsd:string"/>
<xsd:element name = "sobrenome"  type = "xsd:string"/>
<xsd:element name = "email"  type = "xsd:string"/>
<xsd:element name = "ano"  type = "xsd:gYear"/>
```

O atributo `name` determina o nome do elemento e o atributo `type` define o tipo de dado que o elemento pode receber. XML Schema suporta mais de 44 tipos de dados; os mais comuns são apresentados na tabela 2.1.

Tipo de dado	*Exemplo*
string	"Hello World"
boolean	true, false, 1, 0
integer	452
decimal	7.08
float	12.56E3, 12, 12560, 0, -0, infinito, -infinito
time	formato: hh:mm:ss.sss
date	formato: YYYY-MM-DD
gYear	formato: YYYY
gMonth	formato: MM
gDay	formato: DD

Tabela 2.1: Tipos de dados para XML Schema

Elementos Complexos

Os elementos que possuem subelementos e atributos são considerados complexos. Por exemplo, os elementos `livro`, `artigo` e `autor` do DTD da figura 2.2 devem ser declarados como elementos complexos. Na figura 2.3, o elemento `autor` é declarado em XML Schema.

```
<xsd:element name = "autor">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name = "nome"  type = "xsd:string"  minOccurs = "0"  maxOccurs = "1"/>
         <xsd:element name = "sobrenome"  type = "xsd:string"/>
         <xsd:choice>
            <xsd:element name = "endereco"  type = "xsd:string"/>
            <xsd:element name = "email"  type = "xsd:string"/>
         </xsd:choice>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Figura 2.3: Declaração do elemento autor em XML Schema

A *tag* `<xsd:complexType>` inicia a declaração de todo elemento complexo. Os conectores de seqüência e escolha do DTD (, e |) são subtituídos pelos elementos `<xsd:sequence>`

e <xsd:choice>, respectivamente. A quantidade de vezes que um elemento pode aparecer é determinada pelos atributos `minOccurs` (valor mínimo) e `maxOccurs` (valor máximo). Essa definição de cardinalidade é bem mais ampla do que a do DTD. Permite que o usuário determine valores precisos para a ocorrência de cada elemento. O valor padrão é 1 para ambos os atributos.

Quando um elemento aparece várias vezes nas declarações de outros elementos, ele pode passar a ser referenciado. Por exemplo, o elemento `titulo` do DTD da figura 2.2 pertence às declarações dos elementos `livro` e `artigo`. Através do atributo `ref` nas declarações dos elementos, pode-se fazer referência ao elemento `titulo`, que deverá ser declarado primeiro. A figura 2.4 apresenta um exemplo.

```
<xsd:element name = "titulo"  type = "xsd:string"/>
<xsd:element name = "livro">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "titulo"/>
         ...
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Figura 2.4: Exemplo do uso de referência em XML Schema

Utilizar referências para um elemento se assemelha à clonagem de um objeto. O elemento é definido primeiro e pode ser duplicado em outro local através do mecanismo de referências. Os dois elementos passam a ser instâncias da mesma classe. Outra alternativa que possibilita a reutilização de declarações e evita inconsistências é a criação de novas classes, denominadas "*named types*". Essas classes são consideradas novos tipos de dados e podem ser utilizadas através do atributo `type` nas declarações de elementos.

Para se definir uma nova classe, basta dar um nome ao elemento `complexType`, através do atributo `name`. A figura 2.5 apresenta a definição e utilização da classe `autorType`.

2.3.2 Declaração de Atributos

As declarações dos atributos devem aparecer no final da declaração do elemento, por imposição da W3C. Um atributo é declarado da seguinte maneira:

```
<xsd:attribute name = "nomecompleto"  type = "xsd:string"  use = "required"/>
```

Os itens que podem aparecer na declaração de um atributo são:

name: determina o nome do atributo;

ref: apontador para outra declaração de atributo;

```
<!-- Definição da classe -->
    <xsd:complexType name = "autorType">
       <xsd:sequence>
          <xsd:element name = "nome"  type = "xsd:string"  minOccurs = "0"  maxOccurs = "1"/>
          <xsd:element name = "sobrenome"  type = "xsd:string"/>
          <xsd:choice>
             <xsd:element name = "endereco"  type = "xsd:string"/>
             <xsd:element name = "email"  type = "xsd:string"/>
          </xsd:choice>
       </xsd:sequence>
    </xsd:complexType>

<!-- Utilização da classe -->
    <xsd:element name = "livro">
       <xsd:complexType>
          <xsd:sequence>
             <xsd:element ref = "titulo"/>
             <xsd:element name = "autor"  type = "autorType"  minOccurs = "0"  maxOccurs = "unbounded"/>
             ...
          </xsd:sequence>
       </xsd:complexType>
    </xsd:element>
```

Figura 2.5: Definição e utilização da classe autorType no XML Schema

type: define o tipo de dado do atributo, que incluem os tipos apresentados na tabela 2.1 acrescentados de:

- ID - define o identificador único de um elemento;
- IDREF - o valor do atributo é o ID de outro elemento;
- IDREFS - o valor do atributo é uma lista de IDs de outros elementos separados por espaços;
- ENTITY - o atributo deverá receber uma entidade como valor, ou seja, um arquivo externo, um conjunto de caracteres especiais ou outra entidade definida pelo usuário;
- ENTITIES - o valor do atributo é uma lista de entidades separadas por espaços;
- NMTOKEN - trata-se de uma forma restrita do tipo *string*, ou seja, o valor do atributo continua sendo texto, mas restringida a apenas uma palavra;
- NMTOKENS - o valor do atributo é uma lista de NMTOKENs separados por espaços;

default: determina um valor padrão para o atributo;

fixed: determina um valor fixo para o atributo;

use: define se o atributo é opcional (`optional`), necessário (`required`) ou proibido (`prohibited`). Utiliza-se proibido quando um "*named type*" é redefinido. Por exemplo, suponha que exista um "*named type*" denominado FORMA com os atributos comprimento, altura, largura, raio e diâmetro. Durante a declaração de um elemento CAIXA, pode-se utilizar o "*named type*" FORMA definindo que os atributos raio e diâmetro são proibidos. Esta idéia assemelha-se ao mecanismo de herança da orientação a objetos.

2.3.3 Exemplo de XML Schema

A figura 2.6 apresenta um exemplo de XML Schema aplicado ao domínio da bibliografia.

O elemento <`bibliografia`> possui dois elementos: um <`livro`> do tipo `livroType` e um <`artigo`> do tipo `artigoType`. Logo após, há a declaração do elemento simples <`titulo`>. A declaração do tipo `livroType`, a seguir, referencia o elemento <`titulo`> declarado anteriormente, cria um elemento <`autor`> do tipo `autorType` e um elemento <`editor`>, do tipo `editorType`. O tipo `artigoType` referencia o elemento <`titulo`>, cria um elemento <`autor`> do tipo `autorType`, um elemento <`autorcontato`> com atributo `IDautor` e um elemento simples <`ano`>. A declaração do tipo `autorType` inclui os elementos simples <`nome`>, <`sobrenome`>, <`endereco`> ou <`email`>, e o atributo `id`. Por fim, a declaração do tipo `editorType` possui um elemento do tipo `livroType`, criando uma recursão, e o atributo <`nomecompleto`>.

2.4 Manipulando Documentos XML

As seções anteriores concentraram-se nos conceitos básicos sobre XML e nos documentos que permitem sua validação (DTD e XML Schema). Esta seção foca a manipulação do conteúdo de um documento XML.

Várias linguagens de consulta XML surgiram desde 1997, entre elas a LOREL, XML-QL, XML-GL, XSL, XQL e Quilt (ver [9] para um estudo comparativo entre elas). Recentemente, uma nova linguagem foi desenvolvida, a XQuery [4]. Ainda não há nenhuma linguagem de consulta XML oficialmente recomendada pela W3C. Entretanto, XQuery encontra-se em um dos últimos níveis de maturidade no processo de recomendação.

Além das linguagens de consultas XML, há outras maneiras de se manipular um documento XML, como a utilização do DOM [38] ou do XSLT [16].

Para a compreensão dos próximos capítulos desta dissertação, serão apresentados conceitos básicos sobre XPath (base para outras tecnologias XML), DOM, XSLT e as linguagens de consulta XML-QL e XQuery.

```
<?xml version = "1.0"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

   <xsd:element name = "bibliografia">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name = "livro"  type = "livroType"  minOccurs = "0"  maxOccurs = "unbounded"/>
            <xsd:element name = "artigo"  type = "artigoType"  minOccurs = "0"  maxOccurs = "unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name = "titulo"  type = "xsd:string"/>

   <xsd:complexType name = "livroType">
      <xsd:sequence>
         <xsd:element ref = "titulo"/>
         <xsd:element name = "autor"  type = "autorType"  minOccurs = "1"  maxOccurs = "unbounded"/>
         <xsd:element name = "editor"  type = "editorType"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name = "artigoType">
      <xsd:sequence>
         <xsd:element ref = "titulo"/>
         <xsd:element name = "autor"  type = "autorType"  minOccurs = "1"  maxOccurs = "unbounded"/>
         <xsd:element name = "autorcontato">
            <xsd:complexType>
               <xsd:attribute name = "IDautor"  type = "xsd:IDREF"  use = "optional"/>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name = "ano"  type = "xsd:gYear"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name = "autorType">
      <xsd:sequence>
         <xsd:element name = "nome"  type = "xsd:string"  minOccurs = "0"  maxOccurs = "1"/>
         <xsd:element name = "sobrenome"  type = "xsd:string"/>
         <xsd:choice>
            <xsd:element name = "endereco"  type = "xsd:AnyElement"/>
            <xsd:element name = "email"  type = "xsd:string"/>
         </xsd:choice>
      </xsd:sequence>
      <xsd:attribute name = "id"  type = "xsd:ID"  use = "required"/>
   </xsd:complexType>

   <xsd:complexType name = "editorType">
      <xsd:sequence>
         <xsd:element name = "livro"  type = "livroType"  minOccurs = "0"  maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name = "nomecompleto"  type = "xsd:string"  use = "required"/>
   </xsd:complexType>

</xsd:schema>
```

Figura 2.6: Exemplo de um XML Schema para o documento XML da figura 2.1

2.4.1 Linguagem XPath

XPath [17] é uma linguagem de expressão usada para selecionar porções de um documento XML. No XPath, um documento XML é modelado como uma árvore ordenada usando sete tipos de nós: raiz, elemento, atributo, texto, namespace, instrução de processamento e comentário. Por simplicidade, somente os quatro tipos mais utilizados serão enfatizados nesta seção. Um exemplo simples na figura 2.7 ilustra os tipos de nós, usando os símbolos apresentados em [44].

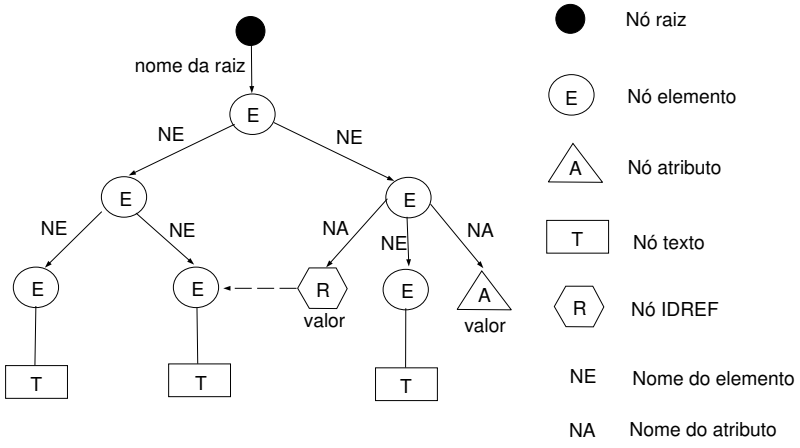

Figura 2.7: Os quatro tipos mais utilizados de nós XPath

- Nó raiz: trata-se de um nó virtual que aponta para o verdadeiro elemento raiz do documento XML.
- Nó elemento: um elemento XML é representado por um nó com formato elíptico. O relacionamento entre nós pai e filho é representado por uma aresta dirigida e rotulada com o nome do elemento. Os nós do tipo elemento podem possuir n ($n \geq 0$) outros nós elemento ou nós texto como filhos. Além disso, podem possuir um conjunto de nós do tipo atributo.
- Nó atributo: o nó triangular representa um atributo do documento XML. O relacionamento com o nó elemento pai é representado por uma aresta dirigida e rotulada com o nome do atributo e seu valor deve aparecer logo abaixo do triângulo. Nós atributo não podem possuir filhos. O nó IDREF é um nó atributo com um símbolo diferenciado (um polígono) para facilitar a visualização. Como é utilizado para fazer referências no próprio documento XML, o nó é acompanhado por uma aresta tracejada que aponta para o elemento referenciado.

- Nó texto: são as folhas da árvore, portanto, não possuem filhos. Representados por retângulos, são acoplados aos nós elemento por arestas não rotuladas.

Os nós seguem uma ordem conhecida como "ordem do documento" (*document order*). Essa ordem é determinada através de uma busca em profundidade da esquerda para a direita, ou seja, de acordo com a ocorrência das *tags* iniciais no documento XML. Os nós atributo de um elemento ocorrem antes dos nós de seus filhos, mas sua ordem relativa é dependente de implementação. A ordenação iniciada ao contrário é conhecida como "ordem reversa do documento" (*reverse order*).

Nas próximas discussões, a expressão *grafo XML* será utilizada referindo-se ao modelo de dados XPath. A figura 2.8 mostra o grafo XML do documento apresentado na figura 2.1. Os números que aparecem dentro dos nós representam a ordem do documento.

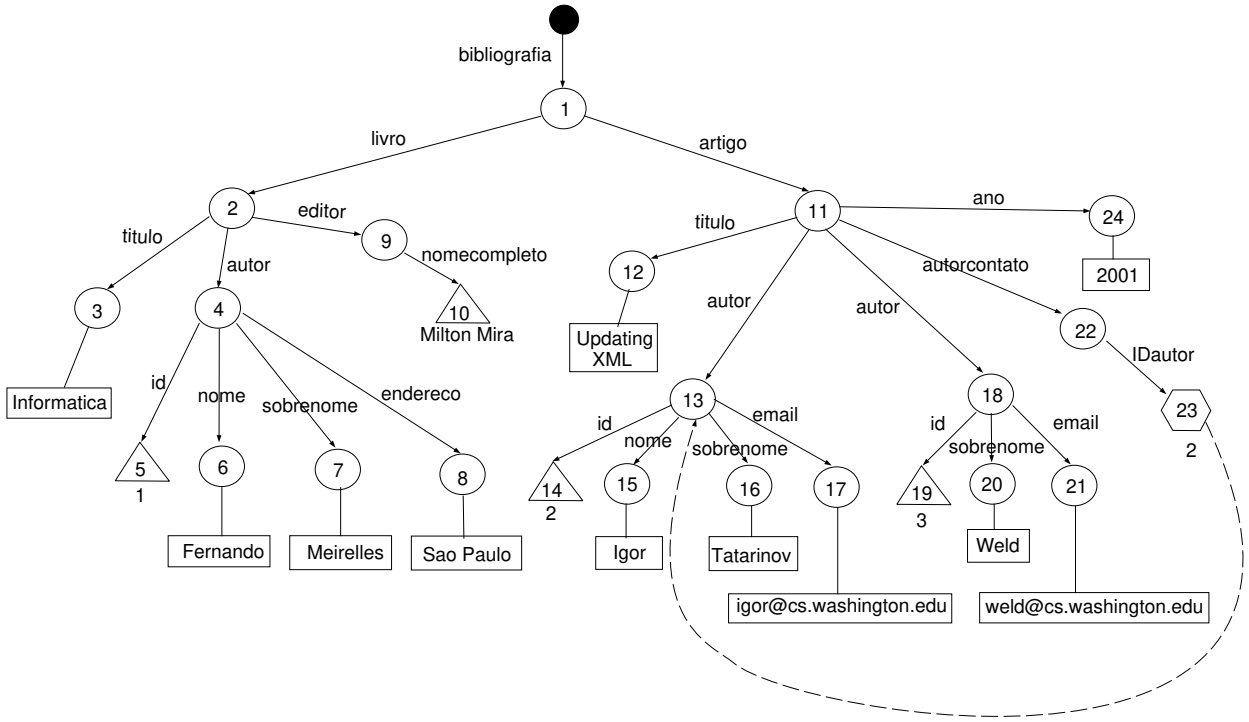

Figura 2.8: Grafo correspondente ao documento XML da figura 2.1

Expressões em XPath comportam-se da mesma maneira que expressões regulares, exceto que elas operam em nós XML ao invés de caracteres. Tanto XPath como expressões regulares

possuem várias barras e parênteses. Apesar disso, são simples de se entender. XPath é mais fácil de ser compreendido através de exemplos, como os que seguem:

/bibliografia/artigo/titulo Seleciona todos os elementos <titulo> que são filhos de um elemento <artigo> que por sua vez é filho de um elemento <bibliografia>, que é a raiz do documento. O resultado pode ser múltiplos elementos <titulo>.

/bibliografia/*/titulo Retorna os elementos <titulo> filhos de qualquer elemento que seja filho do elemento <bibliografia>. O asterisco se associa a qualquer elemento.

//titulo Seleciona todos os elementos <titulo> que aparecem em qualquer lugar do documento XML. A barra dupla indica profundidade arbitrária.

count(//artigo) Retorna o número de elementos <artigo> que aparecem no documento.

//artigo[ano = 2000] Retorna todos os elementos <artigo> que possuam um elemento <ano> cujo valor é 2000. Os colchetes delimitam um "predicado" que age como um filtro para a seleção do resultado.

//autor[@id > 2] Retorna todos os elementos <autor> que possuam um atributo id com valor superior a 2. O sinal @ indica que se trata de um atributo. Note como um atributo pode ser tratado como um inteiro para a comparação em XPath.

//autor[@id] Retorna todos os elementos <autor> que possuam um atributo id com qualquer valor.

//autor/@id Retorna todos os atributos id anexados aos elementos do tipo <autor>.

(//livro | //artigo) [titulo = $tit] Retorna todos os elementos <livro> ou <artigo> que possuam um elemento filho <titulo> com o valor igual a variável $tit.

//ano[. = 2000] Retorna todos os elementos <ano> que possuam um valor igual a 2000. O "." na expressão representa o nó atual, que é similar ao ponteiro "*this*" nas linguagens orientadas a objetos.

(//artigo)[1]/text() Retorna os nós texto do primeiro elemento <artigo> do documento.

2.4.2 XSLT - Extensible Stylesheet Language Transformations

XSLT [16] serve para transformar um documento XML em outro documento diferente, no formato XML, HTML ou texto. XSLT usa XPath para identificar os nós do documento XML que formarão o documento resultante.

Um documento XSLT é um documento XML que possui o elemento raiz <xsl:stylesheet> e uma coleção de regras de *template*. Cada regra determina qual elemento a ser encontrado no documento XML (especificado pelo atributo match), as condições a serem avaliadas e qual resultado será retornado. A figura 2.9 apresenta um documento XSLT para recuperar os títulos de artigos publicados em 2001.

```
<?xml version = "1.0"?>
<xsl:stylesheet>
     <xsl:template match = "/bibliografia">
        <resultados>
           <xsl:apply-templates/>
        </resultados>
     </xsl:template>
     <xsl:template match = "artigo[ano = 2001]">
        <resultado>
           <xsl:value-of select = "/titulo">
        </resultado>
     </xsl:template>
</xsl:stylesheet>
```

Figura 2.9: Documento XSLT para recuperar títulos de artigos publicados em 2001

O elemento <xsl:template match = "/bibliografia"> seleciona o elemento raiz do documento XML. Nesta regra, cria-se o elemento <resultados> no documento resultante. O elemento <xsl:apply-templates/> é usado para determinar que as próximas regras de *template* sejam aplicadas aos filhos da raiz. O elemento <xsl:template match = "artigo [ano = 2001]"> verifica se o subelemento <ano> do elemento <artigo> é igual a 2001. Para os nós compatíveis, criam-se elementos <resultado> no documento resultante. O elemento <xsl:value-of> especifica que o texto do elemento <resultado> é o conteúdo do subelemento <titulo>.

A figura 2.10 apresenta o documento resultante quando o documento XSLT da figura 2.9 é aplicado ao documento XML da figura 2.1.

```
<resultados>
      <resultado> Updating XML</resultado>
</resultados>
```

Figura 2.10: Documento resultante da aplicação do documento XSLT da figura 2.9 no documento XML da figura 2.1

2.4.3 DOM - Document Object Model

DOM [14, 38] é uma API (*Applications Programming Interface*) que modela um documento XML como uma árvore e define uma forma de acesso e manipulação dos seus objetos. Assim,

DOM permite que programas acessem e manipulem dinamicamente a estrutura e conteúdo dos documentos XML.

No modelo de dados do DOM, os elementos e atributos de um documento XML são representados como nós retangulares e os textos como nós retangulares com bordas arredondadas, como pode ser visto na figura 2.11.

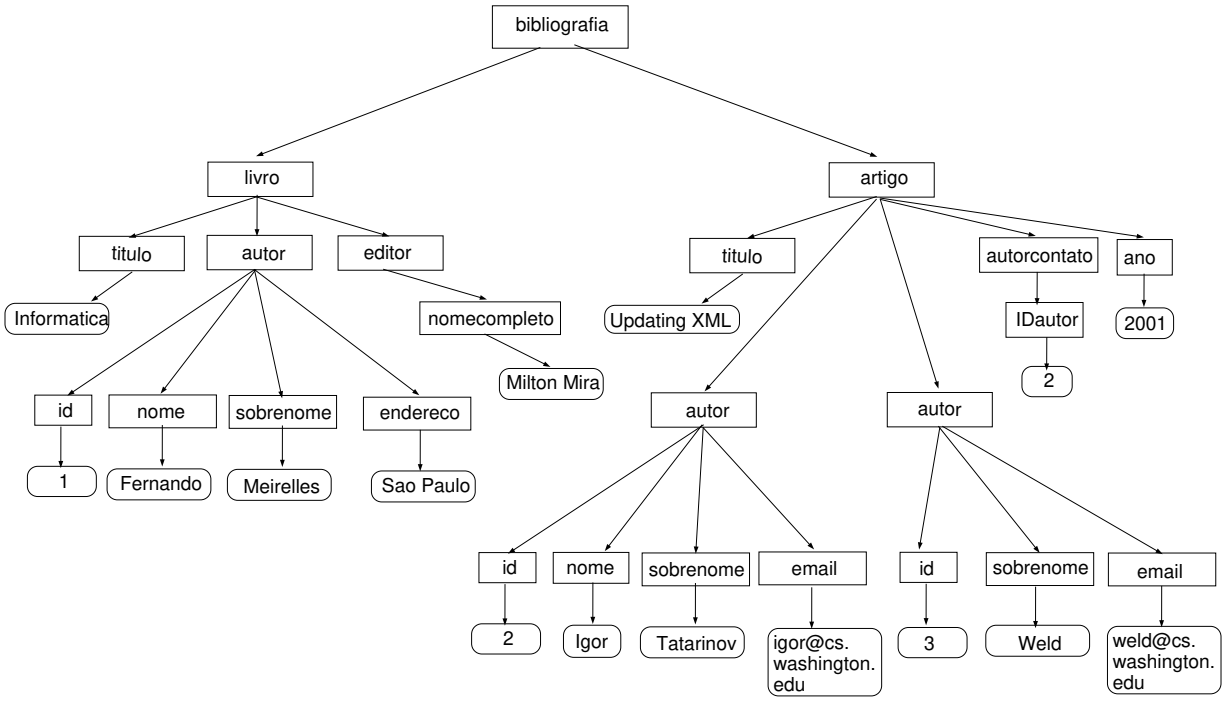

Figura 2.11: Árvore DOM correspondente ao documento XML da figura 2.1

O modelo de dados do DOM é similar, mas não igual ao modelo de dados do XPath. As diferenças entre eles incluem:

- DOM possui uma quantidade maior de tipos de nós, como nós para fragmentos do documento; e
- XPath não representa declarações do DTD, caso haja um no início do documento XML.

Após o *parsing* e criação da árvore DOM, os nós podem ser acessados através de funções, como as da tabela 2.2.

Função	*Descrição*
getDocumentElement()	Recupera a raiz do documento XML
getChildNodes()	Recupera os nós filho do elemento atual
getFirstChild()	Recupera o primeiro filho do elemento atual
getNextSibling()	Recupera o nó irmão do elemento atual
getNodeName()	Recupera o nome do elemento
getNodeValue()	Recupera o conteúdo do elemento
setNodeValue()	Atualiza o conteúdo do elemento
appendChild()	Adiciona um novo nó na árvore
removeChild()	Remove um nó da árvore
replaceChild()	Substitui um nó da árvore

Tabela 2.2: Funções para manipulação dos nós da árvore DOM

2.4.4 XML-QL

A linguagem XML-QL [21, 22] utiliza duas cláusulas, WHERE e CONSTRUCT. Na cláusula WHERE, os elementos do documento são selecionados e na cláusula CONSTRUCT, eles são reagrupados para formar um documento XML como resultado.

Suponha que se deseja saber quais os títulos e editores dos livros escritos pelo autor "Fernando Meirelles". A figura 2.12 apresenta a consulta escrita em XML-QL, considerando o documento XML da figura 2.1.

```
WHERE <bibliografia>
          <livro>
                <titulo>$t</titulo>
                <autor>
                      <nome>Fernando</nome>
                      <sobrenome>Meirelles</sobrenome>
                </autor>
                <editor nomecompleto = $e/>
          </livro>
      </bibliografia> IN "www.unicamp.br/bib.xml"
CONSTRUCT <resultado>
             <titulo>$t</titulo>
             <editor>$e</editor>
          </resultado>
```

Figura 2.12: Exemplo de uma consulta em XML-QL

Esta consulta procura cada elemento <livro> do documento XML encontrado em www.unicamp.br/bib.xml que possui pelo menos um elemento <titulo>, um elemento <autor> e um elemento <editor>. Além disso, o subelementos <nome> e <sobrenome> devem ser iguais a "Fernando" e "Meirelles", respectivamente. Para cada elemento <livro> encontrado, a variável $t recebe o valor do elemento <titulo> e a variável $e recebe o valor do atributo nomecompleto. A cláusula CONSTRUCT estrutura o resultado da consulta, especificando o local onde cada variável deverá aparecer. O documento resultante dessa

consulta é apresentado na figura 2.13.

```
<resultado>
      <titulo>Informatica</titulo>
      <editor>Milton Mira</editor>
</resultado>
```

Figura 2.13: Documento resultante da consulta XML-QL da figura 2.12

2.4.5 XQuery

A linguagem XQuery [4], na verdade, começou como Quilt [13], uma linguagem desenvolvida por Jonathan Robie, Don Chamberlin, e Daniela Florescu. Os autores foram influenciados por outras linguagens como XQL, XML-QL e SQL. Como mencionado anteriormente, XQuery será, em breve, a linguagem de consulta XML oficial.

XQuery utiliza fortemente o XPath, tanto que as versões 1.0 do XQuery e 2.0 do XPath estão sendo desenvolvidas pelo mesmo grupo da W3C e suas especificações são entrelaçadas. Além das expressões em XPath, XQuery proporciona a construção de consultas mais complexas, através de expressões FLWOR.

As expressões FLWOR (pronunciada "*flower*") são os blocos construtores da linguagem XQuery. O nome originou-se das palavras-chave FOR, LET, WHERE, ORDER BY e RETURN. A cláusula FOR provê um mecanismo de iteração entre uma seqüência de valores. Por exemplo:

`for $x in (1 to 3) return ($x, $x + 10)`

A cláusula FOR primeiramente avalia a expresssão que segue a palavra "in". Para cada item da seqüência, a variável `$x` assume seu valor. A cláusula RETURN no final da expressão indica o que deve ser retornado. Para cada valor que `$x` assume, a cláusula RETURN é avaliada. Para o exemplo, o resultado é:

1, 11, 2, 12, 3, 13

A cláusula LET permite a designação de variáveis. A figura 2.14 apresenta um exemplo.

```
LET $artigo := document("bibliografia.xml")/bibliografia/artigo
FOR $autor in $artigo/autor
RETURN <resultado> {$autor/sobrenome} </resultado>
```

Figura 2.14: Consulta XQuery com a cláusula LET

Nesta consulta, a função `document` retorna a raiz do arquivo do documento da figura 2.1 e a variável `$artigo` recebe os elementos <`artigo`>, filhos do elemento <`bibliografia`>.

A cláusula FOR percorre cada subelemento `<autor>` da variável `$artigo` e retorna o sobrenome do autor. O resultado da consulta é apresentado na figura 2.15.

```
<resultado>
        <sobrenome>Tatarinov</sobrenome>
        <sobrenome>Weld</sobrenome>
</resultado>
```

Figura 2.15: Documento resultante da consulta XQuery da figura 2.14

As tuplas resultantes ainda podem ser selecionadas com a cláusula WHERE e ordenadas com a cláusula ORDER BY. Neste caso, a cláusula RETURN é avaliada uma vez para cada tupla sobrevivente da cláusula WHERE e ordenada de acordo com a cláusula ORDER BY. Por exemplo, uma consulta que seleciona os autores com o atributo `id` maior do que 1 pode ser escrita da maneira apresentada na figura 2.16.

```
LET $artigo := document("bibliografia.xml")/bibliografia/artigo
FOR $autor in $artigo/autor
WHERE $autor/@id > 1
ORDER BY $autor/@id
RETURN <resultado> {$autor} </resultado>
```

Figura 2.16: Consulta XQuery para recuperar autores com atributo id maior do que 1

A consulta avaliada sobre o documento XML da figura 2.1 é apresentada na figura 2.17.

```
<resultado>
        <autor id = "2">
                <nome>Igor</nome>
                <sobrenome>Tatarinov</sobrenome>
                <email>igor@cs.washington.edu</email>
        </autor>
        <autor id = "3">
                <sobrenome>Weld</sobrenome>
                <email>weld@cs.washington.edu</email>
        </autor>
</resultado>
```

Figura 2.17: Documento resultante da consulta XQuery da figura 2.16

Além das expressões FLWOR, a linguagem XQuery também suporta funções e operadores para facilitar comparações entre valores, manipulações de caracteres, cálculos matemáticos, conversão de tipos de dados, etc. XQuery também possibilita a definição de novas funções pelos desenvolvedores.

Nos capítulos 3 e 4, vários métodos de armazenamento e recuperação de documentos XML serão analisados. Eles envolvem diferentes idéias, desde técnicas *ad hoc* até algoritmos sofisticados de tradução de consultas. Os conceitos deste capítulo serão necessários para a melhor compreensão dos métodos e exemplos posteriores.

Capítulo 3

Armazenamento de Documentos XML

O problema do armazenamento de documentos XML em banco de dados relacionais envolve duas fases. A primeira é a criação de um esquema relacional adequado, que permita a execução das consultas com maior eficiência possível. Nesta fase, define-se como os elementos e atributos de um documento XML serão armazenados nas tabelas, um processo denominado mapeamento.

A segunda fase é a tradução de consultas escritas em linguagem XML[1] para consultas SQL, explorando os benefícios do esquema relacional proposto.

Este capítulo apresenta uma visão geral das técnicas publicadas na área de armazenamento dos documentos XML. A seção 1 apresenta a classificação, as seções de 2 a 8 explicam como os métodos funcionam, as seções 9 e 10 apresentam análises comparativas e a seção 11 fecha o capítulo apresentando as conclusões e algumas questões em aberto.

3.1 Classificação

Como a existência de uma estrutura (DTD ou XML Schema) para um documento XML não é obrigatória, a primeira classificação divide as técnicas em dois grandes grupos: os mapeamentos independentes de estrutura e os mapeamentos dependentes de estrutura.

Os mapeamentos independentes de estrutura ainda podem ser subdivididos da seguinte forma:

- Mapeamentos dependentes do conteúdo - são aqueles que definem o esquema relacional de acordo com o conteúdo do documento XML. Se o conteúdo muda, o esquema relacional deve ser redefinido.

[1]Utilizaremos o termo linguagem XML para denotar uma linguagem genérica de consulta sobre documentos XML; linguagens específicas serão mencionadas nos métodos a serem analisados.

- Mapeamentos baseados em arestas - dado um grafo XML, estes mapeamentos armazenam as arestas do grafo.
- Mapeamentos baseados em caminhos - estes mapeamentos preocupam-se em armazenar os caminhos entre o nó raiz e algum outro nó do grafo XML.
- Mapeamentos baseados em intervalos - são aqueles que utilizam a posição de um elemento, atributo ou texto dentro do documento XML. O intervalo pode ser contado por caracteres no texto ou por palavras, como será visto posteriormente.

Para os mapeamentos dependentes de estrutura, faz-se a seguinte subclassificação:

- Mapeamentos dependentes do DTD - são aqueles que mapeiam as declarações de elementos e as declarações de atributos para um esquema relacional.
- Mapeamentos dependentes do XML Schema - são aqueles que utilizam os benefícios do XML Schema, como a definição de tipos de dados, para a criação do esquema relacional.

Os mapeamentos utilizados em produtos comerciais são classificados como um grupo especial, devido à falta de informações detalhadas sobre eles.

A figura 3.1 apresenta a classificação completa.

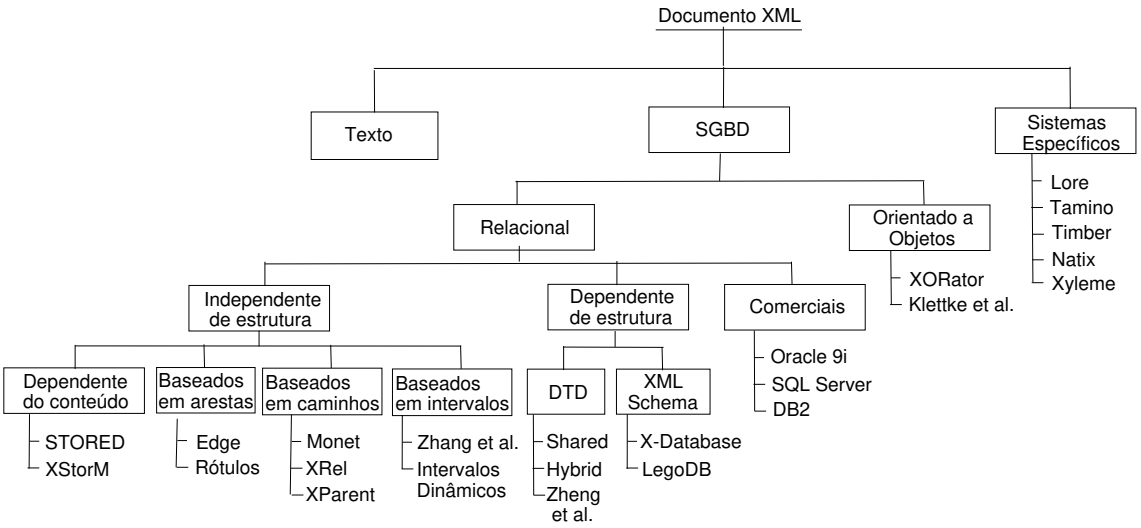

Figura 3.1: Classificação dos métodos de armazenamento dos documentos XML

Além das semelhanças que possibilitam a classificação apresentada na figura 3.1, nota-se uma evolução cronológica dos métodos de armazenamento de documentos XML. Em 1999 e 2000 surgiram métodos mais *ad hoc*, como os dependentes de conteúdo e os baseados em arestas. Os primeiros métodos envolvendo DTD também surgiram nessa época. Os anos de 2001 e 2002 marcam o aparecimento dos métodos baseados em caminhos, do uso do XML Schema na criação das tabelas e dos primeiros produtos comerciais com suporte a XML. Em 2003, os métodos passam a adotar técnicas sofisticadas como o uso de intervalos dinâmicos e de otimização. A figura 3.2 apresenta a linha do tempo do desenvolvimento dos métodos de armazenamento de documentos XML em banco de dados relacionais. Os detalhes de cada método serão abordados nas seções seguintes.

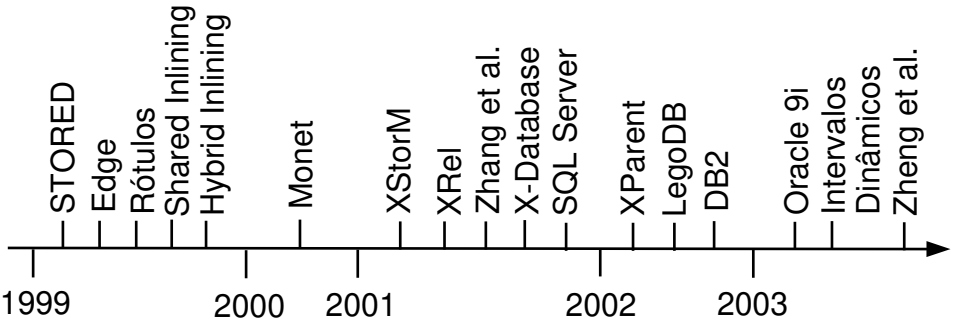

Figura 3.2: Linha do tempo dos métodos de armazenamento dos documentos XML

3.2 Mapeamentos Dependentes do Conteúdo

3.2.1 Abordagem STORED

O trabalho desenvolvido por Deutsch et al. [23], propõe o mapeamento do modelo de dados semi-estruturado para o modelo de dados relacional através de uma linguagem específica, denominada STORED (*Semistructured TO Relational Data*).

Primeiramente, os dados semi-estruturados são modelados como um grafo (semelhante ao da figura 2.8) e faz-se uma análise de seu conteúdo. Pretende-se, nessa fase, definir quais são os padrões mais freqüentes encontrados no grafo, ou seja, as subárvores que cobrem melhor todos os elementos e atributos do documento. Para isso, utiliza-se uma extensão do algoritmo *WL* criado por Wang e Liu [86], baseado em técnicas de *data mining*. Por exemplo, suponha que, aplicando-se o algoritmo WL ao grafo da figura 2.8, um dos padrões encontrados seja o padrão apresentado na figura 3.3.

```
bibliografia.artigo:   {titulo[1], autor[*]:{id[1], sobrenome[1], email[1]},
                        autorcontato[1]:{IDautor[1]}, ano[1]}
```

Figura 3.3: Padrão encontrado pelo algoritmo WL no grafo XML da figura 2.8

Este padrão possui a seguinte notação: subelementos aparecem delimitados pelas chaves e a quantidade de cada elemento aparece entre colchetes. O padrão determina que a maioria dos elementos do tipo `artigo` possuem um `titulo`, vários elementos do tipo `autor`, um elemento `autorcontato` e um elemento `ano`. Os elementos do tipo `autor` possuem um elemento `id`, um elemento `sobrenome` e um elemento `email`. O elemento `autorcontato` possui um elemento `IDautor`. Note que o elemento `nome` está ausente no autor, pois trata-se de um elemento que não aparece com freqüência no grafo XML da figura 2.8, ou seja, não está nas duas subárvores de autor.

O segundo passo da abordagem é converter cada padrão para uma ou mais consultas STORED, contendo as cláusulas FROM, WHERE e STORE. Essas consultas determinam como os elementos e atributos são armazenados nas tabelas. A cláusula FROM define quais elementos devem ser avaliados, a partir da raiz de uma subárvore. Por exemplo, a seqüência iniciada por `FROM bibliografia.artigo: $X` determina que se avalie a subárvore de cada elemento `artigo`. Todas as outras variáveis que aparecem na cláusula FROM devem se associar com os elementos descendentes na subárvore avaliada em um determinado momento. A cláusula WHERE especifica uma condição e a cláusula STORE define como as variáveis da cláusula FROM são armazenadas nas tabelas.

Seguindo o exemplo, o padrão geraria as duas consultas STORED da figura 3.4. A tabela R2 seria criada porque o elemento `autor` é multivalorado (*).

```
C1 =   FROM bibliografia.artigo:  $X
       {titulo:  $T, autorcontato:  {IDautor:  $N}, ano:  $A}
       STORE R1($X, $T, $N, $A)

C2 =   FROM bibliografia.artigo:$X.autor:  $Y
       {id:  $I, sobrenome:  $S, email:  $E}
       KEY $Y
       STORE R2($X, $Y, $I, $S, $E)
```

Figura 3.4: Consultas STORED para o padrão da figura 3.3

As tabelas R1 e R2 armazenam os números dos nós dos elementos `artigo` e `autor` associados às variáveis $X e $Y respectivamente. Além disso, armazenam os valores dos nós que se associam às outras variáveis da cláusula FROM. A cláusula KEY da consulta C2 determina explicitamente o acréscimo da variável $Y à chave primária da tabela R2.

Para garantir que nenhuma informação seja perdida, os elementos que não foram armazenados no esquema proposto são modelados como grafos denominados "*overflow*". Estes grafos podem ser armazenados em qualquer repositório de dados semi-estruturados. Os mapeamentos são realizados através de consultas STORED de *overflow*. Por exemplo, o elemento `nome` pertencente ao elemento `autor` não aparece no padrão da figura 3.3, entretanto está presente no grafo XML da figura 2.8. Essa informação será armazenada através de uma consulta STORED de *overflow*, apresentada na figura 3.5:

```
O1 =   FROM bibliografia.artigo.autor:  $Y
       {id:  $I, $N: _, sobrenome:  $S, email:  $E}
       WHERE $N = nome
       OVERFLOW G1($N)
```

Figura 3.5: Consulta STORED de *overflow*

Para cada variável $N que for encontrada com o rótulo **nome** serão armazenadas: (1) a variável $Y, referente ao nó `autor`, para determinar a qual subárvore se refere, (2) a aresta (Y, N, Valor). A quantidade de mapeamentos de *overflow* pode ser minimizada quando alguns parâmetros são definidos para o algoritmo WL, como a quantidade mínima que uma subárvore deve aparecer no grafo XML para se tornar um padrão. Um valor menor pode garantir maior cobertura do grafo.

STORED permite consultas sobre o documento semi-estruturado, como a da figura 3.6 para recuperar títulos de artigos publicados em 2001.

```
Q =   SELECT $T
      FROM bibliografia.artigo:  $X
      {titulo:  $T, ano:$A}
      WHERE $A = 2001
```

Figura 3.6: Consulta STORED para recuperar artigos publicados em 2001

Para a avaliação da consulta Q, STORED cria uma "regra de inversão" para cada consulta gerada para fazer o mapeamento (consultas C1 e C2 da figura 3.4). Uma regra de inversão especifica como os campos da tabela devem ser hierarquicamente distribuídos para formar um grafo. A figura 3.7 mostra a regra de inversão para a consulta C1 da figura 3.6.

A regra de inversão lê a tabela R1 e constrói um grafo iniciando pelo nó bibliografia. Para cada artigo encontrado, criam-se os nós titulo, autorcontato (com IDautor como filho) e ano. As palavras iniciadas pela letra s (s-artigo, s-titulo, etc.) são na verdade funções para garantir que os dados retornados pertencem ao mesmo artigo (aquele associado à variável $X no momento).

```
FROM R1($X, $T, $N, $A)
CONSTRUCT bibliografia:  s-bibliografia()
       {artigo:  s-artigo($X)
              {titulo:  s-titulo($X),
               autorcontato:  s-autorcontato($X)
                     {IDautor:  s-IDautor($X)},
               ano:  s-ano($X)}}
```

Figura 3.7: Regra de inversão para a consulta C1

O objetivo é reconstruir o grafo XML, unindo todas as regras de inversão. Após a reconstrução do grafo, a consulta Q pode ser avaliada sobre ele, percorrendo a subárvore especificada na cláusula FROM. A tuplas que satisfazem a condição definida na cláusula WHERE são retornadas.

As desvantagens dessa abordagem são:

- geração de tabelas com grande quantidade de valores nulos, principalmente se o grafo do documento XML for totalmente desestruturado, ou seja, sua estrutura é tão irregular que vários padrões diferentes são encontrados; e
- grafos desestruturados e atualizações no documento XML geram mapeamentos de *overflow*, afetando o desempenho das consultas devido à necessidade de se integrar tabelas relacionais com repositórios de dados semi-estruturados. Além disso, se houver grande quantidade de mapeamentos de *overflow*, o processo todo deve ser reiniciado.

3.2.2 Abordagem XStorM

A idéia utilizada na técnica de Wang et al. [87] é muito similar ao da abordagem STORED: faz-se uma busca no documento XML pelos padrões mais freqüentes e criam-se tabelas para mapeá-los. Os elementos não mapeados são armazenados em tabelas *overflow*.

Diferente do STORED, que possui uma linguagem específica para o mapeamento, XStorM utiliza o DOM (seção 2.4.3), um padrão aberto. Através de um algoritmo de *data mining*, selecionam-se as subárvores que aparecem mais vezes na árvore DOM. Essas subárvores são chamadas *k-tree-expression*, onde k denota o número de folhas da subárvore. A figura 3.8 apresenta uma *6-tree-expression* (destacada em negrito) para a árvore DOM da figura 2.11.

Cada *k-tree-expression* é diretamente mapeada para uma tabela: o número do nó raiz passa a ser a chave primária e as folhas passam a ser os campos. Por exemplo, a *6-tree-expression* da figura 3.8 será mapeada para a tabela `Livro = (livroid, titulo, id, nome, sobrenome, endereco, nomecompleto)`. O que não for mapeado para o esquema relacional, será armazenado em uma tabela denominada *overflow* com uma chave estrangeira

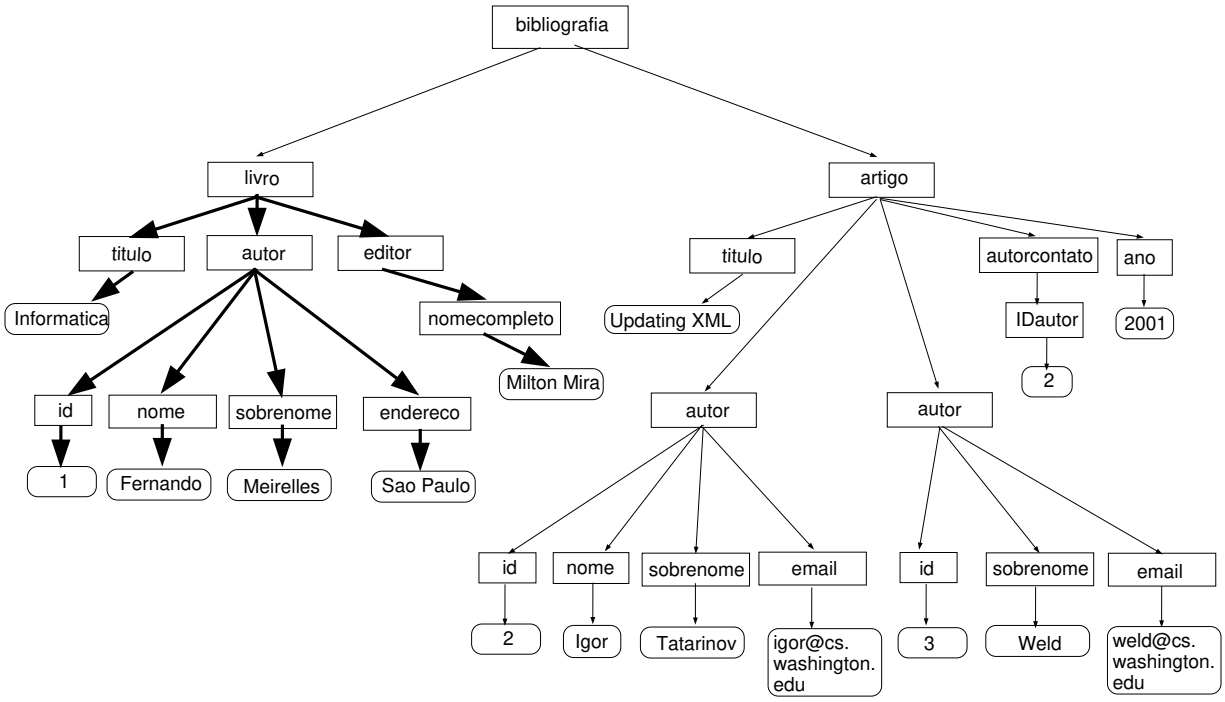

Figura 3.8: Árvore DOM com uma 6-tree-expression

identificando a subárvore a que pertence. A utilização de tabelas relacionais para *overflow* evita o problema de integração de dados entre diferentes sistemas que acontecia no STORED.

A desvantagem dessa abordagem é a utilização do DOM, pois toda a estrutura criada para um documento XML é armazenada em memória. Documentos grandes podem afetar o desempenho ou até mesmo impossibilitar o processo de mapeamento. Pelo artigo de Wang et al. [87], não foi possível determinar se valores multivalorados são mapeados para novas tabelas e nem como as consultas são realizadas.

3.3 Mapeamentos Baseados em Arestas

3.3.1 Abordagem Edge

Há várias maneiras de se armazenar as arestas e folhas de um grafo XML em esquemas relacionais. O trabalho desenvolvido por Florescu e Kossmann [31, 32] apresenta quatro modos para se armazenar as arestas e dois modos de se armazenar folhas, totalizando oito esquemas diferentes. Apenas os dois mais relevantes serão descritos.

A abordagem mais simples é armazenar todas as arestas em uma única tabela, denominada tabela **Edge**, que possui o seguinte esquema:

```
Edge (Source, Target, Number, Label, Flag, Value)
```

Armazena-se o nó origem de uma aresta no campo `Source` e o nó destino no campo `Target`. O campo `Number` guarda a ordem da aresta em relação ao nó pai. O campo `Label` armazena o rótulo da aresta, enquanto o valor do campo `Flag` indica se o nó de destino é uma referência ou uma folha. O campo `Value` armazena o valor do elemento, normalmente do tipo #PCDATA. A figura 3.9 mostra a tabela Edge populada para o grafo da figura 2.8.

Source	*Target*	*Number*	*Label*	*Flag*	*Value*
0	1	1	bibliografia	-	-
1	2	1	livro	-	-
1	11	2	artigo	-	-
2	3	1	titulo	string	"Informatica"
2	4	2	autor	-	-
2	9	3	editor	-	-
4	5	1	id	string	"1"
4	6	2	nome	string	"Fernando"
4	7	3	sobrenome	string	"Meirelles"
4	8	4	endereco	string	"Sao Paulo"
9	10	1	nomecompleto	string	"Milton Mira"
11	12	1	titulo	string	"Updating XML"
11	13	2	autor	-	-
11	18	3	autor	-	-
11	22	4	autorcontato	-	-
11	24	5	ano	string	"2001"
13	14	1	id	string	"2"
13	15	2	nome	string	"Igor"
13	16	3	sobrenome	string	"Tatarinov"
13	17	4	email	string	"igor@cs.washington.edu"
18	19	1	id	string	"3"
18	20	2	sobrenome	string	"Weld"
18	21	3	email	string	"weld@cs.washington.edu"
22	23	1	IDautor	ref	"2"

Figura 3.9: Tabela Edge para o grafo XML da figura 2.8

Por exemplo, a tupla (`13, 16, 3, sobrenome, string, "Tatarinov"`) descreve a terceira aresta que parte do nó 13. A aresta possui um rótulo (*sobrenome*) e um valor ("Tatarinov").

O algoritmo de tradução de consultas XML-QL para SQL não foi detalhado por Florescu e Kossmann em [31]. Entretanto, discutem-se como algumas classes de consultas XML podem ser traduzidas. Em geral, a reconstrução de um fragmento do documento envolve muitas operações de junção. Por exemplo, uma consulta simples para recuperar os elementos artigos (`/bibliografia/artigo`) envolve 2 operações de junção para encontrar os números dos nós dos elementos `artigo`, mais várias outras operações de junção para recuperar todos os filhos.

Uma das desvantagens dessa abordagem é que não há distinção entre elementos e atributos no mapeamento, impossibilitando uma reconstrução exata do documento original. Além disso, a reconstrução do documento envolve uma quantidade de operações de junção muito grande, de maneira recursiva. Dependendo do documento, a quantidade excessiva de operações de junção obriga o término precoce do processamento da consulta.

3.3.2 Abordagem dos Rótulos

Outro esquema de mapeamento proposto por Florescu e Kossmann [31, 32] sugere o agrupamento de todos os elementos ou atributos com o mesmo rótulo em uma tabela. Essa abordagem corresponde ao particionamento horizontal da tabela Edge, usando `Label` como o campo de particionamento. Conseqüentemente, o número de tabelas criadas é exatamente a quantidade de rótulos diferentes existentes em um documento XML. Cada tabela possui o seguinte esquema:

Tabela (Source, Target, Number, Flag, Value)

Os campos têm o mesmo significado da abordagem Edge. O nome da tabela é o rótulo escolhido para o particionamento.

A figura 3.10 apresenta as tabelas Sobrenome e Titulo. Outras tabelas seriam criadas para cada um dos rótulos, totalizando 15 tabelas.

Titulo				
Source	*Target*	*Number*	*Flag*	*Value*
2	3	1	string	"Informatica"
11	12	1	string	"Updating XML"

Sobrenome				
Source	*Target*	*Number*	*Flag*	*Value*
4	7	3	string	"Meirelles"
13	16	3	string	"Tatarinov"
18	20	2	string	"Weld"

Figura 3.10: Tabelas Titulo e Sobrenome criadas pelo particionamento da tabela Edge

Há uma diferença entre o desempenho de consultas na tabela Edge e nas tabelas particionadas devido à quantidade de dados manipulados. Na abordagem dos Rótulos, as tabelas possuem menor cardinalidade do que a tabela Edge e apenas aquelas necessárias às consultas são utilizadas.

3.4 Mapeamentos Baseados em Caminhos

3.4.1 Abordagem Monet

A abordagem apresentada por Schmidt et al. [64, 65] particiona o grafo XML durante o *parsing*. Para cada caminho partindo da raiz até algum nó do grafo, gera-se uma tabela, inclusive para os nós texto. Por exemplo, o caminho da raiz para o conteúdo do elemento `ano` da figura 2.8 é *bibliografia/artigo/ano/CDATA*.

Cada tabela possui dois campos:

Tabela (Source, Target)

O campo `Source` armazena o nó de origem, enquanto o campo `Target` armazena o nó de destino ou o valor de uma folha. Além disso, cada tabela é nomeada com seu respectivo caminho. A figura 3.11 apresenta as tabelas populadas para o grafo XML da figura 2.8.

```
                              bibliografia/livro   = {⟨1,2⟩}
                        bibliografia/livro/titulo  = {⟨2,3⟩}
                   bibliografia/livro/titulo/CDATA = {⟨3,"Informatica"⟩}
                          bibliografia/livro/autor = {⟨2,4⟩}
                       bibliografia/livro/autor/@id = {⟨4,5⟩}
                 bibliografia/livro/autor/@id/CDATA = {⟨5,"1"⟩}
                       bibliografia/livro/autor/nome = {⟨4,6⟩ }
                 bibliografia/livro/autor/nome/CDATA = {⟨6,"Fernando"⟩ }
                  bibliografia/livro/autor/sobrenome = {⟨4,7⟩ }
           bibliografia/livro/autor/sobrenome/CDATA = {⟨7,"Meirelles"⟩ }
                  bibliografia/livro/autor/endereco = {⟨4,8⟩ }
           bibliografia/livro/autor/endereco/CDATA = {⟨8,"Sao Paulo"⟩ }
                        bibliografia/livro/editor = {⟨2,9⟩}
          bibliografia/livro/editor/@nomecompleto = {⟨9,10⟩ }
    bibliografia/livro/editor/@nomecompleto/CDATA = {⟨10,"Milton Mira"⟩ }
                               bibliografia/artigo = {⟨1,11⟩ }
                         bibliografia/artigo/titulo = {⟨11,12⟩ }
                   bibliografia/artigo/titulo/CDATA = {⟨12,"Updating XML"⟩ }
                         bibliografia/artigo/autor = {⟨11,13⟩, ⟨11,18⟩ }
                       bibliografia/artigo/autor/@id = {⟨13,14⟩, ⟨18,19⟩ }
                 bibliografia/artigo/autor/@id/CDATA = {⟨14,"2"⟩, ⟨19,"3"⟩ }
                       bibliografia/artigo/autor/nome = {⟨13,15⟩ }
                 bibliografia/artigo/autor/nome/CDATA = {⟨15,"Igor"⟩ }
                  bibliografia/artigo/autor/sobrenome = {⟨13,16⟩, ⟨18,20⟩ }
           bibliografia/artigo/autor/sobrenome/CDATA = {⟨16,"Tatarinov"⟩, ⟨20,"Weld"⟩ }
                     bibliografia/artigo/autor/email = {⟨13,17⟩, ⟨18,21⟩ }
               bibliografia/artigo/autor/email/CDATA = {⟨17,"igor@cs..."⟩, ⟨21,"weld@cs..."⟩ }
                    bibliografia/artigo/autorcontato = {⟨11,22⟩ }
         bibliografia/artigo/autorcontato/@IDautor = {⟨22,23⟩ }
bibliografia/artigo/autorcontato/@IDautor/CDATA = {⟨23,"2"⟩ }
                             bibliografia/artigo/ano = {⟨11,24⟩ }
                       bibliografia/artigo/ano/CDATA = {⟨24,"2001"⟩ }
```

Figura 3.11: Tabelas da abordagem Monet para o grafo XML da figura 2.8

Note que para um exemplo simples, Monet criou 32 tabelas. A excessiva fragmentação do conteúdo do documento em várias tabelas é a principal desvantagem dessa abordagem.

Em compensação, o tamanho do banco de dados é reduzido, devido a existência de apenas dois campos para cada tabela.

Monet suporta consultas escritas na linguagem OQL. A figura 3.12 apresenta um exemplo de uma consulta OQL que recupera os títulos de artigos publicados em 2001.

```
SELECT x
FROM bibliografia/artigo y,
       y/titulo/CDATA x,
       y/ano/CDATA z
WHERE z = 2001
```

Figura 3.12: Consulta em OQL para recuperar títulos de artigos publicados em 2001

A consulta consiste de dois blocos, uma especificação dos elementos envolvidos (cláusulas SELECT e FROM) e as restrições (cláusula WHERE). O processamento da consulta ocorre através de operações de junção unindo todos os artigos encontrados (variável y) com a tabela `bibliografia/artigo/titulo` e a tabela `bibliografia/artigo/titulo/CDATA` (variável x). O mesmo ocorre para a variável z. Aplica-se a condição definida na cláusula WHERE e todas as tuplas associadas à variável x são retornadas.

Essa abordagem facilita consultas que utilizam expressões regulares (do tipo /, //). Em relação às abordagens Edge e dos Rótulos, os mapeamentos baseados em caminhos evitam algumas operações de junção para as reconstruções dos caminhos, tornando as consultas mais eficientes.

3.4.2 Abordagem XRel

Essa abordagem, apresentada por Shimura et al. em [75, 76], enumera todos os caminhos entre a raiz e cada nó do grafo XML, exceto os nós texto, e os armazena em uma tabela. Além disso, essas informações são combinadas com as informações sobre regiões.

A região de um nó é um par de valores que representa as posições inicial e final do nó no documento XML. Por exemplo, a região do nó 2 (`livro`) é (15, 172) porque o sinal $<$ da *tag* inicial é o caractere número 15 do documento XML e o sinal $>$ da *tag* final é o caractere número 172. As regiões dos nós atributos são formadas acrescentando-se 1 à posição inicial do nó pai. Exemplo: o nó `autor` possui região (50, 126) e a região de seu atributo `id` é (51, 51). Essas regiões foram criadas com o intuito de se preservar informações sobre a hierarquia do documento XML, pois através de operações simples de menor ($<$) e maior ($>$), pode-se verificar se um elemento está contido em outro. Regiões também descartam a necessidade do grafo XML, já que os valores das regiões podem ser retirados diretamente do documento XML.

Além da tabela que armazena os caminhos, XRel cria outras três tabelas. Uma para armazenar os relacionamentos pai-filho entre os nós, denominada tabela `Element`. Uma para armazenar os valores dos atributos (tabela `Attribute`) e outra para armazenar os valores dos elementos (tabela `Text`). Esquema relacional:

```
Path (pathID, pathexp)
Element (docID, pathID, start, end, index, reindex)
Attribute (docID, pathID, start, end, value)
Text (docID, pathID, start, end, value)
```

O campo `pathID` da tabela `Path` identifica um caminho especificado no campo `pathexp`. Os campos `docID`, `pathID`, `start`, `end` e `value` das outras tabelas representam o identificador do documento XML, o identificador do caminho, a posição inicial da região, a posição final de uma região e o valor texto do nó, respectivamente. Os campos `index` e `reindex` da tabela `Element` representam a posição de nós irmãos em relação ao nó pai, em ordem do documento e ordem reversa, respectivamente. Esses dois últimos campos servem para facilitar o processamento de consultas. A figura 3.13 apresenta as quatro tabelas do XRel para o documento da figura 2.1.

Quando um documento XML é atualizado, as posições dos elementos mudam e as regiões passam a ser inúteis. Essa definição de região não minimiza os efeitos de atualizações no banco de dados. Uma alternativa proposta por Kha et al. em [47] seria utilizar o RRC (*Relative Region Coordinate*), ou seja, as posições relativas dos nós. Ao invés de calcular a distância de um nó a partir da raiz, sua distância seria calculada a partir do nó pai, minimizando os efeitos das atualizações.

XRel apresenta um algoritmo detalhado para tradução de expressões XPath em SQL. O trunfo da abordagem é utilizar os caminhos armazenados na tabela `Path` para o processamento das consultas. A idéia básica do algoritmo de tradução de consultas é processar a expressão definida em XPath comparando-a com os caminhos armazenados, através do operador LIKE do SQL. Encontrados os identificadores dos caminhos, pode-se realizar operações de junção com as outras tabelas quando necessário. As regiões são utilizadas para definir a ordem dos elementos no documento XML resultante.

3.4.3 Abordagem XParent

O esquema proposto por Jiang et al. [44, 45] para mapeamento dos documentos XML possui quatro tabelas:

```
LabelPath (ID, Len, Path)
DataPath (Pid, Cid)
Element (PathID, Did, Ordinal)
Data (PathID, Did, Ordinal, Value)
```

docID	pathID	start	end	index	reindex
1	1	1	472	1	1
1	2	15	172	1	1
1	3	22	49	1	1
1	4	50	126	1	1
1	6	36	56	1	1
1	7	57	88	1	1
1	8	89	118	1	1
1	9	127	164	1	1
1	11	173	457	1	1
1	12	183	211	1	1
1	13	212	321	1	2
1	13	322	408	2	1
1	15	228	244	1	1
1	16	245	276	1	1
1	16	337	363	1	1
1	17	277	313	1	1
1	17	364	400	1	1
1	18	409	433	1	1
1	20	434	448	1	1

(a) Tabela Element

docID	pathID	start	end	value
1	5	51	51	"1"
1	10	128	128	"Milton Mira"
1	14	213	213	"2"
1	14	323	323	"3"
1	19	410	410	"2"

(b) Tabela Attribute

docID	pathID	start	end	value
1	3	30	40	"Informatica"
1	6	42	49	"Fernando"
1	7	68	76	"Meirelles"
1	8	99	107	"Sao Paulo"
1	12	191	202	"Updating XML"
1	15	234	237	"Igor"
1	16	256	264	"Tatarinov"
1	17	284	305	"igor@cs.washington.edu"
1	16	348	351	"Weld"
1	17	371	392	"weld@cs.washington.edu"
1	20	439	442	"2001"

(c) Tabela Text

pathID	pathexp
1	/bibliografia
2	/bibliografia/livro
3	/bibliografia/livro/titulo
4	/bibliografia/livro/autor
5	/bibliografia/livro/autor/@id
6	/bibliografia/livro/autor/nome
7	/bibliografia/livro/autor/sobrenome
8	/bibliografia/livro/autor/endereco
9	/bibliografia/livro/editor
10	/bibliografia/livro/editor/@nomecompleto
11	/bibliografia/artigo
12	/bibliografia/artigo/titulo
13	/bibliografia/artigo/autor
14	/bibliografia/artigo/autor/@id
15	/bibliografia/artigo/autor/nome
16	/bibliografia/artigo/autor/sobrenome
17	/bibliografia/artigo/autor/email
18	/bibliografia/artigo/autorcontato
19	/bibliografia/artigo/autorcontato/@IDautor
20	/bibliografia/artigo/ano

(d) Tabela Path

Figura 3.13: Tabelas da abordagem XRel para o documento XML da figura 2.1

A tabela `LabelPath`, de maneira similar ao XRel, armazena os caminhos entre o nó raiz e outros nós do grafo XML. O campo `ID` os identifica e o campo `Len` armazena a quantidade de arestas que aparecem no caminho. Os campos `Pid` e `Cid` da tabela `DataPath` armazenam os números dos nós de origem e dos nós de destino das arestas do grafo XML, respectivamente. Nas tabelas `Element` e `Data`, `PathID` é uma chave estrangeira para o campo `ID` da tabela `LabelPath` e os campos `Did` armazenam o número do nó de destino da última aresta do

caminho. Os campos `Ordinal` armazenam a ordem dos nós em relação a seus irmãos e `Value` armazena o valor de elementos e atributos. A figura 3.14 apresenta as tabelas preenchidas para o domínio da bibliografia (figura 2.8).

ID	*Len*	*Path*
1	1	/bibliografia
2	2	/bibliografia/livro
3	3	/bibliografia/livro/titulo
4	3	/bibliografia/livro/autor
5	4	/bibliografia/livro/autor/@id
6	4	/bibliografia/livro/autor/nome
7	4	/bibliografia/livro/autor/sobrenome
8	4	/bibliografia/livro/autor/endereco
9	3	/bibliografia/livro/editor
10	4	/bibliografia/livro/editor/@nomecompleto
11	2	/bibliografia/artigo
12	3	/bibliografia/artigo/titulo
13	3	/bibliografia/artigo/autor
14	4	/bibliografia/artigo/autor/@id
15	4	/bibliografia/artigo/autor/nome
16	4	/bibliografia/artigo/autor/sobrenome
17	4	/bibliografia/artigo/autor/email
18	3	/bibliografia/artigo/autorcontato
19	4	/bibliografia/artigo/autorcontato/@IDautor
20	3	/bibliografia/artigo/ano

(a) Tabela `LabelPath`

PathID	*Did*	*Ordinal*	*Value*
3	3	1	"Informatica"
6	6	1	"Fernando"
7	7	1	"Meirelles"
8	8	1	"Sao Paulo"
12	12	1	"Updating XML"
15	15	1	"Igor"
16	16	1	"Tatarinov"
17	17	1	"igor@cs.washington.edu"
16	20	1	"Weld"
17	21	1	"weld@cs.washington.edu"
20	24	1	"2001"
5	5	1	"1"
10	10	1	"Milton Mira"
14	14	1	"2"
14	19	1	"3"
19	23	1	"2"

(b) Tabela `Data`

PathID	*Ordinal*	*Did*
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
7	1	7
8	1	8
9	1	9
10	1	10
11	1	11
12	1	12
13	1	13
13	2	18
14	1	14
14	1	19
15	1	15
16	1	16
16	1	20
17	1	17
17	1	21
18	1	22
19	1	23
20	1	24

(c) Tabela `Element`

Pid	*Cid*
1	2
1	3
2	3
2	4
2	9
4	5
4	6
4	7
4	8
9	10
11	12
11	13
11	18
11	22
11	24
13	14
13	15
13	16
13	17
18	19
18	20
18	21
22	23

(d) Tabela `DataPath`

Figura 3.14: Tabelas da abordagem XParent para o documento XML da figura 2.8

XParent suporta consultas simples elaboradas em XQuery (do tipo /, //, *), através de um algoritmo similar ao do XRel, ou seja, as expressões da consulta são avaliadas usando o operador LIKE do SQL.

XParent é muito semelhante ao XRel, tanto em relação às tabelas geradas, como no processamento das consultas. O que diferencia os dois é a utilização de regiões pelo XRel, opção que gera operações de junção teta na avaliação de consultas, porque envolvem os operadores menor (<) e maior (>). Segundo Jiang et al. em [44], as operações de equijunção (as que usam somente o sinal de =) apresentam melhor desempenho que as operações de junção teta.

3.5 Mapeamentos Baseados em Intervalos

3.5.1 Abordagem de Zhang et al.

O trabalho desenvolvido por Zhang et al. [88] propõe o armazenamento de documentos XML em duas tabelas. As tabelas são uma adaptação do índice invertido, uma técnica muito popular na área de recuperação de informações. A primeira tabela, denominada `Elements`, possui o seguinte esquema relacional:

```
Elements (term, docno, begin, end, level)
```

O campo `term` armazena o nome do elemento e o campo `docno`, o número do documento XML. Os campos `begin` e `end` armazenam as posições inicial e final do elemento no documento, respectivamente. As posições são contadas por palavras no texto e não por caracteres, como ocorria no XRel [76]. Por fim, o campo `level` armazena o nível hierárquico do elemento no documento XML.

A segunda tabela, denominada `Texts`, apresenta o seguinte esquema relacional:

```
Texts (term, docno, wordno, level)
```

O campo `term` armazena a palavra e o campo `wordno` armazena sua posição no documento XML. Os outros campos possuem o mesmo significado da tabela `Elements`.

A figura 3.15 mostra as tabelas populadas para a figura 2.1.

Esta abordagem suporta algumas consultas em XPath que se beneficiam da idéia dos intervalos, como expressões com / e //. Estes tipos de expressões são fáceis de serem avaliadas através de operações de junção que comparem as posições dos elementos. Se um elemento estiver contido em outro, sua posição inicial será maior que a do elemento pai, enquanto sua posição final será menor, comprovando que ele é mesmo um elemento filho.

term	*docno*	*begin*	*end*	*level*
<bibliografia>	1	1	67	1
<livro>	1	2	27	2
<titulo>	1	3	5	3
<autor>	1	6	20	3
@id	1	7	9	3
<nome>	1	10	12	4
<sobrenome>	1	13	15	4
<endereco>	1	16	19	4
<editor>	1	21	26	3
@nomecompleto	1	22	25	3
<artigo>	1	28	66	2
<titulo>	1	29	32	3
<autor>	1	33	46	3
@id	1	34	36	3
<nome>	1	37	39	4
<sobrenome>	1	40	42	4
<email>	1	43	45	4
<autor>	1	47	57	3
@id	1	48	50	3
<sobrenome>	1	51	53	4
<email>	1	54	56	4
<autorcontato>	1	58	62	3
@IDautor	1	59	61	3
<ano>	1	63	65	3

(a) Tabela Elements

term	*docno*	*wordno*	*level*
Informatica	1	4	3
1	1	8	3
Fernando	1	11	4
Meirelles	1	14	4
Sao	1	17	4
Paulo	1	18	4
Milton	1	23	3
Mira	1	24	3
Updating	1	30	3
XML	1	31	3
2	1	35	3
Igor	1	38	4
Tatarinov	1	41	4
igor@cs.washington.edu	1	44	4
3	1	49	3
Weld	1	52	4
weld@cs.washington.edu	1	55	4
2	1	60	3
2001	1	64	3

(c) Tabela Texts

Figura 3.15: Tabelas da abordagem de Zhang el al. para o documento XML da figura 2.1

3.5.2 Abordagem dos Intervalos Dinâmicos

Esta abordagem, desenvolvida por DeHaan et al. [19], armazena todo o documento XML em apenas uma tabela denominada T, com os campos `s` para o nome do elemento (atributo ou texto), `l` para a posição inicial e `r` para a posição final. As posições são calculadas contando as palavras. A figura 3.16 apresenta a tabela simplificada para o documento XML da figura 2.1.

Esta abordagem suporta consultas em XQuery, inclusive expressões FLWOR. Vários *templates* são definidos para a avaliação das consultas. Por exemplo, o *template* para se encontrar elementos filho de um determinado nó é apresentado na figura 3.17. A idéia é a mesma da abordagem anterior, de Zhang et al. [88], verifica-se se um elemento está contido no outro.

Os *templates* para as consultas FLWOR são mais complexos e envolvem operações matemáticas nos valores dos intervalos (por isso o nome dinâmico). À medida que uma variável se associa a um valor (pelas cláusulas LET ou FOR), os valores de seus intervalos vão sendo modificados.

Dentre os mapeamentos independentes de estrutura, essa abordagem é a única a suportar consultas FLWOR, sendo esta sua maior vantagem.

s	*l*	*r*
<bibliografia>	1	67
<livro>	2	27
<titulo>	3	5
Informatica	4	4
<autor>	6	20
@id	7	9
1	8	8
<nome>	10	12
Fernando	11	11
<sobrenome>	13	15
Meirelles	14	14
<endereco>	16	19
⋮	⋮	⋮

Figura 3.16: Tabela T simplificada da abordagem de Intervalos Dinâmicos para o documento XML da figura 2.1

```
SELECT u.s
FROM T as u
WHERE EXISTS (
       SELECT *
       FROM T as v
       WHERE v.l < u.l AND u.r < v.r)
```

Figura 3.17: Template de uma consulta SQL para encontrar nós filho

3.6 Mapeamentos Dependentes do DTD

3.6.1 Abordagem Shared Inlining e Hybrid Inlining

Ao converter DTDs para esquemas relacionais, é bastante natural tentar mapear as declarações de elementos para tabelas e as declarações de atributos para campos das tabelas. Entretanto, este tipo de mapeamento pode gerar uma quantidade excessiva de tabelas, muitas delas redundantes. As técnicas denominadas Shared Inlining e Hybrid Inlining, propostas por Shanmugasundaram et al. [69, 74], tentam resolver o problema incluindo o maior número possível de descendentes de um elemento em uma única relação.

A primeira parte do processo envolve a simplificação do DTD, pois alguns DTDs são muito complexos para serem diretamente mapeados para um esquema relacional. As transformações de simplificação não afetam a eficiência do mapeamento. Há três tipos de transformações:

1. transformações de nivelamento para a conversão de uma declaração aninhada para uma representação plana.

```
(a, b)* → a*, b*
(a, b)?  → a?, b?
(a | b) → a?, b?
```

2. transformações de simplificação para a redução do número de indicadores de ocorrência.

```
a** → a*
a*?  → a*
a?* → a*
a?? → a?
```

3. transformações de agrupamento para reunir subelementos que possuem o mesmo nome.

```
...a*,...a*,... → a*, ...
...a*,...a?,... → a*, ...
...a?,...a*,... → a*, ...
...a?,...a?,... → a*, ...
 ...a,...a,... → a*, ...
```

Além dessas transformações, todos os indicadores "+" são transformados em indicadores "*". Assim, um elemento declarado como

```
<!ELEMENT A ((B | C | E)?,(E?|(F?, (B,B)*))*)>
```

seria transformado para `<!ELEMENT A (B*,C?,E*,F*)>`.

A segunda etapa cria o *grafo DTD*, de maneira similar ao grafo XML. Basicamente, os elementos, atributos e indicadores passam a ser os nós do grafo. Por exemplo, a figura 3.18 mostra o grafo DTD referente ao documento DTD da figura 2.2.

O esquema relacional criado para um determinado DTD é a união dos conjuntos de tabelas criadas para cada elemento. Para se definir as tabelas de um elemento, criou-se uma estrutura denominada *grafo elemento*. Para se construir um grafo elemento, deve-se realizar uma busca em profundidade no grafo DTD a partir do nó do elemento escolhido. Quando um elemento já visitado for alcançado, a aresta passa a ser tracejada significando que há uma recursão. A figura 3.19 apresenta o grafo elemento criado a partir do nó `editor`.

Criam-se tabelas para os elementos precedidos por "*" ou por uma aresta tracejada, para os elementos que não são atingidos por nenhuma aresta ou que são atingidos por mais de uma aresta no grafo elemento. Os outros elementos do grafo tornam-se campos das tabelas de seus nós pai. A figura 3.20 apresenta o esquema relacional gerado pelo método Shared Inlining para o grafo DTD da figura 3.18. O campo `parentID` armazena o identificador do elemento pai e o campo `parentCODE` foi criado para determinar o tipo do elemento pai, já

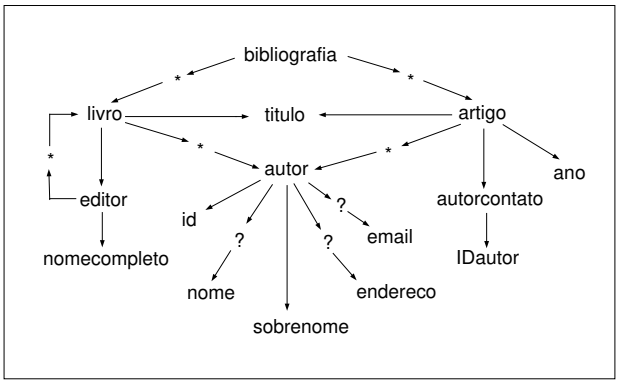

Figura 3.18: Grafo DTD da figura 2.2

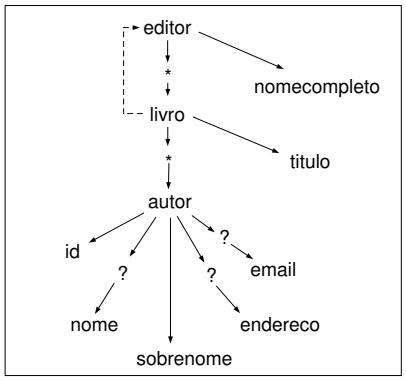

Figura 3.19: Grafo elemento para `editor`

que um elemento pode ter elementos pais de tipos diferentes (ex.: titulo pode ser descendente de artigo ou livro).

A diferença da abordagem Shared Inlining para a Hybrid Inlining é que nesta os elementos atingidos por mais de uma aresta também passam a ser campos em tabelas. Por exemplo, `titulo` passa a ser campo nas tabelas `livro` e `artigo`. Essa modificação evita algumas operações de junção em consultas que englobam esses elementos. A figura 3.21 apresenta o esquema relacional gerado pelo método Hybrid Inlining para o grafo DTD da figura 3.18.

Ambas as abordagens suportam consultas escritas em XML-QL. Não há detalhamento do algoritmo, apenas alguns exemplos. Basicamente, os elementos da cláusula CONSTRUCT do XML-QL passam para o SELECT da consulta SQL. Da mesma maneira, os elementos que aparecem na cláusula WHERE do XML-QL são identificados e passam a fazer parte da

bibliografia	(bibliografiaID)
livro	(livroID, livro.parentID, livro.parentCODE, livro.editor.nomecompleto)
artigo	(artigoID, artigo.parentID, artigo.ano, artigo.autorcontato.IDautor)
autor	(autorID, autor.parentID, autor.parentCODE, autor.id, autor.nome, autor.sobrenome, autor.endereco, autor.email)
titulo	(tituloID, titulo.parentID, titulo.parentCODE, titulo)

Figura 3.20: Tabelas da abordagem Shared Inlining para a figura 3.18

bibliografia	(bibliografiaID)
livro	(livroID, livro.parentID, livro.parentCODE, livro.editor.nomecompleto, livro.titulo)
artigo	(artigoID, artigo.parentID, artigo.ano, artigo.autorcontato.IDautor, artigo.titulo)
autor	(autorID, autor.parentID, autor.parentCODE, autor.id, autor.nome, autor.sobrenome, autor.endereco, autor.email)

Figura 3.21: Tabelas da abordagem Hybrid Inlining para a figura 3.18

cláusula FROM da consulta SQL. Além disso, se necessário, algumas operações de junção são criadas para representar a hierarquia entre os elementos da cláusula WHERE. Por exemplo, para a consulta XML-QL da figura 2.12, deve-se criar uma operação de junção para a expressão `livro/autor`.

3.6.2 Abordagem de Zheng et al.

O trabalho de Zheng et al. [89] reduz o problema de se encontrar um esquema relacional para um documento XML em um problema de otimização. Dado um documento XML associado a um DTD e um conjunto de consultas com pesos diferentes, deseja-se encontrar qual o melhor esquema relacional que possa ser adotado para minimizar o custo de execução das consultas.

Primeiramente, o DTD é mapeado para um grafo semelhante ao da figura 3.18, com a diferença que somente os operadores "*" são considerados. Após essa etapa, o grafo é particionado em fragmentos. Um fragmento é um subgrafo conexo que não pode conter o símbolo "*", que denota um elemento multivalorado. O grafo DTD particionado em 5 fragmentos é apresentado na figura 3.22.

O mapeamento do grafo particionado para o esquema relacional é quase direto: cada fragmento passa a ser uma tabela onde os nós terminais passam a ser campos da tabela. Se o fragmento for precedido pelo símbolo "*", cria-se uma chave estrangeira com o sufixo `parentID`. Para o grafo da figura 3.22, 5 tabelas são criadas (figura 3.23).

Em seguida, traduzem-se as consultas escritas em XQuery para consultas em SQL e calcula-se o custo da execução das mesmas utilizando o esquema relacional proposto. Este é o estado inicial.

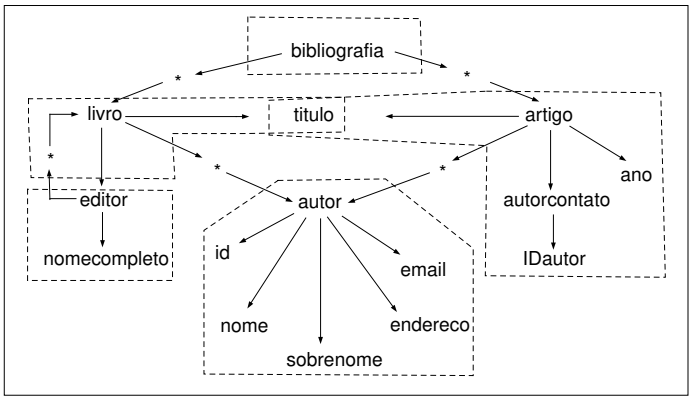

Figura 3.22: Grafo DTD particionado em 5 fragmentos

```
T1 = (bibliografia)

T2 = (livro, livro.titulo, livro.parentID)

T3 = (artigo, artigo.titulo, artigo.autorcontato.IDautor,
      artigo.ano, artigo.parentID)

T4 = (editor, editor.nomecompleto, editor.parentID)

T5 = (autor, autor.id, autor.nome, autor.sobrenome,
      autor.email, autor.endereco, autor.parentID)
```

Figura 3.23: Tabelas criadas para o grafo da figura 3.22

Para se encontrar diferentes esquemas relacionais, algumas transformações podem ser aplicadas ao grafo DTD, de tal maneira que haja mudanças nos fragmentos. Por exemplo, pode-se criar um fragmento que englobe somente o elemento `titulo`. Com um esquema relacional novo, os custos são recalculados e define-se um novo estado.

Escolhe-se o estado com menor custo e o grafo DTD passa por novas transformações. Aplica-se o processo sucessivamente até que o melhor esquema seja selecionado.

A vantagem desta abordagem é a possibilidade de verificar diferentes configurações para o esquema relacional, não apresentado por nenhuma abordagem discutida anteriormente.

3.7 Mapeamentos Dependentes do XML Schema

3.7.1 Abordagem X-Database

O sistema X-Database, desenvolvido por Varlamis e Vazirgiannis [83], propõe o mapeamento direto das declarações do XML Schema para tabelas ou campos no banco de dados relacional.

Basicamente, cada elemento complexo torna-se uma tabela e os elementos simples e atributos tornam-se campos. No mapeamento de elementos complexos, uma questão importante surge: elementos complexos podem conter (através do **"type"**) ou referenciar (através do **"ref"**) outros elementos complexos. Isso implica na criação de novas tabelas para relacionamentos muitos para muitos e na definição de algumas restrições para que o esquema relacional reflita corretamente o XML Schema, como chave estrangeira.

Há quatro possíveis casos para a existência dos subelementos:

1. Elemento complexo A possui exatamente uma referência a um elemento complexo B.

   Significa que um elemento B deve ser criado antes que qualquer elemento A possa referenciá-lo. Trata-se de um relacionamento um para muitos, não sendo necessário a criação de uma nova tabela. Apenas duas tabelas são criadas, uma para A e outra para B. Quando um elemento B for removido, todas as referências a ele também devem ser removidas.

2. Elemento complexo A possui várias referências (`maxOccurs = "unbounded"`) a um elemento complexo C.

   Significa que elementos do tipo C devem ser criados antes de serem referenciados por elementos do tipo A. Trata-se de um relacionamento muitos para muitos, então deve-se criar uma tabela intermediária (A-C-link) para armazenar a ordem das referências. Somando-se as tabelas criadas para A e C, três tabelas são criadas. Quando um elemento A ou C for destruído, todos os registros relacionados em A-C-link são removidos.

3. Elemento complexo A contém exatamente um elemento complexo D.

   Isso significa que o elemento D só pode existir dentro de A. Trata-se de um relacionamento um para um e duas tabelas são criadas, uma para A e outra para D. Quando A for destruído, o elemento D também deve ser destruído.

4. Elemento complexo A contém vários (`maxOccurs = "unbounded"`) elementos complexos E.

   Esse relacionamento é um para muitos, já que elementos do tipo A não podem compartilhar os mesmos subelementos do tipo E. Quando A for removido, todos os elementos E também devem ser removidos do banco de dados.

```
bibliografia =   (bibliografiaID)
livroType =      (livroID, titulo, bibliografia-ref, editorType-ref)
artigoType =     (artigoID, titulo, ano, bibliografia-ref)
autorcontato =   (autorcontatoID, IDautor, artigoType-ref)
autorType =      (autorID, nome, sobrenome, endereco,
                 email, id, livroType-ref, artigoType-ref)
editorType =     (editorID, livroType-ref, nomecompleto)
```

Figura 3.24: Tabelas geradas pelo X-Database para o XML Schema da figura 2.6

A figura 3.24 apresenta o esquema relacional gerado para o XML Schema da figura 2.6.

Como bibliografia contém vários elementos livros e artigos, seguindo o caso 4, criaram-se as chaves estrangeiras `bibliografia-ref` nas tabelas livroType e artigoType. O mesmo ocorre na tabela autorType, que possui duas chaves estrangeiras com sufixo ref no nome. Note a criação da tabela autorcontato, pois trata-se de um elemento complexo, e da recursão gerada entre as tabelas livroType e editorType. Os outros campos das tabelas são elementos simples ou atributos mapeados de forma direta.

X-Database não suporta nenhum tipo de linguagem de consulta XML. Criou-se um módulo especial denominado DBCommand para que algumas consultas simples possam ser avaliadas.

3.7.2 Abordagem LegoDB

A idéia principal da abordagem apresentada por Bohannon et al. [5, 6, 33, 34] é analisar possíveis mapeamentos e selecionar o que for melhor para uma determinada aplicação, uma abordagem semelhante ao trabalho de Zheng et al. [89]. Para chegar ao melhor esquema relacional possível, além do XML Schema, dados estatísticos sobre o documento XML e as consultas utilizadas também são levados em consideração.

A figura 3.25 apresenta o *framework* utilizado para o desenvolvimento da ferramenta LegoDB, que automatiza todo o processo de mapeamento.

O primeiro módulo processa o XML Schema e os dados estatísticos do documento XML para gerar o *p-schema*, uma versão concisa dos elementos contidos no XML Schema, acrescida de informações sobre tamanhos e valores dos dados. A figura 3.26 mostra, como exemplo, o *p-schema* inicial com os elementos `artigo` e `autor` para o XML Schema da figura 2.6 associado ao documento XML da figura 2.1.

Os dados estatísticos para textos (`String<#tamanho, #distintos>`) indicam o tamanho do dado em bytes e o número de valores distintos. Para dados escalares, utiliza-se `Scalar<#tamanho, #min, #max, #distintos>` correspondendo ao tamanho, valores mínimo e máximo e o número de valores distintos. A notação `*<#quantidade>` indica o número médio de elementos `Autor` dentro de cada elemento `Artigo`.

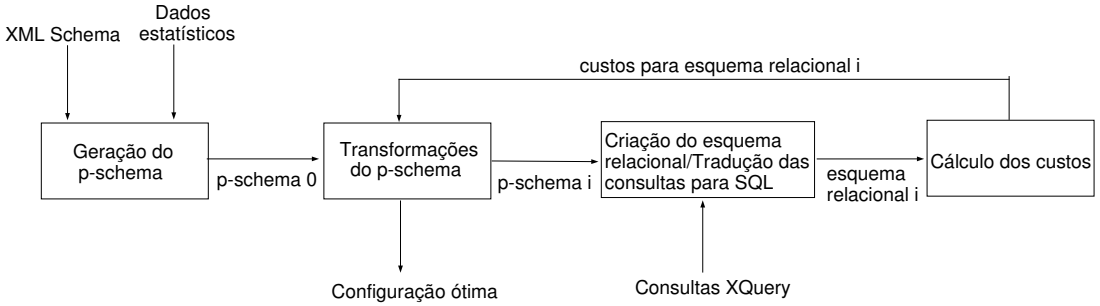

Figura 3.25: *Framework* do LegoDB

```
type Artigo =
   artigo [ titulo [String<#8, #1>],
            Autor*<#2>,
            autorcontato[@IDautor [Integer <#4, #2, #2, #1>]],
            ano [Integer <#4, #2001, #2001, #1>]]

type Autor =
   autor [ @id[Integer <#4, #2, #3, #2>],
           nome [String<#8, #1>],
           sobrenome [String<#8, #2>],
           endereco [String<#8, #2>] | email [String<#8, #2>]]
```

Figura 3.26: *p-schema* inicial do LegoDB com os elementos `artigo` e `autor`

O próximo módulo aplica algumas transformações ao *p-schema* inicial a fim de gerar diferentes *p-schemas*, implicando em diferentes esquemas relacionais. Essas transformações envolvem operações de *outlining* (elemento aninhado com declaração separada), *inlining* (inserção de um elemento da declaração de seu pai), distribuição (`(a,(b|c)) == (a,b|a,c)` e `(a,[d|e] == a[d]|a[e])`), separação (`a+ == a,a*`) e utilização de valores nulos (`(a|b)` $\subset$ `(a?, b?)`). A figura 3.27 apresenta três possíveis *p-schemas* para o elemento `autor`.

O terceiro módulo faz o mapeamento de cada *p-schema* para um esquema relacional, de forma praticamente direta. Para cada tabela, cria-se uma chave (sufixo `id`) e, se houver necessidade, uma chave estrangeira (prefixo `parent`). A figura 3.28 apresenta os mapeamentos para os elementos `artigo` e `autor`. O mapeamento do elemento `livro` seria obtido de forma semelhante.

Além da criação do esquema relacional, o terceiro módulo recebe as consultas em XQuery e as traduzem para SQL. No LegoDB, a tradução é realizada em duas fases. A primeira fase normaliza a consulta XQuery, desmembrando as expressões XPath. A figura 3.29 apresenta um exemplo. A consulta XQuery recupera títulos de artigos publicados em 2001. Note que na consulta normalizada, as variáveis que são campos em tabelas recebem o prefixo `v`.

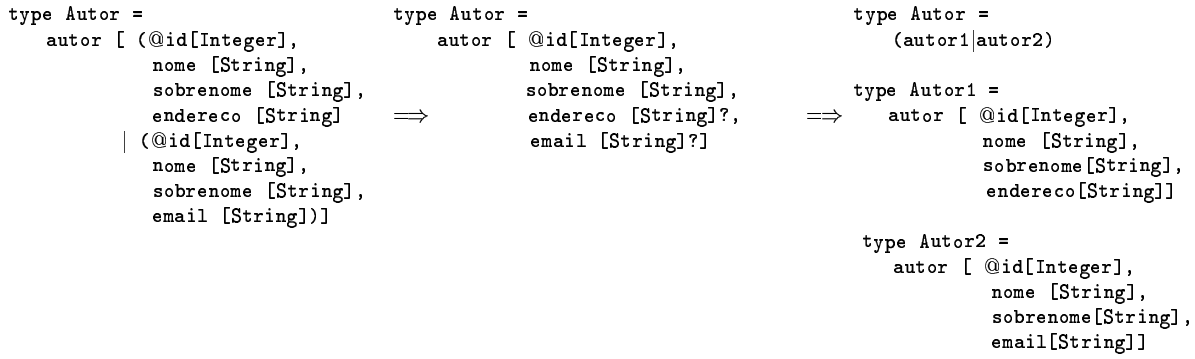

Figura 3.27: Exemplos de *p-schemas*. (a) Transformação `(a,(b|c)) == (a,b|a,c)` . (b) Transformação `(a|b)` $\subset$ `(a?, b?)`. (c) Transformação `(a,[d|e] == a[d]|a[e])`.

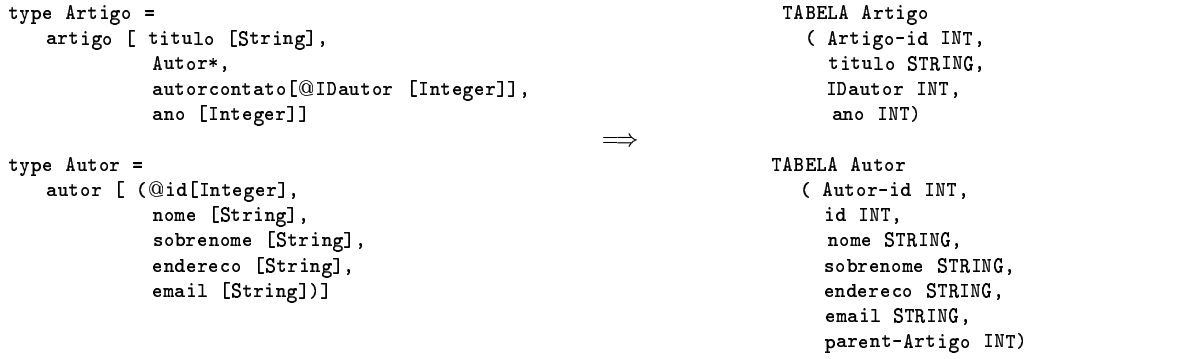

Figura 3.28: Mapeamento do *p-schema* para esquema relacional no LegoDB

A transformação de uma consulta normalizada em XQuery é direta. Uma váriável que apareça na cláusula RETURN do XQuery passa para a cláusula SELECT do SQL. Identifica-se a tabela associada a cada variável que aparece nas cláusulas WHERE e FROM do XQuery e ela passa para a cláusula FROM do SQL. A cláusula WHERE do SQL é mapeada diretamente da cláusula WHERE do XQuery.

O último módulo do LegoDB recebe o esquema relacional de um determinado *p-schema* e as consultas SQL com seus correspondentes pesos, que variam de acordo com a aplicação. Por exemplo, supondo que haja 4 consultas, pode-se determinar duas configurações diferentes: C1 = (Q1:0.4, Q2:0.4, Q3:0.1, Q4:0.1) e C2 = (Q1:0.1, Q2:0.1, Q3:0.4, Q4:0.4). Este módulo faz uma estimativa dos custos das consultas que servirão como parâmetro para a escolha do melhor esquema relacional.

Os custos são enviados para o segundo módulo, que escolhe o *p-schema* com melhores resultados. Para o *p-schema* escolhido, aplicam-se novas transformações para a geração de outras configurações, posteriormente repassadas aos próximos módulos. O processo é repetido até que a configuração atual não possa mais ser melhorada.

```
CONSULTA XQUERY

    FOR $artigo IN document("bibliografia.xml")/bibliografia/artigo
    WHERE $artigo/ano = 2001
    RETURN $artigo/titulo

CONSULTA XQUERY NORMALIZADA

    LET $bibdoc := document("bibliografia.xml")
    FOR $bib IN $bibdoc/bibliografia, $artigo IN $bib/artigo, $v-titulo IN $artigo/titulo
        $v-ano IN $artigo/ano
    WHERE $v-ano = 2001
    RETURN $v-titulo
```

Figura 3.29: Normalização de consulta XQuery no LegoDB

3.8 Mapeamentos em Produtos Comerciais

3.8.1 Oracle

A Oracle Corporation começou a dar suporte ao XML a partir do Oracle 8i [3]. O objetivo era que algumas funcionalidades fossem estendidas a fim de permitir o armazenamento de documentos XML. Com o crescimento e amadurecimento do XML, a Oracle criou um conjunto de funcionalidades denominado Oracle XML DB [26, 37]. Esse conjunto, implementado no Oracle 9i e 10g, permite o armazenamento e recuperação dos documentos XML de maneira mais eficiente do que no Oracle 8i.

Na área de armazenamento, a principal inovação foi a criação de um novo tipo de dados, o XMLType. Pode-se utilizar XMLType para armazenar um documento XML em um campo da tabela. O tipo de dados inclui alguns métodos que possibilitam a manipulação do conteúdo do documento XML, possibilitando a recuperação de dados. O Oracle oferece duas opções para o armazenamento do tipo XMLType: através do CLOB ou mapeando elementos e atributos para o banco de dados.

1. CLOB (Character Large Objects)

   A utilização desse tipo de armazenamento é indicado quando o documento não possui estrutura. A indexação textual permite que operadores como CONTAINS sejam utilizados para encontrar dados no texto. A vantagem é que sempre há uma cópia exata do documento no banco de dados. Entretanto, algumas funcionalidades do SQL não podem ser exploradas.

2. Mapeamento de elementos e atributos

   O mapeamento de elementos e atributos de um documento XML para o banco de dados objeto-relacional só pode ser feito se há uma estrutura, o XML Schema. Basicamente,

atributos e elementos que possuem apenas texto como conteúdo são mapeados para campos. Elementos com subelementos se tornam objetos (*object types*) e listas de elementos são mapeadas para coleções. Os mapeamentos podem ser explicitamente determinados pelo desenvolvedor da aplicação ou podem ser realizados pelo XML DB através da inserção de informações no XML Schema. Por exemplo, a figura 3.30 mostra o XML Schema com mapeamentos para uma versão simplificada do elemento autor.

```
<xsd:complexType name = "autorType"xdb:defaultTable = "AUTOR"/>>
   <xsd:sequence>
      <xsd:element name = "nome"  type = "xsd:string"  xdb:SQLName = "NOME"/>
      <xsd:element name = "sobrenome"  type = "xsd:string"  xdb:SQLName = "SOBRENOME">
   </xsd:sequence>
   <xsd:attribute name = "id"  type = "xsd:ID"  use = "required"  xdb:SQLName = "ID"  xdb:SQLType = "integer"/>
</xsd:complexType>
```

Figura 3.30: XML Schema do elemento autor com informações extras do Oracle

As informações extras definem que a tabela AUTOR será criada com os campos NOME, SOBRENOME e ID (especificados pelo SQLName). O campo ID será do tipo `integer` (especificado pelo SQLType). Tipos de dados não especificados para o banco de dados seguem o mesmo tipo do documento XML. Caso o desenvolvedor não insira nenhum tipo de informação, o Oracle utiliza um mapeamento padrão.

Consultas podem ser enviadas ao Oracle através de funções do padrão SQL/XML [36]. As principais funções são apresentadas na tabela 3.1.

Função	*Descrição*
existsnode()	Recebe uma expressão em XPath e retorna verdadeiro se existe um nó no documento XML com aquela expressão. Caso contrário, retorna falso.
extract()	Recebe uma expressão em XPath e retorna o nó ou a subárvore relativa àquela expressão formatado como documento XML.
extractvalue()	Recebe uma expressão em XPath cuja folha seja um nó texto e retorna seu valor.

Tabela 3.1: Funções SQL/XML para realização de consultas no Oracle

Suponha que o documento XML da figura 2.1 esteja armazenado como um XMLType no Oracle com o nome de bibliografia. A consulta para recuperar os títulos de artigos publicados em 2001 é apresentada na figura 3.31.

Nesta consulta, se a avaliação da expressão dentro da função `existsNode` for 1, ou seja, verdadeira, o elemento `<titulo>` é retornado.

```
SELECT extract(object-value,'/bibliografia/artigo/titulo')
FROM BIBLIOGRAFIA
WHERE existsNode(object-value,'/bibliografia/artigo[ano="2001"]')= 1
```

Figura 3.31: Consulta no Oracle para recuperar títulos de artigos publicados em 2001

3.8.2 IBM

O DB2 da IBM suporta XML através do XML Extender [15, 39]. Há duas opções de armazenamento, a coluna XML (*XML column*) e a coleção XML (*XML collection*).

A coluna XML é indicada para o usuário que deseja armazenar o documento original, acessa apenas alguns elementos e faz poucas atualizações. O documento XML é armazenado em um campo na tabela com um dos três tipos: (a) XMLVARCHAR para documentos pequenos, (b) XMLCLOB para documentos grandes e (c) XMLFILE para documentos que devem ser armazenados fora do DB2. O usuário deve determinar quais os elementos e atributos serão consultados e mapeá-los para tabelas denominadas "*side tables*". Essas tabelas são indexadas para tornar as operações de consulta mais eficientes. A definição do mapeamento é feita através do arquivo DAD (*Data Access Definition*), um documento XML com informações extras sobre as tabelas que devem ser criadas e os elementos ou atributos que se tornarão campos dessas tabelas. Por exemplo, a figura 3.32 apresenta um DAD simplificado que poderia ser definido para o elemento `autor` do documento XML da figura 2.1.

```
<?xml version = "1"?>
<DAD>
      <Xcolumn>
            <table name = "autor">
                  <column name = "nome"  type = "varchar"  path = "/bibliografia/livro/autor/nome"
                     multi-occurrence = "YES"/>
            </table>
      </Xcolumn>
</DAD>
```

Figura 3.32: Exemplo de DAD para uma coluna XML

O elemento `table` determina qual o nome da tabela e o elemento `column` determina qual campo deve ser criado, especificando seu nome, tipo, caminho desde a raiz e se há múltiplas ocorrências do elemento no documento.

Uma coleção XML deve ser utilizada quando o usuário deseja acessar ou atualizar os dados com freqüência. Neste modo de armazenamento, todos os elementos e atributos devem ser mapeados para campos de várias tabelas. Um arquivo DAD deve ser definido pelo usuário para esse fim. Por exemplo, para o documento XML

```
<?xml version = "1"?>
<artigo>
      <titulo>Updating XML</titulo>
      <ano>2001</ano>
</artigo>
```

o DAD da figura 3.33 poderia ser definido.

```
<?xml version = "1"?>
<DAD>
   <Xcolletion>
      <root-node>
         <element-node name = "artigo">
            <element-node name = "titulo">
               <text-node>
                  <column name = "titulo">
               </text-node>
            </element-node>
            <element-node name = "ano">
               <text-node>
                  <column name = "ano">
               </text-node>
            </element-node>
         </element-node>
      </root-node>
   </Xcolletion>
</DAD>
```

Figura 3.33: Exemplo de DAD para uma coleção XML

O DAD é definido caminhando pelo grafo XML, identificando os elementos e atributos que serão armazenados e criando os campos correspondentes, como o campo `titulo` para o elemento `titulo`.

Assim como no Oracle 9i, a recuperação de dados ocorre através de funções. As UDFs (*User-Defined Functions*) recebem uma expressão em XPath como parâmetro, para localizar o elemento. Há funções que recuperam o documento XML inteiro, como **Content()**, e funções que recuperam tipos de dados específicos, como **extractInteger()**, **extractReal()**, **extractVarchar()**, etc.

3.8.3 Microsoft

O Microsoft SQL Server 2000 [62] suporta o armazenamento de documentos XML através de anotações no XML Schema, de maneira similar à solução da Oracle. A estrutura passa a ser conhecida como "*XSD annotated schema*". Essas anotações são novos atributos nos elementos, contendo referências ao SQL para a criação das tabelas.

A figura 3.34 apresenta um exemplo de um XSD *annotated schema* simplificado para o elemento `autor`.

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns:sql = "urn:schemas-microsoft:mapping-schema">
<xsd:element name = "autor"  sql:relation = "Autores">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="nome"  sql:field="AutorNome"  type="xsd:string"  minOccurs="0"  maxOccurs="1"/>
         <xsd:element name = "sobrenome"  sql:field = "AutorSobrenome"  type = "xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name = "id"  sql:field = "AutorID"  type = "xsd:ID"  use = "required"/>
   </xsd:complexType>
</xsd:element>
```

Figura 3.34: Exemplo de XSD annotated schema

A definição das tabelas é feita através do atributo `sql:relation`, e a dos campos através do `sql:field`.

Quando não há um mapeamento definido pelo usuário, utiliza-se um mapeamento padrão determinando que os elementos complexos são mapeados para tabelas e elementos simples e atributos são mapeados para campos da tabela.

SQL Server aceita consultas em XPath, se houver um *XSD annotated schema* definido. Para que uma consulta seja executada, primeiramente ela passa por um processo de reescrita, transformando-se em uma consulta SQL com a cláusula FOR XML EXPLICIT (definição na seção 4.2.7). Esta cláusula formata o resultado final da maneira especificada no *schema*.

3.9 Análise Comparativa dos Métodos de Armazenamento

Apresentam-se, a seguir, as características que foram analisadas nos métodos de armazenamento de documentos XML. As tabelas 3.2 e 3.3 apresentam um resumo.

3.9.1 Qualidade do esquema relacional

Não há nenhuma métrica definida na literatura para se avaliar a qualidade de um esquema relacional gerado a partir de um dos métodos de mapeamento. Neste trabalho, três características serão avaliadas: o número de tabelas geradas, a quantidade de valores nulos e o tamanho das tabelas. Isoladamente, a quantidade de tabelas geradas não determina se o esquema tem boa qualidade. Entretanto, associada à abordagem, pode influenciar o desempenho das consultas.

Para as abordagens STORED e XStorM, o número de tabelas criadas depende do documento XML e o pior caso é quando o documento está totalmente desestruturado. Neste caso, várias consultas de *overflow* são criadas, o que afeta o desempenho devido ao problema de integração de dados. Valores nulos ocorrem quando alguma subárvore possui menos atribu-

tos do que o padrão determinado pelo algoritmo de *data mining*. Os tamanhos das tabelas também dependem da estruturação do documento XML. Documentos estruturados geram menor quantidade de tabelas, com maior tamanho, e vice-versa.

A tabela Edge, apesar de ser simples, gera valores nulos para os nós intermediários do grafo XML (os que unem a raiz e as folhas). O mesmo ocorre no caso da abordagem dos Rótulos. A quantidade de tabelas, para essas abordagens, não influencia muito o desempenho das consultas, pois, mesmo para consultas simples, a quantidade de operações de junção necessárias continua a mesma. O tamanho das tabelas é proporcional à quantidade de arestas no grafo XML, já que o número de campos é fixo.

Monet não gera nenhum valor nulo, em compensação cria uma grande quantidade de tabelas. A excessiva fragmentação do documento em diversas tabelas dificulta certos tipos de consultas, como as que envolvem valores de vários caminhos diferentes ou as que exigem reconstrução de uma subárvore. Em relação ao tamanho das tabelas, provavelmente Monet é uma das abordagens mais econômicas, porque os caminhos são armazenados como metadados no banco de dados, sobrando apenas números de nós e valores das folhas nas tabelas.

XRel, XParent possuem exatamente 4 tabelas, que não geram valores nulos e possuem tamanhos similares (dependentes da quantidade de caminhos). Nestas abordagens, o esquema relacional é melhor utilizado para a avaliação das consultas, evitando a necessidade de tantas operações de junção como nas abordagens anteriores.

As abordagens baseadas em intervalos também possuem um número fixo e pequeno de tabelas (de uma a duas tabelas) e não geram valores nulos. A abordagem de intervalos dinâmicos é ainda mais econômica em relação ao tamanho das tabelas, pois possui apenas 1 tabela com 3 campos.

A característica de qualidade do esquema relacional é difícil de ser avaliada nas abordagens dependentes de estrutura e nos produtos comerciais, uma vez que os esquemas relacionais são gerados dinamicamente. Em relação aos valores nulos, mesmo que os documentos XML sejam válidos, pode ser que eles não possuam todos os elementos do esquema relacional, ou seja, omitam os elementos declarados como opcionais na estrutura.

3.9.2 Preservação das Restrições

Com exceção do X-Database, todas as outras abordagens se preocupam com a estrutura do próprio documento XML, DTD ou XML Schema, e esquecem as restrições semânticas. Por exemplo, no DTD da figura 2.2, IDREF pode ser considerada uma chave estrangeira, enquanto todos os atributos do tipo #REQUIRED deveriam ser valores não nulos.

O X-Database preserva restrições relativas aos valores obrigatórios, chaves estrangeiras e restrições de domínio.

Em um trabalho complementar proposto por Lee e Chu [52], há uma extensão para que a abordagem Hybrid Inlining preserve as restrições. Nenhuma das outras abordagens possui

esta característica.

3.9.3 Grau de automação

Esta característica demonstra a necessidade de interação do desenvolvedor/usuário em todo o processo.

As abordagens dependentes do conteúdo exigem que alguns parâmetros do algoritmo de *data mining* sejam estipulados previamente. Alguns desses parâmetros são: quantidade máxima de tabelas, quantidade máxima de campos nas tabelas, quantas vezes uma subárvore precisa aparecer no grafo DTD para ser um padrão, entre outros.

Para as abordagens baseadas em arestas, caminhos e intervalos, um *parser* deve ser desenvolvido para preencher as tabelas.

Para as abordagens dependentes de estrutura não foi possível determinar o grau de automação. Entretanto, supõe-se que um documento XML válido seja facilmente decomposto para as tabelas.

Os produtos comerciais exigem conhecimentos avançados sobre XML Schema, XPath e SQL para que sejam escritas as anotações e consultas.

3.9.4 Suporte a atualizações

Apesar de existirem poucos trabalhos sobre atualizações em documentos XML [79, 10], este tópico avalia o impacto das atualizações sobre os esquemas relacionais.

Nas abordagens dependentes do conteúdo, as atualizações geram consultas de *overflow*, que, dependendo da quantidade, acabam exigindo o reinício de todo o processo.

Nas abordagens baseadas em arestas e no XParent, os valores dos nós devem ser modificados. No Monet, mudanças na estrutura do grafo XML geram novas tabelas. No XRel e nas abordagens baseadas em intervalos, todos os valores das posições ficam inutilizadas.

Atualizações em documentos XML que seguem um DTD ou XML Schema não afetam o esquema relacional.

3.9.5 Reconstrução do Documento XML

Duas questões podem ser consideradas na reconstrução de um documento XML. A primeira é ordem dos elementos, entre relacionamentos pai e filho e entre nós irmãos. A segunda diz respeito ao processamento envolvido.

Não foi possível verificar por [23] se STORED permite a reconstrução do documento XML. As abordagens baseadas em arestas geram consultas SQL aninhadas para a recuperação de nós filhos e suportam ordem entre os irmãos através do campo `Number`. Como não diferenciam atributos de elementos, não conseguem reconstruir o documento original.

Devido a quantidade de tabelas geradas na abordagem Monet, a reconstrução do documento XML também é afetada pelas operações de junção. Além disso, não suporta ordem entre irmãos. XParent também exige várias consultas SQL com operações de junção, entretanto suporta ordem entre irmãos através do campo `Ordinal`. XRel e as abordagens baseadas em intervalos reconstroem o documento usando operações junção teta, preservando a ordem entre nós irmãos.

Nas abordagens dependentes de estrutura, criam-se chaves estrangeiras para mapear relacionamentos pai e filho, mas nenhuma mencionou a questão da ordem entre irmãos.

Nos produtos comerciais há a opção de se armazenar o documento XML original como CLOB.

3.9.6 Linguagem de consulta suportada

XQuery é a linguagem candidata com mais chances de ser oficializada. Por ser recente, a maioria das abordagens ainda não a implementaram. A tabela 3.3 apresenta qual a linguagem adotada por cada abordagem.

3.9.7 Classes de consultas suportadas

Por ser recente, muitos sistemas ainda não incorporaram todas as funcionalidades do XQuery.

As abordagens baseadas em arestas suportam consultas simples para recuperação do número de um nó, seleção por valores e expressões do tipo / e //. As abordagens baseadas em caminhos e a abordagem de Zhang et al. [88] foram desenvolvidas para suportar as expressões que envolvam / e //. A abordagem de intervalos dinâmicos suporta consultas mais complexas, inclusive expressões FLWOR.

Hybrid Inlining suporta expressões simples (/ e //). Para a abordagem de Zheng et al. [89] não foi possível determinar a classe de consultas suportadas. O LegoDB suporta, além de expressões simples, consultas FLWOR não aninhadas.

O Oracle e o SQL Server suportam consultas simples do tipo / e //, já o DB2 suporta somente extração de tipos de dados definidos pelas UDFs.

3.9.8 Suporte à recursão

Para as abordagens dependentes de estrutura, deve-se avaliar o suporte à recursão. Dos trabalhos apresentados, apenas a abordagem Hybrid Inlinig preocupou-se em mapear a recursão para o esquema relacional.

	Número de tabelas geradas	Quantidade de valores nulos	Tamanho das tabelas geradas	Preservação das Restrições
STORED	depende do conteúdo do documento XML	ocorre quando uma subárvore não segue o padrão	depende da estrutura do documento	não preserva
XStorM	depende do conteúdo do documento XML	ocorre quando uma subárvore não segue o padrão	depende da estrutura do documento	não preserva
Edge	1	ocorre para os nós intermediários	depende da quantidade de arestas	não preserva
dos Rótulos	depende da quantidade de arestas no grafo XML	ocorre para os nós intermediários	depende da quantidade de arestas	não preserva
Monet	depende da quantidade de caminhos do grafo XML	evita valores nulos	econômico - apenas nós e valores	não preserva
XRel	4	evita valores nulos	depende da quantidade de caminhos	não preserva
XParent	4	evita valores nulos	depende da quantidade de caminhos	não preserva
Zhang et al.	2	evita valores nulos	econômico	não preserva
Intervalos Dinâmicos	1	evita valores nulos	econômico - 1 tabela com 3 campos	não preserva
Hybrid Inlining	depende do DTD	pode haver valores nulos		não preserva
Zheng et al.	depende do DTD	pode haver valores nulos		não preserva
X-Database	depende do XML Schema	pode haver valores nulos		preserva valores não nulos, chaves estrangeiras e restrições de domínio
LegoDB	depende do XML Schema	pode haver valores nulos		não preserva
Oracle 9i	depende do XML Schema	pode haver valores nulos		não preserva
DB2	depende do DAD	pode haver valores nulos		não preserva
SQL Server	depende do XSD annotated schema	pode haver valores nulos		não preserva

Tabela 3.2: Sumário dos métodos de armazenamento de documentos XML - Parte A

	Grau de Automação	Suporte a atualizações	Reconstrução do documento XML	Linguagem de consulta suportada	Classe de Consultas Suportadas	Suporte à Recursão
STORED	definição de parâmetros para o algoritmo WL	gera consultas *overflow*		STORED		
XStorM	definição de parâmetros para o algoritmo WL	gera tabelas *overflow*		nenhuma		
Edge	desenvolvimento de *parser*	muda os valores dos nós	gera várias junções e não recria o documento original	XML-QL	recuperação do nº do nó, seleções, expressões / ,//	
dos Rótulos	desenvolvimento de *parser*	muda os valores dos nós	gera várias junções e não recria o documento original	XML-QL	recuperação do nº do nó, seleções, expressões / ,//	
Monet	desenvolvimento de *parser*	cria novas tabelas	gera várias junções, não suporta ordem entre irmãos	OQL		
XRel	desenvolvimento de *parser*	inutiliza as regiões	utiliza junções teta	XPath	expressões /, //	
XParent	desenvolvimento de *parser*	muda os valores dos nós	gera várias junções, suporta ordem entre irmãos	XQuery	expressões /, //,*	
Zhang et al.	desenvolvimento de *parser*	inutiliza os intervalos	utiliza junções teta	XPath	expressões /, //	
Intervalos Dinâmicos	desenvolvimento de *parser*	inutiliza os intervalos	utiliza junções teta	XQuery	expressões /, // e FLWOR	
Hybrid Inlining	indefinido	não afeta	junções entre chaves estrangeiras	XML-QL	expressões /, //	suporta
Zheng et al.	indefinido	não afeta	junções entre chaves estrangeiras	XQuery	indefinido	não suporta
X-Database	indefinido	não afeta	junções entre chaves estrangeiras	possui módulo especial DBCommand		não suporta
LegoDB	indefinido	não afeta	junções entre chaves estrangeiras	XQuery	expressões /, // e FLWOR	não suporta
Oracle 9i	exige conhecimentos em XML Schema	não afeta	CLOB	funções SQL/XML	expressões /, //	não suporta
DB2	exige conhecimentos sobre o DAD	não afeta	CLOB	funções UDF	extração de tipos de dados	não suporta
SQL Server	exige conhecimentos em XML Schema	não afeta	CLOB	XPath	expressões /, //	não suporta

Tabela 3.3: Sumário dos métodos de armazenamento de documentos XML - Parte B

3.10 Análises de Desempenho

Na literatura, não foi encontrado nenhum trabalho que envolva análise de desempenho de todos os métodos anteriormente apresentados. Entretanto, há pelo menos dois trabalhos que comparam diferentes formas de se armazenar um documento XML, incluindo bancos de dados específicos e bancos de dados relacionais. Esta seção apresenta um breve resumo de seus resultados, focado apenas no desempenho dos bancos de dados relacionais.

No trabalho de Tian et al. [80], analisam-se três métodos: Edge, abordagem dos Rótulos e Hybrid Inlining. A seguir, o resumo dos resultados obtidos:

- na reconstrução do documento XML, o tempo gasto pelo Hybrid Inlining é, em média, 45% menor que os tempos dos dois outros métodos; e
- Hybrid Inlining também apresenta os melhores resultados em relação às consultas do tipo / e //. Isso porque as abordagens Edge e dos Rótulos envolvem muitas operações de junção, enquanto algumas consultas podem ser executadas com apenas um SELECT nas tabelas do Hybrid Inlining.

O trabalho de Schmidt et al. [66] apresenta uma análise comparativa para demonstração do *benchmark* proposto, o XMark. O grupo que desenvolveu este trabalho preferiu não identificar os métodos de armazenamento e limitou-se a dizer que dois deles, nomeados A e B, são independentes de estrutura. O método A armazena os dados em apenas uma tabela, enquanto B apresenta excessiva fragmentação dos dados em várias tabelas. Já o método C utiliza o DTD para gerar o esquema relacional. A seguir, o resumo dos resultados obtidos:

- para consultas do tipo / e // que envolvem recuperação de valores, o método C apresentou os melhores resultados, chegando a superar os outros tempos em mais de 60%;
- o método A foi melhor nas consultas que envolvem a função agregada COUNT, por exemplo, uma consulta que conte a quantidade de autores de um documento XML.

3.11 Conclusões e Questões em Aberto

Pode-se chegar a algumas conclusões analisando as tabelas 3.2 e 3.3:

- a abordagem dos intervalos dinâmicos possui o esquema relacional com melhor qualidade dentre as abordagens independentes de estrutura, segundo as métricas avaliadas;
- poucas abordagens consideram a preservação das restrições de integridade na definição do esquema relacional;

- os métodos, em geral, exigem a interação dos desenvolvedores da aplicação. No caso dos produtos comerciais, conhecimentos avançados na área são necessários, impedindo o desenvolvimento de aplicações complexas;
- os mapeamentos independentes de estrutura são muito afetados por atualizações no documento XML, enquanto os mapeamentos dependentes de estrutura não são afetados;
- em geral, a reconstrução do documento XML original exige o processamento de muitas operações de junção, a não ser que se armazene uma cópia do documento;
- poucas abordagens utilizam a linguaguem XQuery e as que a suportam, não implementam todas as suas funcionalidades;
- a utilização de intervalos pelas abordagens independentes de estrutura apresentou vantagens no mapeamento das consultas XML para SQL;
- dos métodos apresentados, o LegoDB é o mais completo, pela utilização do XML Schema, do XQuery e por suportar uma classe maior de consultas; e
- as análises de desempenho contempladas na literatura indicam que a utilização das informações sobre a estrutura de um documento XML (DTD ou XML Schema) parece ser um fator decisivo para que o método atinja bons índices de desempenho.

A seguir, discutem-se algumas questões em aberto:

1. Apesar de existirem trabalhos analisando o desempenho de algumas das abordagens apresentadas, como visto anteriormente na seção 3.10, não há nenhum trabalho completo, envolvendo abordagens dependentes e independentes de estrutura e que seja específico para bancos de dados relacionais.
2. Nenhuma das abordagens dependentes de estrutura utiliza a técnica de intervalos. A tradução das consultas pode ser facilitada pelo armazenamento das posições dos nós, como visto em [19]. A união das abordagens é uma questão em aberto.
3. Como discutido na seção 3.9.2, com exceção do X-Database [83], nenhuma técnica considera as restrições semânticas. Não há trabalhos que analisem o impacto do mapeamento das restrições na avaliação das consultas.
4. Devido à diferença entre o XQuery e o SQL, nenhuma das técnicas implementa todas as funcionalidades oferecidas pelo XQuery. A mais completa delas, a abordagem de Intervalos Dinâmicos, desenvolveu operadores especiais que estendem as operações do

SQL, se aproximando das técnicas aplicadas aos sistemas de banco de dados específicos. Não há um estudo aprofundado sobre quais classes de consultas XQuery podem ser traduzidas para SQL.

5. Também é uma questão em aberto o tratamento da recursão nos documentos DTD e XML Schema, tanto para a criação do esquema relacional, como para a tradução de consultas XML para SQL. A recursão já foi definida pelo padrão SQL:1999, porém as implementações existentes ainda não suportam a sintaxe e a semântica do padrão. Futuras implementações do SQL podem facilitar as traduções de consultas.

6. Quando um esquema relacional é criado, paralelamente uma visão poderia ser definida. No caso das abordagens dependentes de estrutura, o DTD ou XML Schema auxiliariam a definição dessa visão. Se existir uma visão, o problema da tradução de consultas pode ser reduzido para o problema de composição de consultas, amplamente discutido nos métodos de publicação de dados.

No capítulo 4 serão apresentados os métodos de publicação de dados. Estes métodos fazem o mapeamento inverso, ou seja, documentos XML são gerados a partir de dados armazenados em um banco de dados relacional. Assim como nos métodos de armazenamento, uma das questões importantes é a tradução de consultas em linguagem XML para consultas em SQL.

Capítulo 4

Publicação de Dados Utilizando Documentos XML

Para intercâmbio na *Web*, os dados armazenados em bancos de dados relacionais precisam ser mapeados para XML, um processo denominado publicação (*XML publishing*). O mapeamento é complexo porque os modelos de dados diferem significantemente. Dados relacionais são normalizados em várias relações e, normalmente, o esquema é proprietário. Em contraste, XML é uma estrutura aninhada, não normalizada e seu esquema é público. Publicar dados XML envolve junções de tabelas, seleções e projeções dos dados para a criação da hierarquia do documento XML final.

O documento XML resultante do processo de publicação nada mais é do que uma visão sobre o banco de dados relacional. Portanto, ela pode ser virtual ou materializada. Na visão materializada, um documento XML é criado a partir de todos os dados armazenados no banco de dados relacional. As aplicações podem acessar os dados diretamente, ou seja, não há necessidade de requisitá-los aos SGBD. Entretanto, a visão materializada pode precisar de atualizações freqüentes. A maior questão neste cenário é o custo de processamento gasto na criação da visão devido à quantidade de dados manipulados.

Na visão virtual, aplicações solicitam dados através de uma consulta escrita em linguagem XML sobre esta visão. A consulta XML passa por um processamento, responsável por selecionar na visão apenas os dados necessários à resposta. O resultado desse processo, normalmente uma nova consulta, é traduzido para uma ou mais consultas SQL, que serão posteriormente executadas pelo SGBD. As tuplas da tabela resultante são agrupadas para formar o documento XML final. Este cenário, apresentado na figura 4.1, é o mais comum, porque garante dados atualizados e utiliza o processamento de consultas dos SGBDs relacionais.

Uma questão importante neste cenário é a tradução de uma consulta XML em uma ou mais consultas SQL. O processo pode ser complexo devido à própria complexidade da visão

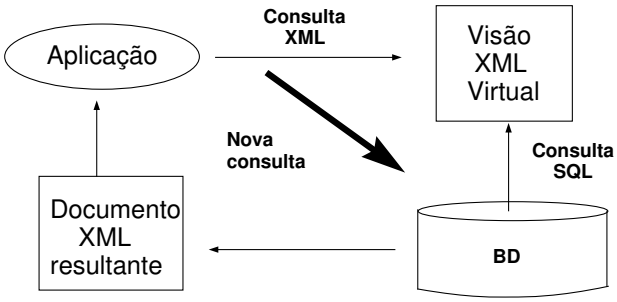

Figura 4.1: Cenário dos métodos de publicação com visão virtual

XML e das consultas XML, dificultando a criação de algoritmos para esse fim. Outra questão é que as consultas geradas automaticamente normalmente não são tão eficazes como as que escrevemos à mão, demandando a criação de algoritmos de otimização.

Este capítulo apresenta uma visão geral dos métodos de publicação contemplados na literatura. A seção 1 apresenta a classificação, as seções 2 e 3 explicam como os métodos funcionam, a seção 4 apresenta uma análise comparativa entre eles e a seção 5 finaliza o capítulo com as conclusões e alguns dos problemas em aberto.

Para as seções seguintes, adotou-se o esquema relacional da figura 4.2.

```
Artigo(artigoid,titulo, autorcontato, ano)
Livro(livroid, titulo, editor)
Autor(id, nome, sobrenome, endereco, email)
ArtigoAutor(artigoid, id)
LivroAutor(livroid, id)
```

Figura 4.2: Esquema relacional para o domínio da biblioteca

4.1 Classificação

O cenário comum é criar uma visão virtual a partir do esquema relacional do banco de dados. Como a visão também pode ser considerada um esquema, podemos dizer que define-se um esquema global como visão de um esquema local. Por exemplo, para se criar um esquema global envolvendo títulos de livros, pode-se utilizar a tabela Livro:

```
Livro(livroid, titulo, editor) ⇒ TituloLivro (livroid, titulo)
```

Essa abordagem é conhecida como GAV (*global-as-view*). A abordagem oposta, denominada LAV (*local-as-view*), define o esquema do banco de dados como uma visão do esquema global, ou seja, o conteúdo do banco de dados é descrito como uma consulta sobre as tabelas do esquema global. Por exemplo, suponha que o esquema global possua a tabela Artigo e que no banco de dados deseja-se armazenar artigos publicados antes de 1990 e artigos publicados depois de 2000. Duas tabelas podem ser definidas:

```
ArtigosAntigos (artigoid, titulo) ⇒ Artigo (artigoid, titulo, autorcontato, ano) e ano < 1990
ArtigosRecentes (artigoid, titulo) ⇒ Artigo (artigoid, titulo, autorcontato, ano) e ano > 2000
```

A primeira classificação agrupa os métodos de publicação em relação ao tipo de abordagem que utilizam, GAV ou LAV, como pode ser visto na figura 4.3.

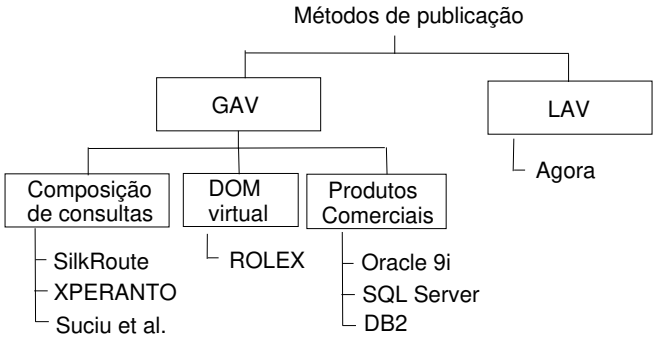

Figura 4.3: Classificação dos métodos de publicação

Os métodos de publicação que utilizam a abordagem GAV ainda podem ser divididos da seguinte maneira:

- Métodos que utilizam composição de consultas - são aqueles que se enquadram no cenário mais comum. As consultas XML são compostas com a visão virtual e uma nova consulta é gerada e traduzida para SQL.
- Método que utiliza uma árvore DOM virtual - trata-se de um método que simula as operações do DOM, sem criar a árvore fisicamente.
- Métodos utilizados em produtos comerciais - são classificados em um grupo especial devido à falta de informações detalhadas sobre eles.

Ao contrário do que ocorre nos métodos de mapeamento de documentos XML para bancos de dados relacionais, o aspecto cronológico não é tão marcante nos métodos de publicação. A maioria dos métodos foram desenvolvidos entre 2001 e 2002, como mostra a figura 4.4.

Nas seções 2 e 3, os métodos serão melhor detalhados.

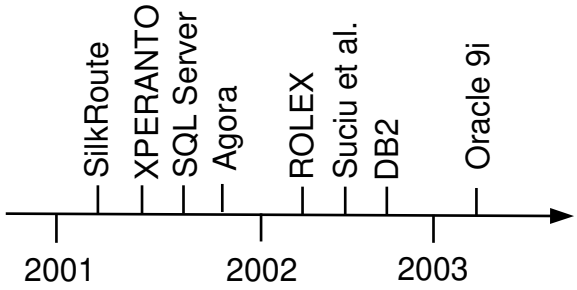

Figura 4.4: Linha do tempo do desenvolvimento dos métodos de publicação

4.2 Métodos de publicação que utilizam a abordagem GAV

4.2.1 Projeto SilkRoute

O projeto SilkRoute [28, 29, 30], desenvolvido por Fernández et al., trata o problema da publicação de dados criando uma visão virtual ou materializada dos dados armazenados no banco de dados relacional.

SilkRoute utiliza uma linguagem declarativa denominada RXL (*Relational to XML Transformation Language*), especificamente desenvolvida para criar uma visão virtual dos dados armazenados no banco de dados relacional. Uma consulta RXL possui as cláusulas FROM, WHERE e CONSTRUCT. A cláusula FROM determina quais tabelas serão manipuladas e as associa a variáveis. A cláusula WHERE especifica as condições necessárias para a realização da consulta. A cláusula CONSTRUCT cria as *tags* do documento XML e determina quais campos das tabelas formarão seu conteúdo. A consulta RXL precisa ser escrita manualmente pelo desenvolvedor da aplicação ou outra pessoa que conheça o esquema relacional. A figura 4.5 apresenta uma consulta RXL simplificada para o domínio da bibliografia (esquema relacional da figura 4.2).

O elemento <`artigo`> é construído a partir da tabela Artigo, associada à variável `$art`. Os subelementos de <`artigo`> possuem como conteúdo os valores dos campos com o mesmo nome. Por exemplo, o elemento <`titulo`> é preenchido com o valor do campo `titulo`. Para os elementos <`autor`>, define-se uma consulta RXL aninhada, com as operações de junção necessárias presentes na cláusula WHERE. Os subelementos de <`autor`> são formados pelos campos da tabela Autor. O elemento <`livro`> é obtido de maneira semelhante ao elemento <`artigo`>.

Uma aplicação que necessita recuperar dados deverá formular uma consulta em XML-

```
CONSTRUCT
<bibliografia>
       {
       FROM Livro $l
       CONSTRUCT
              <livro>
              ⋮
              </livro>
       }
       {
       FROM Artigo $art
       CONSTRUCT
              <artigo>
                     <titulo>$art.titulo</titulo>
                     {
                     FROM Autor $a, ArtigoAutor $x
                     WHERE $art.artigoid = $x.artigoid and $a.id = $x.id
                     CONSTRUCT
                            <autor id = $a.id>
                                   <nome>$a.nome</nome>
                                   <sobrenome>$a.sobrenome</sobrenome>
                                   <email>$a.email</email>
                            </autor>
                     }
                     <autorcontato IDautor = $art.autorcontato/>
                     <ano>$art.ano</ano>
              </artigo>
       }
</bibliografia>
```

Figura 4.5: Consulta RXL para a criação da visão virtual no SilkRoute

QL (visto anteriormente na seção 2.4.4), baseando-se na visão virtual criada pela consulta RXL. Como exemplo, suponha que deseja-se saber quais os títulos de artigos publicados em 2001. Considerando a visão gerada pela consulta RXL da figura 4.5, a consulta XML-QL é apresentada na figura 4.6.

```
WHERE <bibliografia>
          <artigo>
                <titulo>$t</titulo>
                <ano> 2001 </ano>
          </artigo>
       </bibliografia> IN "www.unicamp.br/visao.xml"
CONSTRUCT <resultado>
             <artigo>
                   <titulo>$t</titulo>
             </artigo>
          </resultado>
```

Figura 4.6: Consulta XML-QL para recuperar títulos de artigos publicados em 2001

Tanto a consulta XML-QL como a consulta RXL são enviadas ao primeiro módulo do

SilkRoute, o módulo Compositor. A arquitetura do sistema é apresentada na figura 4.7.

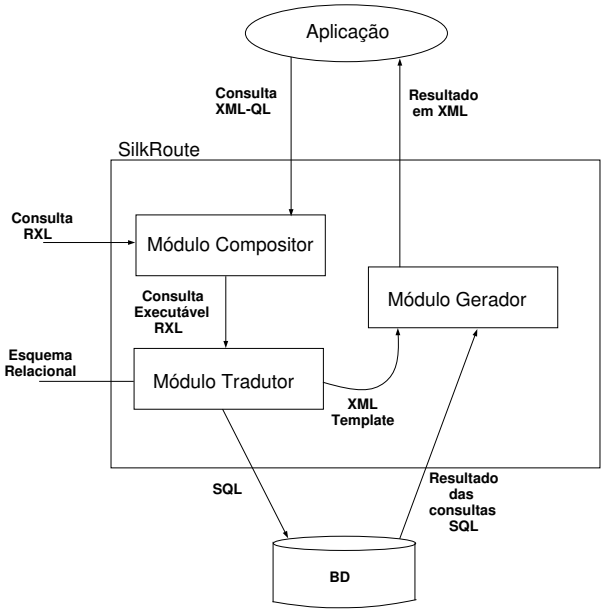

Figura 4.7: Arquitetura do sistema SilkRoute

No módulo Compositor, realiza-se uma composição entre as duas consultas para gerar uma nova consulta RXL denominada consulta executável. A composição de consultas objetiva a minimização da quantidade de dados necessários à formação do documento XML resultante. A idéia do algoritmo de composição é associar os elementos que aparecem na cláusula WHERE da consulta XML-QL diretamente com os elementos que aparecem na cláusula CONSTRUCT do RXL, eliminando os elementos que não fazem nenhuma associação. Além disso, as condições da cláusula WHERE do XML-QL também passam a ser as condições da consulta RXL executável. A consulta executável para o domínio da bibliografia é mostrada na figura 4.8. Note que as *tags* do resultado são as mesmas da consulta XML-QL e a condição `ano = 2001` aparece na cláusula WHERE.

A consulta executável é enviada ao Módulo Tradutor, junto com o esquema relacional, para que seja decomposta em uma ou mais consultas SQL. O Módulo Tradutor também gera um *template* XML que servirá de entrada para o módulo Gerador, contendo a estrutura das *tags* que formarão o documento resultante.

O algoritmo de tradução de consultas do SilkRoute utiliza uma representação intermediária para as consultas RXL denominada "*view tree*". Trata-se de um grafo que modela o relacionamento entre as tabelas envolvidas na resposta. Por exemplo, se houver uma consulta aninhada como ocorre na figura 4.5 envolvendo as tabelas Artigo e Autor, na *view*

```
CONSTRUCT
   <resultado>
      {
            FROM Artigo $art
            WHERE $art.ano = 2001
            CONSTRUCT
                  <artigo>
                        <titulo>$art.titulo</titulo>
                  </artigo>
      }
   </resultado>
```

Figura 4.8: Consulta RXL executável para o domínio da bibliografia

tree deverá existir uma aresta conectando-as com o rótulo "*", indicando que o elemento <artigo> possui vários subelementos <autor>. Se o elemento possuir exatamente um subelemento, a aresta recebe rótulo 1. Como a consulta RXL executável da figura 4.8 gera uma *view tree* sem arestas, apresentaremos na figura 4.9 a *view tree* simplificada para a consulta RXL da figura 4.5.

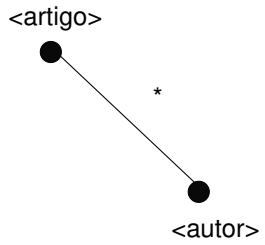

Figura 4.9: Exemplo de *view tree* no SilkRoute

A partir da *view tree*, criam-se as consultas SQL. Uma aresta com rótulo 1 gera uma operação de junção simples, enquanto um aresta com o asterisco gera uma operação de *outer join*. Utiliza-se *outer join* porque pode existir, por exemplo, artigos sem autores, entretanto os dados do artigo devem aparecer no resultado final. A consulta SQL gerada para o exemplo da bibliografia é mostrada na figura 4.10.

SilkRoute aplica algumas transformações à *view tree*, de maneira que diferentes consultas SQL sejam geradas. Calculam-se os custos da execução das mesmas pelo SGBD e escolhe-se o conjunto de consultas SQL mais eficiente. Esta técnica assemelha-se ao trabalho de Zheng et al. [89] e do LegoDB [6] apresentadas no capítulo 3.

Por fim, executam-se as consultas SQL e o Módulo Gerador reúne os resultados para gerar o documento XML final com as *tags* previamente especificadas no *template*.

```
SELECT art.titulo
FROM Artigo as art
WHERE art.ano = 2001
```

Figura 4.10: Consulta SQL gerada no SilkRoute para o domínio da bibliografia

SilkRoute também oferece a reconstrução do documento XML completo, ou seja, a visão materializada. Basta enviar a consulta RXL definida para a criação da visão virtual direto para o Módulo Gerador.

4.2.2 Projeto XPERANTO

O projeto XPERANTO [70, 71, 72, 73], desenvolvido por Shanmugasundaram et. al, é um sistema intermediário entre o banco de dados relacional e a aplicação, responsável por fornecer uma visão XML dos dados e receber consultas formuladas em XQuery.

Primeiramente, cria-se uma visão virtual padrão (*default*). Trata-se de uma visão simples, pois cada tabela do banco de dados é mapeada para um elemento com a *tag* `<nome-da-tabela>` e suas tuplas tornam-se subelementos do tipo `<row>`. A figura 4.11 mostra a visão virtual simplificada para o domínio da bibliografia.

```
<db>
  <artigo>
    <row>
      <artigoid>1</artigoid><titulo>Updating XML</titulo><autorcontato>2</autorcontato><ano>2001</ano>
    </row>
  </artigo>
  <autor>
    <row>
      <id>1</id><nome>Fernando</nome><sobrenome>Meirelles</sobrenome><endereco>Sao Paulo</endereco>
    </row>
    <row>
      <id>2</id><nome>Igor</nome><sobrenome>Tatarinov</sobrenome><email>igor@cs.washington.edu</email>
    </row>
    <row>
      <id>3</id><sobrenome>Weld</sobrenome><email>weld@cs.washington.edu</email>
    </row>
  </autor>
  .
  .
  .
</db>
```

Figura 4.11: Visão virtual padrão do XPERANTO

O usuário pode definir uma visão mais adequada à sua aplicação formulando-a em XQuery sobre a visão padrão. A figura 4.12 mostra a criação da visão denominada bibliografia em XQuery. O documento resultante é semelhante ao da figura 2.1.

```
create view bibliografia as(
  return <bibliografia> {
    for $livro in view("default")/livro/row
      return
       <livro>
         <titulo>$livro/titulo</titulo>
            {for $livautor in view("default")/livro-autor/row,
               $autor in view("default")/autor/row,
            where $livautor/livroid = $livro/livroid and $livautor/id = $autor/id
            return
              <autor id = $autor/id>
                <nome>$autor/nome</nome>
                <sobrenome>$autor/sobrenome</sobrenome>
                <endereco>$autor/endereco</endereco>
              </autor>}
         <editor nomecompleto=$livro/editor /> }
       </livro>
       <artigo>
       ...
       construído de maneira similar ao elemento <livro>
       ...
       </artigo>
  </bibliografia> )
```

Figura 4.12: Consulta XQuery para criação de uma nova visão em XPERANTO

Pode-se criar vários níveis de abstração, ou seja, definição de visões mais específicas para uma determinada aplicação. Após a definição da nova visão, formulam-se consultas sobre ela. Por exemplo, uma consulta que deseja recuperar os títulos de artigos publicados em 2001 é mostrada na figura 4.13.

```
<resultado> {
    FOR $artigo IN VIEW("bibliografia")/bibliografia/artigo
    WHERE $artigo/ano = 2001
    RETURN
       <artigo>
          <titulo>$artigo/titulo</titulo>
       </artigo> }
 </resultado>
```

Figura 4.13: Consulta XQuery para recuperar títulos de artigos publicados em 2001

O processo de tradução da consulta XQuery para SQL inicia-se pela sua composição com a consulta que cria a visão (figura 4.12). O objetivo da composição é transformar consultas complexas em consultas mais simples, eliminando informações desnecessárias. Para isso, XPERANTO utiliza uma representação interna denominada XQGM (*XML Query Graph Model*). XQGM é um grafo formado por um conjunto de operadores e funções para navegação. A maioria dos operadores possuem o mesmo significado do SQL, como `select`, `project`, `join` e `orderby`. As funções de navegação servem para criar e manipular os

elementos e atributos do documento XML. Englobam funções para recuperação de dados (como *GetContents()* para recuperar subelementos e *getAttributes()* para recuperar atributos) e de construção de elementos (como *cr8Elem()* para criar um elemento e *cr8XMLFrags()* para criar uma fragmento do documento XML). A manipulação do grafo XQGM é totalmente transparente à aplicação. A figura 4.14 apresenta o XQGM simplificado para a consulta XQuery de criação da visão bibliografia (figura 4.12).

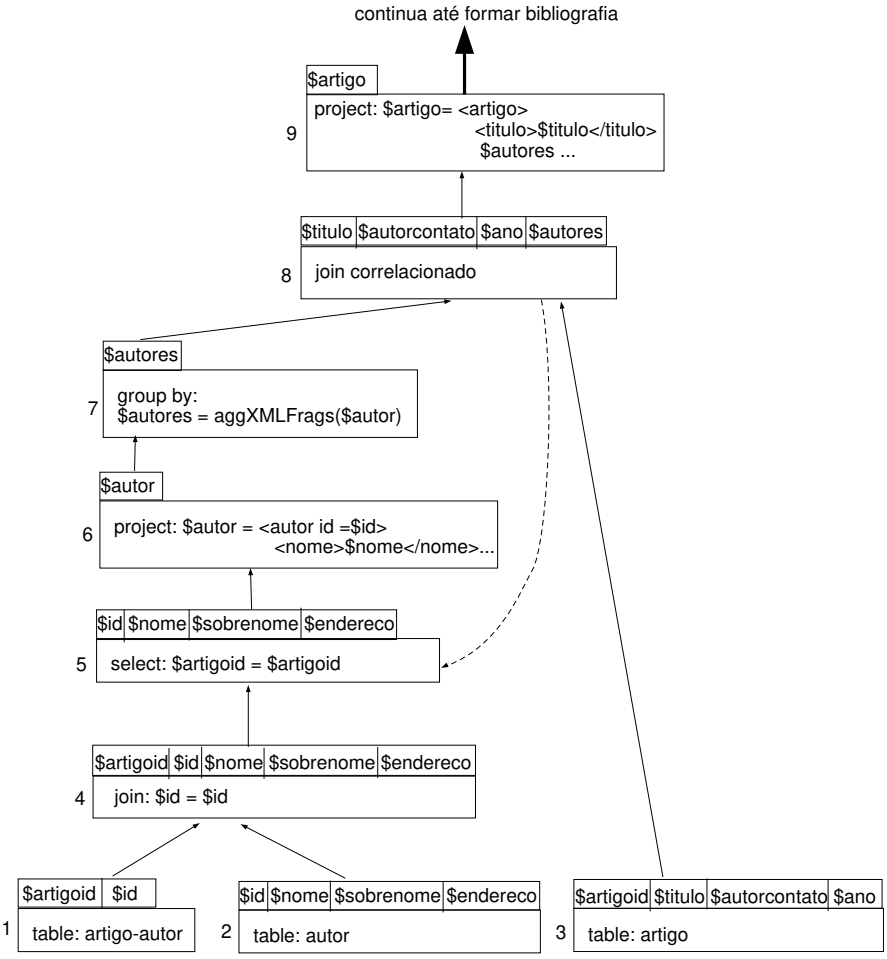

Figura 4.14: Consulta XQuery na representação XQGM para a visão bibliografia

Nas caixas 1, 2 e 3 projetam-se os campos das tabelas correspondentes (os nomes dos campos aparecem no topo da caixa). Na caixa 4, realiza-se uma operação de junção no campo `id` do autor. Na caixa 5, selecionam-se os autores para um determinado artigo, especificado pelo campo `artigoid`. Criam-se as *tags* para o elemento <`autor`> na caixa 6 e na caixa 7,

eles são agrupados. Na caixa 9, cria-se o elemento <artigo>. A caixa 8 apenas indica que as consultas são aninhadas. O elemento <livro> seria criado de maneira similar.

Na verdade, cada caixa possui várias funções de navegação especificando como os elementos são gerados. Por exemplo, a figura 4.15 expande a caixa 6, detalhando a construção o elemento autor.

```
cr8Elem(autor,
        cr8AttList(cr8Att(id, $id)),
        cr8XMLFragList (cr8Elem(nome, cr8AttList(), cr8XMLFragList($nome)),
                        cr8Elem(sobrenome, cr8AttList(), cr8XMLFragList($sobrenome)),
                        cr8Elem(endereco, cr8AttList(), cr8XMLFragList($endereco))))
```

Figura 4.15: Detalhamento da construção do elemento autor com funções de navegação

A função `cr8Elem(Tag, Atts,Clist)` cria um elemento com a *tag* `Tag`, os atributos `Atts` e os subelementos `Clist`. A função `cr8Att(Nome, Val)` cria um atributo com nome `Nome` e conteúdo `Val`.

A figura 4.16 apresenta o XQGM para a consulta da figura 4.13. A caixa 1 indica qual visão será avaliada. Na caixa 2, recuperam-se todos os elementos e atributos filhos de bibliografia, enquanto na caixa 3, a hierarquia aninhada do XML é retirada. Na caixa 4, selecionam-se os filhos que são elementos com *tag* `artigo` e na caixa 5, recuperam-se seus subelementos. Na caixa 6, os subelementos de <artigo> são desaninhados e na caixa 7 há a seleção do elemento <ano>. Na caixa 8, extrai-se o conteúdo do elemento <ano> e na caixa 9, retira-se a estrutura aninhada. Na caixa 10, verifica-se o texto do elemento <ano>. Se for igual a 2001, a caixa 11 retorna os títulos pertencentes ao resultado.

A criação dos grafos XQGM é realizada no Módulo Parser XQuery do XPERANTO. A figura 4.17 apresenta a arquitetura completa do sistema.

O processamento de composição das consultas é feito no Módulo Compositor de Visões. Faz-se a composição aplicando-se algumas regras que simplificam funções correspondentes. Por exemplo, considere a função `cr8Elem(Tag, Atts,Clist)` e a função `getAttributes (elemento)`. Se elas forem aplicadas ao mesmo elemento, o retorno da função `getAttributes()` será o segundo parâmetro da função `cr8Elem()`. Se a caixa 5 da figura 4.16 for composta com a caixa 9 da figura 4.14, a consulta fica reduzida ao conteúdo do elemento <artigo>. Como estas, há outras 10 regras de composição para simplificar as consultas.

O resultado do algoritmo de composição é um novo grafo XQGM. A figura 4.18 apresenta o XQGM após a composição. Note que a condição (ano = 2001) passou para o XQGM da figura 4.14 e toda árvore do elemento <autor> foi retirada, pois era desnecessária à resposta.

O grafo XQGM passa para o Módulo Gerador de Consultas SQL que definirá uma única consulta SQL. Para melhorar seu desempenho, esta consulta será decorrelacionada[1]. A figura

[1]No decorrelacionamento, uma consulta SQL correlacionada passa a ser expressa por uma operação de

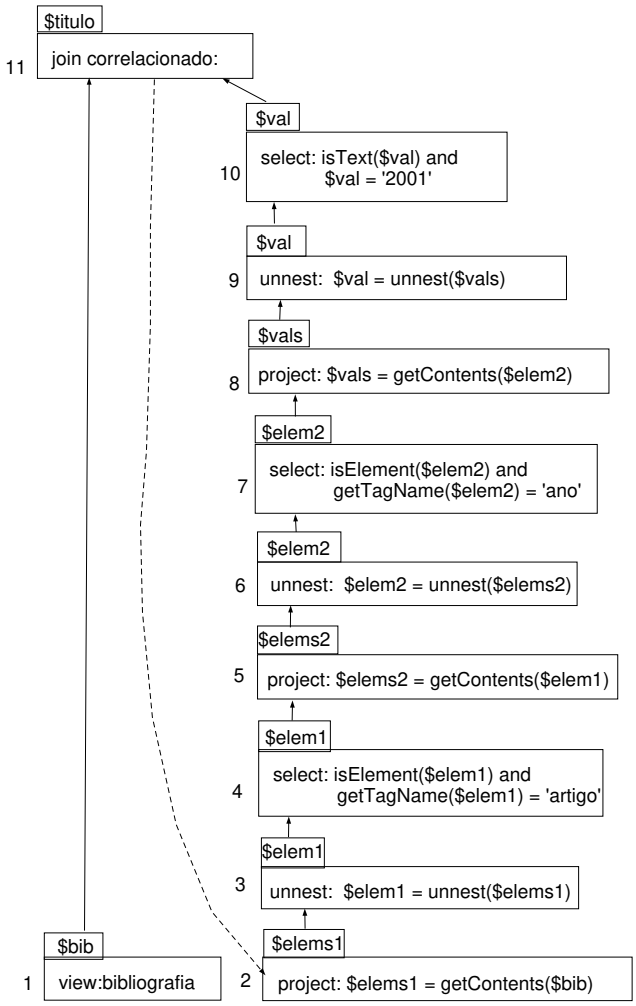

Figura 4.16: Consulta em XQuery para recuperar os títulos de artigos publicados em 2001 na representação XQGM

4.19 mostra o decorrelacionamento da consulta da figura 4.14. Faz-se a junção dos autores selecionados diretamente com a tabela artigo (caixa 5), ao invés de se executar uma condição de seleção correlacionada.

O último módulo, o Módulo Inserção de Tags, serve para estruturar o resultado das consultas executadas no SGBD.

junção entre as tabelas.

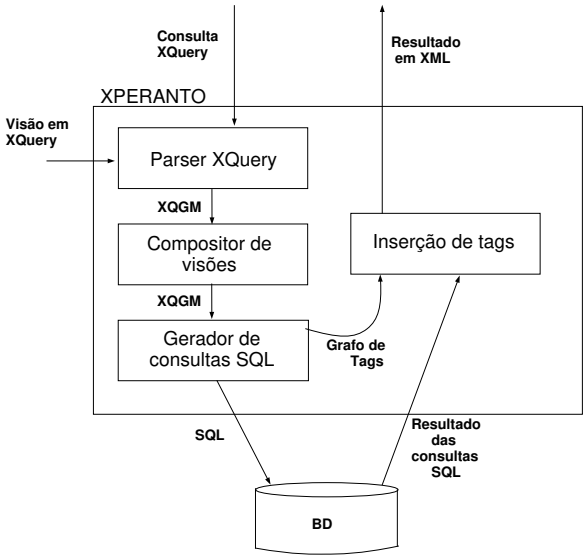

Figura 4.17: Arquitetura do sistema XPERANTO

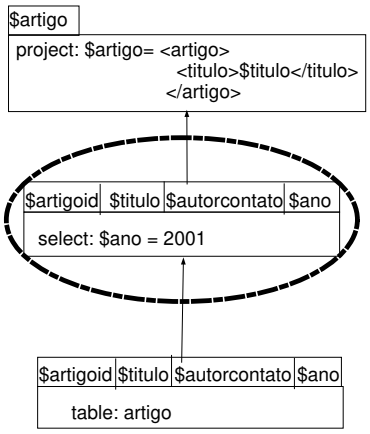

Figura 4.18: Grafo XQGM após composição das consultas

4.2.3 Método de tradução de consultas XSLT para SQL

O trabalho de Suciu et al. [43] apresenta um algoritmo para tradução de programas escritos em XSLT para consultas SQL. Considera-se que já exista uma visão virtual com todos os dados do banco de dados relacional, como a *view tree* do SilkRoute [28]. Neste contexto, o sistema avalia a consulta XSLT sobre a visão virtual, gera as consultas SQL, as executa

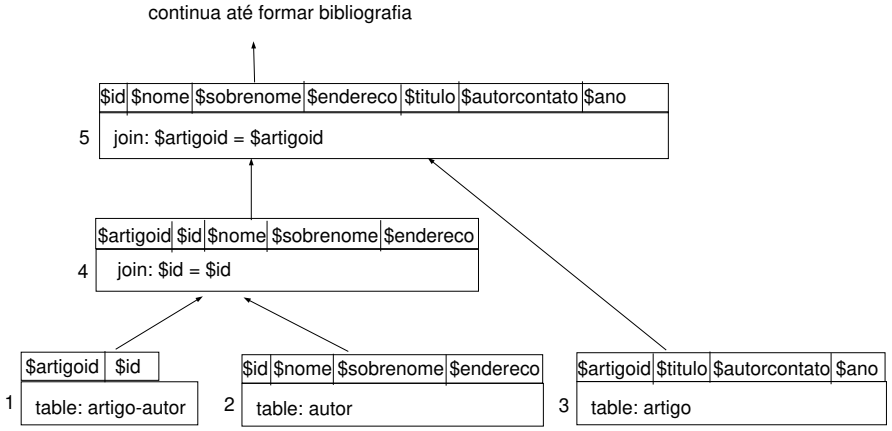

Figura 4.19: Grafo XQGM da figura 4.14 decorrelacionado

no SGBD e cria o documento XML resultante. A figura 4.20 apresenta a arquitetura do sistema.

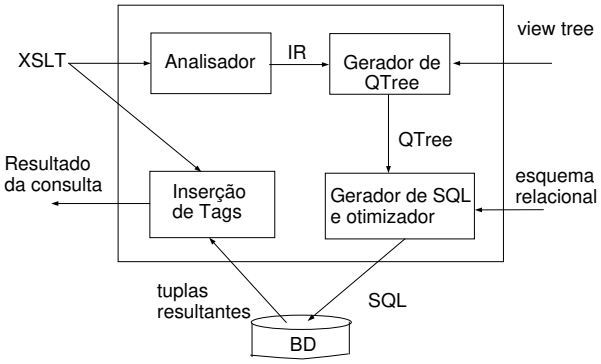

Figura 4.20: Arquitetura do sistema de tradução de consultas XSLT para SQL

Além do documento XSLT, o sistema recebe a *view tree*, que define quais consultas SQL devem ser executadas para a recuperação de cada elemento XML. Suponha que já exista a visão virtual bibliografia.

No Módulo Analisador, o documento XSLT é convertido para uma representação funcional: para cada regra de *template* do XSLT, especifica-se uma função, com parâmetros e chamadas para outras funções. Para o exemplo da figura 2.9 que recupera títulos de artigos publicados em 2001, duas funções são geradas:

```
f0-raiz($default) = f1($default/bibliografia)
f1-bibliografia($default) = if ($default/artigo/ano = 2001),
                            return ($default/artigo/titulo)
```

A função `f0` recebe a visão como parâmetro e chama a função `f1` enviando o elemento <`bibliografia`> como parâmetro. Por sua vez, `f1` verifica a condição expressa no `if` e se ela for verdadeira, retorna o valor especificado pelo `return`.

Ainda no Módulo Analisador, cria-se uma representação intermediária (IR), identificando as chamadas entre as funções. Como no exemplo há duas funções, o grafo IR é bem simples (figura 4.21).

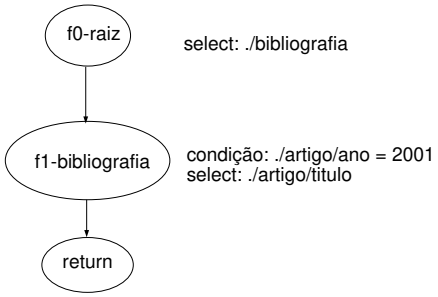

Figura 4.21: IR para o documento XSLT da figura 2.9

Passa-se o IR para o Módulo Gerador de QTree. A QTree não passa de uma composição entre o IR e a visão virtual, que captura o caminho percorrido pela consulta na visão, as condições impostas aos nós e os valores retornados (precedido por #). A figura 4.22 apresenta a QTree para o exemplo, com apenas um nó. Nesta QTree fica explícito que a consulta percorre os subelementos de <`artigo`> e se `V1` (ano) for igual a 2001, o valor de <`titulo`> é retornado.

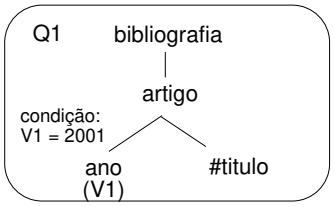

Figura 4.22: Exemplo de QTree

A QTree é repassada para o módulo Gerador de SQL e Otimizador, responsável pela geração das consultas SQL. Cada nó da QTree se torna uma consulta SQL, que serão otimi-

zadas e enviadas ao SGBD. Para o exemplo, apenas uma consulta SQL é gerada, apresentada na figura 4.23.

```
SELECT titulo
FROM Artigo
WHERE ano = 2001
```

Figura 4.23: Consulta SQL gerada para a QTree da figura 4.22

No módulo Inserção de Tags, inserem-se as *tags* para a criação do documento XML final.

4.2.4 Projeto ROLEX

O sistema ROLEX (*Relational On-Line Exchange with XML*) [7, 8], desenvolvido por Korth et al., traz uma abordagem nova para a recuperação de dados armazenados em banco de dados relacionais. ROLEX considera que o documento XML textual provido pelos sistemas descritos anteriormente não é suficiente para a maioria das aplicações. Isso porque o desenvolvedor dessas aplicações precisa manipular o documento através de um *parser*, como o DOM (definido na seção 2.4.3).

O projeto ROLEX provê uma visão do banco de dados na forma de uma árvore DOM para que a aplicação possa navegar pelos nós da árvore através das operações providas pela interface, como navegação do nó pai para um nó filho, do nó filho para o nó pai, navegação entre nós irmãos ou até um nó com uma *tag* específica. Entretanto, a árvore DOM não é criada fisicamente. As operações do DOM são dinamicamente mapeadas para consultas em SQL, evitando o custo da geração do documento XML textual e o *parsing* para a criação da árvore DOM. A figura 4.24 mostra a visão geral de como o sistema ROLEX funciona.

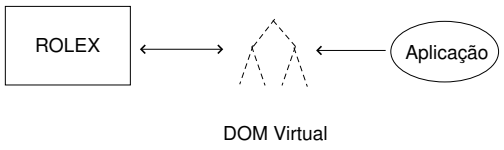

Figura 4.24: Funcionamento do Sistema ROLEX

ROLEX primeiramente cria uma visão do banco de dados relacional em forma de árvore, como mostra a figura 4.25.

Cada nó da árvore possui um cabeçalho contendo uma *tag*, uma variável e um ou mais parâmetros. Além disso, o nó possui uma consulta SQL. As tuplas resultantes da consulta se

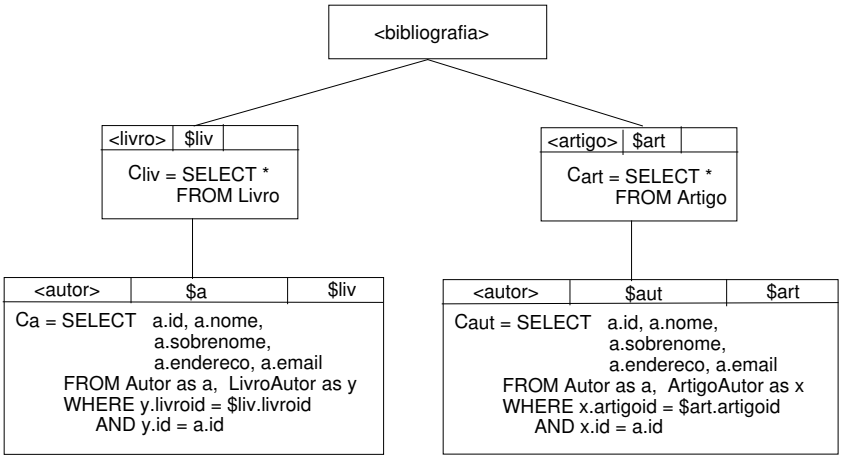

Figura 4.25: Visão do banco de dados relacional no ROLEX

tornarão elementos do documento XML com a *tag* especificada no cabeçalho. Por exemplo, cada tupla resultante da consulta C_{art} (primeiro nó filho à direita) se tornará um elemento <artigo> do documento XML. Os campos podem ser mapeados para subelementos ou atributos, entretanto, esses detalhes não serão mostrados. A variável do cabeçalho identifica a consulta para que ela possa ser utilizada posteriormente como parâmetro. Por exemplo, a consulta C_{aut} (nó folha à direita) utiliza um campo do resultado da consulta C_{art} através da variável `$art` passada como parâmetro.

Em seguida, cria-se um "plano de navegação" para cada nó da árvore. Esse plano de navegação define um subplano especificando como a consulta deverá ser executada e um índice para armazenar o resultado obtido após a execução.

Por fim, criam-se procedimentos que irão simular as operações da árvore DOM. Cada procedimento determina quais as consultas da árvore ROLEX deverão ser executadas para que o resultado requisitado pela operação DOM seja obtido. Como exemplo, considere o procedimento que simula a operação ``navegar-para-filho(f)''. Suponha que a consulta do nó f seja C_f. Antes de executar a consulta C_f, as consultas dos nós predecessores na árvore ROLEX devem ter sido executadas. Para saber isso, basta verificar o índice do plano de navegação de cada nó. Se ele existir, significa que a consulta já foi materializada e o resultado pode ser utilizado na execução de C_f. Senão, executa-se a consulta correspondente e armazena-se o resultado no índice.

ROLEX otimiza os subplanos dos planos de navegação dos nós através de perfis de navegação. Se uma aplicação requer informações de uma subárvore com maior freqüência (por exemplo, faz mais consultas em artigos), o subplano de execução das consultas SQL é

otimizado através de decorrelacionamento (visto anteriormente na seção 4.2.2).

Em um trabalho mais recente [54], os autores utilizam a estrutura da árvore ROLEX para processar um subconjunto de consultas elaboradas em XSLT [16]. A idéia básica neste trabalho é construir uma árvore para a consulta XSLT, fazer uma composição dela com a árvore ROLEX para criar uma nova árvore, mais simplificada e que materialize apenas os dados necessários.

4.2.5 Oracle

O Oracle 9i e 10g [26, 37] suportam a criação de visões através de funções da linguagem SQL/XML [36], apresentadas na tabela 4.1.

Função	*Descrição*
`xmlElement()`	cria um elemento
`xmlAttributes()`	adiciona atributos a um elemento
`xmlForest()`	cria uma floresta de elementos
`xmlAgg()`	cria apenas um elemento para uma coleção de elementos

Tabela 4.1: Funções do SQL/XML para criação de visões no Oracle

A figura 4.26 mostra como seria a definição de uma visão virtual no Oracle utilizando as funções da tabela 4.1. A função `xmlForest()` constrói os elementos <`sobrenome`> e <`email`>. A função `xmlAttributes()` define o atributo `id`. A segunda função `xmlElement()` agrupa os elementos previamente contruídos dentro do elemento <`autor`>. Por fim, a primeira função `xmlElement()` agrega todos os elementos do tipo <`autor`> no elemento <`listaautores`>.

```
SELECT xmlElement("listaautores",
                    xmlElement("autor", xmlAttributes(a.id as "id"),
                                  xmlForest(a.sobrenome as "sobrenome", a.email as "email"))
FROM Autor as a
```

Figura 4.26: Definição de visão virtual no Oracle

As consultas são realizadas sobre a visão utilizando as funções do SQL/XML apresentadas na tabela 3.1, já que a visão pode ser armazenada como um XMLType no banco de dados.

4.2.6 IBM

O DB2 XML Extender [15, 39] suporta a geração de documentos XML através de dois procedimentos armazenados (*stored procedures*): `dxxGenXML()` e `dxxRetrieveXML()`. Os dois

só funcionam se um arquivo DAD (definido na seção 3.8.2) já foi utilizado para mapear os elementos de um documento XML para o banco de dados relacional. Os procedimentos armazenados também utilizam o DAD para reconstruir o documento XML, fazendo o processo inverso.

Outra maneira de publicar dados armazenados no DB2 é utilizando o XML for Tables [40]: baseado no XPERANTO [72], ele cria visões do banco de dados relacional e transforma consultas XML em consultas SQL. Como trata-se de um projeto ainda em pesquisa, XML for Tables suporta uma linguagem similar ao XQuery. Todos os processos são realizados através de procedimentos armazenados, como `table()` para a criação de uma visão padrão e `xperanto.xperantosp()` para envio das consultas XML.

4.2.7 Microsoft

No SQL Server [62], os usuários podem executar consultas em SQL e obter o resultado em documento XML através do uso da cláusula FOR XML. A cláusula é alocada no final da consulta SQL e inclui um dos três modos: RAW, AUTO ou EXPLICIT. O modo RAW cria um elemento `<row>` para cada tupla da tabela resultante. A figura 4.27 apresenta um exemplo para o modo RAW aplicado ao domínio da bibliografia.

```
---- CONSULTA SQL ----

SELECT nome, sobrenome
FROM Autor
FOR XML RAW

---- DOCUMENTO XML RESULTANTE ----

<row nome = "Fernando"  sobrenome = "Meirelles"/>
<row nome = "Igor"  sobrenome = "Tatarinov"/>
<row sobrenome = "Weld"/>
```

Figura 4.27: Exemplo do uso do FOR XML no modo RAW

No modo AUTO, o documento XML passa a ser melhor estruturado. As tabelas da cláusula FROM do SQL se transformam em elementos e os campos da cláusula SELECT passam a ser subelementos ou atributos. A figura 4.28 apresenta um exemplo.

No modo EXPLICIT, o desenvolvedor controla a estrutura do documento XML resultante. Entretanto, a consulta SQL deve ser escrita adicionando-se as informações explicitamente. A figura 4.29 apresenta um exemplo. As informações sobre o aninhamento dos elementos aparecem entre colchetes na cláusula SELECT. A linha `[resultado!1!nome!element]`, por exemplo, indica que o sistema deve criar uma nova *tag* para `resultado` e `nome` será um de seus subelementos.

```
 ---- CONSULTA SQL ----

 SELECT nome, sobrenome
 FROM Autor
 FOR XML AUTO

 ---- DOCUMENTO XML RESULTANTE ----

<autor>
       <nome>Fernando</nome>
       <sobrenome>Meirelles</sobrenome>
</autor>
<autor>
       <nome>Igor</nome>
       <sobrenome>Tatarinov</sobrenome>
</autor>
<autor>
       <sobrenome>Weld</sobrenome>
</autor>
```

Figura 4.28: Exemplo do uso do FOR XML no modo AUTO

```
 ---- CONSULTA SQL ----

 SELECT 1 as TAG
        nome as [resultado!1!nome!element],
        sobrenome as [resultado!1!sobrenome!element],
 FROM Autor
 FOR XML EXPLICIT

 ---- DOCUMENTO XML RESULTANTE ----

<resultado>
       <nome>Fernando</nome>
       <sobrenome>Meirelles</sobrenome>
</resultado>
<resultado>
       <nome>Igor</nome>
       <sobrenome>Tatarinov</sobrenome>
</resultado>
<resultado>
       <sobrenome>Weld</sobrenome>
</resultado>
```

Figura 4.29: Exemplo do uso do FOR XML no modo EXPLICIT

Além do uso do FOR XML, pode-se criar um documento XML a partir de um *XSD annotated schema* (definido da seção 3.8.3). As informações extras inseridas no *schema* definem a visão do banco de dados relacional.

Como visto anteriormente, pode-se formular consultas em XPath para obter dados que seguem o *XSD annotated schema*. Estas consultas são transformadas em consultas SQL com a cláusula FOR XML no modo EXPLICIT.

4.3 Método de publicação que utiliza a abordagem LAV

4.3.1 Projeto Agora

O projeto Agora [56, 57], desenvolvido por Manolescu et al., possui uma abordagem completamente diferente dos trabalhos apresentados anteriormente. O sistema Agora adota a abordagem LAV. A figura 4.30 apresenta a arquitetura do sistema.

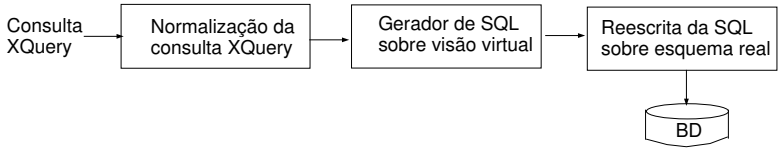

Figura 4.30: Arquitetura do sistema Agora

O esquema global é quase uma representação da estrutura do documento XML, como pode ser visto na figura 4.31.

```
Document (docID, docURIID, rootElemID)
URI (uriID, uriValID)
ProcInstr (piID, piValID, piVal2ID)
QName (qNameID, qnPrefixID, qnLocalID)
Attribute (attrID, attrElID, attrNameID, attrValID)
Element (elID, elQNameID, elTypeID)
Namespace (nsID, nsValID, nsURIID)
Comment (commID, commValID)
Value (valID, value)
Child (parentID, childID, childValID, chilIndex)
```

Figura 4.31: Esquema global do sistema Agora

No Agora, uma consulta XQuery deve ser elaborada sobre o esquema global. Esta consulta é normalizada para que se adeque a um dos *templates* que permitem a tradução direta para SQL. Como exemplo, a figura 4.32 apresenta o *template* T1 que recupera valores de atributos, ou seja, uma consulta do tipo `E1/@attName`, onde `E1` é o elemento e `attName` é o nome do atributo. As funções `S(E1)`, `F(E1)` e `W(E1)` calculam as cláusulas FROM, WHERE e SELECT do elemento `E1`, respectivamente, de maneira recursiva.

O último passo é reescrever as consultas SQL geradas para consultas SQL sobre o esquema local. Essas consultas são executadas e as *tags* inseridas para criação do documento XML final.

```
SELECT a.attrID
FROM F(E1), Attribute as a, Value as v
WHERE W(E1) and a.attrElID = S(E1)
          and a.attrNameID = v.valID and v.value = attName
```

Figura 4.32: Um exemplo de *template* para tradução de consultas no sistema Agora

4.4 Análise Comparativa dos Métodos de Publicação

Apresentam-se, a seguir, as características que foram analisadas nos métodos de publicação de dados relacionais em documento XML. A tabela 4.2 mostra um resumo.

4.4.1 GAV x LAV

A abordagem GAV, como visto anteriormente, permite que uma visão seja definida a partir de um esquema relacional proprietário. A maior vantagem dessa abordagem, segundo Levy et al. [53], é que os algoritmos de composição de consultas são mais simples do que os algoritmos de reescrita de consultas, utilizados pela abordagens que adotam LAV. Entretanto, adicionar novas fontes de dados, como ocorre nos sistemas de integração, não é trivial. Em contraste, na abordagem LAV, cada fonte é descrita isoladamente sobre o esquema global. É tarefa do sistema determinar como os dados vão interagir para responder às consultas.

No caso da fonte de dados ser puramente relacional, a abordagem GAV é mais recomendada, segundo Deutsch et al. [25]. Com exceção do sistema Agora, os outros trabalhos apresentados utilizam a abordagem GAV.

4.4.2 Definição de Visões

Quando uma consulta XML é escrita, ela especifica quais dados do documento XML são necessários. Normalmente, esses dados somam apenas uma fração de todo o documento. Por isso, a visão virtual deve ser utilizada para a recuperação de dados, evitando a materialização de dados que não farão parte da resposta. Além da visão virtual, seria interessante permitir que o usuário também recupere todos os dados em um só documento, ou seja, crie uma visão materializada.

No SilkRoute, a visão virtual é criada a partir do RXL e o Módulo Compositor faz a composição das consultas retirando os dados desnecessários da visão. Também suporta materialização dos dados.

XPERANTO permite a definição de uma visão virtual através de uma consulta XQuery e seu Módulo Compositor de Visões retira as subárvores desnecessárias do grafo XQGM. Se o usuário desejar, também pode materializar os dados.

Os sistemas ROLEX, Agora e o trabalho de Suciu et al. [43] criam visões virtuais para que as consultas sejam formuladas. Entretanto, não permitem a materialização.

No Oracle as visões são definidas através de funções do SQL/XML. Após sua construção, ela pode ser armazenada como XMLType no banco de dados para posteriores consultas.

No DB2, utilizam-se procedimentos armazenados associados a um DAD previamente definido para criar uma visão materializada. Para visões virtuais, pode-se utilizar o XML for Tables.

No SQL Server pode-se definir uma visão virtual através de uma consulta SQL com a cláusula FOR XML no modo EXPLICIT.

4.4.3 Grau de Automação

Na área da computação, espera-se que todos os processos sejam automatizados. Alguns dos trabalhos apresentados exigem que o usuário/desenvolvedor da aplicação defina algumas entradas para o algoritmo, exigindo conhecimentos avançados sobre o funcionamento interno do sistema.

No SilkRoute, XPERANTO e no trabalho de Suciu et al. [43], as visões virtuais devem ser especificadas pelo desenvolvedor. O XPERANTO também permite que a visão padrão seja utilizada.

Para o ROLEX e o sistema Agora não foi possível verificar pelos artigos se são completamente automatizados.

No Oracle, a visão deve ser definida pelo desenvolvedor através de anotações no XML Schema. Oracle também permite a criação de uma visão padrão através da função SYS-XMLGEN(). Cada tupla da tabela se torna um elemento, semelhante ao processo do XPERANTO (figura 4.11).

No DB2, o desevolvedor deve especificar o arquivo DAD antes de utilizar os procedimentos armazenados. E no SQL Server, toda a consulta SQL FOR XML EXPLICIT é escrita pelo desenvolvedor, uma tarefa bastante complexa.

4.4.4 Linguagem suportada

XQuery está se tornando a linguagem padrão para consultas XML e sua adaptação aos algoritmos é de grande importância para a utilização ampla dos sistemas discutidos neste trabalho.

SilkRoute utiliza uma tecnologia proprietária, a linguagem RXL. Já o ROLEX e o trabalho de Suciu el al. [43] utilizam DOM e XSLT respectivamente. Oracle e DB2 utilizam funções que recebem uma expressão XPath como parâmetro. O SQL Server aceita consultas elaboradas em XPath.

XPERANTO e Agora são os únicos sistemas apresentados que utilizam XQuery.

4.4.5 Redundância de Dados

Seria recomendado que os sistemas permitissem que dados previamente materializados por outras consultas fossem armazenados para acelerar a execução das consultas, como sugere Deutsch et al. em [24]. Neste caso, haveria dados redundantes no banco de dados. Apenas o ROLEX aborda essa questão, através dos índices dos planos de navegação.

4.4.6 Algoritmos de Tradução de Consultas

Os algoritmos de tradução de consultas podem ser avaliados através de duas métricas [49]: sua funcionalidade, ou seja, a classe de consultas XML suportadas, e o desempenho em termos de eficiência das consultas SQL resultantes.

Funcionalidade - Como SilkRoute, Rolex, o trabalho de Suciu et al. [16] e DB2 não utilizam XQuery e nem XPath, serão descartados dessa análise. No XPERANTO, a classe de consultas suportadas não é bem definida. O sistema Agora suporta consultas do tipo filho (/), descendente (//), atributo, expressões FLWOR e expressões lógicas e aritméticas nos dados escalares. Oracle e SQL Server suportam consultas XPath simples, como / e //.

Desempenho - não há nenhum trabalho conhecido comparando a qualidade das consultas SQL geradas pelos vários algoritmos. Entretanto, a maioria deles possuem algum tipo de otimização para as consultas SQL geradas (característica abordada separadamente).

4.4.7 Otimização

Devido à estrutura linear do banco de dados relacional, oposto à estrutura aninhada do XML, gerar um documento XML de um banco de dados relacional sempre envolve avaliação de múltiplas consultas SQL contendo várias operações de junção para reconstruir a estrutura em árvore do XML. A presença de operações de junção nas consultas SQL pode afetar bastante o desempenho, dependendo do plano para sua execução. Um otimizador para essas consultas pode acelerar a construção do documento XML final.

O SilkRoute possui um algoritmo de otimização para decidir entre a criação de uma consulta SQL com várias subconsultas aninhadas, várias consultas SQL separadas ou um meio-termo entre essas duas opções. As decisões são baseadas nos custos das consultas providos pelo SGBD. O XPERANTO cria apenas uma consulta SQL, utiliza decorrelacionamento e ainda repassa todo o processamento pesado da consulta para o SGBD. No Rolex, perfis de navegação são mapeados para determinar quais consultas serão decorrelacionadas. No trabalho de Suciu el al. [43], algumas técnicas simples são apresentadas para otimizar consultas aninhadas que utilizam IN e funções agregadas. O sistema Agora não possui nenhum método de otimização.

Os produtos comerciais não possuem nenhuma técnica específica de otimização, devido à existência do otimizador do próprio SGBD.

Além das técnicas adotadas nos trabalhos anteriores, alguns autores propõem algumas abordagens de otimização para as consultas SQL. Em [2], Kotidis et al. apresentam uma abordagem para a otimização de consultas que geram nós irmãos para um mesmo pai. Por exemplo, as consultas da figura 4.33 para reconstrução do elemento <artigo> poderiam ser otimizadas fazendo-se uma consulta SQL apenas.

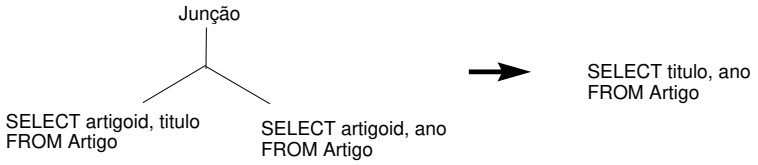

Figura 4.33: Técnica de otimização de consultas proposta por Kotidis et al.[2]

	SilkRoute	XPERANTO	ROLEX	[43]	Agora	Oracle	DB2	SQL Server
Abordagem	GAV	GAV	GAV	GAV	LAV	GAV	GAV	GAV
Definição de visões	VV,VM	VV,VM	VV	VV	VV	VV, VM	VV, VM	VV
Automatização	define a visão RXL	define a visão em XQuery	indefinido	define a visão	indefinido	define o XSD annotated schema	define o DAD	define a SQL FOR XML EXPLICIT
Linguagem XML	XML-QL	XQuery	DOM,XSLT	XSLT	XQuery	funções	UDFs com XPath	XPath
Redundância	não suporta	não suporta	índices	não suporta	não suporta	não suporta	não suporta	não suporta
Funcionalidade		indefinido			FLWR, /, //, etc.	simples (/, //)		simples (/, //)
Otimização	algoritmo baseado em custos	decorrela-cionamento	perfis de navegação	técnicas simples	não possui	não possui	não possui	não possui

Legenda: GAV - global-as-view LAV - local-as-view
VV - visão virtual VM - visão materializada

Tabela 4.2: Sumário dos algoritmos para publicação de dados XML

4.5 Conclusões e Questões em Aberto

Uma análise comparativa entre os métodos vistos anteriormente é dificultada pelo fato deles envolverem características muito distintas. Como ainda não há um estudo sobre o desempenho dos sistemas, nossa análise é apenas qualitativa.

Pode-se chegar a algumas conclusões analisando a tabela 4.2:

- a abordagem GAV foi amplamente adotada pelos métodos, com exceção do sistema Agora;
- em geral, os sistemas ainda exigem a interação do desenvolvedor das aplicações no processo de publicação de dados;
- XQuery não foi amplamente adotada como linguagem de consulta XML, porque é uma linguagem muito recente;
- com exceção do sistema Agora, as classes de consultas suportadas pelas abordagens englobam apenas consultas simples do tipo / e //;
- com exceção dos produtos comercias, os sistemas possuem alguma técnica de otimização para consultas SQL;
- dentre os sistemas apresentados, o mais completo é o XPERANTO. Como nosso trabalho considera apenas fontes relacionais, a abordagem GAV é a mais adequada. XPERANTO se engloba nesse contexto e ainda utiliza a linguagem XQuery.

A seguir, algumas questões em aberto nessa área:

1. Na indústria, discute-se a questão da padronização dos documentos para intercâmbio de dados através da definição de uma estrutura, que pode ser XML Schema ou DTD. Tudo indica que a médio prazo XML Schema obsoletará o DTD, mas atualmente a maioria dos documentos XML estruturados ainda utilizam DTD. Conseqüentemente, surge a necessidade de desenvolver técnicas de publicação que criem documentos seguindo esses padrões.
2. Ainda está indefinido quais classes de consultas XQuery podem ser traduzidas para consultas SQL. Segundo Manolescu et al. [56], os algoritmos publicados na literatura não lidam com eficiência os níveis de aninhamento e agrupamento dos elementos e atributos dos documentos XML.
3. A recursividade em documentos XML não foi abordada por nenhum dos trabalhos apresentados. Não está claro se o SQL suporta recursão na criação de visões (virtual ou materializada) e ainda não há algoritmos capazes de transformar consultas XML do tipo "//" sobre visões recursivas.

4. Como mencionado anteriormente, não há nenhum estudo que compare o desempenho dos algoritmos apresentados.

5. A questão da otimização de consultas SQL previamente geradas, apesar de ter sido discutida, recebeu menos ênfase que o problema da composição de consultas nos trabalhos apresentados. Segundo Krishnamurthy et al. [50], otimização pode se tornar intratável, mesmo para cenários mais simples, como consultas que usam "/". Os autores propõem o uso de algoritmos "inteligentes" para a tradução das consultas ao invés de aplicar algoritmos de otimização. Qual o melhor algoritmo de otimização e qual a melhor opção entre criar novos algoritmos de tradução versus criar novos algoritmos de otimização, ainda são questões em aberto.

Capítulo 5

Conclusões

XML está rapidamente se tornando o formato padrão para representação de dados na *Internet*. Além disso, está sendo utilizado em aplicações diversas, como intercâmbio e integração de dados, entre outros. Apesar de toda expectativa em torno do XML, há uma forte tendência de que os dados continuem sendo armazenados em bancos de dados relacionais, devido ao nível de maturidade adquirido pelos SGBDs relacionais. Para que as duas tecnologias possam coexistir, várias técnicas foram desenvolvidas. Elas se subdividem em técnicas para armazenamento de documentos XML em bancos de dados relacionais e técnicas de publicação de dados em documentos XML. Paralelamente, surgiram algoritmos de tradução de consultas XML para SQL. Esta dissertação apresentou uma revisão do estado da arte nesta área.

A primeira contribuição foi a classificação dos métodos de armazenamento em dois subgrupos, os dependentes e os independentes de estrutura, como visto na figura 3.1. A segunda contribuição foi a revisão detalhada dos métodos, com exemplos relacionados ao documento XML da bibliografia. A terceira contribuição foi a avaliação qualitativa dos métodos, tanto os de armazenamento como os de publicação. Pode-se tirar algumas conclusões gerais sobre as análises:

- por ser uma linguagem recente, XQuery ainda não foi amplamente adotada pelos métodos;
- nota-se uma evolução cronológica nos métodos, partindo de soluções *ad hoc* para sistemas que envolvem algoritmos de otimização, de composição de consultas, tradução de consultas XML para SQL, etc;
- não é possível definir quais métodos são mais eficientes sem uma análise de desempenho;
- não há nenhum sistema completo, que englobe técnicas de armazenamento e técnicas de publicação e ainda utilize o mesmo algoritmo de tradução de consultas.

A quarta contribuição foi a discussão das questões em aberto. Apesar da grande quantidade de artigos publicados na área, ainda há muito o que ser feito.

Como trabalho futuro, alguns tópicos podem ser considerados:

- determinação das classes de consultas XQuery que podem ser traduzidas para SQL;
- desenvolvimento de um método completo, que envolva armazenamento de documentos XML e publicação de dados em XML; e
- estudo dos métodos de atualização de documentos XML. Trata-se de uma área importante e recente, com poucos trabalhos completamente desenvolvidos. A questão das atualizações é dificultada pelo fato de ainda não existir na especificação de XQuery consultas para esse fim.

Referências Bibliográficas

[1] Aguilera, V.; Cluet, S.; Veltri, P.; Vodislav, D. & Wattez, F. Querying XML Documents in Xyleme. Relatório Técnico, Verso/INRIA, 2000. Disponível em: http://citeseer.nj.nec.com/aguilera00querying.html. Acesso em: 01/07/2004.

[2] Amer-Yahia, S.; Kotidis, Y. & Srivastava, D. XML Publishing: Look at Siblings too! In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003. Disponível em: http://www.research.att.com/~sihem/publications/ICDE03.pdf. Acesso em: 01/07/2004.

[3] Banerjee, S.; Krishnamurthy, V.; Krishnaprasad, M. & Murthy, R. Oracle8i - The XML Enabled Data Management System. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000, p. 561–568. Disponível em: http://www.grandpoohbah.net/Sandeepan/icde99_paper.pdf. Acesso em: 01/07/2004.

[4] Boag, S.; Chamberlin, D.; Fernández, M.; Florescu, D.; Robie, J. & Siméon, J. XQuery 1.0: An XML Query Language, 2003. Disponível em: http://www.w3.org/TR/xquery/. Acesso em: 01/07/2004.

[5] Bohannon, P.; Freire, J.; Haritsa, J. R.; Ramanath, M.; Roy, P. & Simeon, J. LegoDB: Customizing Relational Storage for XML Documents. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002. Disponível em: http://citeseer.nj.nec.com/540973.html. Acesso em: 01/07/2004.

[6] Bohannon, P.; Freire, J.; Roy, P. & Simeon, J. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002. Disponível em: http://citeseer.nj.nec.com/465172.html. Acesso em: 01/07/2004.

[7] Bohannon, P.; Korth, H. F. & Narayan, P. P. S. The Table and the Tree: On-Line Access to Relational Data through Virtual XML Documents. In *Informal Proceedings of the 4th International Workshop on the Web and Databases (WebDB)*,

2001. Disponível em: http://www.bell-labs.com/user/ppsnarayan/papers/webdb01/rolex.pdf. Acesso em: 01/07/2004.

[8] Bohannon, P.; Korth, H. F.; Narayan, P. P. S.; Shenoy, P. & Ganguly, S. Optimizing View Queries in ROLEX to Support Navigable Result Trees. In *Proceedings of 28th International Conference on Very Large Databases (VLDB)*, 2002. Disponível em: http://www.vldb.org/conf/2002/S04P04.pdf. Acesso em: 01/07/2004.

[9] Bonifati, A. & Ceri, S. Comparative Analysis of Five XML Query Languages. *SIGMOD Record 29*, 1 (2000), 68–79. Disponível em: http://citeseer.ist.psu.edu/bonifati00comparative.html. Acesso em: 01/07/2004.

[10] Braganholo, V.; Davidson, S. & Heuser, C. On the Updatability of XML Views over Relational Databases. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2003. Disponível em: http://www.inf.ufrgs.br/~vanessa/artigos/webdb2003.pdf. Acesso em: 01/07/2004.

[11] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M. & Maler, E. Extensible Markup Language (XML) 1.0 (Third Edition), 2004. Disponível em: http://www.w3.org/TR/REC-xml. Acesso em: 01/07/2004.

[12] Campbell, C.; Eisenberg, A. & Melton, J. XML Schema. *SIGMOD Record 32*, 2 (2003). Disponível em: http://www.acm.org/sigmod/record/issues/0306/D7-Standard-JimMelton.pdf. Acesso em: 01/07/2004.

[13] Chamberlin, D.; Robie, J. & Florescu, D. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Informal Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, 2000. Disponível em: http://citeseer.nj.nec.com/chamberlin00quilt.html. Acesso em: 01/07/2004.

[14] Chase, N. Understanding DOM, 2001. Disponível em: http://jmvidal.cse.sc.edu/csce590/spring02/understanding-dom.pdf. Acesso em: 01/07/2004.

[15] Cheng, J. & Xu, J. XML and DB2. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000, p. 569–576. Disponível em: http://www-306.ibm.com/software/data/db2/extenders/xmlext/xmlextbroch.pdf. Acesso em: 01/07/2004.

[16] Clark, J. XSL Transformations (XSLT) Version 1.0, 1999. Disponível em: http://www.w3.org/TR/xslt. Acesso em: 01/07/2004.

[17] Clark, J. & DeRose, S. XML Path Language (XPath) Version 1.0, 1999. Disponível em: http://www.w3.org/TR/xpath. Acesso em: 01/07/2004.

[18] Costello, R. L. XML Schema Tutorial, 2002. Disponível em: http://www.xfront.com/xml-schema.html. Acesso em: 01/07/2004.

[19] DeHaan, D.; Toman, D.; Consens, M. & Ozsu, T. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *SIGMOD Record*, 2003. Disponível em: http://db.uwaterloo.ca/~ddbms/publications/xml/sigmod03.pdf. Acesso em: 01/07/2004.

[20] Deitel, H. M.; Deitel, P. J.; Nieto, T. R.; Lin, T. M. & Sadhu, P. *XML How to Program*. Prentice Hall, New Jersey, USA, 2001.

[21] Deutsch, A.; Fernandez, M.; Florescu, D.; Levy, A. & Suciu, D. XML-QL: A Query Language for XML, 1998. Disponível em: http://www.w3.org/TR/NOTE-xml-ql. Acesso em: 01/07/2004.

[22] Deutsch, A.; Fernandez, M.; Florescu, D.; Levy, A. & Suciu, D. A Query Language for XML. In *Proceedings of 8th International World Wide Web Conference (WWW)*, 1999. Disponível em: http://citeseer.ist.psu.edu/deutsch98query.html. Acesso em: 01/07/2004.

[23] Deutsch, A.; Fernandez, M. & Suciu, D. Storing Semistructured Data with STORED. In *SIGMOD Record*, 1999, p. 431–442. Disponível em: http://citeseer.nj.nec.com/409457.html. Acesso em: 01/07/2004.

[24] Deutsch, A. & Tannen, V. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proceedings of 29th International Conference on Very Large Databases (VLDB)*, 2003. Disponível em: www.vldb.org/conf/2003/papers/S07P03.pdf. Acesso em: 01/07/2004.

[25] Deutsch, A. & Tannen, V. Reformulation of XML Queries and Constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, 2003. Disponível em: http://feast.ucsd.edu/People/alin/papers/icdt-03.pdf. Acesso em: 01/07/2004.

[26] Drake, M. & Banerjee, S. Oracle XML DB. Technical White Paper, 2004. Disponível em: http://otn.oracle.com/tech/xml/xmldb/Current/TWP.pdf. Acesso em: 01/07/2004.

[27] Fallside, D. C. XML Schema Part 0: Primer, 2001. Disponível em: http://www.w3.org/TR/xmlschema-0/. Acesso em: 01/07/2004.

[28] Fernández, M.; Morishima, A. & Suciu, D. Efficient Evaluation of XML Middleware Queries. In *SIGMOD Record*, 2001. Disponível em: http://citeseer.nj.nec.com/fernandez01efficient.html. Acesso em: 01/07/2004.

[29] Fernández, M.; Morishima, A.; Suciu, D. & Tan, W.-C. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin 24*, 2 (2001), 12–19. Disponível em: http://www.research.att.com/~mff/files/_F292063957.pdf. Acesso em: 01/07/2004.

[30] Fernández, M.; Tan, W. & Suciu, D. Silkroute: Trading between Relations and XML. In *Proceedings of the 9th International World Wide Web Conference (WWW)*, 2000. Disponível em: http://citeseer.ist.psu.edu/fernndez99silkroute.html. Acesso em: 01/07/2004.

[31] Florescu, D. & Kossmann, D. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Relatório Técnico, INRIA, 1999. Disponível em: http://citeseer.nj.nec.com/florescu99performance.html. Acesso em: 01/07/2004.

[32] Florescu, D. & Kossmann, D. Storing and Querying XML data using an RDMBS. *IEEE Data Engineering Bull. 22*, 3 (1999), 27–34.

[33] Freire, J.; Ramanath, M.; Haritsa, J. & Roy, P. Searching for Efficient XML-to-Relational Mappings. In *Proceedings of the First International XML Database Symposium (XSym)*, 2003. Disponível em: http://citeseer.ist.psu.edu/584943.html. Acesso em: 01/07/2004.

[34] Freire, J. & Simeon, J. Adaptive XML Shredding: Architecture, Implementation, and Challenges. In *Proceedings of the First VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT)*, 2002. Disponível em: http://www.cse.ogi.edu/~juliana/pub/eextt2002.pdf. Acesso em: 01/07/2004.

[35] GmbH, S. S. E. P. Introduction to XML, 1998. Disponível em: http://www.xml.org/xml/step_intro_to_xml.shtml. Acesso em: 01/07/2004.

[36] Group, I. H. T. SQL/XML, 2004. Disponível em: http://www.sqlx.org. Acesso em: 01/07/2004.

[37] Higgins, S. Oracle XML DB Developer´s Guide. Manual do desenvolvedor, 2003. Disponível em: http://download-west.oracle.com/docs/cd/B13789_01/appdev.101/b10790.pdf. Acesso em: 01/07/2004.

[38] Hors, A. L.; Hégaret, P. L.; Wood, L.; Nicol, G.; Champion, J. R. M. & Byrne, S. Document Object Model (DOM) Level 3 Core Specification Version 1.0, 2004. Disponível em: http://www.w3.org/TR/DOM-Level-3-Core. Acesso em: 01/07/2004.

[39] IBM. XML Extender Administration and Programming Version 8. Technical Paper, 2002. Disponível em: http://publibfp.boulder.ibm.com/epubs/pdf/c2712340.pdf. Acesso em: 01/07/2004.

[40] IBM. XML for Tables. Relatório Técnico, 2003. Disponível em: http://www.alphaworks.ibm.com/tech/xtable. Acesso em: 01/07/2004.

[41] International Organization for Standardization, 1947. Disponível em: http://www.iso.org. Acesso em: 01/07/2004.

[42] Jagadish, H.; Al-Khalifa, S.; Lakshmanan, L.; Nierman, A.; Paparizos, S.; Patel, J.; Srivastava, D. & Wu, Y. Timber: A Native XML Database. Relatório Técnico, University of Michigan, 2002. Disponível em: http://citeseer.ist.psu.edu/jagadish02timber.html. Acesso em: 01/07/2004.

[43] Jain, S.; Mahajan, R. & Suciu, D. Translating XSLT Programs to Efficient SQL Queries. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, 2002. Disponível em: http://www.cs.washington.edu/homes/ratul/papers/www2002-xslt.pdf. Acesso em: 01/07/2004.

[44] Jiang, H.; Lu, H.; Wang, W. & Yu, J. X. Path Materialization Revisited: An Efficient Storage Model for XML Data. In *Proceedings of the 13th Australasian Database Conference (ADC)*, Melbourne, Australia, 2002. Disponível em: http://citeseer.nj.nec.com/jiang02path.html. Acesso em: 01/07/2004.

[45] Jiang, H.; Lu, H.; Wang, W. & Yu, J. X. XParent: An Efficient RDBMS-Based XML Database System. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, USA, 2002. Disponível em http://citeseer.nj.nec.com/480360.html. Acesso em 01/07/2004.

[46] Kanne, C.-C. & Moerkotte, G. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000, p. 198. Disponível em: http://citeseer.nj.nec.com/kanne99efficient.html. Acesso em: 01/07/2004.

[47] Kha, D. D.; Yoshikawa, M. & Uemura, S. An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, 2001, p. 313–320. Disponível em: http://citeseer.nj.nec.com/kha01xml.html. Acesso em: 01/07/2004.

[48] KLETTKE, M. & MEYER, H. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *Informal Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, 2000. Disponível em: http://citeseer.nj.nec.com/klettke00xml.html. Acesso em: 01/07/2004.

[49] KRISHNAMURTHY, R.; KAUSHIK, R. & NAUGHTON, J. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proceedings of the First International XML Database Symposium (XSym)*, 2003. Disponível em: http://www.cs.wisc.edu/~sekar/research/xmltosqlsurvey.pdf. Acesso em: 01/07/2004.

[50] KRISHNAMURTHY, R.; KAUSHIK, R. & NAUGHTON, J. F. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004. Disponível em: http://www.cs.wisc.edu/~sekar/application/sekar_intelligence.pdf. Acesso em: 01/07/2004.

[51] LAURENT, S. Why XML?, 1998. Disponível em: http://www.simonstl.com/articles/whyxml.htm. Acesso em: 01/07/2004.

[52] LEE, D. & CHU, W. W. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, 2000. Disponível em: http://citeseer.ist.psu.edu/277279.html. Acesso em: 01/07/2004.

[53] LEVY, A. Logic-Based Techniques in Data Integration. In *Workshop on Logic-Based Artificial Intelligence*, 1999. Disponível em: http://citeseer.nj.nec.com/391746.html. Acesso em: 01/07/2004.

[54] LI, C.; BOHANNON, P.; KORTH, H. & NARAYAN, P. Composing XSL Transformations with XML Publishing Views. In *SIGMOD Record*, 2003. Disponível em: http://www.cen.uiuc.edu/~cli/417-li.pdf. Acesso em: 01/07/2004.

[55] LIE, H. W. & BOS, B. Cascading Style Sheets, 1996. Disponível em: http://www.w3.org/TR/REC-CSS1. Acesso em: 01/07/2004.

[56] MANOLESCU, I.; FLORESCU, D. & KOSSMANN, D. Answering XML Queries over Heterogeneous Data Sources. In *Proceedings of 27th International Conference on Very Large Databases (VLDB)*, 2001. Disponível em: http://citeseer.nj.nec.com/451483.html. Acesso em: 01/07/2004.

[57] MANOLESCU, I.; FLORESCU, D.; KOSSMANN, D.; XHUMARI, F. & OLTEANU, D. Agora: Living with XML and Relational. In *Proceedings of 26th International Conference on Very Large Databases (VLDB)*, 2000. Disponível em: http://citeseer.ist.psu.edu/manolescu00agora.html. Acesso em: 01/07/2004.

[58] Martinelli, M. XML : A Technical Introduction, 2000. Disponível em: http://www.iei.pi.cnr.it/~Martinelli/XML/doc/XML-A_Technical_Introduction.PDF. Acesso em: 01/07/2004.

[59] McHugh, J.; Abiteboul, S.; Goldman, R.; Quass, D. & Widom, J. Lore: A Database Management System for Semistructured Data. *SIGMOD Record 26*, 3 (1997), 54–66. Disponível em: http://citeseer.nj.nec.com/mchugh97lore.html. Acesso em: 01/07/2004.

[60] Raggett, D.; Hors, A. L. & Jacobs, I. HTML 4.01 Specification, 1999. Disponível em: http://www.w3.org/TR/html401. Acesso em: 01/07/2004.

[61] Runapongsa, K. & Patel, J. Storing and Querying XML Data in Object-Relational DBMSs. In *Proceedings of the 9th Conference on Extending Database Technology (EDBT)*, 2002. Disponível em: http://www-personal.umich.edu/~krunapon/research/XORator.pdf. Acesso em: 01/07/2004.

[62] Rys, M. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, 2001. Disponível em http://www.csd.uch.gr/~hy561/Papers/msxml-icde01.pdf. Acesso em 01/07/2004.

[63] Schmelzer, R. The Pros and Cons of XML, 2001. Disponível em: http://www.zapthink.com/reports/proscons-view.html. Acesso em: 14/05/2003.

[64] Schmidt, A. *Processing XML in Database Systems.* Tese de Doutorado, Universiteit van Amsterdam, 2002. Disponível em: http://www.cwi.nl/htbin/ins1/publications. Acesso em: 01/07/2004.

[65] Schmidt, A.; Kersten, M.; Windhouwer, M. & Waas, F. Efficient Relational Storage and Retrieval of XML Documents. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2000, p. 47–52. Disponível em: http://citeseer.nj.nec.com/schmidt00efficient.html. Acesso em: 01/07/2004.

[66] Schmidt, A.; Waas, F.; Kersten, M.; Carey, M.; Manulescu, I. & Busse, R. XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002. Disponível em http://www.vldb.org/conf/2002/S30P01.pdf. Acesso em 01/07/2004.

[67] Schöning, H. & Wäsch, J. Tamino - An Internet Database System. In *Proceedings of the 7th Conference on Extending Database Technology (EDBT)*, 2000.

[68] Scowen, R. Extended BNF - A Generic Base Standard, 1998. Disponível em: http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf. Acesso em: 01/07/2004.

[69] Shanmugasundaram, J. *Bridging Relational Technology and XML.* Tese de Doutorado, University of Wisconsin, Madison, 2001. Disponível em: http://www.cs.cornell.edu/People/jai/papers/Dissertation.pdf. Acesso em: 01/07/2004.

[70] Shanmugasundaram, J.; Carey, M.; Florescu, D.; Ives, Z.; Lu, Y.; Shekita, E. & Subramanian, S. XPERANTO: Publishing Object-Relational Data as XML. In *Informal Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, 2000, p. 105–110. Disponível em: http://citeseer.nj.nec.com/carey00xperanto.html. Acesso em: 01/07/2004.

[71] Shanmugasundaram, J.; Carey, M.; Kiernan, J.; Shekita, E. & Subramanian, S. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Databases (VLDB)*, 2000, p. 646–648. Disponível em: http://citeseer.nj.nec.com/article/carey00xperanto.html. Acesso em: 01/07/2004.

[72] Shanmugasundaram, J.; Kiernan, J.; Shekita, E. J.; Fan, C. & Funderburk, J. Querying XML Views of Relational Data. In *Proceedings of 27th International Conference on Very Large Databases (VLDB)*, 2001, p. 261–270. Disponível em: http://citeseer.ist.psu.edu/shanmugasundaram01querying.html. Acesso em: 01/07/2004.

[73] Shanmugasundaram, J.; Shekita, E.; Barr, R.; Carey, M.; Lindsay, B.; Pirahesh, H. & Reinwald, B. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 27th International Conference on Very Large Databases (VLDB)*, 2001, p. 133–154. Disponível em: http://citeseer.ist.psu.edu/shanmugasundaram00efficiently.html. Acesso em: 01/07/2004.

[74] Shanmugasundaram, J.; Tufte, K.; Zhang, C.; He, G.; DeWitt, D. J. & Naughton, J. F. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, 1999, p. 302–314. Disponível em: http://citeseer.nj.nec.com/295071.html. Acesso em: 01/07/2004.

[75] Shimura, T.; Yoshikawa, M. & Uemura, S. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proceedings of the 10th International Conference and Workshop on Database and Expert Systems Applications (DEXA)*, 1999, p. 206–217. Disponível em: http://citeseer.nj.nec.com/shimura99storage.html. Acesso em: 01/07/2004.

[76] Shimura, T.; Yoshikawa, M.; Uemura, S. & Amagasa, T. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases. *ACM Transactions on Internet Technology (TOIT)* (2001). Disponível em: http://db-www.aist-nara.ac.jp/members/Yoshikawa/paper/TOIT2001-authorCopy.pdf. Acesso em: 01/07/2004.

[77] Sperberg-McQueen, C. M. & Burnard, L. A Gentle Introduction to SGML, 1993. Disponível em: ftp://www.ucc.ie/pub/sgml/p2sg.ps. Acesso em: 01/07/2004.

[78] Sperberg-McQueen, C. M. & Burnard, L. A Gentle Introduction to XML, 2002. Disponível em: http://www.tei-c.org/Guidelines2/gentleintro.pdf. Acesso em: 01/07/2004.

[79] Tatarinov, I.; Ives, Z. G.; Halevy, A. Y. & Weld, D. S. Updating XML. In *SIGMOD Record*, 2001. Disponível em: http://citeseer.nj.nec.com/tatarinov01updating.html. Acesso em: 01/07/2004.

[80] Tian, F.; DeWitt, D.; Chen, J. & Zhang, C. The Design and Performance Evaluation of Alternative XML Storage Strategies. In *SIGMOD Record*, 2002. Disponível em http://www.cs.wisc.edu/~czhang/doc/publications/feng6page.pdf. Acesso em 01/07/2004.

[81] Unicode Consortium, 1988. Disponível em: http://www.unicode.org. Acesso em: 01/07/2004.

[82] van der Vlist, E. Using W3C XML Schema, 2001. Disponível em: http://www.xml.com/pub/a/2000/11/29/schemas/part1.html. Acesso em: 01/07/2004.

[83] Varlamis, I. & Vazirgiannis, M. Bridging XML-Schema and Relational Databases: a System for Generating and Manipulating Relational Databases using Valid XML Documents. In *Proceedings of the ACM Symposium on Document Engineering (DOCENG)*, 2001. Disponível em http://citeseer.nj.nec.com/483894.html. Acesso em 01/07/2004.

[84] World Wide Web Consortium, 1994. Disponível em: http://www.w3.org. Acesso em: 01/07/2004.

[85] Walsh, N. A Technical Introduction to XML, 1997. Disponível em: http://www.arbortext.com/resources/white_papers.htm. Acesso em: 01/07/2004.

[86] Wang, K. & Liu, H. Discovering Typical Structures of Documents: A Road Map Approach. In *21st Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, 1998, p. 146–154. Disponível em: http://citeseer.nj.nec.com/wang98discovering.html. Acesso em: 01/07/2004.

[87] Wang, W. Q.; Lee, M.-L.; Ooi, B. C. & Tan, K.-L. XStorM: A Scalable Storage Mapping Scheme for XML Data. *Proceedings of 10th International World Wide Web Conference (WWW) 4*, 1-2 (2001), 101–119. Disponível em: http://citeseer.nj.nec.com/488036.html. Acesso em: 01/07/2004.

[88] Zhang, C.; Naughton, J. F.; DeWitt, D. J.; Luo, Q. & Lohman, G. M. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Record*, 2001. Disponível em: http://citeseer.nj.nec.com/zhang01supporting.html. Acesso em: 01/07/2004.

[89] Zheng, S.; Wen, J.-R. & Lu, H. Cost-Driven Storage Schema Selection for XML. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2003. Disponível em http://research.microsoft.com/asia/dload_files/group/mediasearching/2002p/cost-XML-%20DASFAA.pdf. Acesso em 01/07/2004.