JÚLIO CESAR FRANCHETTI STELMATCHUK

XTREM: UMA FERRAMENTA PARA ARMAZENAMENTO DE DOCUMENTOS XML EM BANCO DE DADOS RELACIONAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Marcos Sfair Sunye

Co-orientadora: Prof.ª Carmem Satie Hara

CURITIBA
2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Júlio Cesar Franchetti Stelmatchuk, avaliamos o trabalho intitulado, "*XTREM: Uma Ferramenta para Armazenamento de Documentos XML em Banco de Dados Relacionais* ", cuja defesa foi realizada no dia 19 de novembro de 2003, às dezesseis horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 19 de novembro de 2003.

Prof. Dr. Marcos Sfair Sunye
DINF/UFPR – Orientador

Prof. Dr. José Tarcisio Pires Trindade
UEM - Membro Externo

Profª. Dra. Laura Sanchez Garcia
DINF/UFPR - Membro Interno

AGRADECIMENTOS

À **Deus**

Por ter me dado a oportunidade de fazer esta dissertação de mestrado e por ter me ajudado sempre que precisei.

Aos meus orientadores **Marcos Sfair Sunye** e **Carmem Satie Hara**

Por toda atenção, todo direcionamento, todo apoio e todo empenho para que esta dissertação se tornasse possível.

Aos meus **familiares**, especialmente meus pais **Julio Stelmatchuk** e **Geny Franchetti Stelmatchuk**, e minha esposa **Lilian Queler Bolonha Stelmatchuk**

Por todos os incentivos morais e financeiros, que foram essenciais para o desenvolvimento e conclusão desta dissertação.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Com a disseminação da Internet, houve a necessidade de criar-se um formato de dados que facilitasse a publicação e a extração de dados neste meio. Assim surgiu o formato XML (*Extended Markup Language*), uma versão simplificada do SGML (*Standard Generalized Markup Language*).

Atualmente, dados XML disponíveis na Internet em geral não são armazenados localmente neste formato, mas em um banco de dados. Para que o processo de publicação e extração de dados seja automatizado, existe a necessidade de transformar dados de um banco de dados em XML e vice-versa.

Como a tecnologia de banco de dados mais utilizada atualmente é a relacional, este trabalho concentra-se no problema de transformação de dados XML para o modelo relacional, através da especificação e implementação do XTREM (*XML to Relational Mapping*), uma ferramenta para gerar o esquema de um banco de dados relacional para armazenar documentos XML.

Ao contrário de trabalhos anteriores que baseiam-se na estrutura do documento XML para a geração do esquema, o XTREM baseia-se em restrições de integridade semântica, mais especificamente chaves XML [9].

O sistema XTREM foi baseado no artigo [13] onde os autores propõem uma nova linguagem para o mapeamento de XML para relacional. Uma grande vantagem desta linguagem é a não utilização da DTD (*Data Type Definition*) para fazer o mapeamento, o que a torna mais genérica, visto que alguns documentos XML não possuem DTD.

Esta dissertação não tem apenas o objetivo de apresentar um método para criação do esquema de tabelas relacionais de um documento XML, ela tem também o objetivo de apresentar uma solução de busca dos dados no documento XML mapeado, para inseri-los em suas tabelas resultantes.

Com o uso da ferramenta XTREM, o mapeamento de documentos XML para bases de dados relacionais torna-se mais próximo do tradicional, pelo fato de serem utilizadas

dependências funcionais para se efetuar a normalização das tabelas geradas, e pelo fato de ser utilizada uma linguagem que possibilita a extração de apenas uma parte dos documentos XML.

A ferramenta XTREM gera como saída tanto o esquema das relações normalizadas na Terceira Forma Normal (3FN), quanto a transformação do documento XML para o esquema relacional.

CAPÍTULO 1

INTRODUÇÃO

Há muitos anos existem formatos de dados que possibilitam a troca de informações entre diferentes plataformas computacionais. Dentre eles, é possível citar o FITS (*Flexible Image Transport System*), voltado para a comunidade de astronomia, e o GRIB (*Grid in Binary*), específico para troca de informações meteorológicas (veja [11] para uma lista de formatos existentes). A maior parte destes formatos atende uma área ou comunidade específica e tem como característica serem formatos "auto-descritivos", isto é, contém tanto as informações sobre os dados (metadados), como os dados em si.

Com a disseminação da Internet, houve a necessidade de se criar um formato de dados que facilitasse a publicação e a extração de dados neste meio. Assim, surgiu o SGML (*Standard Generalized Markup Language*), um padrão que formaliza a estrutura de um documento possibilitando a sua portabilidade. Mais tarde, surgiu a necessidade de extrair apenas a essência do SGML para torná-lo mais simples. Surgiu assim, o formato XML.

```
<cadastro>
  <cliente clieid="1">
    <numcliente> 1 </numcliente>
    <nome> Jose da Silva </nome>
    <contato contid="5">
      <email> jose@xxx.com.br </email>
    </contato>
  </cliente>
  <cliente clieid="2">
    <nome> Maria Costa </nome>
    <contato contid="6">
      <telefone> 224-7376 </telefone>
    </contato>
  </cliente>
</cadastro>
```

Figura 1.1: Documento XML

Documentos XML são semi-estruturados, isto é, contêm dados caracterizados por maior heterogeneidade (que os dados estruturados) e podem não corresponder a um esquema

específico [1]. O exemplo da Figura 1.1 ilustra um documento XML sobre clientes de uma loja de componentes eletrônicos.

Note que em um documento XML os metadados são representados através de marcações (*tags*), como por exemplo `<cliente>`, e também através dos níveis de encaixamento. Ou seja, o fato de `<numcliente> 1 </numcliente>` estar dentro de `<cliente>` indica que o numcliente é referente ao cliente José da Silva. Além disso, visto que XML é um modelo semi-estruturado, os dados não precisam ser homogêneos. No exemplo da Figura 1.1 o contato com o primeiro cliente é feito através de `email`, enquanto que com o segundo, através do `telefone`.

São estas características da linguagem, ser semi-estruturada e auto-descritiva, que a tornaram tão popular para a troca de informações tanto na Internet como em outras aplicações.

Atualmente, dados XML disponíveis na Internet em geral não são armazenados localmente neste formato, mas em um banco de dados. Por exemplo, suponha que a loja de componentes eletrônicos possui uma página na Internet com a sua tabela de preços. Esta página é criada a partir de um banco de dados local que exporta seus dados em XML. Periodicamente, a loja faz um levantamento de preços das lojas concorrentes que também mantêm uma tabela de preços disponível em XML para poder sempre oferecer preços competitivos. Para que o processo de publicação e extração de dados seja automatizado, existe a necessidade de transformar dados de um banco de dados em XML e vice-versa. Como a tecnologia de banco de dados mais utilizada atualmente é a relacional, este trabalho concentra-se no problema de transformação de dados XML para o modelo relacional. A direção oposta, de relacional para XML, é mais simples, já que o modelo relacional é estruturado.

Este trabalho apresenta o XTREM (*XML to Relational Mapping*), uma ferramenta para gerar o esquema de um banco de dados relacional para armazenar documentos XML. Ao contrário de trabalhos anteriores que baseiam-se na estrutura do documento XML para a geração do esquema, o XTREM baseia-se em restrições de integridade semântica, mais especificamente chaves XML [9]. A entrada para o XTREM é composta por três partes:

- uma regra geral que mapeia os dados XML que se deseja armazenar para uma relação universal;
- as chaves XML que são válidas no documento XML;
- o documento XML a ser armazenado.

A saída da ferramenta é uma decomposição da relação universal na Terceira Forma Normal (3FN) baseada nas dependências funcionais propagadas das chaves XML. Além disso, dado um documento XML, a ferramenta faz a transformação do documento nas relações normalizadas. A arquitetura da ferramenta está ilustrada na figura 1.2. O módulo *Propagador de Chaves* foi implementado por Jing Qin e detalhado em [13]. Fazem parte da dissertação a especificação e implementação do módulo *Normalizador*, utilizando técnicas tradicionais de normalização do modelo relacional [5, 2], e a implementação do módulo *Transformador XML-Rel*, como especificado em [12].

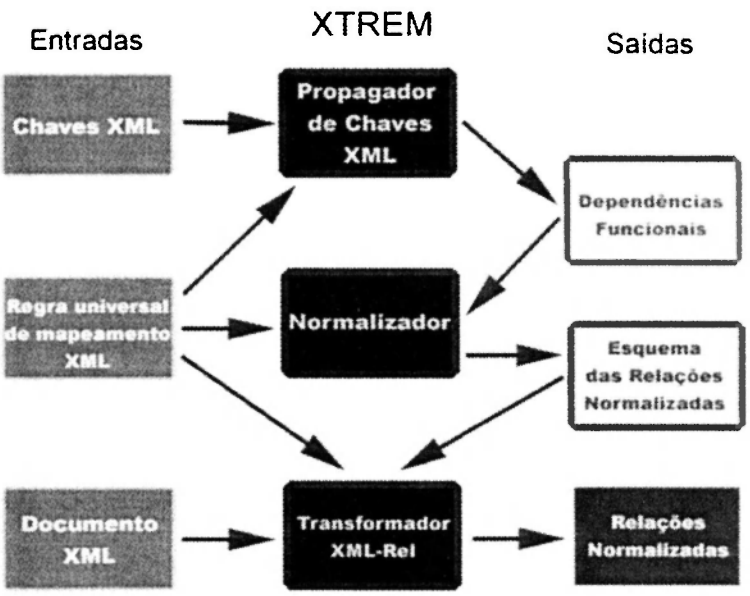

Figura 1.2: Arquitetura do XTREM

Os capítulos seguintes estão organizados da seguinte forma: o capítulo 2 apresenta os modelos de dados relacional e XML, juntamente com a definição de chaves XML; o capítulo 3 descreve os trabalhos relacionados de transformação de documentos XML para o modelo relacional. A especificação da ferramenta XTREM e a comparação com os

métodos estudados é detalhada no capítulo 4 e a sua implementação é apresentada no capítulo 5. O capítulo 6 apresenta as conclusões.

CAPÍTULO 2

MODELOS DE DADOS

Este capítulo contém uma breve introdução aos dois modelos de armazenamento de dados utilizados na ferramenta, o Relacional e o XML, bem como as seguintes restrições de integridade: dependências funcionais do modelo relacional e chaves XML.

2.1 O Modelo Relacional

No modelo relacional um *esquema* é formado por um conjunto de *esquemas de tabelas* como R1(a, b, c) e R2(c, d) [3]. Nestas expressões, R1 e R2 são os nomes das tabelas, e a, b, c e d são os nomes das colunas. Na prática, também é preciso especificar os tipos de dados dessas colunas. Uma instância de um esquema nada mais é do que um conjunto de tabelas. Assim, a forma mais comum de representar tais tabelas é como ilustrado na Figura 2.1.

r1:

a	b	c
a1	b1	c1
a2	b2	c2

r2:

c	d
c1	d1
c2	d2
c3	d3

Figura 2.1: Exemplo de instância do modelo relacional

Uma tabela consiste de um conjunto de dados dispostos em forma de linhas e colunas. As linhas são conjuntos de dados que possuem os mesmos atributos. Cada coluna é um atributo das linhas, e todos os valores de uma mesma coluna devem ser do mesmo tipo.

Sendo uma tabela um conjunto de linhas, não é possível que existam duas linhas idênticas em uma mesma tabela, pois conjuntos não permitem a existência de elementos repetidos. Porém, geralmente não é necessário conhecer os valores de todas as colunas

para identificar uma linha, mas apenas um subconjunto delas. Por exemplo, com dados de pessoas armazenados em uma tabela definida por Pessoa (RG, nome, endereco), dado um RG é possível identificar uma única linha da tabela sabendo-se que não existem duas pessoas com o mesmo RG. Ou seja, com o RG de uma pessoa, é possível identificar no máximo uma linha da tabela e, assim, obter o seu nome e endereço. Este conjunto de atributos é chamado de *chave primária* da tabela. Uma chave primária é um tipo especial de uma restrição de integridade mais expressiva chamada *dependência funcional*, que será visto a seguir.

2.1.1 Dependências Funcionais

Uma dependência funcional (DF) é uma restrição de integridade que envolve dois conjuntos de colunas de uma tabela. Elas referem-se à semântica dos dados e ajudam a evitar redundâncias, inconsistências e anomalias que podem surgir tanto na inclusão quanto na exclusão de dados de um banco de dados.

Considere uma tabela R(a1, a2, ..., an). Uma dependência funcional é denotada por:

$X \rightarrow Y$

Onde X e Y são subconjuntos de {a1, a2, ..., an}. Ela especifica uma restrição sobre as possíveis linhas que podem existir em uma instância r de R. A restrição estabelece que para quaisquer linhas t1 e t2 em r, se t1[X]=t2[X], então t1[Y]=t2[Y], onde t1[X] representa os valores das colunas X da linha t1, e t1[Y] representa os valores das colunas Y da linha t1. Em outras palavras, para duas linhas quaisquer de r, se as colunas de X forem iguais em ambas as linhas, então as colunas de Y também serão iguais em ambas as linhas. Assim, é dito que Y é dependente funcionalmente de X. Note que:

- Se uma DF estabelece que em qualquer instância r de R não pode existir mais que uma linha com um dado valor de X, isto implica que $X \rightarrow Y$ para quaisquer subconjuntos de colunas Y de R.
- Se $X \rightarrow Y$ em R, isto não significa que $Y \rightarrow X$ em R.

Uma DF não pode ser automaticamente inferida a partir de uma instância, mas deve

ser definida por alguém que conheça a semântica da aplicação. Por exemplo, considere a instância da tabela `Automóveis` mostrada na Figura 2.2. Embora num primeiro momento possa parecer que `{Modelo, Estilo}` → `Preço`, não se pode afirmar isso a menos que se saiba que isso é verdade para todas as instâncias da tabela `Automóveis`.

Modelo	Estilo	Motor	Veloc_Max*	Preço*	Mercado	Competidor
Corsa	Sedan	1.0	150	19000	Doméstico	Uno
Corsa	Hatch	2.0	200	23000	Esportivo	Gol
Fiesta	Hatch	1.0	160	14000	Doméstico	Uno
Fiesta	Hatch	1.3	170	16000	Doméstico	Palio

* Os valores são fictícios.

Figura 2.2: Instância do esquema da tabela Automóveis

Examinando o exemplo com mais cuidado, pode-se perceber que `Preço` é funcionalmente dependente de `Modelo`, `Estilo` e `Motor`, ou seja, `{Modelo, Estilo, Motor}` → `Preço`. Uma outra restrição existente nesta tabela é que `Veloc_Máx` é dependente funcionalmente de `Modelo` e `Motor`: `{Modelo, Motor}` → `Veloc_Máx`.

A chave primária de uma tabela é considerada um caso especial de dependências funcionais . Ou seja, se um subconjunto de atributos X é chave de uma tabela, então X → R, onde R é o conjunto de todos os atributos da tabela.

2.1.1.1 Axiomas de Armstrong

Os Axiomas de Armstrong [22] são regras de inferência que mostram como um conjunto de dependências funcionais implicam em outras dependências funcionais. Para a definição dessas regras, considere uma tabela com um conjunto U de colunas. As regras são:

- *Reflexividade*: se $Y \subseteq X \subseteq U$, então $X \rightarrow Y$. Essa é uma regra trivial, pois se Y é subconjunto de X, então X determina Y.
- *Incremento*: se $X \rightarrow Y$ e Z é um subconjunto de U, então $(X \cup Z) \rightarrow (Y \cup Z)$.
- *Transitividade*: se $X \rightarrow Y$ e $Y \rightarrow Z$, então $X \rightarrow Z$.

Os Axiomas de Armstrong são completos, porque dado um conjunto F de DFs, estas regras nos permitem computar todas as DFs que são consequências de F. No entanto, outras regras podem ser definidas para simplificar a geração destas dependências. Dentre estas regras, é possível citar as regras de união e decomposição:

- *União:* se $X \rightarrow Y$ e $X \rightarrow Z$ então $X \rightarrow (Y \cup Z)$
- *Decomposição:* se $X \rightarrow (Y \cup Z)$, então $X \rightarrow Y$, e $X \rightarrow Z$

Estas regras não são básicas porque podemos utilizar os Axiomas de Armstrong para prová-las como mostrado abaixo.

Regra de União

1. $X \rightarrow Y$, premissa
2. $X \rightarrow Z$, premissa
3. $X \rightarrow (X \cup Y)$ por incremento de X em 1
4. $(X \cup Y) \rightarrow (Y \cup Z)$ por incremento de Y em 2
5. $X \rightarrow (Y \cup Z)$ por transitividade de 3 e 4

Regra de Decomposição

1. $X \rightarrow (Y \cup Z)$, premissa
2. $(Y \cup Z) \rightarrow Y$ por reflexividade em 1
3. $X \rightarrow Y$ por transitividade de 1 e 2.
4. $(Y \cup Z) \rightarrow Z$ por reflexividade em 1
5. $X \rightarrow Z$ por transitividade de 1 e 4.

Uma vez que as regras de união e decomposição são corretas, podemos afirmar que qualquer conjunto de DFs F é equivalente a um conjunto de DFs F', onde todas as dependências em F' são da forma $X \rightarrow A$, onde A é um atributo. Ou seja, em F' todas as DFs possuem um único atributo no seu lado direito. No restante do trabalho assumiremos que todas as DFs são desta forma.

2.1.1.2 Fechamento de conjunto de dependências funcionais

Dado um conjunto de F de dependências funcionais, seu fechamento (representado por F^+) é o conjunto de todas as dependências funcionais que podem ser derivadas de F pela aplicação dos Axiomas de Armstrong. Por exemplo: seja $F = \{A \rightarrow B\}$. Seu fechamento é formado pelo conjunto $F^+ = \{A \rightarrow B; A \rightarrow A; B \rightarrow B\}$

Dado um conjunto F de DFs, e um conjunto de colunas X, o fechamento de X em F, representado por X^+, é o conjunto de todas as colunas que são determinados por X em F. A Figura 2.3 mostra um algoritmo para computar X^+[6]:

```
Algoritmo Closure(X, F)
Entrada: Um conjunto de colunas X e um conjunto de DFs F.
Saída: Um conjunto de colunas que forma o fechamento de X.
início
  Dep_Antiga := vazio;
  Dep_Nova := X;
  enquanto (Dep_Antiga != Dep_Nova)
  {
    Dep_Antiga := Dep_Nova;
    para cada DF (W → A) em F
      se W ⊆ Dep_Antiga
      Dep_Nova := Dep_Nova ∪ {A};
  }
  retorne(Dep_nova);
fim.
```

Figura 2.3: Algoritmo para computar X^+

O algoritmo Closure pode ser utilizado para verificar se uma DF $X \rightarrow A$ faz parte do fechamento de um conjunto de DFs F, como ilustrado no algoritmo Membership [4] da Figura 2.4.

Baseado no conceito de fechamento, os conceitos de Cobertura e Cobertura não redundante serão vistos a seguir.

```
Algoritmo Membership(F, X → A)
Entrada: Um conjunto de DFs F e uma DF X → A.
Saída: Verdadeiro, se X → A ∈ F+.
início
  se A ∈ Closure(X, F)
    retorne(verdadeiro);
  senão
    retorne(falso);
fim.
```

Figura 2.4: Algoritmo que determina se uma DF $X \rightarrow A$ é consequência de um conjunto de DFs F

2.1.1.3 Cobertura

Dados dois conjuntos de dependências funcionais F e G, se $F^+ = G^+$ e F é composto por um número menor de DFs que G, então é F é uma cobertura para G.

2.1.1.4 Cobertura não redundante

Se F é uma cobertura para G e não existe nenhum subconjunto F' de F tal que $F'^+ = F^+$, então F é um conjunto não redundante.

A Figura 2.5 mostra um algoritmo para computar a cobertura não redundante de um conjunto de DFs [5].

```
Algoritmo Nonredundant(G)
Entrada: Um conjunto de DFs G.
Saída: Um conjunto DFs F que é uma cobertura não redundante de G.
início
  F := G;
  para toda DF X → A ∈ G
    se Membership(F − {X → A}, X → A)
      F := F − {X → A};
  retorne(F);
fim.
```

Figura 2.5: Algoritmo para computar cobertura não redundante

2.2 XML

XML (*eXtensible Markup Language*) é um novo padrão adotado pelo World Wide Web Consortium (W3C) para complementar a HTML (*HyperText Markup Language*) para troca de dados na Internet. XML é uma forma de representação de dados semi-estruturados, projetada para a Internet. Aparentemente, XML é muito parecido com HTML, mas a diferença básica entre ambos é que XML trabalha especificamente com o conteúdo, servindo como um banco de dados de modo texto, enquanto HTML trabalha tanto com o conteúdo como com a apresentação dos dados na página.

2.2.1 Sintaxe

Em HTML existem marcas no texto que fazem sua formatação, como `<b>...</b>` para que o texto tenha seja apresentado em **negrito**, `<i>...</i>` para *itálico* e assim por diante. Em XML novas marcas podem ser definidas à vontade, como por exemplo `<cliente>...</cliente>`.

No banco de dados relacional o componente básico é a coluna, enquanto que em XML o componente básico é o elemento, que nada mais é do que um texto delimitado por marcas combinantes como `<cliente>` e `</cliente>`. Uma expressão como `<cliente>` é chamada de marca de início e `</cliente>` é chamada de marca de fim. Um elemento pode conter um texto, outros elementos ou uma mistura de texto e elementos, como ilustrado na Figura 1.1 do Capítulo 1.

Para que estas marcações estejam de acordo com o que se quer representar, um documento pode ter uma DTD (*Data Type Definition*) que defina a gramática do documento.

Uma DTD consiste no nome da marca raiz do documento, opcionalmente seguida por várias declarações de marcação que indica as marcas permitidas no documento e sua estrutura associada. A figura 2.6 apresenta a DTD do documento XML da figura 1.1.

Nesta DTD, a primeira linha especifica qual será a raiz do documento, nesse caso `cadastro`. O símbolo ? (interrogação) significa que o elemento em questão pode aparecer nenhuma ou apenas uma vez no elemento em que está contido. O símbolo * (asterisco)

```
<!DOCTYPE cadastro [
  <!ELEMENT cadastro (cliente)*>
  <!ELEMENT cliente (numcliente?, nome, contato*)>
  <!ATTLIST cliente clieid>
  <!ELEMENT numcliente (#PCDATA)>
  <!ELEMENT nome (#PCDATA)>
  <!ELEMENT contato (email?, telefone?)>
  <!ATTLIST contato contid>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT telefone (#PCDATA)>
]>
```

Figura 2.6: DTD para o XML da figura 1.1

significa que o elemento em questão pode aparecer nenhuma ou mais vezes no elemento em que está contido. (#PCDATA) indica que o elemento em questão é um texto, semelhante a um *string*. Um outro tipo de representação de uma DTD é através de grafos, onde um grafo (N, E) consiste de um conjunto N de nós e um conjunto E de arestas: associado a cada aresta $e \in E$ há um par (ordenado) (o, d), onde o é o nó origem e d é o nó destino. Assim, uma DTD pode ser representada como um grafo com nós rotulados, onde cada nó representa um elemento do documento ou um caractere "curinga" (como *), como mostrado na figura 2.7.

2.2.2 Expressões de Caminho

Os documentos XML podem ser encarados como grafos com arestas rotuladas, onde, partindo-se da raiz, existe um único caminho para se atingir cada nó. Neste caso, é usada

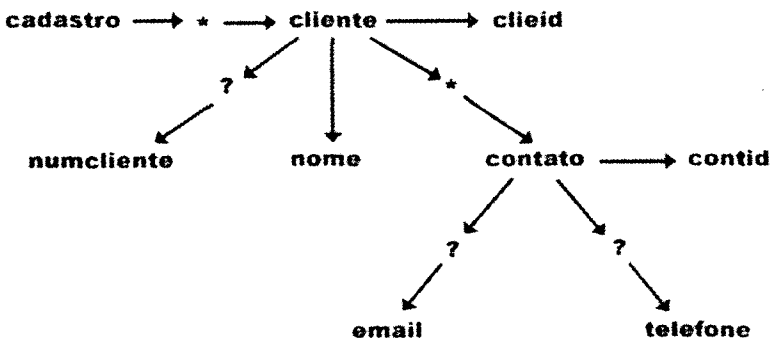

Figura 2.7: Representação de uma DTD em forma de grafo

a notação (o, r, d) para uma aresta rotulada de r do nó o para o nó d.

Uma seqüência de arestas rotuladas, $l_1.l_2.....l_n$ é denominada de expressão de caminho.

Por exemplo, utilizando o exemplo da figura 2.8, o caminho cadastro.cliente.nome retorna o conjunto de nós que corresponde ao conjunto de strings (texto) {"José da Silva", "Maria Costa"}.

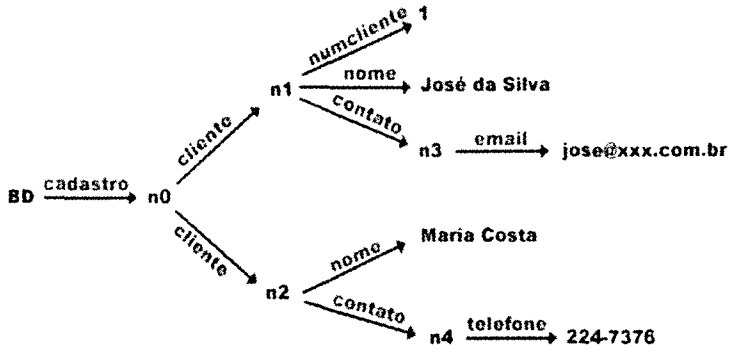

Figura 2.8: Um documento XML em forma de grafo

Mais especificamente, o resultado de uma expressão de caminho $l_1.l_2.....l_n$ é um conjunto de nós v_n, tais que existam arestas $(r, l_1, v_1), (v_1, l_2, v_2), ..., (v_{n-1}, l_n, v_n)$ no grafo de dados, onde r é a raiz do grafo.

Para extrair todos os nomes das pessoas desse documento XML, pode-se utilizar um símbolo "curinga" que casa com qualquer elemento. Assim, se _ for tal símbolo, cadastro._.nome será a expressão de caminho que obtém todos os nomes no documento que tenham uma aresta entre cadastro e nome. Neste caso, o resultado seria o mesmo do exemplo acima, pois existe apenas um elemento entre cadastro e nome, que é cliente.

Agora, se não se conhece quantas arestas existem entre cadastro e nome, seria necessário usar a operação de Estrela de Kleene, representada por *. Por exemplo, para fazer uma consulta de todos os nomes existentes na base de dados onde o primeiro rótulo é cadastro e o ultimo é nome, a expressão de caminho cadastro._*.nome poderia ser utilizada. Dessa maneira, pode haver zero ou mais arestas entre os rótulos cadastro e nome.

Além disso, existe também o símbolo "curinga" + que significa uma ou mais repetições de uma aresta e também existe o símbolo ? que significa zero ou uma ocorrência de uma aresta.

A sintaxe de expressões regulares em caminhos é definida por:

$$e = l \mid \epsilon \mid \_ \mid e\text{`.'}e \mid \text{`('}e\text{`)'} \mid e\text{`*'} \mid e\text{`+'} \mid e\text{`?'}$$

onde l é um rótulo, e é uma expressão e ϵ é a expressão vazia.

2.2.3 Chaves para XML

Assim como existe a definição de chaves primárias para o modelo relacional, existem diversas propostas para a definição de chaves para XML. Neste trabalho, será usada a definição de chaves como descrito em [9].

O conceito de chaves em XML é diferente do conceito de chaves em banco de dados relacionais. Enquanto em bancos de dados relacionais uma chave é um conjunto de colunas que juntas determinam uma única linha, em XML uma chave é um conjunto de elementos e/ou atributos que determinam um elemento, ou um nó se o documento for representado em forma de grafo, como na Figura 2.7.

Em XML, as chaves podem ser usadas para prover integridade em um documento e também para a otimização de consultas. Existem dois tipos de chaves XML [9]: absoluta e relativa.

2.2.3.1 Chave Absoluta

Uma chave absoluta possui a seguinte forma [10, 9]:

`(A, {C1, ..., Cn})`

onde `A` é uma expressão de caminho chamada *Caminho Alvo*, e `{C1, ..., Cn}` é um conjunto de expressões de caminho chamados de *Caminhos Chave.* Em um documento XML em forma de grafo, `[[A]]` representa o conjunto de nós atingidos quando o caminho `A` é percorrido a partir da raiz. Os nós em `[[A]]` satisfazem a chave se, e somente se, para

dois nós quaisquer *n1* e *n2* em `[[A]]`, se eles têm os mesmos valores para os *Caminhos Chave*, então eles devem ser o mesmo nó.

Por exemplo, considere um documento XML de cadastro de livros. Uma chave absoluta que assegura que um nome identifica unicamente um livro no cadastro é expressa da seguinte forma:

`(livro, {nome})`

Assim, neste documento, não podem existir dois livros com o mesmo nome.

2.2.3.2 Chaves Relativas

Alguns documentos XML necessitam de uma estrutura de chaves hierárquica. As chaves relativas [10, 9] foram criadas para atender esta necessidade.

As chaves relativas nada mais são que uma representação hierárquica de chaves absolutas. Elas possuem a seguinte forma:

`(R, (A, C))`,

onde `R` é uma *chave raiz* de um documento, e `(A, C)` é a chave para o subdocumento cuja raiz está contida em `[[R]]`.

Continuando o exemplo do cadastro de livros, com chaves relativas é possível expressar a seguinte restrição:

`(livro, {nome})`,

`(livro, (capitulo, {numero}))`.

A primeira restrição é uma *chave absoluta* que assegura que `nome` identifica apenas um `livro` no documento XML. A segunda restrição assegura que um `capítulo` é identificado pelo seu `número`, mas apenas dentro do contexto do livro onde ele está contido. Em outras palavras, para identificar unicamente um capítulo no documento, são necessários o nome do livro e o número do capítulo.

2.3 Comparação entre os modelos

O modelo relacional é estruturado e o modelo XML é semi-estruturado. No modelo semi-estruturado, é inexistente ou opcional a descrição dos tipos de seus dados. Normalmente, o próprio nome do elemento já diz que tipo de dado será armazenado. Em vista disto, este seus dados podem ser descritos como "sem esquema" ou "auto-descritivos".

Nos modelos estruturados, primeiramente toda a estrutura dos dados é planejada, para somente depois os dados serem armazenados nas tabelas.

Uma das principais vantagens do modelo semi-estruturado é a sua capacidade de variação na estrutura. Por exemplo, suponha que existe um cadastro de pessoas em ambos os modelos, e que em ambos pode-se cadastrar apenas um telefone por pessoa. Agora, surge o problema: é preciso cadastrar mais de um telefone por pessoa. Para fazer isso no modelo semi-estruturado é simples: basta criar mais elementos "dentro" do elemento responsável por armazenar o telefone. Para solucionar o problema no modelo relacional, é necessário criar uma nova tabela com números de telefone, com sua chave primária formada por um código seqüencial e pelo código da pessoa. Em seguida, deve-se povoar essa tabela com os números dos telefones contidos na tabela de origem e eliminar a coluna com o número do telefone na tabela de origem, para que os dados não fiquem duplicados.

Outra vantagem do modelo semi-estruturado é a interoperabilidade. Como seus dados estão descritos na própria instância, qualquer sistema pode ter a capacidade de lê-los.

Por outro lado, no modelo semi-estruturado a perda de espaço pode ser grande, pois para cada dado gravado, o tipo ao qual ele pertence precisa ser descrito.

O mapeamento de Relacional para XML é relativamente simples em comparação com o mapeamento de XML para Relacional. No primeiro, basta visualizar cada linha e cada coluna como elementos onde as colunas estão abaixo das linhas. Por exemplo, uma possível representação das tabelas da Figura 2.1 em XML está ilustrada na Figura 2.9.

Já o mapeamento de XML para Relacional não é tão óbvio. Este é o assunto do próximo capítulo.

```
<r1>
  <linha num=1>
    <a> a1 </a>
    <b> b1 </b>
    <c> c1 </c>
  </linha>
  <linha num=2>
    <a> a2 </a>
    <b> b2 </b>
    <c> c2 </c>
  </linha>
</r1>
<r2>
  <linha num=1>
    <c> c1 </c>
    <d> d1 </d>
  </linha>
  <linha num=2>
    <c> c2 </c>
    <d> d2 </d>
  </linha>
  <linha num=3>
    <c> c3 </c>
    <d> d3 </d>
  </linha>
</r2>
```

Figura 2.9: Mapeamento Relacional para XML

CAPÍTULO 3

MAPEAMENTO XML PARA RELACIONAL

Fazer o armazenamento dos dados do modelo estruturado no modelo semi-estruturado não é uma tarefa muito complexa (como foi visto no capítulo anterior), pois o modelo semi-estruturado engloba o estruturado. O contrário, armazenar dados do modelo semi-estruturado no modelo estruturado, exige um estudo mais aprofundado.

Neste capítulo são apresentadas diversas propostas existentes para o armazenamento de documentos XML em um banco de dados relacional.

Primeiramente serão descritos três métodos diferentes para conversão encontrados em [21]: Basic, Shared e Hybrid.

Em seguida, serão mostrados mais quatro métodos distintos de mapeamento: um método que armazena as arestas do documento XML [16], um método Simples [8] que tem um resultado muito parecido com o Hybrid, o STORED [14] que possui a característica única de "Overflow" e o LegoDB [7], um método que cria um mapeamento baseado nos custos para o processamento de um conjunto de consultas.

3.1 Método Basic

O algoritmo do método Basic para conversão de dados XML para Relacional necessita de uma DTD para ser aplicado. Tomando como exemplo a DTD da Figura 2.6, cria-se um grafo para ela, como na Figura 2.7. Neste grafo, pode-se notar que existem 2 "nós" diferentes dos demais, que são o ? e *. Eles não correspondem a elementos ou atributos na DTD, mas representam particularidades sobre os seus nós imediatamente abaixo. O ? significa que o nó imediatamente abaixo pode ocorrer somente uma ou nenhuma vez dentro do nó pai (nó imediatamente superior). O * significa que o nó imediatamente abaixo pode ocorrer nenhuma ou mais vezes dentro do nó pai.

O funcionamento do método Basic é descrito a seguir. Cria-se uma tabela para cada nó

do grafo. Para toda tabela criada, uma coluna de identificação única é criada (ID). Caso o nó seja um nó folha (não possua descendentes) do grafo, então a tabela possuirá uma coluna com o próprio nome do nó. Caso o nó seja não-folha, para cada nó descendente, este é marcado como visitado e é criado uma nova coluna na tabela, desde que ele não se encaixe nas seguintes restrições:

1. O nó é multi-valorado (está logo após *);
2. O nó já está marcado como visitado (significa que existe uma recursão no grafo).

Caso o nó participe de alguma dessas restrições, então é criada uma nova tabela para ele na qual existirá uma coluna a mais que identifique quem é o pai desta tabela (parentID).

Na Figura 3.1 tem-se as tabelas criadas com este método, usando como entrada a DTD da Figura 2.6.

cliente	(clienteID: integer, cliente.numcliente: string, cliente.nome: string, cliente.contato.email: string, cliente.contato.telefone: string)
nome	(nomeID: integer, nome: string)
numcliente	(numclienteID: integer, numcliente: string)
cliente.contato	(cliente.contatoID: integer, cliente.contato.parentID: integer, cliente.contato.email: string, cliente.contato.telefone: string)
contato	(contatoID: integer, contato.email: string, contato.telefone: string)
email	(emailID: integer, email: string)
telefone	(telefoneID: integer, quantidade: string)

Figura 3.1: Resultado da aplicação do método Basic

3.2 Método Shared

O objetivo principal do método Shared é diminuir o número de tabelas existentes no método Basic. No Shared, os únicos nós que podem ser tabelas são os que:

1. Não são referenciados por nenhum outro nó (como por exemplo, cliente);
2. São multi-valorados (estão logo após *, como por exemplo contato);
3. Estiverem em recursão no grafo;
4. São referenciados por mais de um nó.

Inicialmente, os únicos nós que podem ser escolhidos para serem tabelas principais são os descritos no item 1 (a raiz do documento). Se algum nó filho não se enquadrar nos itens 2, 3 ou 4, ele será colocado dentro da tabela pai. Caso contrário, será criada uma nova tabela para ele, com seus filhos como colunas e mais duas colunas: `parentID` para identificar quem é o nó pai e `parentCODE` para armazenar qual o tipo do nó pai . Todas as tabelas criadas, independentemente de serem pais ou filhas, deverão ter uma coluna de identificação única(ID) e uma coluna que identifique se o nó é raiz ou não(isroot).

Na figura 3.2 tem-se as tabelas criadas com este método.

cliente	(clienteID: integer, cliente.numcliente.isroot: boolean, cliente.numcliente: string, cliente.nome.isroot: boolean, cliente.nome: string)
contato	(contatoID: integer, contato.parentID: integer, contato.parentCODE: integer, contato.email.isroot: boolean, contato.email: string, contato.telefone.isroot: boolean, contato.telefone: string)

Figura 3.2: Resultado da aplicação do método Shared

3.3 Método Hybrid

O algoritmo do método Hybrid é muito parecido com o do método Shared. A única diferença é que neste método se um nó filho for referenciado por mais de um nó (restrição 4 do Shared), então não será criada uma nova tabela para ele, mas uma coluna com seu nome será criada na tabela pai. Como no exemplo utilizado nesta dissertação não existe um elemento referenciado por mais de 1 elemento, as tabelas criadas neste método são idêndicas as criadas no método Shared (Figura 3.2).

3.4 Método que guarda as arestas

Este método de mapeamento não utiliza a DTD do documento. Ele simplesmente assume que o próprio documento XML está representado através de um grafo de arestas rotuladas e ordenadas, onde os rótulos das arestas são os próprios elementos para onde elas apontam.

A ordenação é determinada pela ordem de leitura das arestas, conforme descrito em [16]. Além desta ordenação para as arestas, o sistema exige que todo elemento que possua um filho que não seja um texto (#PCDATA), tenha uma coluna de identificador (ID). Este identificador, juntamente com o número de ordenação de cada aresta, forma a chave primária da tabela.

Cada elemento é representado como um nó do grafo, não havendo diferença entre elementos e atributos. Conseqüentemente, este método não permite que o documento original possa ser reconstituído após a conversão, embora este problema possa ser contornado com mais uma coluna contendo a informação sobre o tipo do elemento em questão.

Este método permite a existência de três tipos distintos de armazenamento dos dados:

1. Tabela de arestas: uma única tabela armazena dados de todas as arestas, onde existe uma coluna que diferencia as arestas através de seus nomes;
2. Tabela binária: tabelas distintas são criadas para cada nome de aresta, ou seja, as arestas são agrupadas por nomes e cada nome identifica uma tabela;
3. Tabela universal: uma única tabela armazena todas as arestas, porém cada nome de aresta forma uma nova coluna desta tabela.

Mas, além destes três tipos de armazenamento, existem mais dois tipos referentes ao conjunto destino das arestas: os destinos que são dados (#PCDATA) ficam armazenados na mesma tabela das arestas; ou os destinos que são dados ficam em tabelas separadas, chamadas de tabelas de dados, onde cada tabela de dados possui somente dados de um mesmo tipo(caractere, inteiro, etc). Estes dois tipos referentes aos dados podem ser aplicados à cada um dos três tipos vistos acima, totalizando assim seis tipos diferentes de mapeamento de XML para relacional.

Para a descrição detalhada dos diferentes formas de armazenamento, será utilizado o documento XML da Figura 1.1 juntamente com uma nova representação em forma de grafo com o ID dos nós, mostrado na Figura 3.3.

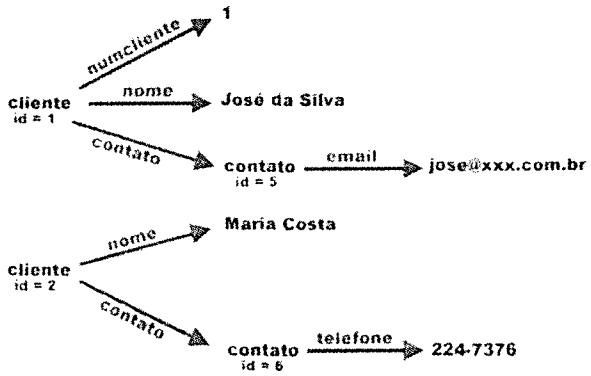

Figura 3.3: Outra representação em forma de grafo do documento XML da Figura 1.1

3.4.1 Tabela de Arestas

A Tabela de Arestas armazena todas as arestas do grafo em uma única tabela. Nesta tabela são gravados o ID do elemento (origem) e do destino de cada aresta do grafo, o nome da aresta, um domínio indicando qual o tipo do dado (se inteiro, string, data ou apenas referência para outro elemento) e um número seqüencial de cada aresta, pois elas são ordenadas com esses números.

A chave primária dessa tabela é formada pelo ID dos elementos de origem e pelo número seqüencial. O esquema da tabela criada neste método é: `Aresta(origem, seqüencial, nome, tipo, destino)`.

A Figura 3.4 mostra um exemplo da aplicação deste método.

origem	seqüencial	nome	tipo	destino
1	1	numcliente	inteiro	1
1	2	nome	texto	Jose da Silva
1	3	contato	referência	5
5	1	email	texto	jose@xxx.com.br
2	1	nome	texto	Maria Costa
...	...	...	...	...

Figura 3.4: Exemplo de Tabela de Arestas

3.4.2 Tabela Binária

É uma tabela que contém todas as arestas que possuem o mesmo nome. Ou seja, cada tabela só poderá conter um tipo de aresta, como uma tabela para a aresta `nome`, outra para `numcliente`, outra para `contato`, etc. A chave primária dessa tabela é formada pelo ID dos elementos de origem e pelo número seqüencial. A estrutura das tabelas criadas neste método é: `Binaria_nome_aresta(origem, seqüencial, tipo, destino)`.

A Figura 3.5 mostra um exemplo da aplicação deste método para a aresta *nome*.

origem	seqüencial	tipo	destino
1	2	texto	Jose da Silva
2	1	texto	Maria Costa

Figura 3.5: Exemplo de Tabela Binária (Binária_nome)

3.4.3 Tabela Universal

Nesta tabela todas as arestas são armazenadas juntas. Esta tabela nada mais é do que a tabela correspondente do resultado de um *outer join* completo de todas as Tabelas Binárias.

Para cada aresta diferente que existir no documento, 3 novas colunas serão criadas nesta tabela, pois para cada tipo de aresta, existe um número seqüencial, um tipo de dado e um destino específico (a coluna de origem é comum à todas as arestas). Considerando que n1, ..., nk são nomes de arestas. A estrutura da tabela criada neste método é: Universal(origem, seqüencial_n1, tipo_n1, destino_n1, ..., seqüencial_nk, tipo_nk, destino_nk).

A Figura 3.6 mostra um exemplo da aplicação deste método.

origem	seqüencial nome	tipo nome	destino nome	seqüencial numcliente	tipo numcliente	destino numcliente	...
1	1	texto	Jose da Silva	2	inteiro	1	...
2	1	texto	Maria Costa	Nulo	Nulo	Nulo	...
...	...	...	...	...	...	...	...

Figura 3.6: Exemplo de Tabela Universal

3.4.4 Mapeando os valores

Os valores, que correspondem ao tipo #PCDATA, podem ser mapeados em dois tipos diferentes: na mesma tabela onde estão as arestas (como vistos até agora), ou em tabelas separadas. Ou seja, são mais dois tipos que podem ser aplicados aos 3 tipos vistos acima, existindo assim, 6 tipos diferentes de mapeamento.

3.4.4.1 Valores em tabelas separadas

Os valores podem ser divididos ainda em tipos (como inteiro ou texto – *string*). Assim, cada tabela possui apenas um tipo de dado, sendo necessária a criação de tantas tabelas quantos forem os tipos de dados usados no mapeamento.

Com o uso de valores em tabelas separadas, as tabelas deste método vistas até agora teriam os valores da coluna destino alterados apenas quando os valores não fossem referência para outro nó.

A estrutura de cada tabela é: Valores_tipo(vID, valor).

Onde vID é a chave, um valor seqüencial que identifica cada valor contido na tabela.

A Figura 3.7 mostra um exemplo da aplicação deste método.

vID	valor
V1	Jose da Silva
V2	Maria Costa
V3	224-7376
...	...

Figura 3.7: Exemplo de Tabela para valores do tipo texto

3.4.4.2 Valores na mesma tabela

Uma alternativa óbvia para armazenar os valores de arestas que não apontam para outro elemento é colocá-los na mesma tabela em que as arestas estão armazenadas, como foi visto nos exemplos de Tabela de Arestas, Tabela Binária e Tabela Universal.

3.4.5 Vantagens e Desvantagens

A vantagem de usar os valores de destino em tabelas separadas é a existência de poucos valores nulos. Em contrapartida, aumenta a quantidade de relacionamentos entre as tabelas, pois os valores de nós-folha são armazenados em tabelas distintas das demais tabelas.

Em relação ao desempenho desses tipos de armazenamento, é possível dizer que quando os valores estão todos na mesma tabela o desempenho é maior. Nos resultados apresentados em [16], a tabela Binária, apesar de possuir maior quantidade de relacionamentos, consegue um resultado superior aos da tabela de Aresta e Universal. As tabelas de Aresta e Universal ficam muito grandes e com muitos valores nulos, o que acarreta em um custo computacional maior.

3.5 Um método simples

O método simples [8] utiliza a DTD do documento para fazer o mapeamento. Nele, o documento não fica armazenado com a estrutura de grafo do documento XML como no método de Florescu e Kossmann [16], o que torna mais difícil a tarefa de reconstruir o documento XML original. O método trata o mapeamento de um documento XML para relacional da seguinte maneira:

Primeiro, o sistema procura por todos os elementos no documento que são complexos, ou seja, elementos que sejam pais de algum outro elemento ou que possuam algum atributo. Para cada um desses elementos, o sistema gera uma tabela contendo uma coluna com o próprio nome da tabela para ser sua chave primária.

Em seguida, o sistema procura por elementos com conteúdo misturado, ou seja, elementos que possuam texto, bem como atributos e elementos filhos. Para cada elemento encontrado, o sistema gera uma tabela separada para armazenar o seu conteúdo texto ligada com a tabela do elemento pai através da chave primária da tabela.

Logo após, o sistema procura por atributos e elementos com dados (PCDATA) que possuam um único valor. Para cada ocorrência, uma coluna é gerada na tabela principal.

Se o elemento é opcional, o sistema permite que a coluna em questão contenha nulos.

Posteriormente, o sistema procura por elementos que possuam mais de um valor ou que possuam elementos filhos que ocorram mais de uma vez. Para cada elemento encontrado nesta condição, o sistema gera uma tabela separada para armazenar os valores, e a tabela fica ligada à do elemento pai pela sua chave primária.

Por fim, para cada elemento filho complexo, o sistema liga a tabela do elemento pai com a tabela do elemento filho através da chave primária da tabela pai.

O resultado do método para a DTD da Figura 2.6 é composto por duas tabelas, como mostrado na Figura 3.8:

Cliente	(clienteNOMETABELA: string, clieid: integer, numcliente: string, nome: string)
Contato	(contatoCHAVE: string, clieid: integer, email: string, telefone: string, contatoNOMETABELApai: string, contatoIDpai: integer)

Figura 3.8: Tabelas geradas pelo método Simples

3.6 O método STORED

No método de mapeamento STORED [14], os dados são mapeados de um modelo semi-estruturado para um modelo misto, ou seja, relacional mais semi-estruturado. Além disso, o método é baseado em uma nova linguagem chamada STORED(*Semistructured TO RElational Data*), e utiliza técnicas de mineração de dados para descobrir automaticamente padrões de dados dentro do documento XML.

STORED é uma linguagem declarativa e mais restritiva que outras linguagens de consulta para dados semi-estruturados, pois ela não possui relacionamentos ou expressões regulares. Segundo os autores, esta restrição faz com que seja possível a reconstrução do documento XML original após o mapeamento.

Uma consulta no STORED consiste de cláusulas FROM-WHERE-STORE, onde a cláusula FROM especifica o padrão a ser encontrado, o caminho a ser percorrido, e as variáveis necessárias para a obtenção dos dados; a cláusula WHERE especifica quais

condições devem ser satisfeitas para que os dados XML sejam extraídos; e a cláusula STORE diz como as variáveis contidas em FROM vão ser armazenadas na forma de tabelas no modelo relacional.

A Figura 3.9 mostra um exemplo de como construir uma tabela contendo todos os contatos dos clientes da loja de componentes eletrônicos do exemplo da Figura 1.1:

```
E1 = FROM cadastro.cliente.contato: $C,
     {telefone: $T, email: $E}
     STORE Tb_contato($C, $T, $E)
```

Figura 3.9: Construção de uma tabela de contatos de clientes com o STORED

O resultado do mapeamento encontra-se na Figura 3.10:

contid	telefone	email
5		jose@xxx.com.br
6	224-7376	

Figura 3.10: Resultado do mapeamento da Figura 3.9

Nesta linguagem, por padrão, o elemento a partir do qual a busca no documento inicia é a primeira variável existente na clausula FROM.

Portanto, neste exemplo, a busca incia nos elementos alcançados atrás do caminho cadastro.cliente.contato. Cada elemento encontrado, associado à variável $C (o símbolo $ indica que C é uma variável). Assim, os dados encontrados serão gravados na tabela Tb_contato, da cláusula STORE, e as suas colunas conterão o ID de contato($C) e os valores dos elementos telefone($T) e email($E), respectivamente.

Caso o usuário queira que a busca inicie por uma variável que não seja a primeira variável da cláusula FROM, é necessário que ela esteja explicitamente definida pela cláusula KEY, como mostrado na Figura 3.11.

O resultado deste mapeamento encontra-se na Figura 3.12.

```
E2 = FROM cadastro.cliente: $C.contato: $CO,
     {telefone: $T, email: $E}
     KEY $CO
     STORE Tb_cadastro($C, $CO, $T, $E)
```

Figura 3.11: Construção de outra tabela de com o STORED

clieid	contid	telefone	email
1	5		jose@xxx.com.br
2	6	224-7376	

Figura 3.12: Resultado do mapeamento da Figura 3.11

3.6.1 Overflow

Uma inovação do Stored com relação aos outros métodos de mapeamento vistos até agora, é a existência da cláusula OVERFLOW para armazenar na forma semi-estruturada porções do documento XML que não satisfazem o padrão de busca da cláusula FROM.

Os dados armazenados no OVERFLOW podem ser usados em buscas futuras que o usuário possa vir a fazer. Um exemplo de utilização deste comando pode ser visto na Figura 3.13.

```
E3 = FROM cadastro: $CA, $CL:_
       WHERE $CL = cliente
OVERFLOW G1($CL)
```

Figura 3.13: Exemplo de Overflow

Neste exemplo, todos os clientes do cadastro serão gravados em um arquivo G1, independente da estrutura que cada nó cliente possui.

3.7 O método LegoDB

No método de mapeamento de dados XML para relacional LegoDB [7], criou-se uma nova linguagem para descrever o esquema do documento XML semelhante ao XML Schema [23], a qual seus autores chamaram de p-Schema.O método de mapeamento baseia-se em dados

estatísticos sobre o desempenho de consultas para obter o esquema de armazenamento do documento XML no modelo relacional.

3.7.1 P-Schema

O XML Schema [23] tem a mesma função de uma DTD para um documento XML. Ou seja, ele serve para fazer a especificação da estrutura do documento, mas, de uma maneira mais detalhada que uma DTD faz.

O p-Schema (*Physical XML Schema*) é uma derivação de XML Schema, possuindo uma sintaxe mais prática e mais fácil de trabalhar que o XML Schema.

A Figura 3.14 ilustra um exemplo de p-schema para a DTD da Figura 2.6.

```
type cliente = [clieid[ integer ],
                numcliente [ integer ],
                contato*,
                nome [ string ]]
type contato = [email[ string ],
                 telefone [ string ]]
```

Figura 3.14: Exemplo de p-Schema

Com o p-Schema do documento XML, o método LegoDB impõe algumas regras para a sua transformação em tabelas. As regras são as seguintes:

- Criar uma tabela para cada type encontrado no p-Schema;
- Para cada tabela criada, criar uma coluna para ser a chave da tabela, contendo o id do elemento;
- Para cada tabela filha, criar uma chave estrangeira para a sua tabela pai;
- Criar uma coluna para cada elemento dentro de outro elemento;
- Se algum elemento é opcional, então a coluna correspondente pode conter nulos.

A Figura 3.15 tem as tabelas para o p-Schema da Figura 3.14.

cliente	(clieid: integer, numcliente: string, nome: string)
contato	(contid: integer, email: string, telefone: string, clienteCHAVEpai: integer)

Figura 3.15: Tabelas para a Figura 3.14

3.7.2 Aplicação do Método LegoDB

A idéia do método LegoDB é procurar fazer o melhor mapeamento possível para um documento XML através de um algoritmo de busca e comparações entre os mapeamentos possíveis.

O algoritmo de busca funciona da seguinte maneira:

Primeiro é feita a transformação do XML Schema do documento para o p-Schema.

Em seguida o algoritmo carrega um conjunto de consultas pré-determinadas que ele deve executar para calcular o custo do p-Schema criado.

Em seguida, o algoritmo aplica uma série de regras de transformação no p-Schema e para cada transformação, o algoritmo calcula o custo de execução do conjunto de consultas, reescrevendo e armazenando em uma lista, o custo de cada transformação.

A transformação que obteve o menor custo é escolhida como o novo p-schema para repetir o passo anterior.

Este processo é repetido até que o algoritmo não encontre mais nenhuma transformação com um custo menor que a obtida até o momento.

Este p-Schema é então usado para gerar o esquema relacional para armazenamento do documento XML.

CAPÍTULO 4

A FERRAMENTA XTREM

O sistema XTREM foi baseado no artigo [13] onde os autores propõem uma nova linguagem para o mapeamento de XML para relacional, similar à proposta apresentada pelo STORED [14].

Uma grande vantagem desta linguagem é a não utilização da DTD para fazer o mapeamento, o que a torna mais genérica, visto que alguns documentos em XML não possuem DTD. A linguagem é apresentada na Seção 4.1.

Conforme apresentado na Figura 1.2, a ferramenta XTREM é dividida em módulos. Na Seção 4.2, é descrito o módulo *Propagador de chaves*, responsável por gerar as dependências funcionais para que o módulo *Normalizador*, apresentado na Seção 4.3, gere um esquema de relações normalizadas. As relações normalizadas são usadas pelo módulo *Transformador XML-rel*, apresentado na Seção 4.4, para gerar um *Script* de inserção dos dados extraídos do documento XML.

4.1 A Linguagem de Transformação

Um programa de transformação é formado por um conjunto de regras que definem o esquema relacional resultante e também de onde os valores armazenados nas tabelas são obtidos no documento XML. Ou seja, dado um esquema de banco de dados relacional R com n tabelas, R = (R_1, ..., R_n), cada tabela é fruto de uma determinada regra aplicada ao documento XML. Uma transformação é da forma:

$$\mathsf{C} = (\mathsf{Regra}(R_1), \ldots, \mathsf{Regra}(R_n))$$

Cada uma dessas regras $\mathsf{Regra}(R_i)$ especifica como povoar as colunas da tabela R_i. Mais especificamente, para cada coluna, é especificada uma expressão de caminho a se percorrer para alcançar o elemento cujo valor será usado para povoar a coluna. O formato geral de uma regra é apresentado abaixo:

nome	email	telefone
Jose da Silva	jose@xxx.com.br	
Maria Costa		224-7376

Figura 4.1: Tabela *Cliente* resultante da transformação

Regra(*nome_tabela*) =

{*nome_coluna*$_1$: val(v_1), *nome_coluna*$_2$: val(v_2), ..., *nome_coluna*$_n$: val(v_n)},

$v_1 \leftarrow v_r.caminho_1$, $v_2 \leftarrow v_i.caminho_2$, ..., $v_n \leftarrow v_j.caminho_n$;

onde $nome\_coluna_1, nome\_coluna_2, \ldots, nome\_coluna_n$ são os nomes das colunas da tabela *nome_tabela* resultante do mapeamento; v_r é a variável padrão da linguagem que guarda o elemento raiz do documento XML; v_1, v_2 e v_n são *variáveis de mapeamento* que identificam nós da representação em forma de grafo do documento XML e *val* é uma função que transforma sub-árvores em uma forma textual que é usada para povoar as colunas da tabela. Por exemplo, a regra abaixo obtém dados sobre clientes para povoar uma tabela Cliente.

Regra(Cliente) =

{nome: val(v_2), email: val(v_4), telefone: val(v_5)},

$v_1 \leftarrow v_r$._*.cliente, $v_2 \leftarrow v_1$.nome, $v_3 \leftarrow v_1$.contato, $v_4 \leftarrow v_3$.email,

$v_5 \leftarrow v_3$.telefone;

Note que a linguagem permite a utilização de "curingas" para o caminho do elemento raiz (e somente para a raiz). No exemplo acima, a utilização da expressão de caminho v_r._*.nf significa que o elemento nf pode estar precedido de qualquer elemento e estar em qualquer nível do grafo. A tabela resultante da transformação acima está ilustrada na Figura 4.1.

4.1.1 Comparação com Outros Métodos de Transformação

Embora esta linguagem seja bastante simples, com pequenas extensões, como um gerador de identificadores para nós do grafo, ela é capaz de representar mapeamentos especificados

com outras técnicas [12], como por exemplo os métodos Hybrid e Simples, apresentados no capítulo 3. Por exemplo, suponha que *id* seja uma função que, gere um identificador único para cada nó de um documento XML em forma de grafo. As regras que especificam o resultado das tabelas *cliente* e *contato* da Figura 3.2, gerados pelo método Hybrid estão especificado a seguir:

Regra(cliente) =

{clienteID: id(v_1), cliente.numcliente: val(v_2), cliente.nome: val(v_3)},

$v_1 \leftarrow v_r$._*.cliente, $v_2 \leftarrow v_1$.numcliente, $v_3 \leftarrow v_1$.nome;

Regra(contato) =

{contatoID: id(v_2), contato.parentID: id(v_1), contato.email: val(v_3),

contato.telefone: val(v_4)},

$v_1 \leftarrow v_r$._*.cliente, $v_2 \leftarrow v_1$.contato, $v_3 \leftarrow v_2$.email, $v_4 \leftarrow v_2$.telefone;

Dos métodos apresentados no Capítulo 3, o que mais se assemelha ao método desta dissertação é o STORED, já que ambas possuem linguagens declarativas, isto é, o mapeamento é definido por regras que selecionam as partes do documento XML que se deseja armazenar.

Uma das principais diferenças entre as duas linguagens é que o STORED permite a utilização de variáveis que armazenam os rótulos dos elementos. Além disso, há diferenças na linguagem de expressões de caminho. Enquanto no STORED o caracter curinga "_" pode ser utilizado para casar com um único elemento, na linguagem usada existe o curinga "*" para casar com uma sequência de zero ou mais elementos.

Com relação a geração do esquema relacional para armazenamento de um documento XML, o método STORED é baseado em técnicas de data mining no documento XML e um conjunto de consultas, e gera um esquema de armazenamento misto: ou seja, parte do documento (a parte mais estruturada) é armazenada numa base relacional, e o restante em uma base semi-estruturada. O XTREM, ao contrário, gera um esquema relacional baseado em dados semânticos, mais precisamente, chaves XML.

4.2 Propagação de chaves XML para dependências funcionais

No XTREM, a geração de um esquema relacional para armazenar um documento XML requer que o usuário defina duas entradas:

- mapeamento do documento XML para uma tabela universal utilizando a linguagem descrita na seção 4.1;
- conjunto de chaves XML que são válidas no documento XML.

Baseado nestas duas entradas, a ferramenta gera um conjunto de dependências funcionais que podem ser provadas serem válidas na relação universal. Os detalhes do algoritmo de propagação de chaves XML para dependências funcionais estão além do escopo deste trabalho e estão descritos em [13]. Porém, o exemplo a seguir ilustra o funcionamento do algoritmo.

Suponha que deseja-se criar uma tabela Cliente (clieid, nome, contid, email, telefone) com valores de um documento XML que respeita a DTD da Figura 2.6 do Capitulo 2. Sua regra de mapeamento está ilustrada abaixo:

Regra(Cliente) =

{clieid: val(v_2), nome: val(v_3), contid: val(v_5), email: val(v_6), telefone: val(v_7)},

$v_1 \leftarrow v_r$._*.cliente, $v_2 \leftarrow v_1$.clieid, $v_3 \leftarrow v_1$.nome, $v_4 \leftarrow v_1$.contato, $v_5 \leftarrow v_4$.contid,

$v_6 \leftarrow v_4$.email, $v_7 \leftarrow v_4$.telefone;

Além disso, as chaves XML válidas no documento são:

(cliente,{clieid})

(cliente,(nome,{}))

(cliente,(contato,{contid}))

(cliente.contato,(email,{}))

(cliente.contato,(telefone,{})).

A primeira chave define que todo cliente é identificado por clieid. A segunda chave define que cada `cliente` possui um único nome. A terceira chave define que para se

encontrar um contato de algum cliente, além de saber qual é esse cliente, é necessário conhecer o contid. A quarta e a quinta chaves definem que cada contato possui um único email e telefone, respectivamente.

O algoritmo desenvolvido em [13] tem como saída as seguintes dependências funcionais:

$$\text{clieid} \rightarrow \text{nome}$$
$$(\text{clieid, contid}) \rightarrow \text{email}$$
$$(\text{clieid, contid}) \rightarrow \text{telefone}$$

A primeira DF define que um `clieid` é o atributo determinante de `nome`.

A segunda DF define que com o valor de `clieid` + `contid` pode-se conhecer o valor de `email`.

E por fim, a terceira DF define que com o valor de `clieid` + `contid` também pode-se encontrar o valor de `telefone`.

O algoritmo de propagação foi implementado por Jing Qin utilizando a linguagem C e é utilizado pela ferramenta XTREM.

4.3 Normalização

A normalização é o processo de tornar uma coleção de tabelas mais simples e mais regulares estruturalmente, de uma maneira progressiva.

Através do uso da normalização, o XTREM é capaz de detectar possíveis anomalias nas tabelas existentes no banco de dados do usuário, pois se as tabelas existentes estiverem diferentes das geradas pela ferramenta, então estas tabelas podem não estar normalizadas corretamente.

O processo de normalização está dividido em 6 níveis (1FN, 2FN, 3FN, FNBC, 4FN e 5FN). Para a grande maioria das aplicações a 3FN (Terceira Forma Normal) é suficiente para atingir o objetivo proposto pela normalização. Portanto, este é o nível máximo de normalização que o XTREM irá alcançar.

As regras de normalização até a 3FN estão descritas abaixo:

- *Primeira forma normal (1FN)*: Uma tabela está na primeira forma normal, se e apenas se, as suas colunas não são multi-valoradas;

- *Segunda forma normal (2FN)*: Uma tabela está na segunda forma normal, se está na 1FN, e se cada coluna não participante da chave primária for totalmente dependente da chave primária;

- *Terceira forma normal (3FN)*: Uma tabela está na terceira forma normal, se está na 2FN, e se cada coluna não participante da chave primária é dependente de forma não transitiva da chave primária.

O módulo Normalizador do XTREM transforma a tabela universal definida pela regra de mapeamento de entrada em um esquema de relações normalizadas em 3FN baseado nas dependências funcionais propagadas pelas chaves XML.

O algoritmo de normalização utilizado pode ser encontrado em [5]. Seus passos estão descritos a seguir:

1. *Eliminar colunas desnecessárias*: Seja F o conjunto de DF´s de entrada. Elimine colunas desnecessárias do lado direito de cada DF em F, produzinho um conjunto G. Uma coluna é desnecessária se sua eliminação não altera o fechamento do conjunto F;

2. *Encontrar cobertura*: Encontre uma cobertura não redundante H de G;

3. *Particionar*: Particione H em grupos tal que todas as DF´s em cada grupo tenham lados esquerdos idênticos;

4. *Concatenar chaves equivalentes*: Seja J um conjunto vazio. Para cada par de grupos H_i e H_j, com lados esquerdos X e Y, respectivamente, concatene H_i e H_j se houver uma bijeção $X \leftrightarrow Y$ em H^+. Para cada bijeção, adicione $X \rightarrow Y$ e $Y \rightarrow X$ em J. Para cada coluna $A \in Y$, se $X \rightarrow A$ está em H, então remova-a de H. Faça o mesmo para cada $Y \rightarrow B$ em H com $B \in X$;

5. *Eliminar dependências trasitivas*: Encontre um $H' \subseteq H^+$ tal que $(H' + J)^+ = (H + J)^+$ e nenhum outro subconjunto de H' tenha essa propriedade. Adicione cada DF de J em seu correspondente grupo em H';

6. *Construir relações*: Para cada grupo, construa uma relação consistindo de todos as colunas que se encontram no grupo. Cada conjunto de colunas que aparecem do lado esquerdo de cada DF no grupo é uma chave candidata da relação.

Para a aplicação do algoritmo descrito acima, foram utilizados: o algoritmo de fechamento de DF´s encontrado em [6] e apresentado na seção 2.1.1.2, o algoritmo de busca de uma cobertura não redundante encontrado em [5], apresentado na seção 2.1.1.4, e no passo de *Eliminar dependências transitivas*, o algoritmo Membership encontrado no artigo [4] e apresentado na seção 2.1.1.2.

Para exemplificar, a aplicação do algoritmo no esquema de tabela universal apresentado na seção 4.2 resulta no seguinte esquema de tabelas:

R1	(clieid, nome)
R2	(clieid, contid, email, telefone)

4.4 Geração da Transformação Normalizada

Com as tabelas já normalizadas, o próximo passo é fazer a extração dos dados do documento XML. Esta tarefa é executada pelo módulo Transformador XML-Rel.

A saída do Transformador XML-Relacional é um *Script* em SQL para povoar as tabelas normalizadas.

Usando o exemplo visto até agora, e baseado nas regras de mapeamento acima, o Transformador XML-Rel resultará no seguinte *Script*:

```
INSERT INTO R1(clieid, nome) VALUES("1", "Jose da Silva")
    /* Chave(s) candidata(s) = [clieid] */
INSERT INTO R2(clieid, contid, email, telefone) VALUES("1", "5", "jose@xxx.com.br", "")
    /* Chave(s) candidata(s) = [clieid, contid] */
INSERT INTO R1(clieid, nome) VALUES("2", "Maria Costa")
```

```
/* Chave(s) candidata(s) = [clieid] */
INSERT INTO R2(clieid, contid, email, telefone) VALUES("2", "6", "", "224-7376")
/* Chave(s) candidata(s) = [clieid, contid] */
```

Para uma melhor visualização do resultado geral da ferramenta XTREM, as tabelas resultantes, incluindo o esquema e os valores, são mostradas na Figura 4.2.

R1:

clieid	nome
1	José da Silva
2	Maria Costa

R2:

clieid	contid	email	telefone
1	5	jose@xxx.com.br	
2	6		224-7376

Figura 4.2: Tabelas resultantes da execução do *Script* gerado pelo XTREM

Vale lembrar que em nenhum módulo do XTREM existe a tarefa de verificação de chaves estrangeiras, ou seja, o sistema não faz, em nenhum ponto, a integridade referencial dos dados. Ele faz apenas a estruturação das tabelas e a extração dos dados.

Em comparação aos métodos de mapeamento apresentados no Capítulo 3, pode-se notar que o resultado apresentado por esta proposta possui o mapeamento mais próximo do tradicional, observando que, diferentemente do STORED, as tabelas resultantes da ferramenta são formadas sem a influência do usuário. Em conseqüência disto, as tabelas geradas pela ferramenta não possuem nomes significativos. Elas são nomeadas automaticamente seguindo uma seqüencia com início igual a 1 e com término igual ao número total de tabelas encontradas.

Além disso, no XTREM não é necessário a criação de identificadores para formar a chave primária das relações, uma vez que estas são geradas à partir das dependências funcionais computadas.

CAPÍTULO 5

IMPLEMENTAÇÃO DO SISTEMA

Neste capítulo é apresentada a implementação dos seguintes módulos componentes do XTREM:

- Propagador de Chaves XML;
- Normalizador;
- Transformador XML-Rel.

Exceto o módulo propagador de chaves, que foi implementado por Jing Qin utilizando a linguagem C, o desenvolvimento dos demais módulos faz parte desta dissertação. Todos foram desenvolvidos na linguagem de programação multi-plataforma Java® da Sun Microsystems.

O interpretador de regras do XTREM foi construído através do uso de um interpretador léxico e um interpretador sintático, gerados pelas ferramentas JLex [18] e Java_Cup [17], respectivamente.

5.1 A linguagem de programação Java

Java vem se tornando um padrão cada vez mais efetivo no mercado, tendo em vista ser uma linguagem simples e ao mesmo tempo robusta, e também devido à sua independência de plataforma.

5.2 JLex

JLex (*Java Lex*) [18] é um gerador de analisadores léxicos. Um analisador léxico quebra um determinado código de entrada em *tokens*, que são as palavras conhecidas que

formam linguagens de programação. Como implementar analisadores léxicos manualmente é uma tarefa entediante, foram criadas ferramentas que fazem este processo para os programadores.

JLex foi baseado na ferramenta Lex, um gerador de analisadores léxicos para o sistema operacional UNIX, que cria um código fonte na linguagem de programação C para um determinado código de entrada. Com a mesma especificação que Lex, JLex foi desenvolvido em Java e tem como saída um código fonte de um analisador léxico também em Java.

O código JLex para a linguagem de transformação apresentada na seção 4.1 pode ser encontrada no Apêndice A.1.

5.3 Java_cup

Java_cup (*Java Based Constructor of Useful Parsers* – Construtor de analisadores úteis baseado em Java) [17] é um gerador de analisadores sintáticos LALR (*LookAhead LR*). Um analisador sintático é um sistema que verifica se um programa está gramaticalmente correto – como o código de um programa escrito em uma linguagem de programação, ou também, no caso do XTREM, regras para extração dos dados de um documento XML.

Java_cup foi baseado no programa YACC [19], um analisador sintático para a linguagem de programação C, mas com a diferença de ter como resultado códigos fonte de analisadores sintáticos escritos em Java.

O código Java_cup para a linguagem de transformação apresentada na seção 4.1 pode ser encontrada no Apêndice A.2.

5.4 Extração dos dados do documento XML

Os dois métodos mais conhecidos para a extração dos dados de um documento XML são: o método DOM [15] e o método SAX [20].

Os dois métodos executam a extração dos dados de forma distinta.

O método DOM lê todo o conteúdo do documento XML e o guarda em memória. O SAX extrai as informações à medida que o documento é lido, de forma que tais informações

não precisam ser gravadas em memória. Elas são apresentadas em métodos, de onde podem ser capturadas pelo sistema.

Ambos têm vantagens e desvantagens: a principal vantagem do DOM é que uma vez lido o documento XML, todos os outros acessos a ele não utilizarão mais o disco rígido, pois seus dados estarão armazenados em memória. Mas, por outro lado, quando um documento XML é muito extenso, ele pode ocasionar erro no sistema, pois a quantidade de memória pode não ser suficiente.

A vantagem do SAX é que ele lê uma versão linearizada do documento XML, não sendo portanto necessário armazená-lo na memória. Porém a manipulação do documento é mais difícil e trabalhosa. Para não limitar a utilização do sistema para documentos XML que possam ser armazenados na memória do computador, optou-se por utilizar o método SAX.

Uma vez definido o método de extração dos dados do documento XML, o próximo passo era encontrar uma ferramenta que possibilitasse o uso de tal método juntamente com a linguagem Java. A ferramenta encontrada foi o pacote Ælfred XML Parser [3], que faz uma leitura "in-order" no documento XML e fornece os dados na medida em que os nós (elementos) são lidos – característica do método SAX.

5.5 A ferramenta XTREM

Para os exemplos contidos nesta sessão, o documento XML de entrada utilizado é o documento representado na Figura 1.1.

O objetivo era fazer uma ferramenta que fosse funcional e simples de utilizar. Seguindo este raciocínio, a interface do XTREM, ilustrada na Figura 5.1, possui apenas uma barra de botões e 2 campos editores. A barra de botões fica no topo da ferramenta. Um dos campos editores situa-se na parte superior, onde a regra universal de mapeamento e as chaves XML válidas para o documento são inseridas; o outro, na parte inferior, retorna para o usuário o *Script* com os comandos de inserção dos dados extraídos nas tabelas normalizadas.

Para utilizar a ferramenta, deve-se seguir os seguintes passos:

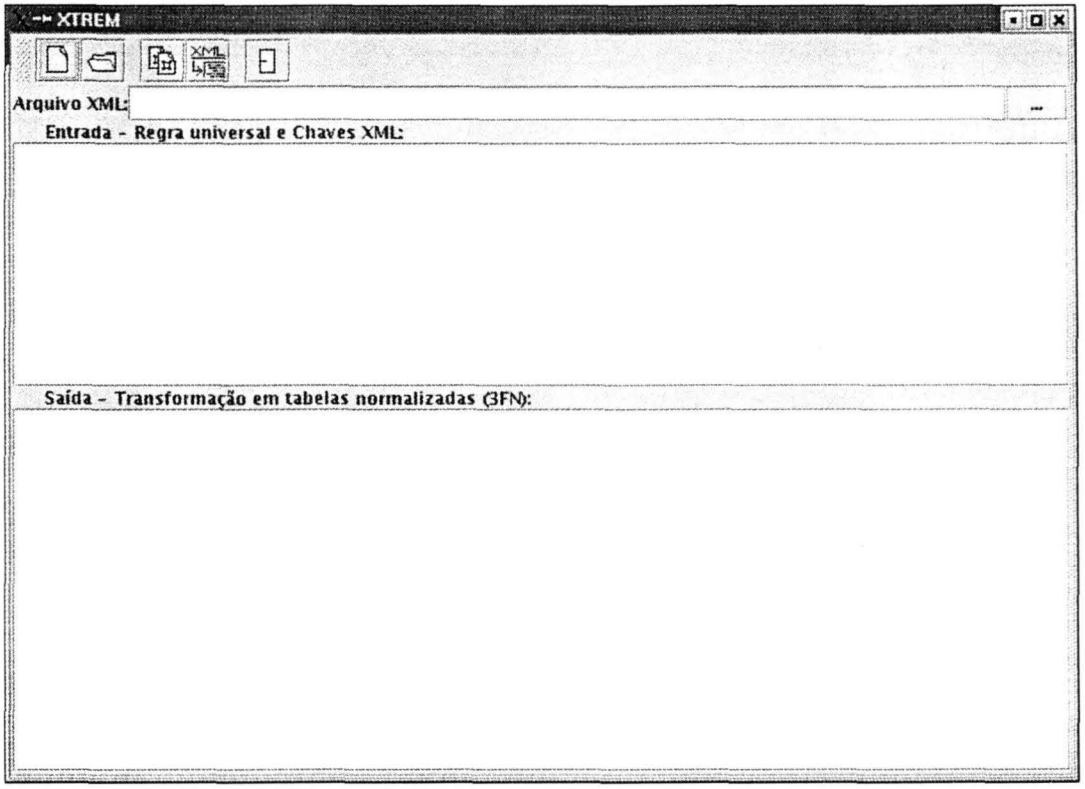

Figura 5.1: Interface do XTREM

- Selecionar o arquivo XML de onde os dados serão extraídos;
- Entrar com a regra universal de mapeamento e as chaves XML (essa entrada pode ser tanto digitada diretamente no XTREM quanto lida de um arquivo);
- Clicar em Analisar (para que a ferramenta analise se a regra está com a sintaxe correta) e em seguida em Extrair (para que a ferramenta gere o esquema das relações normalizadas e busque os dados dentro do documento XML).

Todas estas estapas estão ilustradas nas Figuras 5.2, 5.3 e 5.4, respectivamente.

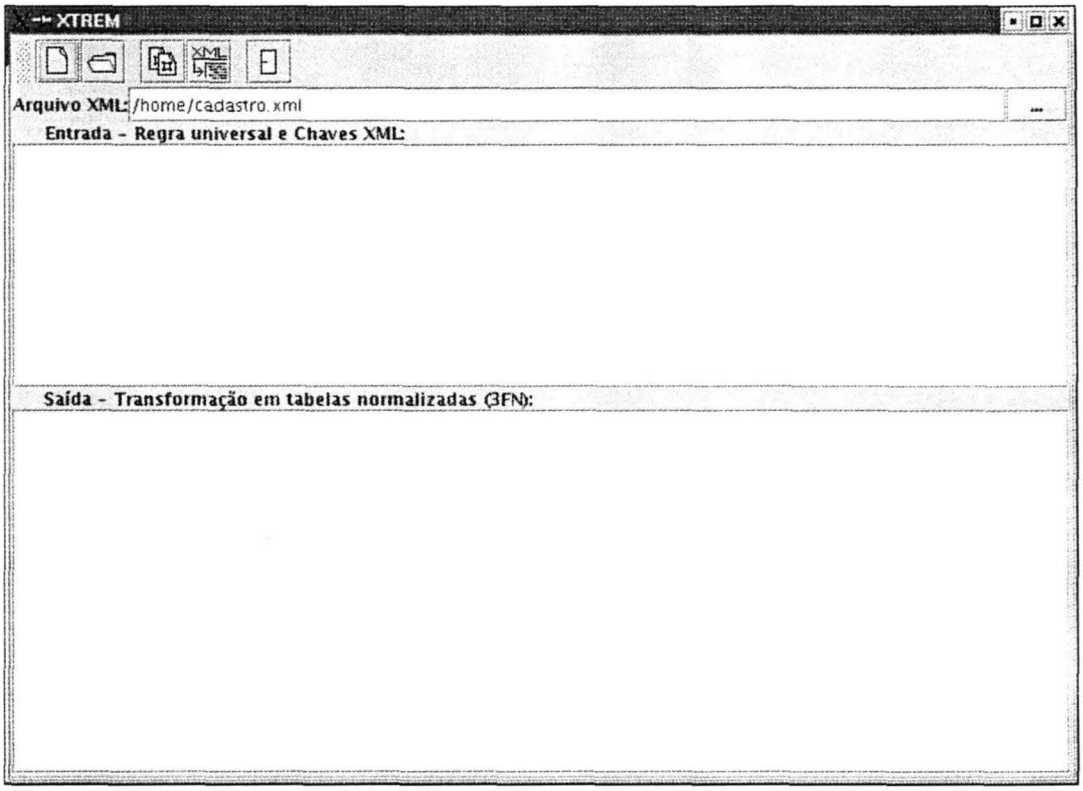

Figura 5.2: Informando o arquivo XML de extração

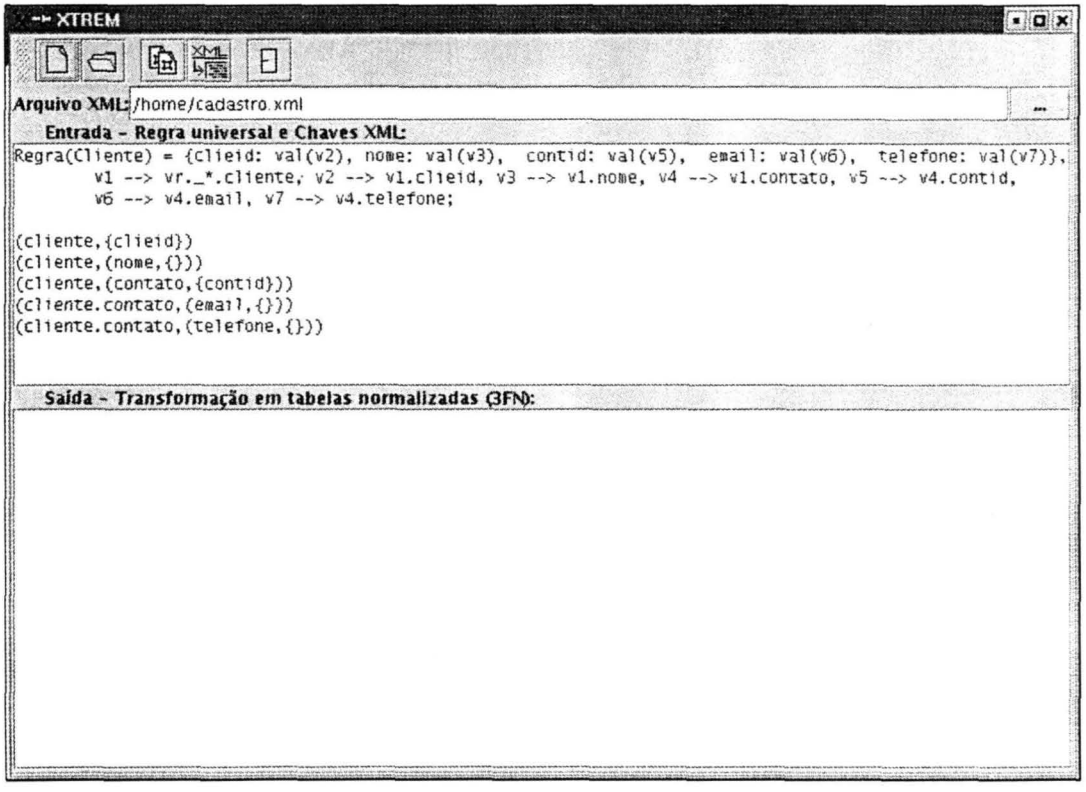

Figura 5.3: Informando a regra universal de mapeamento e as chaves XML

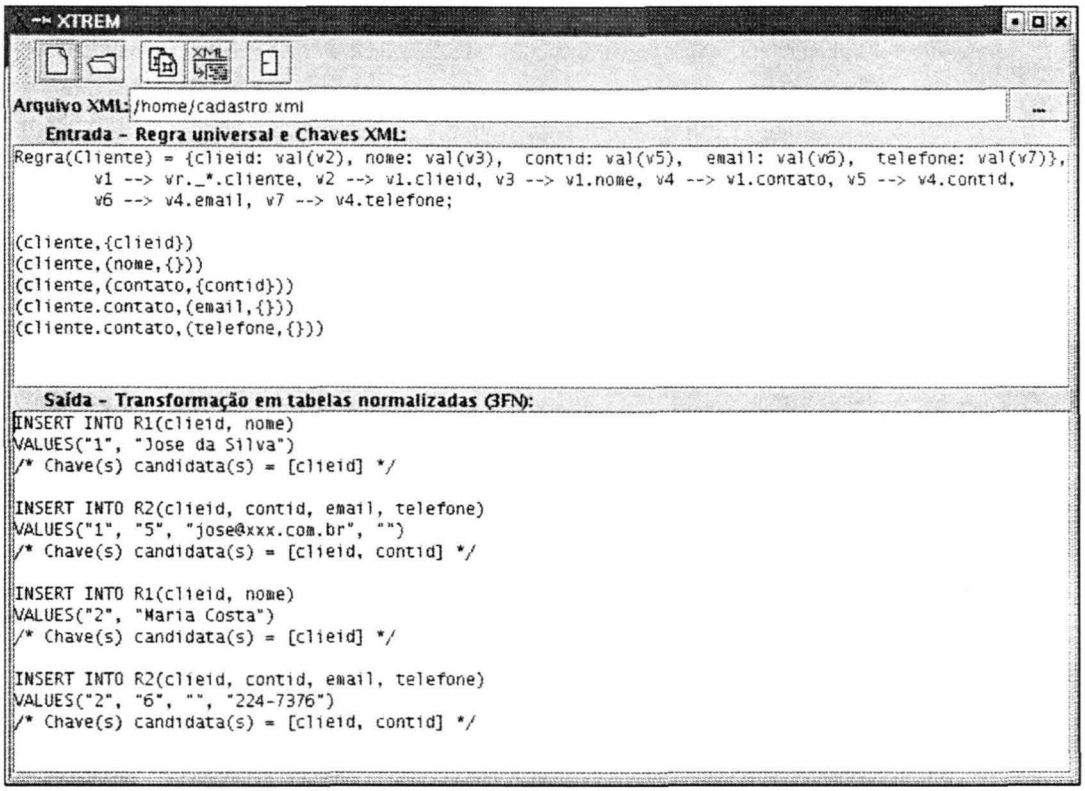

Figura 5.4: Script com os comandos de inserção nas tabelas normalizadas em 3FN

CAPÍTULO 6

CONCLUSÃO

Esta dissertação apresenta a especificação e implementação da ferramenta XTREM, uma ferramenta que gera um esquema de relações normalizadas em terceira forma normal (3FN) e extrai os dados de um documento XML para sua tranformação no esquema relacional resultante.

Ao contrário dos trabalhos para geração de esquemas relacionais apresentados no Capítulo 3, o método apresentado nesta dissertação é baseado em informações semânticas sobre o documento XML, em particular chaves XML. O método de geração do esquema relacional é baseado na transformação de chaves XML em dependências funcionais (DFs) definidas sobre o esquema de uma relação universal. Estas DFs são então utilizadas para fazer a normalização desta relação em 3FN. Assim, no XTREM, a geração do esquema relacional é similar ao projeto lógico tradicional de banco de dados. Além disso, visto que o usuário é responsável pela definição do esquema da relação universal, a ferramenta possibilita a extração de apenas uma porção do documento XML.

Dentre os métodos estudados, o método STORED é o que mais se preocupa tanto com o mapeamento XML-relacional quanto com a extração dos dados do documento XML. Mas, o esquema de suas tabelas resultantes dependem exclusivamente da normalização aplicada pelo utilizador do método.

A ferramenta XTREM também se preocupa com a extração dos dados do documento XML, mas com um diferencial: a normalização das tabelas resultantes do mapeamento é feita automaticamente, tornando-se uma ferramenta diferenciada dos métodos estudados.

Assim, um dos objetivos desta dissertação foi atingido: preencher uma lacuna deixada por estudos anteriores, onde apenas a parte conceitual do processo de mapeamento XML-Relacional é apresentado (a criação do esquema de tabelas relacionais), deixando em branco uma parte importante para o uso efetivo de suas técnicas (a busca dos dados no

documento XML conforme o esquema de tabelas gerado).

Com isso, a maior contribuição desta dissertação é a criação de uma ferramenta capaz de, não apenas encontrar dependências funcionas e utilizá-las para normalizar tabelas resultantes do mapeamento, mas também fazer a extração dos dados do documento XML para serem inseridos nas tabelas normalizadas.

Alguns trabalhos futuros que podem ser realizados para deixar a ferramenta mais completa são: a criação automática de chaves candidatas e a criação de chaves estrangeiras para que a verificação de integridade referencial possa ser feita automaticamente pelo Sistema Gerenciador de Banco de Dados (SGBD) ao executar o *Script* gerado pelo XTREM.

BIBLIOGRAFIA

[1] Serge Abiteboul, Peter Buneman, e Dan Suciu. *Gerenciando dados na Web.* Rio de Janeiro. Editora Campus, 2000.

[2] Serge Abiteboul, Richard Hull, e Victor Vianu. *Foundations of Databases.* Reading, MA, USA. Editora Addison-Wesley, 1995.

[3] Ælfred XML SAX parser, 2002. Disponível em http://www.microstar.com/XML.

[4] Catriel Beeri e Philip A. Bernstein. An algorithmic approach to normalization of relational data base schemas. Relatório Técnico CSRG-73, Department of Computer Science, U. of Toronto, Canada, 1976.

[5] Catriel Beeri e Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM SIGMOD*, 4(1):455–469, 1979.

[6] Philip A. Bernstein. Normalization and functional dependencies in the relational data base model. Relatório Técnico Tech. Rep. CSRG-60, Ph.D. Diss., Department of Computer Science, U. of Toronto, Canada, 1975.

[7] Phil Bohannon, Juliana Freire, Prasan Roy, e Jérôme Siméon. From XML schema to relations: A cost-based approach to XML storage. *ICDE*, 2002.

[8] Ronald Bourret. XML and databases, 1999. http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[9] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, e Wang-Chiew Tan. Keys for XML. *Computer Networks*, 39(5):473 – 487, Agosto de 2002.

[10] Peter Buneman, Wenfei Fan, Jérome Siméon, e Scott Weinstein. Constraints for semistructured data and XML. *ACM SIGMOD*, 30(1), Março de 2001.

[11] Scientific data formats, 2002. Disponível em http://www.cv.nrao.edu/fits/traffic/scidataformats/faq.html.

[12] Susan Davidson, Wenfei Fan, e Carmem Hara. Propagating XML keys to relations. Relatório Técnico MS-CIS-01-33, University of Pennsylvania, 2001.

[13] Susan Davidson, Wenfei Fan, Carmem Hara, e Jing Qin. Propagating XML constraints to relations. *ICDE*, 2003.

[14] Alin Deutsch, Mary Fernandez, e Dan Suciu. Storing semistructured data with STORED. *ACM SIGMOD*, páginas 431–442, 1999.

[15] Document object model - DOM, 2002. Disponível em http://www.w3c.org/DOM.

[16] Daniela Florescu e Donal Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Eng. Bulletin*, 22(3):27–34, 1999.

[17] Analisador sintático Java_cup, 2002. Disponível em http://www.cs.princeton.edu/ appel/modern/java/CUP.

[18] Analisador léxico Jlex, 2002. Disponível em http://www.cs.princeton.edu/ appel/modern/java/JLex.

[19] Steven C. Johnson. Yacc: Yet another compiler compiler. *UNIX Programmer's Manual*, volume 2, páginas 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[20] Simple API for XML - SAX, 2002. Disponível em http://www.saxproject.org.

[21] Jayavel Shanmugasundaran, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, e Jeffrey Naughton. Relational databases for querying xml documents: Limitations and oportunities. *25th VLDB Conference*, páginas 79–90, 1999.

[22] Jeffrey D. Ullman. *Principles of database and knoledgebase systems.* Computer Science Press, 1988.

[23] XML Schema, 2002. Disponível em http://www.w3c.org/XML/Schema.

APÊNDICE A

CÓDIGOS PARA CRIAÇÃO DO INTERPRETADOR DE REGRAS

A.1 JLex

```
package Regra;

import java_cup.runtime.Symbol;
%%
%cup
%eofval{
  return new Symbol(sym.EOF);
%eofval}
%%
"regra" { return new Symbol(sym.REGRA); }
"valor" { return new Symbol(sym.VALOR); }
"id" { return new Symbol(sym.ID); }
":" { return new Symbol(sym.DOIS_PONTOS); }
"," { return new Symbol(sym.VIRGUL); }
";" { return new Symbol(sym.PONTO_VIRGUL); }
"=" { return new Symbol(sym.IGUAL); }
"{" { return new Symbol(sym.ABRE_CHAVE); }
"}" { return new Symbol(sym.FECHA_CHAVE); }
"(" { return new Symbol(sym.ABRE_PARENT); }
")" { return new Symbol(sym.FECHA_PARENT); }
"<-" { return new Symbol(sym.ATRIBUICAO); }
[ \t\r\n\f] { /* ignore white space. */ }
("/"([_a-zA-Z]+[0-9]*))+ { return new Symbol(sym.BARRA_ELEMENTO, new String(yytext())); }
("/@"([_a-zA-Z]+[0-9]*))+ { return new Symbol(sym.BARRA_ATRIBUTO, new String(yytext())); }
([_a-zA-Z]+[0-9]*)+ { return new Symbol(sym.ALFA_NUMERO, new String(yytext())); }
[0-9]+ { return new Symbol(sym.NUMERO, new String(yytext())); }
```

A.2 Java_cup

```
package Regra;

import java_cup.runtime.*;

parser code {:
  public java.util.ArrayList resultado = new java.util.ArrayList();

  public java.util.ArrayList analise(String texto){
    java.io.Reader r = new java.io.StringReader(texto);
    try{
      new parser(new Yylex(r)).parse();
    }
    catch(java.lang.Exception ee){
      javax.swing.JOptionPane.showMessageDialog(null,ee.getMessage());
      System.err.println(ee);
    }
    return resultado;
```

```
  }
:}

terminal PONTO_VIRGUL, IGUAL, ABRE_CHAVE,
         FECHA_CHAVE, ABRE_PARENT, FECHA_PARENT,
         REGRA, ATRIBUICAO, VALOR, ID, DOIS_PONTOS,
         VIRGUL, UNDERS;
terminal String NUMERO, ALFA_NUMERO, BARRA_ELEMENTO, BARRA_ATRIBUTO;

non terminal lista_regras, regra, lista_colunas, coluna, lista_caminhos, caminho;
non terminal String dado, nome_caminho;

lista_regras ::= lista_regras regra | regra;

regra ::= REGRA ABRE_PARENT dado:t
  {: parser.resultado.add(new java.lang.String("tabela  = "+t)); :}
  FECHA_PARENT IGUAL ABRE_CHAVE lista_colunas FECHA_CHAVE VIRGUL lista_caminhos PONTO_VIRGUL;

lista_colunas ::= lista_colunas VIRGUL coluna | coluna;

coluna ::= dado:a DOIS_PONTOS VALOR ABRE_PARENT dado:c
             {: parser.resultado.add(new java.lang.String("coluna  = "+a+"\n"
               +"variavel= "+c+" (valor)")); :} FECHA_PARENT
           | dado:b DOIS_PONTOS ID ABRE_PARENT dado:d
             {: parser.resultado.add(new java.lang.String("coluna  = "+b+"\n"
               +"variavel= "+d+" (id)")); :} FECHA_PARENT
           ;

lista_caminhos ::= lista_caminhos VIRGUL caminho | caminho;

caminho ::= dado:v ATRIBUICAO nome_caminho:c
              {: parser.resultado.add(new java.lang.String("var_cami= "
                +v+"\n"+"caminho = "+c)); :};

nome_caminho ::=   dado:a
                   {: RESULT=a; :}
                  |dado:b BARRA_ELEMENTO:c
                   {: RESULT=new java.lang.String(b)+new java.lang.String(c); :}
                  |dado:d BARRA_ATRIBUTO:e
                   {: RESULT=new java.lang.String(d)+new java.lang.String(e); :}
                  ;

dado ::= ALFA_NUMERO:a {: RESULT=a; :};
```