XML *Publisher*: Um *Framework* para Publicar Dados Objeto-Relacionais Através de Visões XML

Lineu Antonio de Lima Santos

Orientadora: Profª. Drª. Vânia Maria Ponte Vidal

Dissertação apresentada ao Mestrado em Ciência da Computação da Universidade Federal do Ceará (UFC), como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Banco de Dados.

FORTALEZA - CEARÁ
Agosto - 2004

Mestrado em Ciência da Computação
Universidade Federal do Ceará

XML *Publisher*: Um *Framework* para Publicar Dados Objeto-Relacionais Através de Visões XML

Lineu Antonio de Lima Santos[1]

Agosto de 2004

Banca Examinadora

- Profa. Dra. Vânia Maria Ponte Vidal
- Prof. Dr. Marco Antonio Casanova
- Prof. Dr. Ângelo Roncalli A. Brayner

[1] Bolsista CNPQ

À minha esposa Kelly,
inesgotável fonte de amor.

Agradecimentos

A minha querida orientadora Vânia Vidal pelo incentivo, paciência, ensinamentos, dedicação e tempo dispensados no trabalho de acompanhamento e orientação desta dissertação de mestrado.

Ao Professor Dr. Marco Antônio Casanova e ao Professor Dr. Ângelo Brayner, por terem aceitado o convite para participar da banca examinadora e por colaborarem para a melhoria deste trabalho.

À família que me adotou, me abrigou e suportou todas as minhas alterações comportamentais por mais de um ano. Obrigado Tathianne, Constância e Jander, vocês me fizeram sentir amado e querido em uma terra de estranhos. Em especial, à Tathianne por ter sido muito mais que uma amiga, uma verdadeira irmã.

À querida amiga Valdiana, por ter perdido muitos dias e noites me ajudando a concluir este trabalho. Você foi a pessoa que me deu mais ânimo para terminar a dissertação. Obrigado pela suas figuras, sugestões, revisões, suporte técnico e por todas as maças com queijo que costumava me servir. Principalmente, obrigado pela sua amizade em um momento tão difícil da minha vida.

Ao Fernando Lemos, bolsista de iniciação cientifica, pela implementação das ferramentas apresentadas nesta dissertação.

Agradeço também a todos os amigos que de alguma forma contribuíram para a conclusão deste trabalho. Obrigado Wamberg, Tâmara, Denis e Fábio Pinheiro.

A Universidade Federal do Ceará pela oportunidade do grande crescimento profissional e intelectual.

Aos professores e funcionários do Mestrado em Ciência da Computação da Universidade Federal do Ceará pelo apoio à realização do curso.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo auxílio financeiro que possibilitou a realização deste trabalho.

Resumo

Nos últimos anos, a *Web* se tornou o maior ambiente capaz de fornecer acesso a fontes de dados heterogêneas. Nesse contexto, a linguagem XML se firmou como um formato universal para publicação e troca de dados na *Web*. Como a maioria dos dados está armazenada em bancos de dados relacionais ou objeto-relacionais, surgiu a necessidade de definir mecanismos para publicá-los no formato XML. Neste trabalho, apresentamos o XML *Publisher*, um *framework* para publicação de dados armazenados em banco de dados relacionais ou objeto-relacionais no formato XML. O framework proposto permite a publicação de visões XML, de forma que a base de dados pode ser consultada através dessas visões, usando um subconjunto da linguagem *XQuery*, também definido neste trabalho.

No enfoque proposto, para publicar uma visão XML, deve-se definir uma visão de objetos, denominada visão de objetos *default* (VOD), de modo que os objetos da VOD possuem a mesma estrutura dos elementos da visão XML. Consultas definidas sobre o esquema da visão XML são traduzidas em consultas sobre o esquema da respectiva VOD, e que por sua vez são traduzidas, pelo mecanismo de visão do SGBD, em consultas definidas sobre o banco de dados.

Para o XML *Publisher* desenvolveu-se um ambiente para facilitar o processo de definição e manutenção das visões XML publicadas. No ambiente proposto, o processo de definição de uma visão XML começa com o projetista especificando, através de uma interface gráfica, o esquema XML da visão. Em seguida, o esquema da VOD é automaticamente gerado pela ferramenta DSG (***D****efault Object View* ***S****chema* ***G****enerator*). Então, a partir do esquema da VOD gerado e do esquema do banco de dados, o projetista utiliza a ferramenta VBA (*View-By-Assertion*) para gerar de forma gráfica as assertivas de correspondência da visão, as quais especificam os relacionamentos entre o esquema da visão e o esquema do banco. Com base nas assertivas da visão, VBA gera automaticamente a consulta SQL:1999 que realiza o mapeamento definido pelas assertivas da visão.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1
Introdução

1.1 Motivação

Nos últimos anos, a *Web* se transformou no maior ambiente capaz de fornecer acesso a fontes de dados heterogêneas. Nesse contexto, a linguagem XML [58] se firmou como um formato universal para publicação e troca de dados, pois XML é uma linguagem para representação de dados autodescritível e bastante flexível, o que a torna ideal para representar dados estruturados e semi-estruturados.

De fato, muitos grupos industriais e científicos [13][60][6], têm definido esquemas XML públicos que especificam o formato XML dos dados a serem trocados entre suas aplicações. A meta é usar XML como uma "linguagem franca" entre aplicações que trocam dados, tornando possível trocar esses dados sem levar em consideração a plataforma na qual eles estão armazenados ou o modelo de dados no qual eles estão representados [21]. Como a maioria dos dados está armazenada em bancos de dados relacionais, objeto-relacionais ou baseados em objetos, surgiu a necessidade de definir mecanismos para publicá-los no formato XML. Uma forma geral de se publicar esses dados é feita através do uso de visões XML, que podem ser consultadas por aplicações/usuários *Web* utilizando linguagens de consultas próprias para XML, tais como *XML*-QL [15], *XPath* [62], *XQuery* [67] etc. Um mecanismo para consultar visões XML é necessário pois, geralmente, uma aplicação *Web* tem interesse apenas em uma parte dos dados publicados através da visão [44].

Uma visão XML pode ser materializada ou virtual. Na abordagem materializada, os dados são extraídos do banco de dados, transformados em XML e armazenados num repositório qualquer, o qual pode ser um banco de dados XML nativo. Então, consultas sobre a visão XML materializada são processadas diretamente sobre o repositório. Visões são freqüentemente materializadas para acelerar o processamento das consultas,

o que é requisito fundamental em aplicações onde os dados são remotos ou o tempo de resposta é crítico [10]. Por outro lado, visões materializadas devem ser atualizadas para manter sua consistência em relação à base de dados. Na abordagem virtual, os dados permanecem no banco de dados, o que elimina a preocupação com a consistência da visão em relação à fonte de dados. Por essa razão, consultas definidas sobre a visão XML virtual devem ser traduzidas em consultas sobre o banco de dados, sendo que os resultados dessas consultas serão transformados em XML, como é mostrado na Figura 1.1.

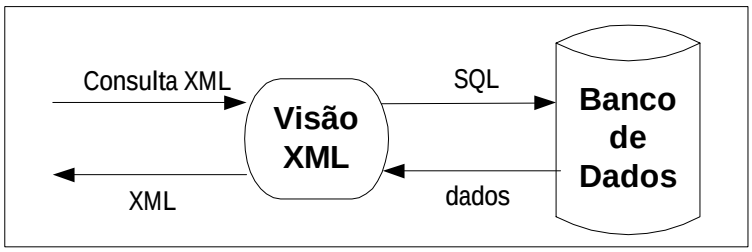

Figura 1.1: Visão XML Virtual

O uso de visões XML também tem uma grande importância em sistemas de integração de dados que adotam uma arquitetura de mediadores baseada em XML [1][3] [32] [53]. O mediador suporta uma visão integrada XML e as fontes locais exportam visões XML. Consultas submetidas ao mediador são decompostas em consultas sobre as visões XML das fontes locais. Consultas sobre as visões XML são então traduzidas em consultas numa linguagem específica da fonte de dados correspondente.

Já existem alguns sistemas como o *XPeranto* [7][12][44][45], o *Silkroute* [21][22] e o SQL *Server* 2000 [41] que dão suporte à publicação de dados relacionais, através de visões XML. No entanto, pouco esforço tem sido empreendido para resolver o problema de publicar dados objeto-relacionais através de visões XML. Nesse contexto o problema é bem mais complexo, pois o modelo objeto-relacional permite o desenvolvimento de estruturas complexas (tabelas aninhadas, tipos de dados estruturados e de referência), além das estruturas relacionais. O *Oracle XSQL Pages Publishing Framework* [33] é um exemplo de *framework* que dá suporte à geração dinâmica de XML na *Web* a partir de dados objeto-relacionais.

Neste trabalho, tratamos o problema de publicar dados objeto-relacionais através de visões XML. A seguir, na Seção 1.2, apresentaremos detalhes sobre algumas propostas para publicação de dados existentes através de visões XML. Na Seção 1.3,

apresentamos nossa proposta. As contribuições alcançadas com este trabalho são mostradas na Seção 1.4.

1.2 Trabalhos Relacionados

A popularidade de XML despertou o interesse de muitos em utilizar SGBD's relacionais e objeto-relacionais para armazenar e consultar documentos XML [5][38][43][46][47][50][54]. No entanto, o foco desse trabalho é a publicação de dados relacionais e objeto-relacionais existentes como documentos XML. Nesta seção apresentamos algumas das propostas para publicação de dados relacionais através de visões XML. Apresentamos também o XSQL *Pages*, um *framework* desenvolvido pela *Oracle* para criação de páginas *Web* dinâmicas que contêm dados XML gerados a partir de um banco de dados objeto-relacional.

1.2.1 SQL *Server* 2000

O Microsoft SQL *Server* 2000 [24][41][69] é um SGBD relacional que possui um mecanismo para publicação de seus dados através de visões XML. O ponto chave desse mecanismo é o conceito de esquema anotado. Um esquema anotado consiste em um documento XML que armazena a estrutura da visão XML juntamente com "anotações" que descrevem o mapeamento entre cada elemento ou atributo da visão XML e o banco de dados. No SQL *Server* 2000, esquemas anotados podem ser definidos usando a linguagem XML *Schema* ou a linguagem XDR (XML *Data Reduced*).

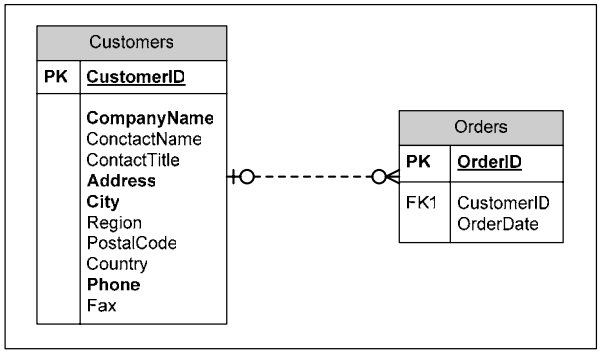

Figura 1.2: Esquema do Banco de Dados *Customers/Orders*

A Figura 1.3 mostra um exemplo de esquema anotado que define uma visão XML simples sobre o banco de dados da Figura 1.2. O elemento *Customer* é mapeado para a tabela *Customer*s usando a anotação *sql:relation*; seu atributo ID é mapeado para o atributo *Customer*ID da tabela *Customer*s com a anotação *sql:field*. Finalmente, o elemento *Order*, que é filho do elemento *Customer*, é definido a partir do relacionamento entre as tabelas *Customer*s e *Order*s. A anotação *sql:relation*ship descreve como a tabela *Customer*s se relaciona à tabela *Order*s. Dentro do elemento *Order* podemos definir todos os seus sub-elementos e atributos.

```
<Schema xmlns="urn:Schemas-microsoft-com:xml-data"
      xmlns:sql="urn:Schemas-microsoft-com:xml-sql">
 <ElementType name="Customer"
          sql:relation="Customers" >
  <AttributeType name="ID"/>
  <attribute type="ID" sql:field="CostumerID"/>
  </attribute>
  <element type="Order">
   <sql:relationship key-relation="Customers"
                key="CustomerID"
                foreign-relation="Orders"
                foreign-key="CustomerID" >
   </sql:relationship>
   …
  </element>
 </ElementType>
</Schema>
```

Figura 1.3: Visão XML no SQL *Server* 2000

O SQL *Server* 2000 possui um utilitário gráfico para definição de esquemas anotados, chamado SQL XML *View Mapper*. Como apresentado na Figura 1.4, esse utilitário carrega o esquema das tabelas do banco de dados e através de uma interface *drag-and-drop*, o usuário pode estabelecer o mapeamento entre o esquema da visão XML e o esquema do banco de dados.

Visões XML definidas no SQL *Server* 2000 podem ser consultadas através de consultas definidas em um subconjunto de *XPath*. Para processar consultas *XPath*, o SQL *Server* 2000 utiliza o esquema anotado para traduzir a *XPath* em uma consulta FOR XML que retorna somente os dados XML requisitados. FOR XML é uma extensão da linguagem SQL para o SQL *Server* 2000 que permite indicar o formato XML do resultado de uma consulta SQL na própria consulta.

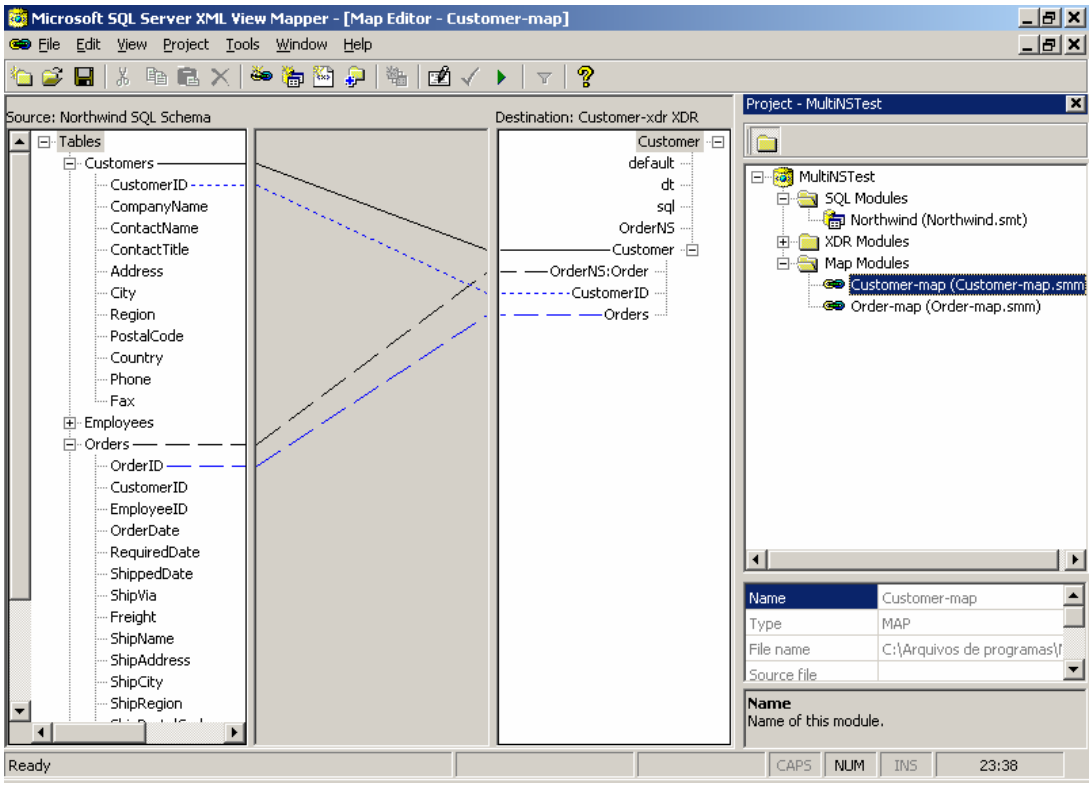

Figura 1.4: SQL *Server* XML *View Mapper*

1.2.2 XSQL *Pages Publishing Framework*

A tecnologia XSQL *Pages* (páginas XSQL) [33], desenvolvida pela *Oracle* [37], combina o poder de XML, XSLT [68] e XSQL para publicar conteúdo dinâmico na *Web* baseado em informações extraídas de bancos de dados objeto-relacionais. Considere, por exemplo, o esquema do banco de dados da Figura 1.5 e a página XSQL da Figura 1.6 (V3.xsql), a qual publica o resultado da consulta SQL:1999 contida em <xsql:query> no formato do esquema XML mostrado na Figura 1.7 (b). Toda vez que uma página XSQL é requisitada através da *Web*, o resultado da consulta SQL:1999 é transformado pelo processador XSQL em um documento XML no formato canônico. A Figura 1.7 (a) mostra o esquema XML do documento XML retornado pelo XSQL para a consulta em V3.xsql. Um exemplo de um documento no formato canônico é mostrado na Figura 1.8. Com o uso da tecnologia XSLT, a estrutura do documento XML canônico pode ser alterada de acordo com o interesse do projetista. Para isso, na página XSQL da Figura 1.6, o atributo href associa um documento stylesheet nomeado V3.xsl, que utiliza uma folha de estilo XSLT para transformar o documento XML canônico no formato do

esquema XML apresentado na Figura 1.7 (b). A correspondência entre os elementos do esquema XML canônico e os elementos do esquema XML também são mostradas na Figura 1.7. A Figura 1.8 mostra um exemplo das transformações aplicadas a um resultado da consulta SQL:1999 na página da Figura 1.6.

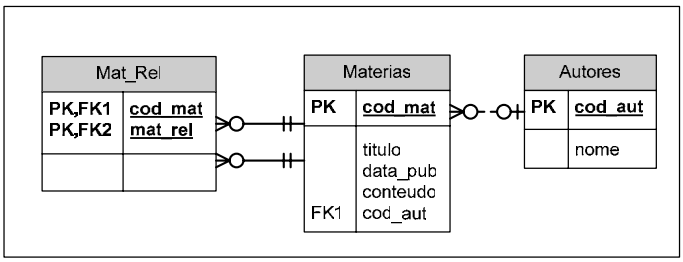

Figura 1.5: Esquema do Banco de Dados Matérias

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="V3.xsl"?>
<xsql:query connection="xml" xmlns:xsql="urn:oracle-xsql">
        SELECT m.titulo as titulo,  m.data_pub as data_publicação,  m.conteudo as conteudo,
                ( SELECT a.nome
                  FROM Autores a
                  WHERE m.cod_aut = a.cod_aut) as autor,
                  CURSOR ( SELECT m2.titulo, m2.data_pub
                                FROM Materias m2, Mat_Rel mr
                                WHERE mr.mat_rel = m2.cod_mat
                                        AND mr.cod_mat = m.cod_mat ) as mat_rel
        FROM Materias m
        WHERE m.titulo = {@titulo}
</xsql:query>
```

Figura 1.6: Página XSQL V3.xsql

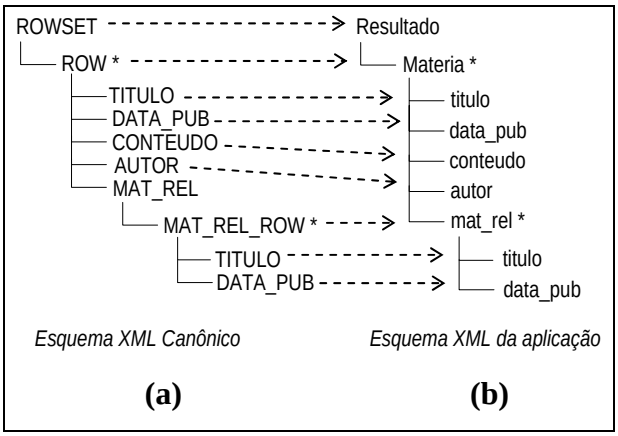

Figura 1.7: Esquemas XML para V3.xsql

Uma página XSQL é mais versátil quando seus resultados podem mudar baseado nos valores de um ou mais parâmetros passados com a requisição [33]. Observe a expressão {@titulo} na cláusula WHERE da consulta SQL:1999 da página XSQL

apresentada na Figura 1.6. Essa expressão denota um parâmetro de nome titulo. A cada requisição feita a essa página, a expressão {@titulo} será substituída pelo valor do parâmetro titulo passado na requisição.

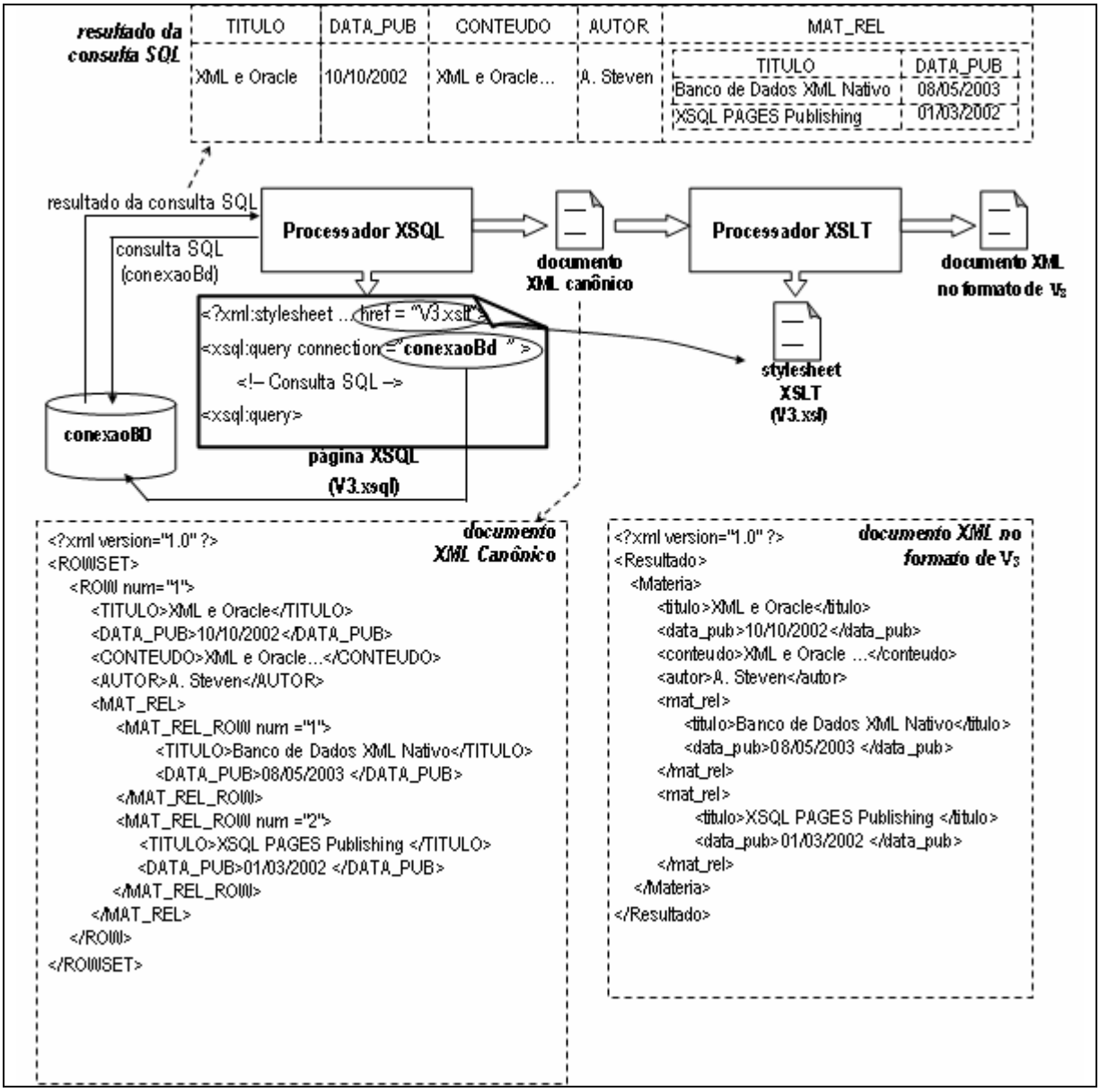

Figura 1.8: Publicação de dados XML baseado em consultas SQL usando XSQL *Pages*

1.2.3 *XPeranto*

O *XPeranto* (XML *Publishing of Entities, Relashionships, And Typed Objects*) [7][12][44][45] é um sistema que funciona como *middleware* entre aplicações e SGBD's relacionais, permitindo a essas aplicações manipularem os dados dos SGBD's

como visões XML, a partir de consultas *XQuery*. Consultas *XQuery* submetidas ao *XPeranto*, são traduzidas em consultas SQL sobre o banco de dados.

Seção

id	descrição
01	Banco de Dados
02	Redes de Computadores

Autor

id	nome	foto	biografia
10	Francisco		...
11	João		...

Matéria

secid	título	resumo	data_pub	autid
01	BDs XML nativo	...	05/05/03	10
01	BDOR	...	10/10/03	10

Figura 1.9: Banco de Dados Matérias

Inicialmente, o *XPeranto* disponibiliza uma Visão XML *Default* do banco de dados existente, a qual especifica a estrutura do banco de dados. Nessa visão, **db** é o elemento raiz. Cada tabela no banco de dados é um elemento filho de **db**. Elementos nomeados por **row** são colocados dentro de cada elemento da tabela. Dentro de um elemento **row**, os nomes das colunas das tabelas aparecem como marcadores e o valor das colunas aparece como texto. A Figura 1.10 mostra a Visão XML *Default* gerada a partir do banco de dados mostrado na Figura 1.9.

```
<db>
        <secao>
                <row><id>01</id><descricao>Banco de Dados</descricao></row>
                <row><id>02</id><descricao>Redes de Computadores</descricao></row>
        </secao>
        <materia>
                <row><secid>01</secid><titulo>BDs XML nativo</titulo>
                        <resumo>...</resumo><data_pub>05/05/03</data_pub><autid>10</autid>
                </row>
                <row><secid>01</secid><titulo>BDOR</titulo>
                        <resumo>...</resumo><data_pub>10/10/03</data_pub><autid>10</autid>
                </row>
        </materia>
        <autor> ...  </autor>
</db>
```

Figura 1.10: Visão XML *Default* de Matérias

Essa visão XML *Default* transforma *XPeranto* numa ferramenta XML pura, ou seja, os usuários não necessitam de conhecimentos além das tecnologias XML para trabalhar com o *XPeranto*. Usuários podem então definir suas próprias visões no topo dessa visão *default* usando *XQuery*, ou simplesmente submeter consultas *XQuery* ao sistema. A Figura 1.11 mostra a Visão XML Seções definida sobre a visão XML *Default* do banco de dados.

```
create view secoes as (
<raiz>{
for $secao in view("default")/secao/row
return
    <secao>{
        <nome>{$secao/descrição}</nome>
        for $materia in view("default")/materia/row
        where $secao/id = $materia/secid
        return
            <materia>{
                <titulo>{$materia/titulo}</titulo>,
                <data_pub>{$materia/data_pub}</data_pub>,
                <resumo>{$materia/resumo}</resumo>,
                <conteudo>{$materia/conteúdo}</ conteudo>,
                let $autor in view("default")/autor/row
                where $autor/id=$materia/autid
                return
                    <autor>{
                        <nome>{$autor/nome}</nome>
                        <foto>{$autor/foto}</foto>
                        < biografia>{$autor/biografia}</biografia>
                    }<autor>
            }</materia>
    }</secao>
}</raiz>
```

Figura 1.11: Visão XML Seções

O processamento das consultas *XQuery* submetidas ao *XPeranto* é mostrado na Figura 1.12. Quando uma consulta *XQuery* é submetida ao *framework*, ela é convertida para uma representação interna chamada XML *Query Graph Model* (XQGM) pelo módulo *XQuery Parser*. O módulo *Query Rewrite & View Composition* realiza uma composição entre a consulta XQGM e a Visão XML sobre a qual a consulta é definida, e executa otimizações para eliminar a construção de fragmentos desnecessários. O módulo *Computation Pushdown* trata da tradução da consulta XQGM modificada em uma consulta SQL sobre o banco de dados e, além disso, gera uma estrutura chamada *Tagger Graph*, que captura a estrutura da consulta XQGM. Essa estrutura é utilizada pelo módulo *Tagger Runtime* para construir o documento XML com o resultado final da consulta.

Recentemente, a IBM tornou o *XPeranto* uma ferramenta comercial chamada IBM XML *For Tables* [25], capaz de processar consultas *XQuery* sobre visões XML de banco de dados DB2.

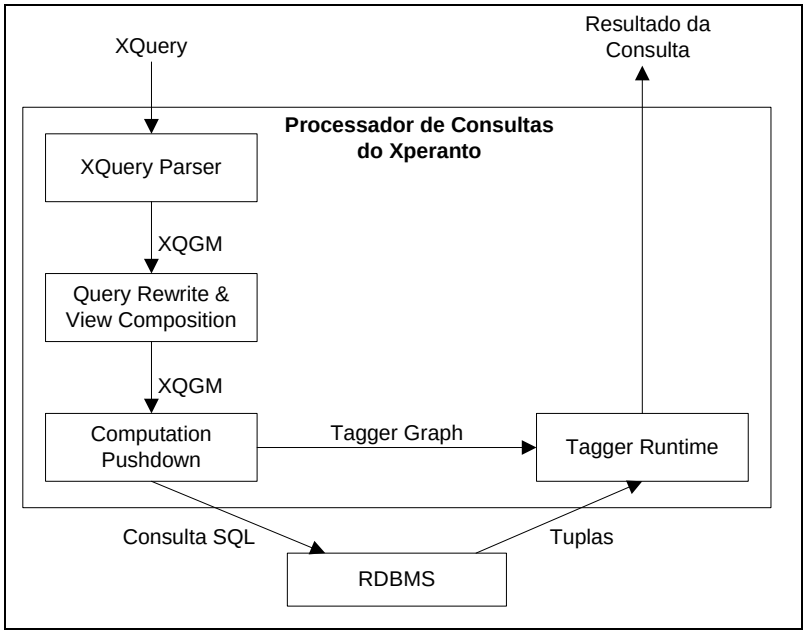

Figura 1.12: Arquitetura do *XPeranto*

1.2.4 *Silkroute*

Silkroute [21][22] é um *framework* para publicação de dados armazenados em um banco de dados relacional no formato XML. Assim como o *XPeranto*, o *Silkroute* serve como *middleware* entre um banco de dados relacional e uma aplicação que acessa esses dados na *Web*. O processo de publicação de dados no formato XML no *Silkroute* é feito de forma similar ao *XPeranto*. No *Silkroute*, esse processo é feito também através dos diversos passos discutidos a seguir.

Primeiro, uma visão XML canônica virtual do banco de dados é gerada automaticamente a partir do esquema do banco de dados. Uma visão XML Canônica no *Silkroute* é muito parecida a uma visão XML *Default* do *XPeranto* e permite ao projetista especificar consultas *XQuery* sobre os dados relacionais.

Em seguida, o projetista especifica uma consulta pública em *XQuery* sobre a visão XML canônica, definindo assim a *visão* XML pública. As aplicações têm acesso somente à visão XML pública e não ao banco de dados.

Para acessar os dados do banco de dados, um usuário/aplicação submete uma consulta *XQuery* sobre a visão XML pública, extraindo somente os dados de interesse da aplicação. O processamento das consultas *XQuery* submetidas ao *Silkroute* é mostrado na Figura 1.13. O módulo *View Forest Composer* do sistema realiza uma composição da consulta da aplicação com a consulta que define a visão XML pública, resultando em uma nova consulta. Essa nova consulta é submetida ao módulo *Planner*, o qual a traduz em uma ou mais consultas SQL equivalentes. Em geral, essa tradução pode produzir diversos conjuntos de consultas SQL que produzem o mesmo resultado da consulta *XQuery*. Esses conjuntos de consultas SQL são chamados de plano de execução. Com o objetivo de selecionar um bom plano de execução de consultas, o módulo *Planner* recebe também como entrada algumas estimativas de custo de consultas produzidas pelo SGBD. Além da tradução, o módulo *Planner* gera um XML Template para construção do documento XML de resposta.

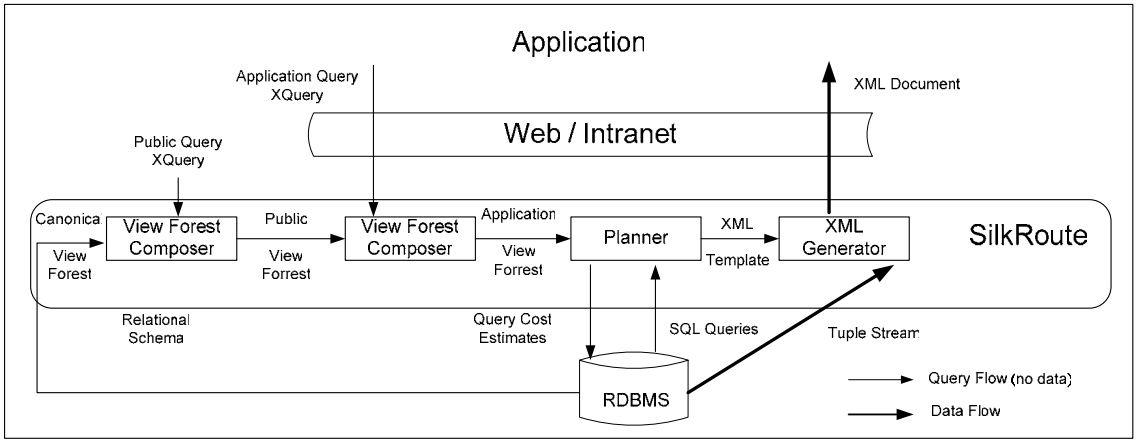

Figura 1.13: Arquitetura do *SilkRoute*

O conjunto de consultas SQL obtido é submetido ao SGBD, que realiza todo o processamento das consultas. As tuplas resultantes das consultas são retornadas para o Silkroute e são manipuladas no módulo *XML Generator*. A partir dessas tuplas e do XML Template, o módulo *XML Generator* produz um documento XML com o resultado das consultas, o qual é retornado para a aplicação.

Internamente, o Silkroute utiliza uma abstração chamada *View Forest* para representação das expressões *XQuery*. A instância do banco de dados relacional (visão canônica), a visão pública e as consultas submetidas ao *framework* são representadas por uma View Forest, que define um mapeamento de um banco de dados relacional para

um documento XML e separa a estrutura do documento XML de saída do processamento que produz o conteúdo do documento. A estrutura é representada por uma ou mais árvores cujos nós são rotulados com os nomes dos elementos XML, nomes de atributos ou tipos atômicos. O processamento é representado por consultas SQL que são relacionadas aos nós. Nenhum valor XML é construído na View Forest, somente o resultado final da consulta SQL. Como exemplo, considere a figura abaixo que mostra o fragmento da View Forest da visão XML canônica que corresponde à tabela Clothing de um banco de dados qualquer:

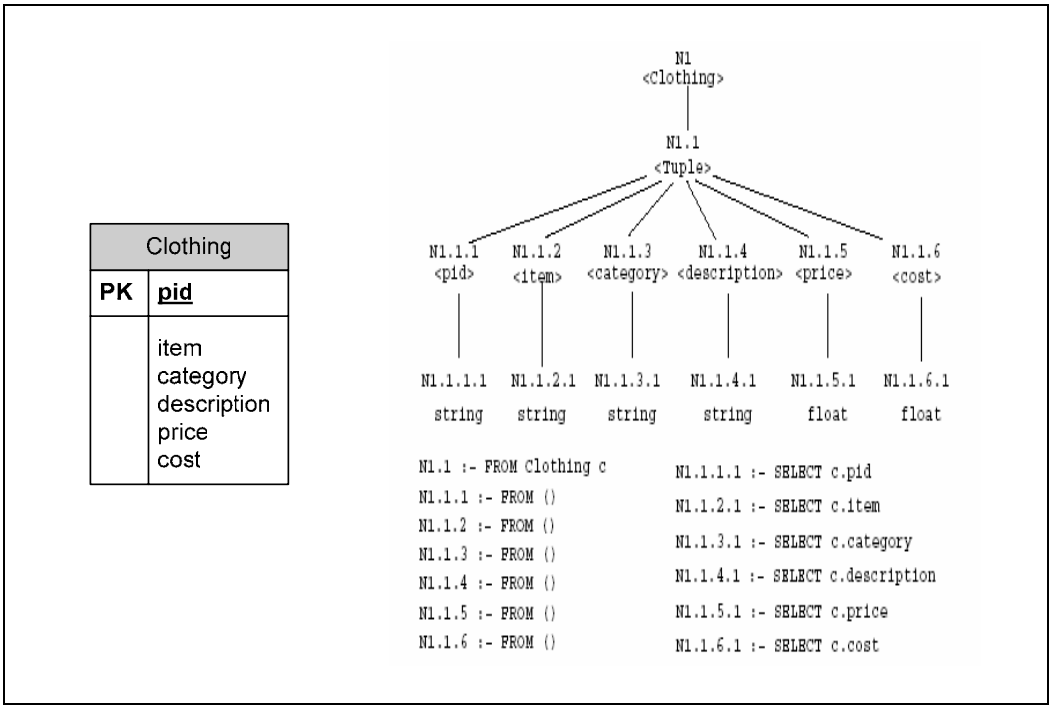

Figura 1.14: Fragmento da *View Forest* Canônica para a tabela *Clothing*

1.2.5 Análise Comparativa

O SQL *Server* 2000, o *XPeranto* e o *Silkroute*, apresentam as seguintes limitações: (i) aplicados somente a bancos de dados relacionais; (ii) existe uma forte dependência entre a visão XML Canônica/Esquema Anotado e o banco de dados, de modo que uma mudança no banco de dados requer a redefinição das Visões XML; (iii) no *XPeranto* e no *Silkroute*, a consulta *XQuery*, que mapeia os elementos da Visão XML Canônica em elementos da visão XML, geralmente é complexa, o que requer conhecimentos avançados de *XQuery* do projetista; (iv) no *XPeranto* e SQL *Server*

2000, não há garantia de processamento eficiente das consultas [24][25]; e (v) no SQL *Server* 2000, a capacidade de consulta é bastante limitada, pois *XPath* suporta apenas seleção e projeção de elementos, mas não reestruturação, como em *XQuery*.

Apesar de criar documentos XML a partir de dados objeto-relacionais, o XSQL *Pages* não é um *framework* para publicação de dados objeto-relacionais através de visões XML, pois consultas sobre as páginas XSQL não são permitidas. Ainda que exista uma forma de parametrizar páginas XSQL, esse mecanismo não é suficiente para permitir a definição de consultas arbitrárias sobre tais páginas.

1.3 *Framework* Proposto

Neste trabalho, apresentamos o XML *Publisher*, um *framework* para publicação de dados objeto-relacionais através de visões XML. No XML *Publisher* os dados são descritos em três níveis de Esquemas como ilustrado na Figura 1.15. Para publicar uma visão XML, o projetista deve definir uma visão de objeto [68], denominada Visão de Objeto *Default* (VOD), de modo que os objetos da VOD possuem a mesma estrutura dos elementos da visão XML. Visões de objeto provêem uma técnica poderosa para determinar visões lógicas, ricamente estruturadas, sobre banco de dados já existente. No *Oracle* 9i, uma visão de objeto é definida através de uma consulta SQL:1999, a qual especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados; e, caso atualizações sejam permitidas, devem ser definidos tradutores (*instead of triggers*) que especificam como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados.

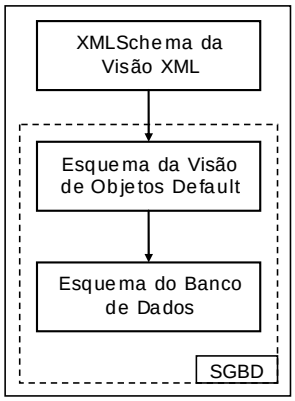

Figura 1.15: Três níveis de esquema

No *framework* proposto, consultas e atualizações definidas sobre o esquema da visão XML são traduzidas em consultas e atualizações sobre o esquema da visão de objeto, e que, por sua vez, são traduzidas pelo mecanismo de visão do SGBD em consultas e atualizações definidas sobre o esquema do banco. A Figura 1.16 mostra os principais componentes do XML *Publisher*. O Módulo de Processamento de Consultas (MPC) é responsável pelo processamento de consultas *XQuery* definidas sobre a Visão XML e o Módulo de Processamento de Atualização (MPA) pelo processamento de atualizações definidas sobre a Visão XML. O XML *Publisher* é disponibilizado como uma aplicação *Web*, de forma que seus serviços podem ser acessados por um cliente através de requisições *HTTP GET*.

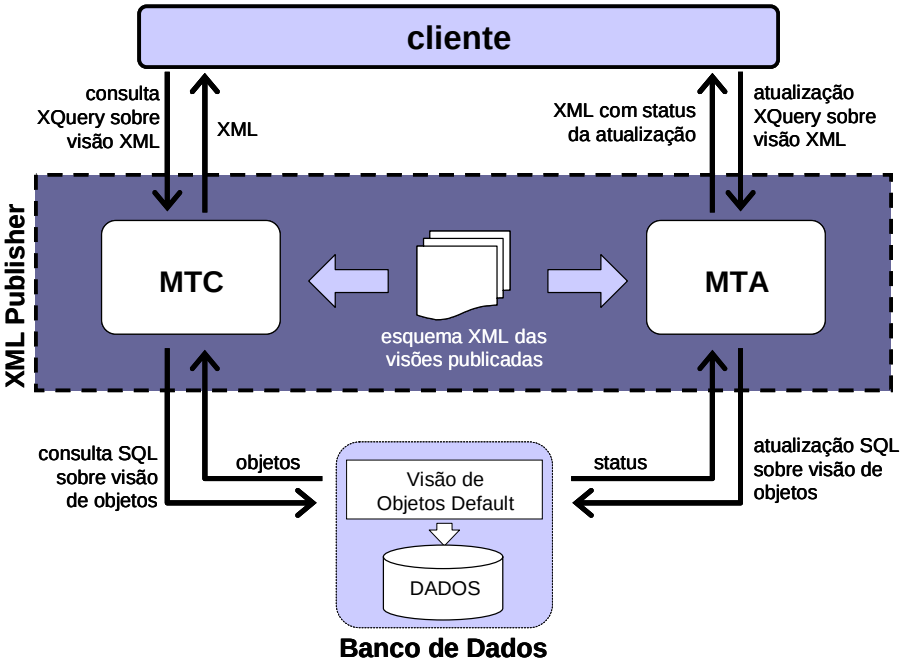

Figura 1.16: Arquitetura do **XML *Publisher***

1.3.1 Processamento de Consultas no XML *Publisher*

Consultas sobre as visões XML publicadas no XML Publisher são definidas através da linguagem *XQuery* [67]. Como apresentado na Figura 1.16, cada consulta *XQuery* submetida ao XML *Publisher* é processada pelo Módulo de Processamento de consulta (MPC) seguindo os seguintes passos. Inicialmente, (i) a consulta *XQuery* é traduzida em uma consulta SQL:1999 sobre a VOD. Dado que a estrutura dos objetos da VOD é compatível com os elementos da visão XML, a tradução é simples e pode ser

feita com base apenas no esquema da visão XML. Em seguida, (ii) a consulta SQL:1999 gerada é processada pelo SGBD. Dessa forma, toda a complexidade do processamento dos dados é delegada ao SGBD. Finalmente, (iii) o resultado da consulta SQL:1999 é transformado, baseado na consulta *XQuery* e no esquema da visão XML, no fragmento XML requisitado pela consulta.

1.3.2 Processamento de Atualizações no XML *Publisher*

Atualizações sobre as visões XML publicadas no XML *Publisher* são definidas através de uma extensão da linguagem *XQuery* [51]. Como apresentado na Figura 1.16, uma atualização submetida ao XML *Publisher* é processada pelo Módulo de Processamento de Atualização (MPA) seguindo os seguintes passos: (i) a atualização *XQuery* é traduzida em uma atualização SQL:1999 definida sobre a VOD correspondente; (2) as atualizações na visão de objeto *default* são por sua vez traduzidas pelos *instead of triggers*, em atualizações definidas sobre o esquema do banco de dados. Em [36] é proposto uma ferramenta para automatizar o processo de geração dos tradutores de atualizações de visões de objeto.

1.4 Contribuições e Organização da Dissertação

Neste trabalho desenvolvemos o módulo de processamento de consultas do XML *Publisher*, o qual denominamos de *XQueryTranslator*. O *XQueryTranslator* transforma uma consulta *XQuery* entrada numa consulta SQL:1999 aplicada à respectiva VOD. Em seguida, o resultado da consulta SQL:1999 é transformado no resultado XML adequado.

Outra contribuição do trabalho consiste no desenvolvimento de um ambiente para facilitar o processo de definição de visões XML no XML *Publisher*. O processo de definição compreende os seguintes passos:

(i) O usuário define, através de uma interface gráfica, o esquema da visão XML;

(ii) A ferramenta DSG (*Default object view Gerator*) gera automaticamente o esquema da visão de objeto *default*;

(iii) O usuário deve então carregar o esquema do banco e, através de uma GUI, definir as assertivas de correspondência do esquema VOD com o esquema do banco de dados [30][40]. Com base no conjunto de assertivas de correspondência da visão, a ferramenta VBA (*View-By-Assertion*) gera automaticamente a consulta SQL:1999 de acordo o mapeamento [19] definido pelas assertivas. VBA lida com o problema da heterogeneidade semântica [52], e permite que mapeamentos complexos possam ser especificados de forma simples. É importante notar que a ferramenta VBA pode ser utilizada para gerar visões de objetos para outros tipos de aplicações, independentemente do XML *Publisher*;

(iv) Os arquivos de configuração do XML *Publisher* são atualizados, de forma que a visão XML finalmente esta definida.

Os Capítulos a seguir estão organizados da seguinte forma. No Capítulo 2 apresentamos algumas características da tecnologia XML abordadas nesse trabalho. No Capítulo 3 tratamos do modelo objeto-relacional, que é utilizado para representar o esquema das visões de objetos e o esquema do banco de dados, o qual pode ser um esquema relacional ou objeto-relacional. No Capítulo 4, damos uma visão geral do XML *Publisher*. No Capítulo 5 apresentamos a ferramenta VBA, o formalismo das assertivas de correspondências da visão e detalhamos o algoritmo para gerar a definição da visão de objetos. No Capítulo 6 descrevemos os algoritmos para o processamento de consultas *XQuery* sobre visões XML do *framework*. Finalmente, no Capítulo 7, mostramos as conclusões e sugestões para trabalhos futuros.

Capítulo 2
XML: Modelo de Dados, Linguagem de Consulta e Esquema

XML se firmou como padrão para publicação e troca de dados na *web*, principalmente devido à sua flexibilidade para representar dados estruturados e semi-estruturados. Neste Capítulo, apresentamos a linguagem XML, abordando tópicos importantes para este trabalho. Na Seção 2.1, apresentamos uma visão geral sobre XML e a sintaxe básica para a criação de documentos. Na Seção 2.2, fazemos uma apresentação dos principais aspectos da linguagem de definição de esquemas XML *Schema* e sua representação gráfica. E na Seção 2.3, mostramos linguagens de consulta para XML.

2.1 XML: Aspectos Gerais

XML (*eXtensible Markup Language*) [58] é uma linguagem de marcação desenvolvida pelo W3C (*World Wide Web Consortium*) [57] para descrever informações. Assim como HTML [59], XML tem origem na SGML (*Standard Generalized Markup Language*), que é um padrão internacional para definição de formatos de representação de texto em meio eletrônico. Entretanto, ao contrário de HTML, que foi projetada para descrever a apresentação dos dados, XML foi projetada para descrever o conteúdo dos dados. A seguir, destacamos alguns aspectos que tornam a linguagem XML mais poderosa que HTML:

- **Linguagem extensível.** XML permite que os usuários definam novos marcadores, de acordo com o domínio que está sendo modelado. Os marcadores servem para descrever o conteúdo de um documento;
- **Independência de conteúdo e apresentação**. XML aborda apenas o conteúdo de um documento. A apresentação pode ser tratada por linguagens específicas, tais como: *Cascading Style Sheets* (CSS) e *eXtensible Stylesheet Language* (XSL);
- **Linguagem flexível**. XML permite a apresentação de um mesmo conteúdo em diversos formatos;
- **Validação**. Um documento XML pode ser associado a um esquema que define a estrutura do documento. Assim, aplicações podem validar seus dados de acordo com um esquema.

XML também pode ser vista como uma metalinguagem, pois, por ser extensível, permite a criação de outras linguagens de marcadores. Estas linguagens podem ser definidas para domínios específicos (matemática, química, comércio eletrônico, entre outros). Assim, os marcadores podem capturar mais semântica sobre os dados que estão sendo modelados, o que não ocorre em um documento HTML, pois seus marcadores têm função apenas de formatar o texto para apresentação.

Um documento XML consiste em vários elementos. Um elemento é descrito por um marcador inicial (i.e., <nome_do_elemento>) e um final (i.e., </nome_do_elemento>) que delimitam o conteúdo do elemento. Cada elemento pode conter texto e elementos aninhados (sub-elementos), como pode ter conteúdo misto ou ser vazio. Quando o elemento é vazio, este deve ser representado de acordo com um dos seguintes formatos: <nome do elemento></nome do elemento> ou <nome do elemento/>. Considere, por exemplo, o documento XML mostrado na figura abaixo:

```
<pessoa>
        <nome>Lineu</nome>
        <e-mail>lineu@lia.ufc.br</e-mail>
</pessoa>
```

Figura 2.1: Exemplo de Documento XML

Os marcadores <pessoa> e </pessoa> descrevem a estrutura do elemento pessoa. O elemento pessoa é composto por dois sub-elementos: nome e e-mail. O conteúdo do

elemento nome é o texto Lineu e o conteúdo do elemento e-mail é o texto lineu@lia.ufc.br. Como podemos observar, os marcadores de um documento são balanceados, ou seja, são fechadas na ordem inversa da qual foram abertos.

Um elemento também pode possuir um ou mais atributos. Estes são especificados no marcador inicial do elemento ou, no caso do elemento vazio, no marcador que define o elemento. O valor de um atributo é considerado texto e deve aparecer entre aspas. Por exemplo, no documento XML mostrado abaixo, a propriedade CPF é um atributo do elemento pessoa. O valor do atributo CPF é 70586796802.

```
<pessoa cpf="70586796802">
    <nome>Lineu </nome>
    <e-mail>lineu@lia.ufc.br </e-mail>
</pessoa>
```

Figura 2.2: Documento XML com atributo

Documentos XML podem ser classificados como bem-formados. Para ser bem-formado, um documento deve obedecer às regras sintáticas definidas na especificação da linguagem XML [58], tais como: (1) conter um ou mais elementos; (2) conter um elemento que contenha todos os outros, chamado de elemento raiz e (3) todos os outros elementos devem estar aninhados, sendo delimitados apropriadamente por seus marcadores. Os documentos XML apresentados anteriormente (Figura 2.1 e Figura 2.2) são documentos XML bem-formados. No entanto, o documento XML apresentado a seguir é um exemplo de documento XML mal-formado, pois o primeiro elemento-nome não possui um marcador final adequado:

```
<pessoa>
    <nome>Lineu </nomes>
    <nome>Vânia </nome>
</pessoa>
```

Figura 2.3: Documento XML mal-formado

Documentos XML também podem ser classificados como válidos. Para ser válido um documento precisa ser bem-formado e deve estar de acordo com a gramática que define sua estrutura. Esta gramática é definida através de um esquema XML, que descreve a estrutura de um documento, determinando os elementos que podem participar deste

documento e também os que podem estar associados a esses elementos. A Seção 2.2 mostra como definir esquemas para documentos XML.

A verificação sintática de um documento XML, que identifica se este é ou não bem-formado, é realizada por um *parser*. Os *parsers* de validação, além de detectar se um documento é bem-formado, são capazes de verificar se o mesmo está de acordo com o seu esquema XML associado, ou seja, verifica se o documento é válido ou não.

2.2 Definindo Esquemas para Documentos XML

Um esquema descreve a estrutura lógica de uma fonte de informação, incluindo os tipos dos dados, os relacionamentos e as restrições que envolvem os dados. Para XML, um esquema é usado para descrever os elementos com seu conteúdo, a lista de atributos de um determinado elemento e as restrições sobre os elementos/atributos. Além disso, um esquema é útil para validar o conteúdo de um documento, ou seja, para determinar se um documento está de acordo com a gramática definida pelo esquema. E essa gramática pode ser reutilizada para validar e definir a estrutura de outros documentos.

A funcionalidade de validar um documento XML é muito importante no contexto de aplicações *web* que trocam informações entre diversas fontes, pois com a definição de esquema é possível verificar se um determinado documento está de acordo com a estrutura esperada, facilitando o processamento de dados pelas aplicações.

A primeira linguagem proposta para definição de esquema XML foi a DTD (*Document Type Definition*) [58]. Entretanto, DTD possui algumas limitações como pode ser observado em [42]. Para sobrepor essas limitações, outras linguagens foram propostas, entre as quais destacamos: XML *Schema* [63] e RDF [61]. Estas linguagens são mais ricas em semântica e oferecem recursos adicionais para definição de esquemas. Uma análise comparativa dessas e de outras linguagens de esquema XML é apresentada em [26].

Os esquemas das visões XML publicadas pelo XML *Publisher* são definidos usando um subconjunto da linguagem XML *Schema*. Sendo assim, na seção a seguir discutimos as características dessa linguagem, destacando apenas as características implementadas no XML *Publisher*.

2.2.1 XML *Schema*

A linguagem XML *Schema* introduz novos construtores que a tornam mais expressiva que DTD e permite não só especificar a sintaxe de um documento XML, mas também: especificar o tipo de dados do conteúdo de cada elemento; herdar sintaxe de outros esquemas; criar tipos de dados simples e complexos; especificar o número mínimo e máximo de vezes que um elemento pode ocorrer; entre outros. Com isso, XML *Schema* pode ser utilizada em uma maior variedade de aplicações, além de ter a vantagem de ser escrita em sintaxe XML, não sendo necessário, portanto, aprender uma nova sintaxe. Essa facilidade nos permite utilizar qualquer ferramenta que trabalha com documentos XML para trabalhar com XML *Schema* [16].

```
<bib>
 <livro autoresref1="A">
  <ano> 1999 </ano>
  <isbn> 3826561422 </isbn>
  <titulo> Transaction Management in Multidatabase Systems
</titulo>
  <editora> Shaker-Verlag </editora>
 </livro>
 <artigo autoresref="A1 A2">
  <ano> 2000 </ano>
  <cdu> 681.31:061.68 </cdu>
  <titulo> Temporal Serialization Graph Testing </titulo>
  <local_ publicacao> XV SBBD </local_ publicacao>
 </artigo>
 <autor id_autor1="A" instituicaoref="I">
  <nome> Ângelo Brayner </nome>
  <e-mail> brayner@unifor.br </e-mail>
 </autor>
 <autor id_autor="A2" instituicaoref="I1 I2">
  <nome> José Maria Monteiro </nome>
  <e-mail> zemaria@lia.ufc.br </e-mail>
 </autor>
<instituicao id_instituicao="I">
  <nome> Unifor </nome>
  <endereco>
    <cidade> Fortaleza </cidade>
    <estado> Ceará </estado>
    <pai> Brasil </pais>
  </endereco>
 </instituicao>
 <instituicao id_instituicao="I2">
  <nome> UFC </nome>
  <telefone> 288-9845 </telefone>
  <endereco>
    <cidade> Fortaleza </cidade>
    <estado> Ceará </estado>
    <pais> Brasil </pais>
  </endereco>
 </instituicao
</bib>
```

Figura 2.4: bib.xml

Na Figura 2.5, apresentamos um exemplo de XML *Schema*. Como pode ser observado, trata-se de um documento XML e o elemento raiz desse documento *Schema*. Em XML *Schema*, todas as declarações de elementos, atributos e tipos devem ser inseridas entre os marcadores do elemento *Schema*.

A seguir, descrevemos os principais componentes necessários para a criação de XML *Schema*s. Para exemplificar o uso destes componentes, utilizamos o esquema da Figura 2.5, que representa a estrutura do documento bib.xml apresentado na Figura 2.4:

```
1 <Schema>
2  <element name=”bib”>
3   <complexType>
4    <sequence>
5      <element name=”livro” type=”Tlivro” minOccurs=”0” maxOccurs=”unbounded”/>
6    <element name=”artigo” type=”Tartigo” minOccurs=”0” maxOccurs=”unbounded”/>
7     <element name=”autor” type=”Tautor” minOccurs=”0” maxOccurs=”unbounded”/>
8     <element name=”instituicao” type=”Tinstituicao” minOccurs=”0” maxOccurs=”unbounded”/>
9    </sequence>
10   </complexType>
11 </element>

12  <complexType name=” Tpublicacao”>
13  <sequence>
14 <element name=”ano” type=”String”/>
15 <sequence>
16    <element name=”isbn” type=”Integer”/>
17    <element name=”cdu” type=”Integer”/>
18   </sequence>
19   <element name=”titulo” type=”String”/>
20  </sequence>
21        <attribute name=”autoresref” type=”IDREFS” use=”required“/>
22 </complexType>

23 <complexType name=”Tartigo”>
24  <complexContent>
25   <extension base=”Tpublicacao” >
26    <sequence>
27     <element name=”local-publicacao” type=”String”/>
28    </sequence>
29   </extension>
30  </complexContent>
31 </complexType>
32 <complexType name=”Tlivro”>
33  <complexContent>
34   <extension base=”Tpublicacao”>
35    <sequence>
36     <element name=”editora” type=”String”/>
37    </sequence>
38   </extension>
39  </complexContent>
40 </complexType>

41 <complexType name=” Tautor”>
42  <sequence>
43   <element name=”nome” type=”String”/>
44   <element name=”e-mail” type=”String”/>
45  </sequence>
46  <attribute name=”id_autor” type=”ID” use=”required“/>
47        <attribute name="instituicaoref" type=”IDREF” use=”optional“/>
48 </complexType>

49 <complexType name=” Tinstituicao”>
50  <sequence>
51   <element name=”nome” type=”String”/>
52          <element name=”telefone” minOccurs=”0” maxOccurs=”“/>
53   <element name=”endereco” type=”Tendereco”/>
54  </sequence>
55        <attribute name=”id_instituicacao” type=”ID” use=”required“/>
56 </complexType>

57 <complexType name=”Tendereco”>
58  <sequence>
59   <element name=”cidade” type=”String”/>
60   <element name=”estado” type=”String”/>
61   <element name=”pais” type=”String”/>
62  </sequence>
63 </complextype>

64 </Schema>
```

Figura 2.5: XML *Schema* de bib.xml

2.2.1.1 Declaração de Elemento

Elementos são declarados utilizando o marcador <element>, onde os atributos name e type definem seu nome e seu tipo, respectivamente. O tipo de um elemento pode também ser especificado através de uma declaração de tipo anônima, como veremos na Seção 2.2.1.3. Na declaração de um elemento também é possível definir a sua cardinalidade, através dos atributos minOccurs e maxOccurs. Por exemplo, a declaração <element

name="autor" type="Tautor" minOccurs="0" maxOccurs="unbounded"/> (linha 7) especifica que o elemento autor é do tipo Tautor e a restrição de ocorrência é opcional e multi-ocorrência.

2.2.1.2 Declaração de Atributo

Atributos são declarados utilizando o marcador <attribute>. Assim como na declaração de elementos, os atributos name e type determinam o nome e o tipo do atributo declarado, respectivamente. Entretanto, não é possível definir valores de mínimo e máximo de ocorrência para atributos, pois cada atributo pode ocorrer no máximo uma vez em um determinado elemento. No entanto, o atributo use do marcador <attribute> é usado para definir se a ocorrência de um atributo é opcional ou obrigatória. O atributo use também é usado para definir outros tipos de restrições como de valor fixo e de valor padrão. Por exemplo, a declaração <attribute name="id_instituicao" type="ID" use="required"/> (linha 55) especifica que este atributo é do tipo ID, monovalorado e obrigatório.

2.2.1.3 Definição de Tipos

Os tipos restringem o conteúdo permitido para um elemento ou atributo. Existem duas espécies de tipos em XML *Schema*: tipos simples e tipos complexos. Um tipo simples restringe o conteúdo de um atributo, ou o conteúdo textual de um elemento. Um tipo complexo restringe o conteúdo permitido para elementos, em termos dos seus atributos ou sub-elementos.

Na linguagem XML *Schema*, para definir um tipo complexo, usamos o marcador <complexType>. Os tipos simples não precisam ser definidos, pois a especificação da XML *Schema* contempla um conjunto expressivo de tipos primitivos [64]. No entanto, é possível criar novos tipos de dados simples que correspondem a refinamentos de tipos primitivos. Para esse propósito, usamos o marcador <simpleType>. Os elementos especificados nos XML *Schemas* das visões publicadas através do XML *Publisher* podem ser apenas de tipos complexos ou tipos primitivos.

O atributo name dos marcadores <simpleType> e <complexType> define o nome do tipo que está sendo definido. Quando esses marcadores não possuem o atributo name, dizemos que se trata de uma declaração de tipo anônima. Neste caso, a definição do tipo vem aninhada na declaração de um elemento. Por exemplo, no esquema da Figura 2.5, a declaração do elemento bib (linhas 2 a 11) contém uma definição de tipo anônima, que especifica que o elemento bib contém os elementos artigo, livro, autor e instituicao.

Nas declarações de tipos complexos, além das restrições de ocorrência declaradas para elementos individuais, a XML *Schema* provê restrições a serem aplicadas em grupos de elementos. Essas restrições também podem ser chamadas de delimitadores de grupos. Existem três tipos de delimitador de grupo:

- ***sequence***: Indica que os elementos do grupo devem aparecer no documento na mesma ordem em que foram declarados no esquema. Por exemplo, na Figura 2.5 (linhas 49 a 56), a declaração do tipo complexo **T**instituicao especifica que os elementos nome, telefone e endereco, contidos neste tipo, devem aparecer nesta ordem no documento. Os tipos complexos definidos nos XML *Schema*s das visões XML publicadas no XML *Publisher*, usam apenas esse delimitador de grupo;
- ***all***: Estabelece que todos os elementos do grupo podem aparecer uma ou nenhuma vez, e que eles podem aparecer em qualquer ordem;
- ***choice****:* Indica que somente um dos elementos declarados no grupo pode aparecer no documento.

Um tipo complexo também pode ser definido a partir da extensão de um outro tipo complexo. No esquema da Figura 2.5 a definição do tipo **T**artigo (linhas 23 a 31) especifica que o tipo complexo **T**artigo é definido a partir do tipo complexo **T**publicacao (linha 25). Assim, um elemento do tipo complexo **T**artigo contém os elementos do tipo complexo **T**publicacao, além do elemento local-publicacao. Neste caso, podemos dizer que **T**artigo é um subtipo de **T**publicacao.

2.2.2 Representação Gráfica para XML *Schema*

Nesta seção apresentamos a notação utilizada neste trabalho para representar graficamente a estrutura definida por um XML *Schema*. Esta notação utiliza uma representação estruturada em "árvore de diretórios", onde cada nó da árvore representa um elemento ou atributo declarado no respectivo XML *Schema*.

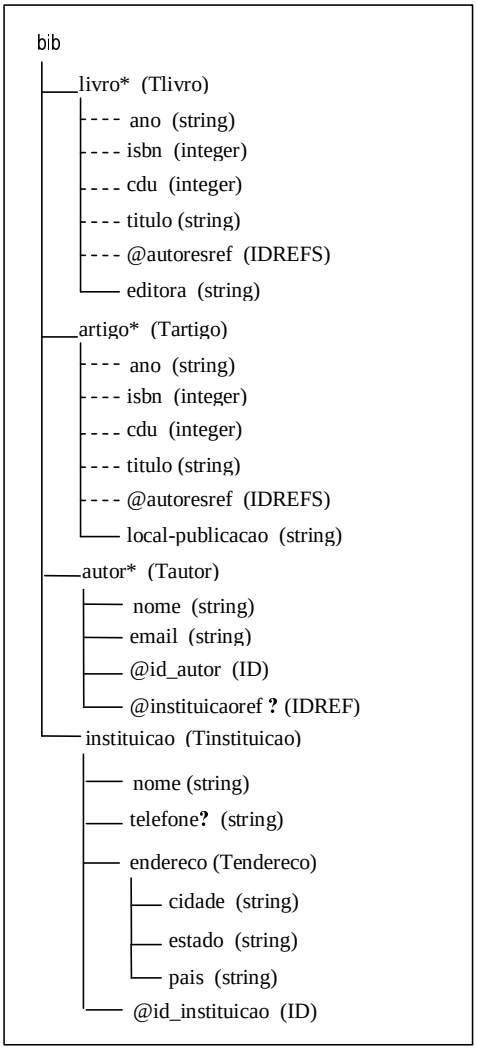

Figura 2.6: Representação gráfica para bib.xml

Para cada declaração de elemento ou atributo é gerado um nó na árvore. Se o elemento for de um tipo complexo, será gerado um nó-filho na árvore para cada item da sua estrutura. Cada nó deve ser rotulado com o nome do elemento ou atributo, seu tipo (se não

for anônimo) e sua restrição de ocorrência. Para atributos, seu nome é precedido do símbolo "@". Se o elemento ou atributo é monocorrência e obrigatório, a restrição de ocorrência é vazia. Se o elemento for monocorrência e opcional, a restrição de ocorrência é "?". Se o elemento for multiocorrência e opcional, a restrição de ocorrência é "*". Finalmente, se o elemento for multiocorrência e obrigatório, a restrição de ocorrência é "+".

A árvore mostrada na Figura 2.6 é a representação gráfica do XML *Schema* apresentado na Figura 2.5. O primeiro nó da árvore é bib, e representa o elemento raiz bib declarado na linha 2 do XML *Schema*. Como podemos perceber no XML *Schema*, a declaração de tipo desse elemento é anônima. Sendo assim, sua representação gráfica não acompanha um tipo como nos demais nós da árvore. Os nós filhos de bib são livro, autor e instituicao, pois no XML *Schema* esses elementos estão declarados como o conteúdo do tipo complexo de bib (linhas 3 a 10 da Figura 2.5). Cada um desses nós filhos são rotulados com o nome do respectivo elemento, o símbolo "*", que denota a restrição de ocorrência zero ou muitos, e o nome do respectivo tipo.

2.3 Linguagens de Consulta para XML

Dados XML diferem de dados de banco de dados relacionais ou orientados a objetos em diversos aspectos, dentre estes podemos destacar o fato de que dados XML não seguem uma estrutura rígida como os esquemas relacionais e orientados a objetos. Assim, linguagens de consultas convencionais como SQL e OQL não são adequadas para definir consultas em documentos XML, portanto, são necessárias linguagens específicas para consultar esses documentos. Muitas linguagens de consulta para XML têm sido propostas, como, XML-QL [23], XSL [68], XQL [15], *XPath* [62] e *XQuery* [67].

As visões XML publicadas no XML *Publisher* podem ser consultadas usando um subconjunto de *XQuery* que será apresentado no Capitulo 6. *XQuery* usa *XPath* para endereçar partes do documento XML consultado. Sendo assim, descrevemos a seguir os principais aspectos das linguagens *XPath* e *XQuery* propostas pelo W3C.

2.3.1 *XPath*

A linguagem *XPath* é uma recomendação do W3C para acessar partes de um documento XML. Esta linguagem usa uma notação de caminhos para navegar através da estrutura hierárquica de um documento XML e foi projetada para ser inserida em outras linguagens chamadas de *host languages*, tais como XSLT e *XQuery*.

XPath modela um documento XML como uma árvore de nós. Este modelo em árvore é apenas conceitual e possui diferentes tipos de nós, tais como, nós-documento, nós-elemento, nós-texto, nós-atributo, entre outros. Para cada tipo de nó, há um modo de determinar seu valor literal (*string-value*). Para alguns tipos de nós o valor é parte do nó; para outros, esse valor é computado a partir do valor de nós descendentes. Por exemplo, o valor de um nó-elemento consiste da concatenação do valor de todos os nós-texto descendentes do nó-elemento na ordem do documento.

O nó-documento encapsula todo o documento XML; ele é o ponto de partida na árvore que descreve o documento XML. Os nós-elemento encapsulam os elementos XML, e podem ter nós-elemento, nós-texto e outros como seus filhos. Nós-elementos também podem conter nós-atributos.

O bloco de construção básico da *XPath* é a expressão. A linguagem provê diversos tipos de expressão que podem ser construídos a partir de palavras-chave, símbolos e operandos. Em geral, os operandos de uma expressão são outras expressões. *XPath* é uma linguagem funcional que permite que vários tipos de expressão sejam aninhados.

A expressão que localiza um nó em uma árvore e retorna uma seqüência de nós distintos na ordem do documento é chamada de expressão de caminho. Tal expressão consiste em um ou mais passos. Cada passo representa um movimento ao longo do documento, em uma determinada direção, e retorna uma lista de nós que servem como um ponto de partida para o próximo passo.

A seguir, exemplificamos algumas expressões de caminho utilizadas para consultar o documento XML bib.xml apresentado na Figura 2.4.

Exemplo 2.1

A expressão de caminho **document**(“bib.xml”)/**bib**/**artigo**/**titulo** consiste em quatro passos: o primeiro passo seleciona o documento XML bib.xml; o segundo seleciona o elemento raiz de bib.xml; o terceiro seleciona os elementos artigo, filhos do elemento raiz e o quarto passo seleciona os elementos titulo dos elementos artigo recuperados no passo anterior. Assim, essa expressão retorna todos os elementos titulo contidos nos elementos artigo, contidos no elemento bib do documento bib.xml. A figura abaixo mostra o XML retornado por essa consulta:

```
<titulo> Temporal Serialization Graph Testing </titulo>
```

Para simplificar os próximos exemplos, utilizamos a variável **$bib** para substituir a expressão **document**(“bib.xml”)/**bib**:

Exemplo 2.2

A expressão de caminho **$bib**/**autor**/**nome**/**text()** retorna o nome de todos os elementos autor contidos em $**bib** (Ângelo Brayner e José Maria Monteiro).

Exemplo 2.3

A expressão de caminho **$bib**/**livro [ano = “1999”]** retorna todos os elementos livro em $**bib** que possuem o elemento ano igual a “1999”. A figura abaixo mostra o XML retornado por essa consulta:

```
<livro autoresref=”A”>
  <ano> 1999 </ano>
  <isbn> 3826561422 </isbn>
  <titulo> Transaction Management in Multidatabase Systems </titulo>
  <editora> Shaker-Verlag </editora>
 </livro>
```

2.3.2 *XQuery*

XQuery [67] é a linguagem de consulta padrão que está sendo elaborada pelo W3C e utiliza o mesmo modelo de dados apresentado na seção anterior. Trata-se de uma linguagem funcional na qual consultas são representadas como expressões. Numa consulta *XQuery* expressões têm a forma de variáveis, expressões de caminhos, chamada a funções, expressões condicionais, construtores de elementos, expressões FLWR etc., e podem ser aninhadas e combinadas usando operadores lógicos e aritméticos.

Para navegar através de caminhos na estrutura de um documento XML, *XQuery* usa expressões de "caminho" cuja sintaxe é uma abreviação da sintaxe de *XPath*. O resultado do cálculo de uma expressão de caminho sobre um documento XML retorna uma floresta ordenada consistindo em nós que satisfazem a expressão e seus descendentes [9]. A ordem do resultado é ditada pela ordem dos elementos dentro do documento XML consultado.

```
<?xml version="1.0"?>
<livros>
        <livro isbn="073571020">
                <titulo>Inside XML</titulo>
                <editora>New Riders</editora>
                <autores>
                        <autor>Steven Holzner</autor>
                </autores>
        </livro
        <livro isbn="8535206485">
                <titulo>Gerenciando Dados na Web</titulo>
                <editora>Campus</editora>
                <autores>
                        <autor>Serge Abiteboul</autor>
                        <autor>Peter Buneman</autor>
                        <autor>Dan Suciu</autor>
                </autores>
        </livro>
</livros>
```

Figura 2.7: Documento liv.xml

Uma característica poderosa de *XQuery* é a presença de expressões FLWOR (*for-let-where-orderby-return*). Uma expressão FLWOR é usada sempre que é necessário interagir sobre elementos de uma coleção. As cláusulas *for-let* fazem variáveis interagirem sobre o resultado de uma expressão ou atribui o valor de expressões arbitrárias a variáveis. A cláusula

where é opcional e permite especificar restrições sobre essas variáveis. A cláusula *order by* é opcional e ordena o resultado da consulta. A cláusula *return* gera a saída da expressão FLWOR, que pode ser um nodo, uma floresta ordenada de nodos, ou um valor primitivo. A cláusula *return* contém uma expressão geralmente composta por construtores de elementos, variáveis e sub-expressões aninhadas. Na Figura 2.7, apresentamos alguns exemplos de consultas *XQuery* para o documento XML liv.xml apresentado:

Exemplo 2.4

A consulta abaixo retorna os títulos de todos os livros. Essa consulta é composta por uma expressão *for-return*. A cláusula **for** especifica uma iteração sobre os elementos livro que são filhos do elemento raiz livros no documento XML liv.xml (função de entrada **document**). Em cada iteração, o respectivo elemento livro será armazenado na variável **$i**. A cláusula **return** especifica que a cada iteração do **for**, o conteúdo textual (função **text**()) do elemento titulo de **$i** deverá ser mostrado:

Consulta	Resultado
for $i in document("liv.xml")/livros/livro **return** **$i**/titulo/**text**()	Inside XML Gerenciando Dados na Web

Figura 2.8: Consulta *XQuery* do Exemplo 2.4

Exemplo 2.5

A consulta abaixo retorna todos os autores do livro "Inside XML". Essa consulta é composta por uma expressão *for-return-return*. A cláusula **For** especifica uma iteração sobre os elementos livro que são filhos do elemento raiz livros no documento XML liv.xml. Em cada iteração, o respectivo elemento livro será armazenado na variável $i. A presença da cláusula **where** especifica que a cláusula **return** será executada apenas quando o titulo do livro for igual a "Inside XML". A cláusula **return** especifica que a cada iteração válida do **For**, o sub-elemento autores de $i deverá ser mostrado.

Consulta	Resultado
for $i in **document**("liv.xml")/livros/livro **where** $i/titulo = "Inside XML" **return** **$i**/autores	<autores> <autor>Steven Holzner</autor> </autores> <autores> <autor>Serge Abiteboul</autor> <autor>Peter Buneman</autor> <autor>Dan Suciu</autor> </autores>

Figura 2.9: Consulta *XQuery* do Exemplo 2.5

Exemplo 2.6

Para cada livro, criar um elemento <livro_autores> contendo o isbn do livro e um sub-elemento <nome_autor> para cada autor desse livro. Esse exemplo enfatiza o uso de construtores de elementos (<livro_autores> e <nome_autor>) e de expressões *for-where-return* aninhadas:

Consulta	Resultado
for $i** in **document**("liv.xml")/livros/livro **return** <livro_autores>{ **$i/@isbn/text(), **for $j in $i**/autores **return** **<**nome_autor**>{** **$j**/autor/text() **}**</nome_autor> }</livro_autores>	<livro_autores> 0735710201 <nome_autor> Steven Holzner </nome_autor> </livro_autores> <livro_autores> 8535206485 <nome_autor> Serge Abiteboul </nome_autor> <nome_autor> Peter Buneman </nome_autor> <nome_autor> Dan Suciu </nome_autor> </livro_autores>

Figura 2.10: Consulta *XQuery* do Exemplo 2.6

Capítulo 3
Modelo Objeto-Relacional

Neste capítulo apresentamos o modelo objeto-relacional baseado no modelo do *Oracle* 9i [35]. Este modelo é resultado da extensão do modelo relacional para suportar os conceitos de orientação a objetos. Na seção 3.1, descrevemos os principais elementos de modelagem que compõem o modelo Objeto-Relacional do *Oracle* 9i. Na seção 3.2, apresentamos uma notação gráfica para representar esquemas objeto-relacionais. Na seção 3.3, abordamos as características orientadas a objetos da linguagem SQL:1999 utilizadas nesta dissertação. Encerramos este Capítulo discutindo sobre o mecanismo de Visões de Objetos implementado no *Oracle* 9i.

3.1 Esquema Objeto-Relacional

No SGBD *Oracle* 9i [37], um esquema objeto-relacional é uma tripla **S**=(**T**, **R**, **I**), onde **T** é um conjunto de definições de tipos, **R** é um conjunto de tabelas e **I** é um conjunto de restrições de integridade. Uma das principais características do modelo objeto-relacional é a capacidade de estender o sistema de tipos do SGBD, ou seja, novos tipos podem ser definidos pelos usuários. Esses tipos são chamados de tipos abstratos de dados (TAD). Um tipo serve como molde para a criação de objetos (instâncias do tipo) e serve para definir a estrutura de dados (atributos) e as operações (métodos) que são comuns às instâncias do tipo. Para criar um TAD em um banco de dados objeto-relacional, usamos o comando CREATE TYPE.

Os atributos de um TAD podem ser classificados como *monovalorados* ou *multivalorados*. O valor de um atributo monovalorado é um objeto (atômico ou estruturado) ou uma referência para um objeto. Já o valor de um atributo multivalorado é uma coleção de objetos ou coleção de referências para objetos. Um atributo multivalorado pode ser representado como uma *Nested Table* (coleção ilimitada de objetos) ou *VArray* (coleção limitada de objetos). Considere, por exemplo, o esquema objeto relacional Pedidos apresentado na Figura 3.1:

- O atributo nome, do tipo $T_{cliente}$, é monovalorado atômico;
- O atributo enderecoEntrega, do tipo T_{pedido}, é monovalorado estruturado cujo valor é uma instância do tipo $T_{endereco}$;
- O atributo cliente_ref, do tipo T_{pedido}, é monovalorado de referência cujo valor é uma referência para uma instância do tipo $T_{cliente}$;

O atributo listaItens, no tipo $T_{pedido,}$ é multivalorado (Nested Table) de tipo T_{lista_item} cujo valor é uma coleção de instâncias de T_{item}.

```
CREATE TYPE Tcliente AS OBJECT
( codigo INTEGER, nome VARCHAR2(50) );

CREATE TYPE Titem AS OBJECT
( codigo INTEGER, produto VARCHAR2(30),
    quantidade INTEGER );

CREATE TYPE Tendereco AS OBJECT
( rua VARCHAR2(30), cidade VARCHAR2(15),
  estado VARCHAR2(2), cep VARCHAR2(8) );

CREATE TYPE Tlista_item AS TABLE OF Titem ;

CREATE TYPE Tpedido AS OBJECT
( codigo INTEGER, data DATE, dataEntrega DATE,
   enderecoEntrega Tendereco, cliente_ref REF Tcliente,
   listaItens Tlista_item);

CREATE TABLE Clientes OF Tcliente
( codigo PRIMARY KEY );

CREATE TABLE Pedidos OF Tpedido
( codigo PRIMARY KEY, cliente_ref REFERENCES Clientes)
NESTED TABLE listaItens STORE AS listaItens_nt_tab;
```

Figura 3.1: Esquema objeto-relacional Pedidos

O modelo objeto-relacional do *Oracle* 9i suporta dois tipos de tabelas: tabelas de tuplas e tabelas de objetos. As tabelas de tuplas são as tabelas do modelo relacional. Uma tabela de objetos possui associado um tipo e os objetos inseridos na tabela possuem um identificador único (OID), permitindo, assim, que os objetos possam ser referenciados por outros objetos através de atributos de referência. A tabela clientes no esquema Pedidos da Figura 3.1 é uma tabela de objetos, cujos objetos são instâncias do tipo $T_{cliente}$. Estes objetos são referenciados pelas instâncias de T_{pedido} através do atributo cliente_ref. O escopo de um atributo de referência (tabela ou visão que contém os objetos referenciados) é especificado com as restrições de escopo [35]. Por exemplo, na tabela Pedidos, a restrição de integridade referencial cliente_ref REFERENCES Clientes especifica que o escopo do atributo de referência cliente_ref é a tabela Clientes.

Antes de inserir um objeto em uma tabela de objetos devemos criá-lo usando um construtor de objetos. Um construtor de objetos é uma função que cria um objeto de um determinado tipo. Considere, por exemplo, o esquema Pedidos apresentado na Figura 3.1. Um exemplo de construtor de objeto para o tipo $T_{cliente}$ é $T_{cliente}$ (1 , “Lineu”). Um objeto do tipo T_{lista_item} pode ser criado com o construtor T_{lista_item} (T_{item} (20, ‘Coca-Cola’, 10), T_{item} (21, ‘Skol’, 10)). Observe que T_{lista_item} é uma Nested Table que contém duas instâncias do tipo T_{item}.

3.2 Notação Gráfica

Neste trabalho usamos uma notação gráfica baseada em [69] para representar um esquema objeto-relacional. A notação adotada estende o diagrama de classes da UML, através do uso de *estereótipos* de forma a permitir modelar, além de tipo de objetos (chamado de classe na UML), também tabelas de objetos, tabelas relacionais e visões de objetos. A Figura 3.2 mostra a representação gráfica do esquema objeto-relacional Pedidos da Figura 3.1. Na notação gráfica, os retângulos com o estereótipo <<Tipo de Objeto>> representam um TAD; os retângulos com o estereótipo <<Tabela de Objeto>> representam tabelas de objetos; os retângulos com o estereótipo <<Tabela Relacional>> representam tabelas de tuplas; e os retângulos com o estereótipo <<Visão de Objeto>> representam uma

tabela virtual de objetos. Os atributos cujos tipos são pré-definidos são escritos dentro dos retângulos. Os demais atributos cujos tipos são definidos pelo usuário são representados por setas rotuladas com o nome do atributo. Usamos setas simples para atributos monovalorados e setas duplas para atributos multivalorados. Para os de referência, adicionamos o estereótipo <<REF>> à seta.

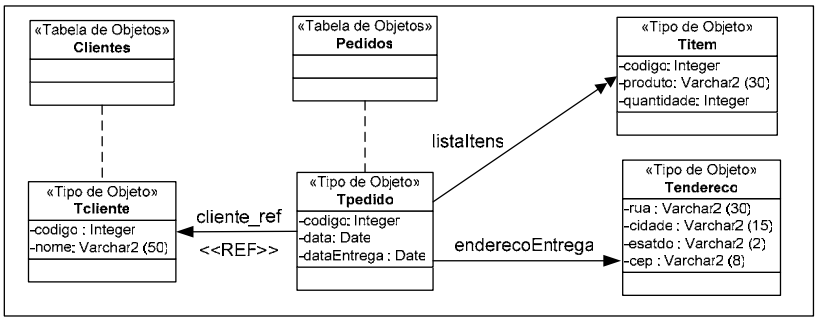

Figura 3.2: Representação gráfica do esquema Pedidos

3.3 Consulta dos Dados Objeto-Relacionais

SQL (*Structured Query Language*) é a linguagem padrão para a definição e manipulação de dados armazenados em banco de dados relacionais. Ela pode ser considerada uma das maiores razões para o sucesso dos bancos de dados relacionais [17]. Em sua mais recente versão (SQL:1999), adicionou novas características para permitir o suporte a dados orientados a objetos. Alguns SGBD's, como DB2 e *Oracle*, já dão suporte às características de orientação a objetos da SQL:1999.

Algumas das funcionalidades que caracterizam SQL:1999 como uma linguagem de definição de dados (DDL), orientados a objetos, foram apresentadas na seção anterior (TAD, tabela de objetos, OID, atributo multivalorado, atributo de referência). A seguir, discutiremos alguns operadores da SQL:1999 como uma linguagem de manipulação de dados (DML) orientados a objeto:

– ***TABLE***

Esse operador permite desaninhar uma coleção de objetos (*Nested Table* ou *Varray*) dentro de uma cláusula FROM. Dessa forma, conseguimos acessar individualmente cada objeto de uma coleção. Considere, por exemplo, o esquema Pedidos apresentado na Figura 3.2. Para retornar o nome de todos os produtos do pedido de número 1, devemos criar a seguinte consulta SQL:1999:

```
SELECT i.produto
FROM Pedidos p,
        TABLE (p.listaItens) i
WHERE p.codigo = 1
```

– ***REF***

Esse operador pode ser usado em consultas para obter uma referência para um objeto. Considere, por exemplo, o esquema Pedidos apresentado na Figura 3.2. A consulta a seguir retorna o OID de todos os clientes que fizeram algum pedido:

```
SELECT REF(p)
FROM Pedidos p
```

– ***CAST-MULTISET***

Esse operador oferece uma forma de sintetizar coleções de objetos (*Nested Table* ou *Varray*) em consultas. Considere, por exemplo, o esquema relacional Pedidos_rel (Figura 3.6.) e o tipo coleção T_{ped} definido pelo comando "CREATE TYPE T_{ped} AS TABLE OF Integer". Na consulta abaixo, para cada cliente será retornado seu nome e uma coleção de objetos contendo os códigos dos seus pedidos:

```
SELECT c.cnome, CAST ( MULTISET (
                        SELECT p.pcodigo
                        FROM Pedidos_rel p
                        WHERE c.ccodigo = p.pcliente
                    ) AS Tped)
FROM Clientes_rel c
```

– ***CURSOR***

O operador **CAST-MULTISET** é fortemente tipado, ou seja, exige que um tipo coleção seja definido para ele. O operador **CURSOR** possui a mesma funcionalidade

do operador **MULTISET**, no entanto, não precisa que um tipo seja definido para ele. A consulta abaixo corresponde à consulta anterior, reescrita com o operador **CURSOR**:

```
SELECT c.cnome,CURSOR( SELECT p.pcodigo
                       FROM Pedidos_rel p
                       WHERE c.ccodigo = p.pcliente)
FROM Clientes_rel c
```

3.4 Visões de Objetos

Um banco de dados geralmente é compartilhado por uma variedade de usuários e deve atender a diferentes requisitos desses usuários. Dessa forma, muitas vezes, os bancos de dados tornam-se grandes e complexos, dificultando assim a sua manipulação. Para facilitar a manipulação dos dados, devem ser fornecidas interfaces que apresentem somente as informações relevantes para cada grupo de usuários. Isto é feito através da definição de visões, que representam modelos simplificados do banco de dados, através dos quais os usuários podem expressar consultas e atualizações. As visões, além de protegerem o acesso aos dados, ajudam a alcançar um certo grau de independência lógica, uma vez que é possível alterar o esquema do banco de dados sem alterar uma visão.

Em banco de dados relacionais, as visões relacionais são tabelas virtuais que têm seus dados derivados das tabelas relacionais nas quais estão baseadas, chamadas tabelas base da visão. Assim como uma visão relacional é uma tabela virtual, uma visão de objetos é uma tabela virtual de objetos. Além das características das visões tradicionais, as visões de objetos possuem as seguintes características:

- **Habilidade para navegar usando referências**

  Assim como nas tabelas de objetos, cada objeto construído a partir de uma visão de objetos possui um OID. Dessa forma, os objetos de uma visão de objetos podem ser referenciados por outros objetos. Utilizar referências para representar conexões entre objetos torna as consultas baseadas em caminhos de referência mais legíveis e compactas do que as consultas feitas com junções.

- **Facilidade para evolução de esquemas**

  As visões de objetos fornecem a flexibilidade de ver, de mais de uma maneira, o mesmo dado relacional ou orientado a objetos. Sendo assim, cada aplicação pode ter seu próprio esquema orientado a objetos sem ter quer mudar o esquema do banco de dados.
- **Consistência e compartilhamento de dados relacionais com novas aplicações baseadas em objetos**
- **Integração de aplicações OO com bancos de dados**

  Visões de objeto provêem uma técnica poderosa para se impor visões lógicas, ricamente estruturadas, sobre banco de dados já existentes. Dessa forma possibilitam a coexistência de aplicações relacionais e orientadas a objetos [55][56], o que torna fácil a introdução de aplicações orientadas a objetos para dados relacionais já existentes sem provocar uma mudança drástica de um paradigma para o outro.

No *Oracle* 9i, para criar uma visão de objetos, primeiro deve-se criar os tipos da visão. A Figura 3.3 contém a definição dos tipos de uma visão Pedidos_v, cuja representação gráfica é mostrada na figura 3.4. Depois, então, define-se a visão através de uma consulta SQL:1999 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados. A Figura 3.5 contém a consulta SQL:1999 que define a visão Pedido_v baseada no esquema do banco de dados relacional Pedido_rel, apresentado na Figura 3.6. A visão Pedidos_v contém um conjunto de objetos do tipo T_{Pedido}, os quais são sintetizados a partir das tuplas das tabelas base.

Consultas SQL:1999, definidas sobre uma visão de objetos, são traduzidas em consultas sobre o banco de dados. Os resultados dessas consultas são, então, integrados para gerar o resultado da consulta inicial [20]. No caso em que atualizações sobre uma visão de objetos são permitidas, devemos definir tradutores (*instead of triggers*) que especifiquem como atualizações sobre a visão de objetos são traduzidas em atualizações especificadas sobre o banco de dados. O código do *instead of trigger* é executado quando inserções, atualizações ou remoções são solicitadas na visão ou em algum atributo multivalorado da visão. A Figura 3.7 mostra, a título de ilustração, o *instead of trigger*

"Adiciona_Cliente", o qual traduz a adição de um pedido na visão Clientes_v (Figura 3.4) em uma adição na tabela relacional Clientes_rel (Figura 3.6).

```
CREATE TYPE Tcliente_v AS OBJECT
( codigo INTEGER, nome VARCHAR2(50) );

CREATE TYPE Titem_v AS OBJECT
( codigo INTEGER, produto VARCHAR2(30),
    quantidade INTEGER );

CREATE TYPE Tendereco_v AS OBJECT
( rua VARCHAR2(30), cidade VARCHAR2(15),
  estado VARCHAR2(2), cep VARCHAR2(8) );

CREATE TYPE Tlista_item_v AS TABLE OF Titem_v ;

CREATE TYPE Tpedido_v AS OBJECT
( codigo INTEGER, data DATE, dataEntrega DATE,
   enderecoEntrega  Tendereco_v,  cliente_ref  REF
  Tcliente_v,
   listaItens Tlista_item_v);
```

Figura 3.3: Definição dos tipos da visão Pedidos_v

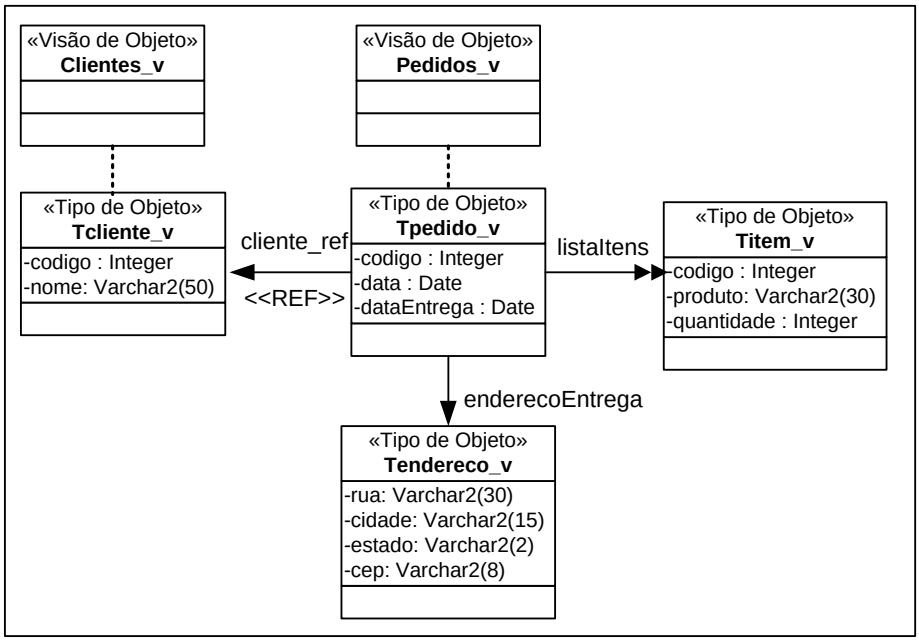

Figura 3.4: Esquema da visão de objetos Pedidos_v

```
CREATE VIEW Pedidos_v OF Tpedido
WITH OBJECT IDENTIFIER (codigo) AS
   SELECT

Tpedido(
          tpedidos_rel•pcodigo,
          tpedidos_rel•pdata,
          tpedidos_rel•pdataEntrega,
          Tendereco ( tpedidos_rel•prua, tpedidos_rel•pcidade, tpedidos_rel•pestado, tpedidos_rel•pcep ),
          (SELECT REF (tcliente)
           FROM Clientes_v tcliente, Clientes_rel tclientes_rel
           WHERE tpedidos_rel•pcliente = tclientes_rel•ccodigo
                    tclientes_rel•ccodigo = tcliente•codigo),
          CAST(MULTISET( SELECT

                                   Titem(
                                        titens_rel•icodigo,
                                        (SELECT tprodutos_rel•pnome
                                         FROM Produtos_rel tprodutos_rel
                                         WHERE titens_rel•iproduto= tprodutos_rel•pcodigo),
                                        titens_rel•iquantidade)
                                  FROM Itens_rel  titens_rel
          ) AS Tlista_item )

FROM Pedidos_rel  tpedidos_rel
```

Figura 3.5: Definição da visão Pedidos_v

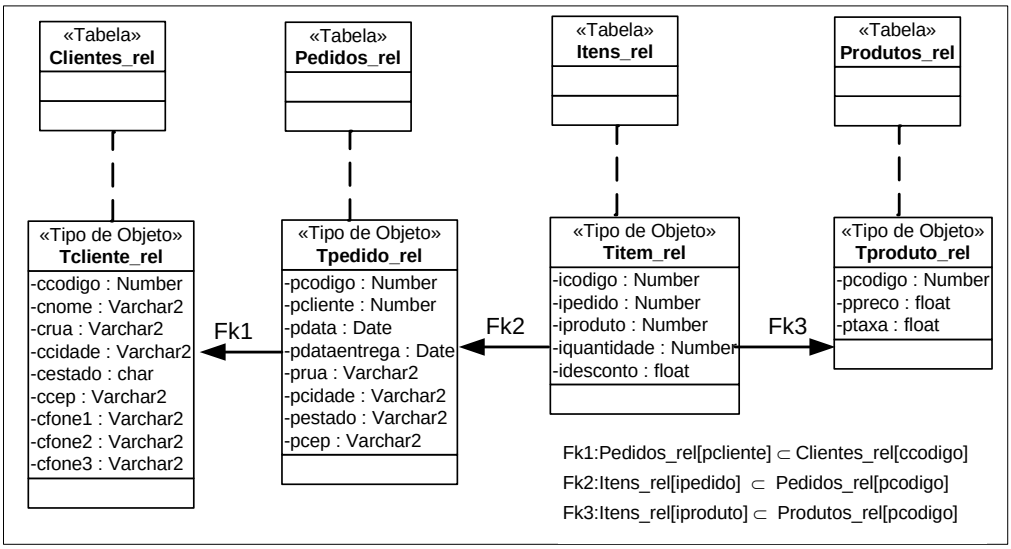

Figura 3.6: Esquema do banco de dados Pedidos_rel

```
CREATE TRIGGER Adiciona_Cliente
INSTEAD OF TRIGGER INSERT ON Clientes_v
FOR EACH ROW
   BEGIN
      INSERT INTO CLIENTES_REL
      (ccodigo, cnome)
       VALUES (:new.codigo, :new.nome)
  END
```

Figura 3.7: ***Instead of trigger*** Adiciona_Cliente

Capítulo 4
XML *Publisher*

Neste Capítulo, apresentamos o XML *Publisher*, um *framework* para publicação (consulta e atualização) de dados objeto-relacionais através de visões XML. Este Capítulo é organizado da seguinte forma. A Seção 4.1 contém uma visão geral do *framework*; na Seção 4.2, descrevemos os passos para publicar visões XML no XML *Publisher*; na Seção 4.3, mostramos como consultas *XQuery* são processadas no *framework*; na Seção 4.4, apresentamos como acessar o XML *Publisher* através de requisições http; na Seção 4.5, descrevemos o algoritmo para gerar o esquema da VOD a partir do esquema da visão XML.

4.1 Introdução

Para publicar uma visão XML no XML *Publisher*, o projetista deve definir uma visão de objetos denominada Visão de Objeto *Default* (VOD), cujos objetos têm a mesma estrutura dos elementos da visão XML. No *framework* proposto, consultas e atualizações sobre a visão XML, são traduzidas em consultas e atualizações sobre o esquema da visão de objeto. Essa tradução é simples, dado que os elementos da visão XML têm estrutura isomórfica à estrutura dos objetos da VOD. As visões de objeto são definidas sobre o esquema do banco de dados, ficando assim a cargo destas resolver o problema do mapeamento de tuplas de tabelas (relacionais ou objeto-relacionais) em objetos de tipo complexo cuja estrutura é compatível com a estrutura do tipo dos elementos da visão XML.

Como foi discutido no Capítulo 3, através do mecanismo de visões de objeto conseguimos facilmente definir visões lógicas, ricamente estruturadas, sobre banco de dados já existente. No *Oracle* 9i, uma visão de objeto é definida através de uma

consulta SQL:1999 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados, e no caso em que atualizações são permitidas devem ser definidos tradutores (*instead of triggers*), os quais especificam como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados.

Nesta dissertação, apresentamos apenas o processo de geração de visões de objeto *default* e o processamento de consultas *XQuery* no *framework*. O processo de geração dos tradutores de atualizações de visões de objetos e o processamento de atualizações no XML *Publisher* são discutidos em [12][36].

4.2 Ambiente para Definição de Visões XML no XML *Publisher*

O XML *Publisher* oferece um ambiente para auxiliar o projetista no processo de definição de visões XML. O processo de definição compreende os seguintes passos: (i) especificação do esquema da visão XML; (ii) geração da respectiva VOD; e (iii) configuração da visão XML no XML *Publisher*.

Passo 1: Especificação do Esquema da visão XML

O processo de definição de visões XML no XML *Publisher* começa com o projetista especificando o esquema da visão XML. Consultas e atualizações *XQuery* sobre a visão XML serão definidas de acordo com esse esquema.

O esquema de cada visão XML deve ser especificado usando a linguagem XML *Schema* [63]. Assim como uma visão de objetos contém um conjunto de objetos do mesmo tipo, uma visão XML no XML *Publisher* contém um conjunto de elementos do mesmo tipo. Sendo assim, o tipo do elemento raiz de cada visão XML definida no XML *Publisher* deve conter apenas uma declaração de elemento multiocorrência, como apresentado na figura a seguir. Observe que o elemento raiz **V** tem um ou vários elementos filhos **E** do tipo T_E. Chamamos o elemento **E** de elemento primário da visão XML. Como veremos a seguir, o nome do elemento raiz determina o nome da VOD e a estrutura de T_E determina a estrutura da VOD.

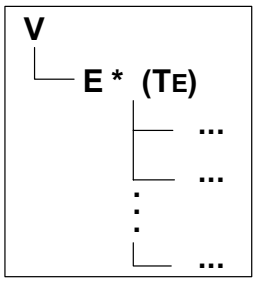

Figura 4.1: Tipo do elemento raiz

No Ambiente para definição de visões do XML *Publisher*, o esquema da visão XML pode ser especificado com o auxílio de uma ferramenta gráfica. Inicialmente, a ferramenta solicita o nome do elemento raiz, o nome do elemento primário e o caminho para o arquivo que irá armazenar o XML *Schema* (Figura 4.2 (a)). Em seguida, o projetista deve especificar o tipo para o elemento primário e seus elementos (Figura 4.2 (b)). Para cada elemento são pedidos seu tipo e sua cardinalidade. No caso de elementos de tipo complexo, então um novo tipo deve ser criado (nome e elementos).

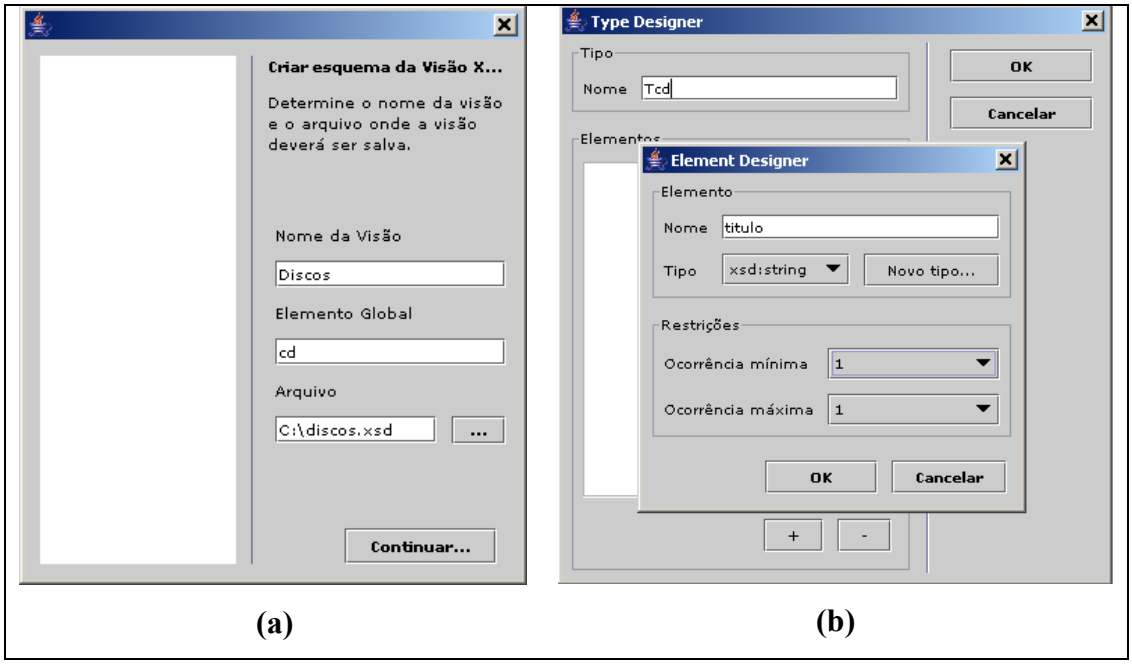

Figura 4.2: Especificando o esquema da visão XML

Passo 2: Geração da VOD

Após especificar o esquema da visão XML, o projetista segue definindo a visão de objetos *default*. Como foi visto no Capítulo 3, para definir uma visão de objetos em um banco de dados objeto-relacional devemos criar os tipos da visão (esquema da

visão) e uma consulta SQL:1999 que especifica como construir os objetos da visão a partir dos objetos do banco de dados. Então, o processo de geração da VOD consiste de dois passos, como discutido a seguir:

Passo 2.1: Geração do Esquema da VOD

Os tipos da VOD têm a mesma estrutura dos tipos complexos da visão XML. Considere, por exemplo, o esquema do banco de dados $\mathbf{B}_1$ apresentado na Figura 4.3. Suponha que o projetista deseja definir a visão XML Discos sobre $\mathbf{B}_1$, cuja estrutura é apresentada na Figura 4.4 (a). O esquema da VOD de Discos é apresentado na Figura 4.4 (b). Observe que elementos monocorrência e multiocorrência da visão XML são diretamente mapeados em atributos monovalorados e multivalorados, respectivamente, da VOD. Por exemplo, o elemento multiocorrência musica é mapeado no atributo multivalorado musica, cujo tipo é composto por objetos do tipo $\mathbf{T}_{musica}$. Os elementos de musica são mapeados em atributos de $\mathbf{T}_{musica}$.

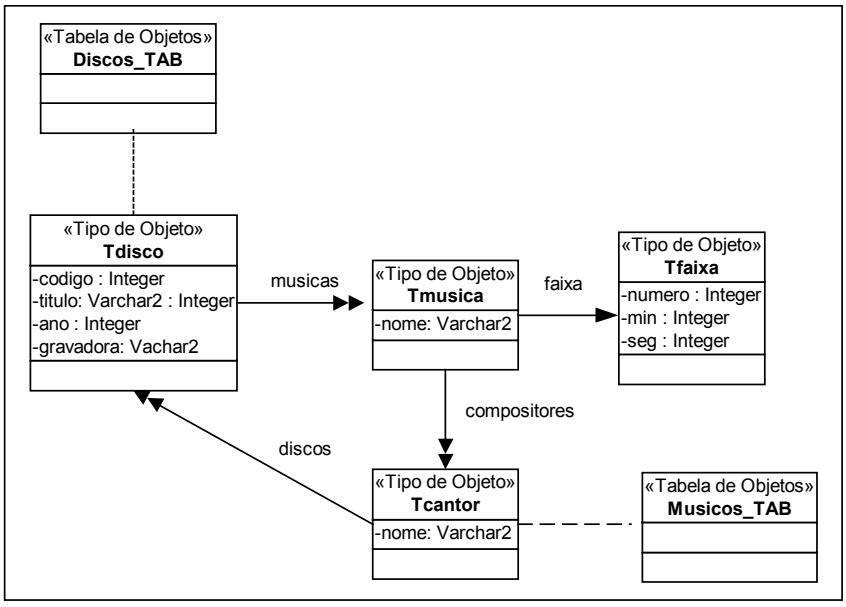

Figura 4.3: Esquema do banco de dados B_1

No Ambiente para definição de visões no XML *Publisher*, a ferramenta DSG (***D**efault Object View **S**chema **G**enerator*) gera automaticamente o esquema da VOD, com base no XML *Schema* da visão XML, desde que este XML *Schema* obedeça às restrições que vêm sendo discutidas neste trabalho. O algoritmo implementado por essa ferramenta é bastante simples, e será apresentado na Seção 4.5.

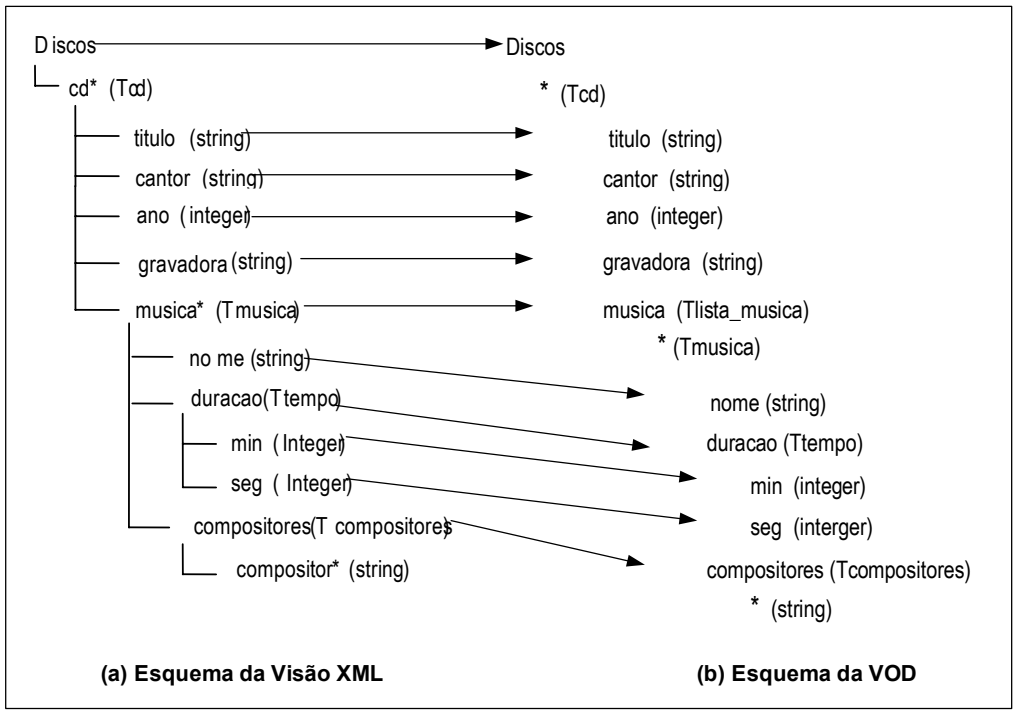

Figura 4.4: Esquemas da Visão Discos

Passo 2.2: Geração da consulta SQL:1999 da VOD

```
CREATE VIEW Discos OF Tcd
WITH OBJECT IDENTIFIER (titulo) AS
   SELECT tdisco.titulo,
          (SELECT tcantor.nome FROM Musicas_TAB
           WHERE REF(tdisco) IN (SELECT * FROM TABLE (tcantor.discos))),
          tdisco.ano, tdisco.gravadora,
          CAST( MULTISET (
                SELECT tmusica.nome,
                       Ttempo (tmusica.faixa.min, tmusica.faixa.seg),
                       CAST(MULTISET (
                            SELECT tcantor.nome
                            FROM TABLE(tmusica.compositores) tcantor2
                       ) AS Tcompositores )
                FROM TABLE(tdisco.musicas) tmusica
          ) AS Tmusica )
   FROM Discos tdisco
```

Figura 4.5: Visão de Objetos *Default* discos

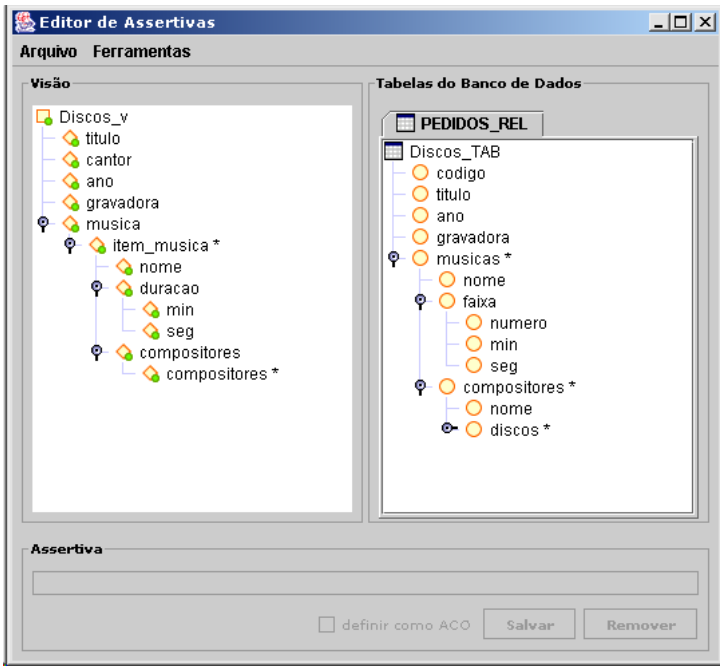

Figura 4.6: GUI para definição de correspondências no VBA

A Figura 4.5 apresenta a consulta SQL:1999 da VOD de Discos definida sobre o banco de dados $\mathbf{B}_1$. Observe que a consulta SQL:1999 definida para essa VOD é bastante complexa, o que requer conhecimentos avançados de SQL:1999. Apesar de complexa, essa consulta pode ser gerada de forma automática no Ambiente de definição de visões do XML *Publisher*, através da ferramenta VBA (*View-By-Assertion*). Com VBA, o projetista inicialmente carrega o esquema da VOD e o esquema do banco de dados e, através de uma GUI, define as assertivas de correspondência entre esses esquemas (Figura 4.6). Com base no conjunto de assertivas de correspondência da visão, VBA gera automaticamente a consulta SQL:1999 de acordo com o mapeamento definido pelas assertivas. As assertivas de correspondências e o algoritmo para gerar a consulta SQL:1999 da VOD serão apresentados no Capítulo 5.

Passo 3: Configurações do XML *Publisher*

Após especificar o esquema da visão XML e criar a consulta SQL:1999 da respectiva VOD, devemos atualizar o arquivo de configuração do XML *Publisher* para efetivar a definição da visão. As configurações do XML *Publisher* são armazenadas em um documento XML de nome WXSConfig.xml, cujo esquema é apresentado na Figura 4.7. Esse documento contém informações que serão utilizadas pelo XML *Publisher* para

o processamento de consultas e atualizações *XQuery*. O documento de configuração armazena as seguintes informações:

- Os nomes das classes Java [14] que implementam o tradutor de consultas *XQuery* e o tradutor de atualização *XQuery*;
- O nome de cada visão XML publicada no XML *Publisher*, as ações disponibilizadas (atualização/consulta), o endereço do XML *Schema* da visão e informações para acessar a respectiva VOD no SGBD.

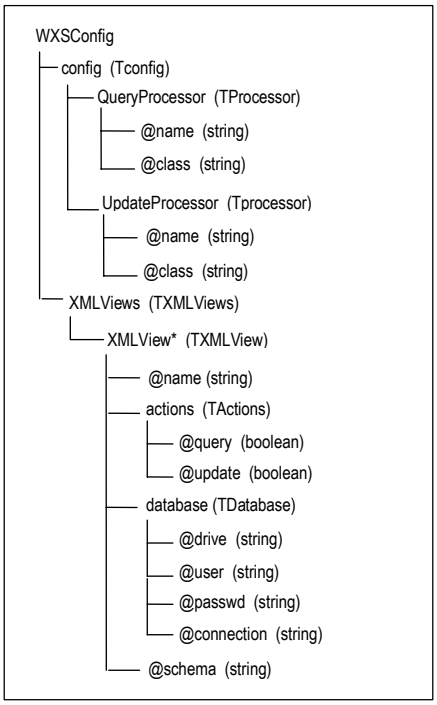

Figura 4.7: Estrutura do arquivo de configuração do XML *Publisher*

Exemplo 4.1

Considere que o documento XML da Figura 4.8 é o arquivo de configuração de alguma instância do XML *Publisher*. A classe Java definida para processar as requisições *ExecuteQuery* é br.ufc.lia.database *XQueryTranslator* (linha 05). A classe Java definida para processar as requisições *UpdateQuery* é br.ufc.lia.database.*XQueryUpdateTranslator* (linha 08).

Esse servidor publica três visões XML: **Jornais** (linha 11), **Pedidos_v** (linha 19) e **Clientes_v** (linha 27). O esquema da visão **Jornais** está armazenado no arquivo file:///M:/XPublisher/xml/config/schema/jornais.xsd (linha 12). A VOD dessa visão está em

um servidor *Oracle* e os dados para acesso desse servidor estão nas linhas 13 a 16. Essa visão é somente-leitura, ou seja, não atualizável (linha 25).

```
01      <?xml version=”1.0” encoding=”ISO-8859-1”?>
02      <WXSConfig>
03        <config>
04          <QueryProcessor name="XQuery Translator"
05                    class="br.ufc.lia.database.XQueryTranslator" />
06
07          <UpdateProcessor name="XQuery Update Translator"
08                    class="br.ufc.lia.database.XQueryUpdateTranslator" />
09        </config>
10        <XMLViews>
11          <XMLView name="Jornais"
12                schema="file:///M:/XPublisher/xml/config/schema/jornais.xsd">
13            <database driver="Oracle.jdbc.driver.OracleDriver"
14                    connection ="jdbc:Oracle:thin:@200.19.179.66:1521:zeus"
15                    user="lineu"
16                    passwd="lineu"/>
17            <actions query="yes" update="no"/>
18          </XMLView>
19          <XMLView name="Pedidos_v"
20                schema=" file:///M:/XPublisher/xml/config/schema/pedidos.xsd ">
21            <database driver="Oracle.jdbc.driver.OracleDriver"
22                    connection ="jdbc:Oracle:thin:@200.19.179.66:1521:zeus"
23                    user="lineu"
24                    passwd="lineu"/>
25            <actions query="yes" update="no"/>
26          </XMLView>
27          <XMLView name="Clientes_v"
28                schema=" file:///M:/XPublisher/xml/config/schema/clientes.xsd ">
29            <database driver="Oracle.jdbc.driver.Oracle Driver"
30                    connection ="jdbc:Oracle:thin:@200.19.179.66:1521:zeus"
31                    user="lineu"
32                    passwd="lineu"/>
33            <actions query="yes" update="no"/>
34          </XMLView>
35        </XMLViews>
36      </WXSConfig>
```

Figura 4.8: WXSConfig.xml

4.3 Processamento de Consultas no XML *Publisher*

Nesta seção, apresentamos uma visão geral sobre o processamento de consultas *XQuery* no XML *Publisher*. Para realizar esse processamento, desenvolvemos um *software* chamado *XQueryTranslator*, o qual será apresentado com mais detalhes no Capítulo 6.

Os passos para o processamento de consultas *XQuery* no XML *Publisher* são apresentados na Figura 4.9. Seja **Q** uma consulta *XQuery* aplicada a uma visão XML $\mathbf{X}_1$. Para processar **Q** o XML *Publisher* procede da seguinte forma:

i. O *XQueryTranslator* traduz **Q** em uma consulta SQL:1999 **S** aplicada a respectiva VOD. Dado que a estrutura dos objetos da VOD tem a mesma estrutura dos elementos da visão XML, a tradução é simples e pode ser feita com base apenas no esquema da visão XML. Em seguida, **S** é submetido ao SGBD.

ii. A consulta **S** gerada é processada pelo SGBD com base na definição da visão de objeto. Dessa forma, toda a complexidade do processamento dos dados é delegada ao SGBD. O resultado de **S** é um conjunto de objetos **O**.

iii. Os objetos de **O** são transformados pelo *XQueryTranslator* em um resultado XML com base em **Q** e no esquema de $\mathbf{X}_1$. Esse é o único passo no qual o *XQueryTranslator* realmente executa alguma manipulação de dados. Mas essa manipulação consiste apenas em percorrer a estrutura dos objetos de **O** para adicionar marcadores (elementos).

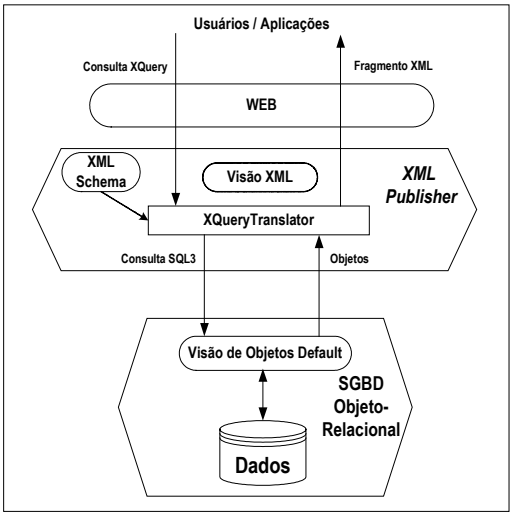

Figura 4.9: Processamento de consultas no XML *Publisher*

Exemplo 4.2

Considere a consulta *XQuery* definida sobre a Visão XML Discos (Figura 4.4). No *XQueryTranslator* essa consulta *XQuery* é traduzida na consulta SQL:1999 abaixo.

Note que essa tradução é simples e direta: as expressões de *for* foram mapeadas em objetos nas cláusulas FROM, a expressão de *where* foi mapeada em um predicado da cláusula WHERE e as expressões de *return* foram mapeadas em projeções da consulta (cláusula SELECT).

```
<resultado>{
        for $i in document("discos.xml")/discos/cd
        where $i/ano = "2003"
        return
            <lancamento>{
                $i/titulo,
                $i/cantor,
                for $j in $i/musica
                return
                        <musica>{
                                $j/nome,
                                $j/duracao
                        }</musica>
            }<lancamento>
}</resultado>
```

Figura 4.10 : Consulta *XQuery* **C**1

```
SELECT i.titulo, i.cantor,
        CURSOR ( SELECT j.nome,
                        j.duracao
                 FROM TABLE (i.musica) j)
FROM Discos i
WHERE i.ano = 2003
```

Figura 4.11 : Consulta **C**2

Suponha que o resultado da consulta acima consiste na tabela de objetos mostrada na Figura 4.12. O resultado contém dois objetos complexos. Cada objeto é composto por dois atributos monovalorados de tipo atômico (**titulo** e **cantor**) e um atributo multivalorado de tipo estruturado. Cada objeto aninhado desse atributo multivalorado é composto por um atributo monovalorado atômico (**nome**) e um atributo monovalorado estruturado (**duracao**). O atributo **duracao** é composto por dois atributos monovalorados atômicos (**min.** e **seg.**).

Em seguida, com base no esquema da visão XML Discos (Figura 4.4) e na consulta *XQuery* (Figura 4.10), o *XQueryTranslator* transforma a tabela de objetos da Figura 4.12 no documento XML da Figura 4.13. Observe que os atributos monovalorados atômicos (**titulo**, **cantor**, **nome**, **min.** e **seg.**) foram mapeados em elementos monocorrência de mesmo nome e tipo simples.

titulo	cantor	Cursor
T1	C1	nome: M1 — Duracao: min. 1, seg. 20 nome: M2 — Duracao: min. 2, seg. 3
T2	C2	nome: M3 — Duração: min 2, seg 50 nome: M4 — Duração: min 3, seg 30

Figura 4.12: Resultado de C_2

```
<resultado>
        <lancamento>
                <titulo>T1</titulo>
                <cantor>C1</cantor>
                <musica>
                        <nome>M1</nome>
                        <duracao>
                                <min>1</min>
                                <seg>20</seg>
                        </duracao>
                </musica>
                <musica>
                        <nome>M2</nome>
                        <duracao>
                                <min>2</min>
                                <seg>30</seg>
                        </duracao>
                </musica>
        </lancamento>
        <lancamento>
                <titulo>T2</titulo>
                <cantor>C2</cantor>
                <musica>
                        <nome>M3</nome>
                        <duracao>
                                <min>2</min>
                                <seg>50</seg>
                        </duracao>
                </musica>
                <musica>
                        <nome>M4</nome>
                        <duracao>
                                <min>3</min>
                                <seg>30</seg>
                        </duracao>
                </musica>
        </lancamento>
</resultado>
```

Figura 4.13: Resultado de C_1

4.4 Acessando o XML *Publisher* na *Web*

O XML *Publisher* foi implementado como um serviço *web* e pode ser acessado por aplicações/usuário através de requisições *HTTP GET*. Essas requisições têm o seguinte formato: http://host/path?REQUEST=operação&param2=value, onde host denota o

endereço do servidor *web* que hospeda o XML *Publisher*, path é o caminho da aplicação XML *Publisher* no servidor *web* e REQUEST é o parâmetro da requisição que indica a operação submetida ao XML *Publisher*. O nome (**param2**) e o valor (**value**) para o segundo parâmetro da requisição variam de acordo com a operação. As operações definidas no XML *Publisher* são:

a) ***GetCapabilities***: operação para recuperar a descrição das capacidades do XML *Publisher*, ou seja, o nome das visões XML que o *framework* publica e as ações suportadas por cada uma das visões (consultas e/ou atualização). A resposta a essa requisição é um documento XML estruturado de acordo com o esquema definido na
b) Figura **4.14**.

c) ***GetSchema***: Essa operação confere ao XML *Publisher* a habilidade de descrever a estrutura de qualquer visão XML que ele publica**.** A resposta a essa requisição será o XML *Schema* da visão XML consultada.

d) ***ExecuteQuery***: através dessa operação, usuários/aplicações submetem consultas *XQuery* ao XML *Publisher*.

e) ***ExecuteUpdate***: operação para submeter atualizações *XQuery* [27][67] às visões publicadas no XML *Publisher*.

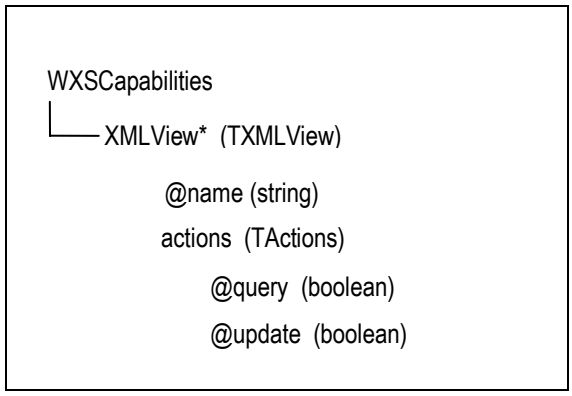

Figura 4.14: XML Schema que define a estrutura da resposta à requisição ***GetCapabilities***

Exemplo 4.3

Suponha que exista uma instância do XML *Publisher* sendo executada a partir do endereço *web* http://www.lia.ufc.br/bd/xmlPublisher e que essa instância publique as seguintes visões XML: **Livros**, **Artigos** e **Revistas.** A requisição ***GetCapabilities*** para esse servidor tem a seguinte forma:

http://www.lia.ufc.br/bd/xmlPublisher?REQUEST=GetCapabilities

O resultado dessa requisição será o XML apresentado na Figura 4.15. Observando esse resultado podemos concluir que a visão **Livros** é a única que não pode ser atualizada, apenas consultada:

```
<WXSCapabilities>
        <XMLView name= "Livros">
                <actions @query= "yes" @update = "no"/>
        </XMLView>
        <XMLView name= "Artigos">
                <actions @query= "yes" @update = "yes"/>
        </XMLView>
        <XMLView name= "Revistas">
                <actions @query= "yes" @update = "yes"/>
        </XMLView>
</WXSCapabilities>
```

Figura 4.15 : Resposta à requisição ***GetCapabilities***

Exemplo 4.4

Para descobrir a estrutura da visão **Livros** (XML *Schema*), deve-se submeter a seguinte requisição ao XML *Publisher*:

http://www.lia.ufc.br/bd/xmlPublisher?REQUEST= GetSchema&VIEW=Livros

Observe que o nome da visão foi passada como o valor do parâmetro VIEW.

Exemplo 4.5

Para processar uma consulta *XQuery* no XML *Publisher*, deve-se submeter a seguinte requisição ao XML *Publisher*:

http://www.lia.ufc.br/bd/xmlPublisher?REQUEST= ExecuteQuery&QUERY=<Consulta>

Note que a consulta é passada como o valor do parâmetro QUERY. Observe que não é preciso passar o nome da visão a ser consultada, como na requisição ***GetSchema***, pois o nome da visão já está definido na própria consulta, como veremos no Capítulo 6.

No Capítulo 6 também apresentamos os algoritmos utilizados no processamento de consultas *XQuery* no XML *Publisher*.

Exemplo 4.6

Para executar uma atualização no XML *Publisher*, deve-se submeter uma requisição ao XML *Publisher* com o seguinte formato:

http://www.lia.ufc.br/bd/xmlPublisher?REQUEST=ExecuteUpdate&QUERY= <Atualização>

Note que a atualização é passada como o valor do parâmetro QUERY. Os algoritmos responsáveis pelo processamento de atualizações *XQuery* no XML *Publisher* são apresentados em [36].

4.5 Algoritmo para Gerar o Esquema da Visão de Objetos *Default*

A Figura 4.16 apresenta o algoritmo **GeraEsquemaOR**, o qual gera o esquema de uma VOD a partir do XML *Schema* da visão XML. Inicialmente o algoritmo chama o procedimento **NomeiaTiposAnonimos** para transformar as definições de tipo anônimo do XML *Schema* em definições de tipos complexos nomeadas, a partir da declaração do elemento primário. Isso garante que todo tipo complexo do XML *Schema* seja mapeado em um tipo do esquema da VOD. Note que se o esquema da visão XML é definido usando o ambiente, o esquema XML gerado já satisfaz esses requisitos.

O algoritmo segue com uma chamada ao procedimento **SubstituiElementoMultiocorrencia** (Figura 4.17). Esse procedimento transforma as definições de elementos multiocorrência do esquema da visão XML em definições de tipo coleção, quando necessário. Um tipo coleção é um tipo complexo do XML *Schema* composto por uma única definição de elemento multiocorrência. Por exemplo, o tipo $\mathbf{T}_{\text{compositores}}$ apresentado no XML *Schema* da Figura 4.4 (a) é do tipo coleção, pois é composto por vários elementos (**compositor**) do mesmo tipo (**string**). Essa transformação é necessária, pois um XML *Schema* só pode ser diretamente mapeado em um esquema objeto-relacional se todos os seus elementos multiocorrência são definidos como tipos coleção. Essa propriedade garante que os elementos multiocorrência

definidos em um XML *Schema* sejam facilmente mapeados em *Nested Tables* ou *Varrays*, como indicado no caso 1 do algoritmo **GeraEsquemaOR**.

GeraEsquemaOR ($\mathbf{S_1}$: esquema da visão XML)
Inicio
  NomeiaTiposAnonimos ($\mathbf{S_1}$)
  SubstituiElementoMultiocorrencia ($\mathbf{S_1}$)
  Para toda definição de tipo complexo **T** em $\mathbf{S_1}$ **faça**
    Caso 1: **T** é do tipo coleção
      Seja $\mathbf{T_A}$ o tipo dos elementos de **T** e **n** sua ocorrência máxima
      Se n é UNBOUNDED **Então**
        Criar uma Nested Table **T** de $\mathbf{T_A}$, definida por:
        "CREATE TYPE **T** AS TABLE OF $\mathbf{T_A}$"
      Senão
        Criar um Varray **T** de $\mathbf{T_A}$ com cardinalidade **n**, definido por:
        "CREATE TYPE **T** AS VARRAY(**n**) OF $\mathbf{T_A}$"
    Caso 2: **T** não é do tipo coleção
      Seja $\mathbf{A_1}$, $\mathbf{A_2}$, ..., $\mathbf{A_n}$ os elementos de **T**, e $\mathbf{T_{A_1}}$, $\mathbf{T_{A_2}}$, ..., $\mathbf{T_{A_n}}$ seus respectivos tipos
      Criar um TAD **T**, definido por:
        "CREATE TYPE **T** AS OBJECT (
          $\mathbf{A_1}$ $\mathbf{T_{A_1}}$,
          $\mathbf{A_2}$ $\mathbf{T_{A_2}}$, ...,
          $\mathbf{A_n}$ $\mathbf{T_{A_n}}$,)"
Fim

Figura 4.16: Algoritmo **GeraEsquemaOR**

Em seguida, o algoritmo **GeraEsquemaOR** transforma cada tipo complexo do esquema da visão XML em um tipo de objeto (tipo abstrato de dado - TAD) no esquema da VOD. Elementos monocorrência são transformados em atributos dos tipos da VOD. Elementos multiocorrência são descartados, uma vez que objetos numa *Nested Table* ou *Varray* não são nomeados.

1. **SubstituiElementoMultiocorrencia** ($\mathbf{S_2}$: esquema da visão XML)
2. **Inicio**
3. **Para** cada tipo complexo **T** do esquema $\mathbf{S_2}$ que não é de tipo coleção **faça**
4. **Para** cada elemento multiocorrência **E** de **T faça**
5. Seja $\mathbf{T_E}$ o tipo do elemento **E**
6. Criar o tipo $\mathbf{T_{lista\_E}}$ composto por um elemento multiocorrência **Item_E** de tipo $\mathbf{T_E}$
7. Modificar o tipo de **E** para $\mathbf{T_{lista\_E}}$
8. Modificar a cardinalidade máxima de **E** para 1
9. **Fim**

Figura 4.17: Procedimento **SubstituiElementoMultiocorrencia**

Note que o esquema da visão XML é modificado apenas em caráter temporário, ou seja, as modificações servem apenas para gerar o esquema da VOD. Consultas *XQuery* submetidas ao XML *Publisher* devem ser definidas sobre o XML *Schema* original.

A seguir, apresentamos um exemplo para ilustrar o funcionamento desse algoritmo.

Exemplo 4.7

Considere o esquema da visão XML Discos apresentado na Figura 4.4 (a). Para gerar o esquema da VOD de Discos, o algoritmo **GeraEsquemaOR** inicialmente chama o procedimento **NomeiaTiposAnonimos**, mas como todos os tipos complexos a partir da declaração do elemento primário do esquema da visão XML são nomeados, a chamada a esse procedimento não altera o esquema. Em seguida, chama o procedimento **SubstituiElementoMultiocorrencia**. Esse procedimento modifica a definição do tipo $\mathbf{T}_{cd}$, pois o elemento **musica** desse tipo é multiocorrência e $\mathbf{T}_{cd}$ não é de tipo coleção.

O procedimento **SubstituiElementoMultiocorrencia** cria um tipo coleção $\mathbf{T}_{lista\_musica}$ que contém vários elementos **Item_musica** do tipo $\mathbf{T}_{musica}$ (linha 6). Além disso, o procedimento modifica a declaração do elemento **musica** em $\mathbf{T}_{cd}$, alterando a cardinalidade desse elemento monocorrência (linha 8) e seu tipo para $\mathbf{T}_{lista\_musica}$ (linha 7). Essas alterações no XML *Schema* da visão XML $\mathbf{X}_1$ são apresentadas na Figura 4.18.

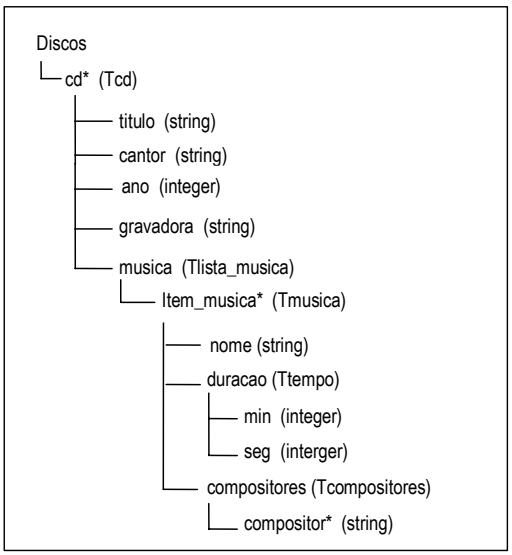

Figura 4.18: Esquema intermediário da visão XML discos

O próximo passo agora é gerar os tipos do esquema da VOD. Os cinco tipos do esquema da Figura 4.18 geram os cinco tipos de objetos no esquema da Figura 4.19. Os tipos $\mathbf{T}_{cd}$, $\mathbf{T}_{musica}$ e $\mathbf{T}_{tempo}$ são tratados pelo caso 2 do algoritmo, pois não são do tipo coleção. Note que os elementos que compõem cada um desses tipos são mapeados em atributos no respectivo tipo de objeto no esquema da VOD. Os tipos $\mathbf{T}_{lista\_musica}$ e $\mathbf{T}_{compositores}$ são tratados pelo caso 1, pois são do tipo coleção. Esses tipos são mapeados em tipos Nested Table da VOD. Observe que os elementos desses dois tipos não são mapeados para o esquema da VOD.

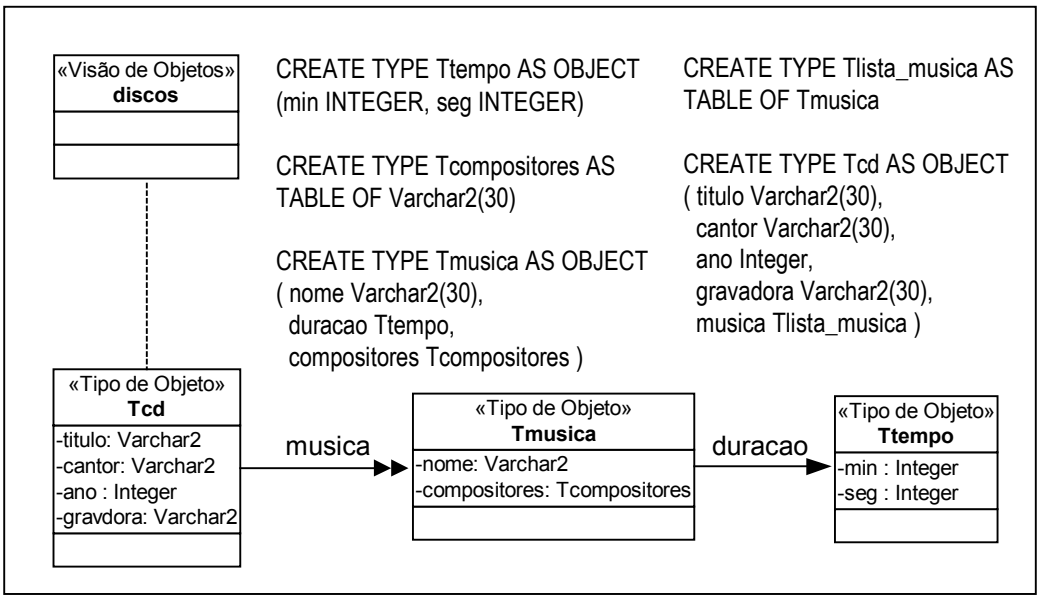

Figura 4.19: Esquema da VOD **Discos**

Capítulo 5

Geração Automática de Visão de Objetos a partir das Assertivas de Correspondências da Visão

Para criar uma visão de objeto, primeiro deve-se criar o tipo dos objetos da visão. Depois, então, define-se a visão através de uma consulta SQL:1999, a qual especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados. Definir uma consulta que realize o mapeamento de tuplas de tabelas relacionais (tabelas *flat*) em objetos de tipos com estrutura complexa é tarefa que exige conhecimentos avançados de SQL:1999. No caso de modificações no esquema do banco de dados, as visões de objeto precisam ser redefinidas.

Neste capítulo, apresentamos um algoritmo que gera automaticamente a definição SQL:1999 de uma visão de objetos a partir das assertivas de correspondência (AC's) da visão. O Capítulo está organizado como se segue. Na Seção 5.1, apresentamos a definição formal dos vários tipos de assertivas de correspondência utilizadas para especificar formalmente o relacionamento entre elementos de esquemas Objeto-Relacional. Na Seção 5.2, discutimos VBA, uma ferramenta gráfica para geração semi-automática de visões de objetos. Na Seção 5.3, apresentamos o algoritmo GeraVisaoDeObjetos, o qual gera a definição de uma visão de objetos a partir das AC's da visão. Neste trabalho, consideramos apenas as visões preservadoras de objetos [37], ou seja, visões onde seus objetos são semanticamente equivalente (S.E.) a objetos de uma tabela base, denominada de tabela *pivô*. Os objetos $\mathbf{o}_1$ e $\mathbf{o}_2$ são S.E. ($\mathbf{o}_1 \equiv \mathbf{o}_2$) sss representam uma mesma entidade do mundo real.

5.1 Assertivas de Correspondências no Modelo Objeto-Relacional

Nesta seção, apresentamos a definição formal dos vários tipos de assertivas de correspondência utilizadas para especificar formalmente o relacionamento entre elementos de esquemas Objeto-Relacional (OR). O formalismo proposto permite especificar várias formas de correspondência, inclusive casos onde os esquemas possuem estruturas diferentes [52]. As AC's da visão de objetos são classificadas em três tipos: (i) AC de Extensão; (ii) AC de Caminho e (iii) AC de Objeto. A seguir, apresentamos algumas definições necessárias para a definição formal dos tipos de assertivas citados acima.

5.1.1 Terminologias

Definição 5.1: Usamos o conceito de **ligação** para representar os relacionamentos entre tipos. Considere $\mathbf{T}_1$ e $\mathbf{T}_2$ tipos de um esquema. Existe uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$ nas situações descritas a seguir:

(i) (**Ligação de atributo de valor**) $\mathbf{T}_1$ contém um atributo cujo tipo é $\mathbf{T}_2$ ou coleção de objetos do tipo $\mathbf{T}_2$. Assim, $\mathbf{a}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Dada uma instância $\mathbf{t}_1$ de $\mathbf{T}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{a}$ retorna o valor do atributo $\mathbf{a}$ (uma instância ou coleção de instâncias de $\mathbf{T}_2$).

(ii) (**Ligação de atributo de referência**) $\mathbf{T}_1$ contém um atributo cujo tipo é uma referência ou uma coleção de referências para objetos do tipo $\mathbf{T}_2$. Assim, $\mathbf{a}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Dada uma instância $\mathbf{t}_1$ de $\mathbf{T}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{a}$ retorna as instâncias de $\mathbf{T}_2$ referenciadas por $\mathbf{t}_1$ através do atributo $\mathbf{a}$.

(iii) (**Ligação de chave estrangeira**) Existe uma chave estrangeira $\mathbf{fk}=\mathbf{R}_1[\mathbf{a}_1,...,\mathbf{a}_n] \subseteq \mathbf{R}_2[\mathbf{b}_1,...,\mathbf{b}_n]$, onde $\mathbf{T}_1$ e $\mathbf{T}_2$ são os tipos[1] das tuplas de $\mathbf{R}_1$ e $\mathbf{R}_2$, respectivamente. Assim, $\mathbf{fk}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Dada uma tupla $\mathbf{t}_1$ de $\mathbf{R}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{fk}$ retorna a tupla $\mathbf{t}_2$ de $\mathbf{R}_2$ tal que $\mathbf{t}_1 \bullet \mathbf{a}_i = \mathbf{t}_2 \bullet \mathbf{b}_i$, para $1 \leq i \leq n$.

[1] Para tratarmos de forma uniforme as tabelas de objetos e as tabelas de tuplas, assumimos que uma dada tabela de tuplas $\mathbf{R}$, $\mathbf{T}_R$ é o tipo das tuplas de $\mathbf{R}$. Este tipo é na verdade, definido implicitamente na própria criação da tabela $\mathbf{R}$.

(iv) (**Inversa de ligação**) $\mathbf{T}_2$ tem uma ligação ℓ: $\mathbf{T}_2 \rightarrow \mathbf{T}_1$ tal que ℓ é uma ligação de atributo de referência ou de chave estrangeira. Então, a inversa da ligação ℓ, dada por ℓ^{-1}: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Este tipo de ligação é chamado de **ligação virtual**. Os demais tipos de ligações são chamados de **ligação direta**.

Definição 5.2: Um objeto pode estar relacionado a outro objeto através da composição de duas ou mais ligações. Considere as ligações, ℓ_1: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$, ℓ_2: $\mathbf{T}_2 \rightarrow \mathbf{T}_3$,..., ℓ_{n-1}: $\mathbf{T}_{n-1} \rightarrow \mathbf{T}_n$, onde $\mathbf{T}_1$, $\mathbf{T}_2$,..., $\mathbf{T}_n$ são tipos de um esquema objeto-relacional. Então, $\varphi = \ell_1 \bullet \ell_2 \bullet ... \bullet \ell_{n-1}$ é um caminho de $\mathbf{T}_1$. Isso significa que as instâncias de $\mathbf{T}_1$ estão relacionadas com as instâncias de $\mathbf{T}_n$ através do caminho φ. Dada uma instância $\mathbf{t}_1$ de $\mathbf{T}_1$, a expressão $\mathbf{t}_1 \bullet \varphi$ retorna uma coleção de objetos $\mathbf{t}_n$ de $\mathbf{T}_n$ tais que existem as instâncias $\mathbf{t}_2$, $\mathbf{t}_3$,..., $\mathbf{t}_{n-1}$ onde $\mathbf{t}_i$ está relacionada com $\mathbf{t}_{i+1}$ através da ligação ℓ_i, para $1 \leq i < n$. Assim, o tipo do caminho φ é $\mathbf{T}_n$ ($\mathbf{T}_\varphi = \mathbf{T}_n$). Se as ligações $\ell_1, \ell_2, ..., \ell_{n-1}$ são monovaloradas, então φ é um **caminho monovalorado**; caso contrário φ é um **caminho multivalorado**. Se ℓ_{n-1} é uma ligação de referência, então φ é um **caminho de referência**; caso contrário φ é um **caminho de valor**.

Definição 5.3: Seja S um esquema OR e R uma tabela de S. Seja σ_S um estado de S. A extensão de R em σ_S, denotado por $\sigma_S(R)$, é o conjunto de objetos que são membros R em σ_S.

5.1.2 Assertivas de Correspondência de Extensão

As Assertivas de Correspondência de Extensão (ACE) especificam correspondências existentes entre a extensão de uma visão com a extensão de sua tabela *Pivô* [49]. No resto desta seção, sejam S um esquema OR, R uma tabela em S e **V** uma visão. Existem dois tipos de ACE's, definidas a seguir:

Definição 5.3: A ACE $[\mathbf{V}] \equiv [\mathbf{R}]$ especifica que, dados σ_S um estado de S e σ_v a extensão de **V** em σ_S, $t_v \in \sigma_v$ sss $\exists\, t_R \in \sigma_S(R)$ tal que $t_v \equiv t_R$.

Definição 5.4: A ACE [**V**] ≡ [**R**[*P*]], onde *P* é um predicado condicional, especifica que, dados σ_S o estado de S e σ_v a extensão de **V** em σ_S, $t_v \in \sigma_v$ sss $\exists\, t_R \in \sigma_S(R)$ tal que $\mathbf{t_R}$ satisfaz a condição *P* e $\mathbf{t_v} \equiv \mathbf{t_R}$.

5.1.3 Assertivas de Correspondência de Caminho

No restante desta seção, considere os tipos $\mathbf{T_v}$ e $\mathbf{T_b}$, onde $\mathbf{T_v}$ e $\mathbf{T_b}$ são semanticamente relacionados, no contexto de uma ACE ou ACC. As ACC's podem ser de três tipos como definidas a seguir:

Definição 5.5: Seja **c** um atributo de $\mathbf{T_v}$ e φ um caminho de $\mathbf{T_b}$, onde a cardinalidade de **c** é igual a cardinalidade de φ. A ACC $[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi}]$ especifica que para quaisquer instâncias $\mathbf{t_v}$ de $\mathbf{T_v}$ e $\mathbf{t_b}$ de $\mathbf{T_b}$, se $\mathbf{t_v} \equiv \mathbf{t_b}$, então $\mathbf{t_v \bullet c} \equiv \mathbf{t_b \bullet \varphi}$. No caso em que **c** é monovalorado, dados $\mathbf{t'_v} = \mathbf{t_v \bullet \varphi_v}$ e $\mathbf{t'_b} = \mathbf{t_b \bullet \varphi}$, então $\mathbf{t'_v} \equiv \mathbf{t'_b}$. No caso em que **c** é multivalorado, temos que $\mathbf{t'_v}$ pertence a $\mathbf{t_v \bullet \varphi_v}$ sss existe $\mathbf{t'_b}$ em $\mathbf{t_b \bullet \varphi}$, tal que $\mathbf{t'_v} \equiv \mathbf{t'_b}$.

Definição 5.6: Suponha $\mathbf{b_1, b_2, ..., b_n}$ atributos de valor atômico de $\mathbf{T_b}$ e **a** um atributo de $\mathbf{T_v}$, cujo tipo $\mathbf{T_a}$ contém atributos atômicos $\mathbf{a_1, a_2, ..., a_n}$. A ACC $[\mathbf{T_v \bullet a}, \{\mathbf{a_1, a_2, ..., a_n}\}] \equiv [\mathbf{T_b}, \{\mathbf{b_1, b_2, ..., b_n}\}]$ especifica que dada uma instância $\mathbf{t_v}$ de $\mathbf{T_v}$ e $\mathbf{t_b}$ de $\mathbf{T_b}$, se $\mathbf{t_v} \equiv \mathbf{t_b}$, então $\mathbf{t_v \bullet a \bullet a_i} = \mathbf{t_b \bullet b_i}$, para $1 \leq i \leq n$.

Definição 5.7: Seja **a** um atributo multivalorado de valor atômico de $\mathbf{T_v}$ e φ um caminho monovalorado de $\mathbf{T_b}$, de tipo $\mathbf{T_\varphi}$. Sejam $\mathbf{c_1}$, $\mathbf{c_2}$,..., $\mathbf{c_n}$ atributos monovalorados de valores atômicos de $\mathbf{T_\varphi}$. A ACC $[\mathbf{T_v \bullet a}] \equiv [\mathbf{T_b \bullet \varphi}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_n}\}]$ especifica que para quaisquer instâncias $\mathbf{t_v}$ de $\mathbf{T_v}$ e $\mathbf{t_b}$ de $\mathbf{T_b}$, se $\mathbf{t_v} \equiv \mathbf{t_b}$, então $\mathbf{t'_v}$ pertence a $\mathbf{t_v \bullet a}$ sss existe um objeto $\mathbf{t'_b}$ em $\{\mathbf{t_b \bullet \varphi \bullet c_1}, \mathbf{t_b \bullet \varphi \bullet c_2}, ..., \mathbf{t_b \bullet \varphi \bullet c_n}\}$, tal que $\mathbf{t'_v} = \mathbf{t'_b}$.

5.1.4 Assertivas de Correspondência de Objeto

As assertivas de correspondências de objeto (ACO) especificam sob que condições dois objetos, os quais são instâncias de tipos semanticamente relacionados, representam o mesmo objeto do mundo real, ou seja, são semanticamente equivalentes.

Definição 5.5.: Considere $\mathbf{T}_v$ um tipo do esquema da visão e $\mathbf{T}_b$ um tipo de uma tabela base, os quais são semanticamente relacionados. Sejam $\mathbf{v}_1$, $\mathbf{v}_2$, ..., $\mathbf{v}_n$ atributos de $\mathbf{T}_v$ e $\mathbf{b}_1$, $\mathbf{b}_2$, ..., $\mathbf{b}_n$ atributos de $\mathbf{T}_b$. A ACO ψ:[$\mathbf{T}_v$, { $\mathbf{v}_1$, $\mathbf{v}_2$, ..., $\mathbf{v}_n$ }] ≡ [$\mathbf{T}_b$, { $\mathbf{b}_1$, $\mathbf{b}_2$, ..., $\mathbf{b}_n$ }] especifica que para qualquer instância $\mathbf{t}_v$ de $\mathbf{T}_v$ e $\mathbf{t}_b$ de $\mathbf{T}_b$, se $\mathbf{t}_v \bullet \mathbf{v}_i = \mathbf{t}_b \bullet \mathbf{b}_i$, $1 \leq i \leq n$, então $\mathbf{t}_v \equiv \mathbf{t}_b$.

5.1.5 Usando Assertivas de Correspondência para Especificar Visões de Objeto

Neste trabalho, uma visão de objeto $\mathbf{V}$ sobre o esquema S é definida por uma 4-tupla $\mathbf{V} = < \mathbf{T}_v, \psi_V, C_V, A_V >$ onde $\mathbf{T}_v$ é o tipo dos objetos de $\mathbf{V}$, ψ_V é uma ACE, C_V é um conjunto de ACC e A_V é um conjunto de ACO. Isso define um mapeamento funcional, denotado DEF_V, de estados σ_S de S em estados da visão, como definido a seguir:

Caso 1: Para ψ_v da forma [$\mathbf{V}$] ≡ [$\mathbf{R}$[P]] , temos:

$$DEF_V(\sigma_S) = \{ \text{ConstrutorT}_v\text{\_T}_R(\mathbf{t}) \mid \mathbf{t} \in \sigma_S(R) \text{ e } P(\mathbf{t}) = \text{true} \},$$

onde ConstrutorT$_v$_T$_R$ ($\mathbf{t}$) cria uma instância $\mathbf{v}$ de $\mathbf{T}_V$ tal que $\mathbf{v} \equiv \mathbf{t}$. Note que se $\mathbf{v} \equiv \mathbf{t}$ então $\mathbf{v}$ deve satisfazer a todas as ACC's de $\mathbf{T}_v$&$\mathbf{T}_b$. Assim, os valores dos atributos do objeto $\mathbf{v}$ criado são definidos de acordo com as assertivas de correspondência de caminho de $\mathbf{T}_v$ & $\mathbf{T}_R$. Na Seção 5.3, apresentamos um algoritmo que a partir das ACCs da visão, gera automaticamente o construtor de objetos da visão.

Caso 2: Para ψ_V da forma [$\mathbf{V}$] ≡ [$\mathbf{R}$] , temos:

$$DEF_V(\sigma_S) = \{ \text{ConstrutorT}_v\text{\_T}_R(\mathbf{t}) \mid t \in \sigma_S(R) \}.$$

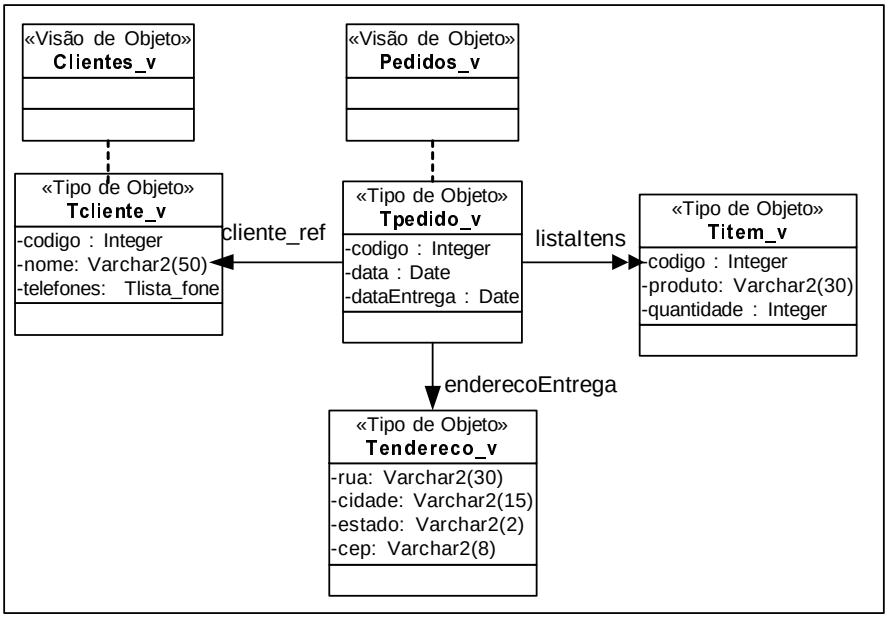

Figura 5.1: Esquema da Visão de Objetos **Pedidos_v**

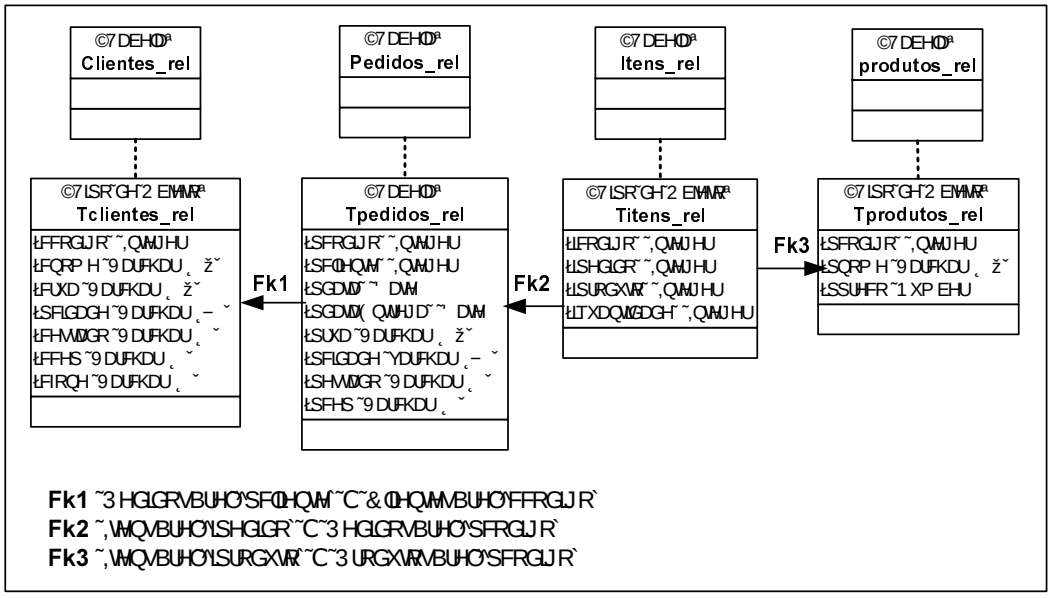

Figura 5.2: Esquema Relacional do Banco de Dados **Pedidos_rel**

Exemplo 5.1

No resto desta seção, considere a visão **Pedidos_v** cujos objetos são do tipo $\mathbf{T}_{pedido}$ (vide Figura 5.1) e o esquema do banco de dados **Pedidos_rel,** apresentado na Figura 5.2. Considere que a ACE de **Pedidos_v** é dada por ψ_1:[**Pedidos_v]** ≡ [**Pedidos_rel].** ψ_1 especifica

que as extensões da visão **Pedidos_v** e da tabela **Pedidos_rel** denotam o mesmo conjunto de objetos do mundo real. As ACC's de **Pedidos_v** são mostradas na **Figura 5.3**.

O processo de geração das ACC's é *top down* e recursivo. Primeiro definimos as ACC's dos atributos de $\mathbf{T}_{pedido\_v}$ com atributos/caminhos da tabela pivô **Pedidos_rel**. No caso de atributos de referência ou de tipos complexo deve-se então recursivamente definir as ACC's dos seus atributos com o seu tipo base.

Das ACC's de $\mathbf{T}_{pedido\_v}$&$\mathbf{T}_{pedidos\_rel}$ temos que dada uma instância $\mathbf{t}_v$ de $\mathbf{T}_{pedido\_v}$, e uma instância $\mathbf{t}_b$ de $\mathbf{T}_{pedidos\_rel}$ tal que $\mathbf{t}_v \equiv \mathbf{t}_b$ então:

(i) $\mathbf{t}_v$•codigo = $\mathbf{t}_b$•pcodigo (de ψ_2)

(ii) $\mathbf{t}_v$•data = $\mathbf{t}_b$•pdata (de ψ_3)

(iii) $\mathbf{t}_v$• dataEntrega = $\mathbf{t}_b$•pdataEntrega (de ψ_4)

(iv) $\mathbf{t}_v$•enderecoEntrega = $\mathbf{T}_{endereco\_v}$($\mathbf{t}_b$•prua, $\mathbf{t}_b$•pcidade, $\mathbf{t}_b$•pestado, $\mathbf{t}_b$•pcep) (de ψ_5)

(v) $\mathbf{t}_v$• cliente_ref ≡ $\mathbf{t}_b$•$\mathbf{Fk}_1$ (de ψ_6). Como o tipo de cliente_ref é uma referência para $\mathbf{T}_{cliente\_v}$, que é um tipo estruturado, deve-se definir as ACC's de $\mathbf{T}_{cliente\_v}$ & $\mathbf{T}_{clientes\_rel}$.

(vi) $\mathbf{t}' \in \mathbf{t}_v$•listaItens sss existe $\mathbf{t} \in \mathbf{t}_b$•$\mathbf{Fk}_2^{-1}$ e $\mathbf{t} \equiv \mathbf{t}'$ (de ψ_7). Como listaItens é uma coleção de objetos de tipo $\mathbf{T}_{item\_v}$, que é um tipo estruturado, deve-se definir as ACC's de $\mathbf{T}_{item\_v}$ & $\mathbf{T}_{itens\_rel}$.

ACC's de $\mathbf{T}_{pedido\_v}$ & $\mathbf{T}_{pedidos\_rel}$
ψ_2: [$\mathbf{T}_{pedido\_v}$•codigo]≡ [$\mathbf{T}_{pedidos\_rel}$•pcodigo]
ψ_3: [$\mathbf{T}_{pedido\_v}$•data] ≡ [$\mathbf{T}_{pedidos\_rel}$•pdata]
ψ_4: [$\mathbf{T}_{pedido\_v}$•dataEntrega] ≡ [$\mathbf{T}_{pedidos\_rel}$•pdataEntrega]
ψ_5: [$\mathbf{T}_{pedido\_v}$•enderecoEntrega], {rua, cidade, estado, cep}] ≡ [$\mathbf{T}_{pedidos\_rel}$,{prua, pcidade], pestado, pcep}]
ψ_6: [$\mathbf{T}_{pedido\_v}$•cliente_ref] ≡ [$\mathbf{T}_{pedidos\_rel}$•fk_1]
ψ_7: [$\mathbf{T}_{pedido\_v}$•listaItens] ≡[$\mathbf{T}_{pedidos\_rel}$•fk_2^{-1}]

ACC's de $\mathbf{T}_{cliente\_v}$ & $\mathbf{T}_{clientes\_rel}$
ψ_9: [$\mathbf{T}_{cliente\_v}$•codigo] ≡ [$\mathbf{T}_{clientes\_rel}$•ccodigo]
ψ_{10}: [$\mathbf{T}_{cliente\_v}$•nome] ≡ [$\mathbf{T}_{clientes\_rel}$•cnome]
ψ_{11}: [$\mathbf{T}_{cliente\_v}$•telefones] ≡ [$\mathbf{T}_{clientes\_rel}$,{cfone1, cfone2, cfone3}]

ACC's de $\mathbf{T}_{item\_v}$ & $\mathbf{T}_{itens\_rel}$
ψ_{13}: [$\mathbf{T}_{item\_v}$•codigo] ≡ [$\mathbf{T}_{itens\_rel}$•icodigo]
ψ_{14}: [$\mathbf{T}_{item\_v}$•produto] ≡ [$\mathbf{T}_{itens\_rel}$•fk_3•pnome]
ψ_{15}: [$\mathbf{T}_{item\_v}$•quantidade] ≡ [$\mathbf{T}_{itens\_rel}$•iquantidade]

Figura 5.3: ACC's de **Pedidos_v**

Das ACC's de $T_{cliente\_v}$ & $T_{clientes\_rel}$ temos que dada uma instância t_v de $T_{cliente\_v}$, e uma instância t_b de $T_{clientes\_rel}$ tal que $t_v \equiv t_b$ então:

(i) $t_v \bullet codigo = t_b \bullet ccodigo$ (de ψ_9)

(ii) $t_v \bullet nome = t_b \bullet cnome$ (de ψ_{10})

(iii) $t_v \bullet$ telefones = $T_{lista\_fone}(t_b \bullet cfone1, t_b \bullet cfone2, t_b \bullet cfone2)$ (de ψ_{11})

Das ACC's de T_{item_v} & T_{itens_rel} temos que dada uma instância t_v de T_{item_v}, e uma instância t_b de T_{itens_rel} tal que $t_v \equiv t_b$ então:

(i) $t_v \bullet codigo = t_b \bullet icodigo$ (de ψ_{13})

(ii) $t_v \bullet produto = t_b \bullet Fk_3 \bullet pnome$ (de ψ_{14})

(iii) $t_v \bullet$ quantidade = $t_b \bullet iquantidade$ (de ψ_{15})

As assertivas de correspondência de objeto de **Pedidos_v** são mostradas na Figura 5.4. As ACO's são inferidas a partir das ACC's e das chaves primárias das tabelas do banco de dados. Por exemplo, da ACC ψ_2 e da chave primária da tabela **Pedidos_rel**, inferimos a ACO ψ_8: $[T_{pedido\_v}, \{codigo\}] \equiv [T_{pedidos\_rel}, \{pcodigo\}]$ que especifica que para quaisquer instâncias t_v de T_{pedido_v} e t_b de $T_{pedidos\_rel}$, se $t_v \bullet codigo = t_b \bullet pcodigo$, então $t_v \equiv t_b$.

Como mostraremos na Seção 5.3 a definição SQL:1999 de uma visão de objetos pode ser gerada automaticamente a partir das ACs da visão. A Figura 5.5 mostra a definição SQL:1999 da Visão que realiza o mapeamento definido pelas Assertivas.

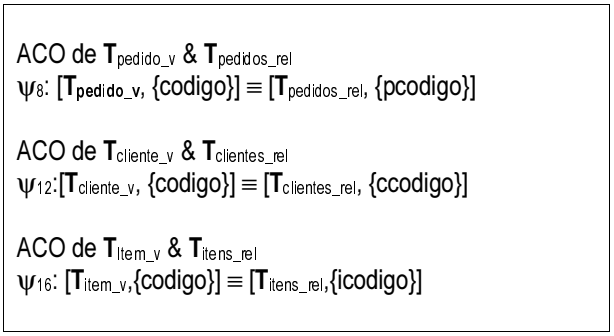

Figura 5.4: ACO's de **Pedidos_v**

```
CREATE VIEW Pedidos_v OF T_pedido
WITH OBJECT IDENTIFIER (codigo) AS
   SELECT
1. T_pedido(                                                    Construtor_T_pedido-T_pedidos_rel
2.        t_pedidos_rel•pcodigo,                                                    (de ψ3 )
3.        t_pedidos_rel•pdata,                                                      (de ψ4 )
4.        t_pedidos_rel•pdataEntrega,                                               (de ψ5 )
5.        T_endereco ( t_pedidos_rel•prua, t_pedidos_rel•pcidade, t_pedidos_rel•pestado, t_pedidos_rel•pcep ),  (de ψ6 )
6.        (SELECT REF (t_cliente)                                                   (de ψ7 )
          FROM Clientes_v t_cliente, Clientes_rel t_clientes_rel
          WHERE t_pedidos_rel•pcliente = t_clientes_rel•ccodigo
                t_clientes_rel•ccodigo = t_cliente•codigo),                         (da ACO ψ9)
7.        CAST(MULTISET( SELECT                                                     (de ψ8 )
                                                    Construtor_T_item-T_itens_rel
                T_item(
8.                   t_itens_rel•icodigo,                                         (de ψ14)
9.                   (SELECT t_produtos_rel•pnome                                 (de ψ15)
                      FROM Produtos_rel t_produtos_rel
                      WHERE t_itens_rel•iproduto= t_produtos_rel•pcodigo),
10.                  t_itens_rel•iquantidade)                                     (de ψ16)
                FROM Itens_rel  t_itens_rel
11.       ) AS T_lista_item )

FROM Pedidos_rel  t_pedidos_rel
```

Figura 5.5 : Definição da visão de objetos **Pedidos_v**

5.2 Gerando Visões de Objetos com VBA

Nesta seção, apresentamos VBA (*View-By-Assertions*), uma ferramenta semi-automática para gerar visões de objetos no *Oracle* 9i [37]. Como estudo de caso, considere o esquema do banco de dados **Pedidos_rel** apresentado na Figura 5.2. Suponha que o usuário deseja criar, usando VBA, a visão de objeto **Pedidos_v** (Figura 5.1)**,** cujos objetos são do tipo T_{pedido}.

A seção começa com o usuário definindo o nome da visão e o nome do seu tipo. Em seguida, o usuário deve criar os atributos do tipo da visão (Figura 5.6 (a)). Para isso, usa a tela de criação de atributo (Figura 5.6 (b)). Além do nome, são pedidos o tipo do atributo, sua cardinalidade e se o mesmo é de valor ou de referência. No caso de atributos de tipo estruturado, então um novo tipo deve ser criado (nome e atributos). No caso de atributos de referência, deve-se definir seu escopo (visão de objeto referenciada). Se a visão ainda não

existe, deverá ser criada. Quando utilizamos VBA na geração de visões de objeto *default* para o XML *Publisher*, a definição dos atributos do tipo da visão é feita automaticamente pela ferramenta GeraEsquemaVOD, como foi mostrado no Capítulo 4.

Após definir os atributos do tipo da visão, o usuário deve definir as assertivas de correspondência da visão. Primeiro, o usuário deve definir a ACE. Para isso, o usuário seleciona, de uma lista de tabelas e visões do banco de dados, a tabela base ou visão base, e, se necessário, define a condição de seleção. Para a visão **Pedidos_v,** a tabela **Pedidos_rel** é selecionada e a ACE ψ_1:[**Pedidos_v**] ≡ [**Pedidos_Rel**] é gerada.

Em seguida, o usuário deve realizar o *matching* [30][39][40] do tipo da visão com o tipo base. Para isso, VBA exibe uma tela contendo, em formato de árvore de diretório, a estrutura do tipo da visão e a estrutura do tipo base (Figura 5.6 (c)). Observe que para o tipo base, além dos atributos, são mostradas as demais ligações do tipo (ligações de chave estrangeira e inversas), de modo que o usuário possa definir caminhos que naveguem por essas ligações. Para definir a ACC de um atributo da visão, o usuário relaciona graficamente o atributo da visão com um atributo ou caminho do tipo da tabela base. A seguir mostramos como gerar as ACC's para alguns dos atributos do tipo $\mathbf{T}_{pedido}$:

- Para definir a ACC do atributo codigo, o usuário seleciona codigo no esquema da visão e pcodigo no esquema do banco de dados. Ao clicar no botão "Salvar", VBA mostra a ACC gerada (ψ_3: [$\mathbf{T}_{pedido}$•codigo] ≡ [$\mathbf{T}_{pedidos\_rel}$•pcodigo]);
- Para definir a ACC do atributo listaItens, o usuário seleciona listaItens no esquema da visão e fk_2^{-1} no esquema do banco. Observe que para o atributo listaItens o "*" denota que o atributo é multivalorado e $\mathbf{T}_{item}$, especificado entre parênteses, é o tipo dos objetos em listaItens. Como $\mathbf{T}_{item}$ é um tipo estruturado, o usuário deve então definir recursivamente as correspondências para os atributos de $\mathbf{T}_{item}$.
- Para definir a ACC do atributo enderecoEntrega, o qual não tem correspondência direta com nenhum atributo do tipo base, o usuário seleciona o atributo enderecoEntrega e a tabela base **Pedidos_rel**, e, então, salva. Em seguida, o usuário define a correspondência para cada atributo de $\mathbf{T}_{endereco}$ com os atributos de $\mathbf{T}_{pedidos\_rel}$. A ACC gerada é ψ_6:[$\mathbf{T}_{pedido}$•enderecoEntrega,{rua, cidade, estado, cep}] ≡ [$\mathbf{T}_{pedidos\_rel}$, {prua, pcidade, pestado, pcep}].

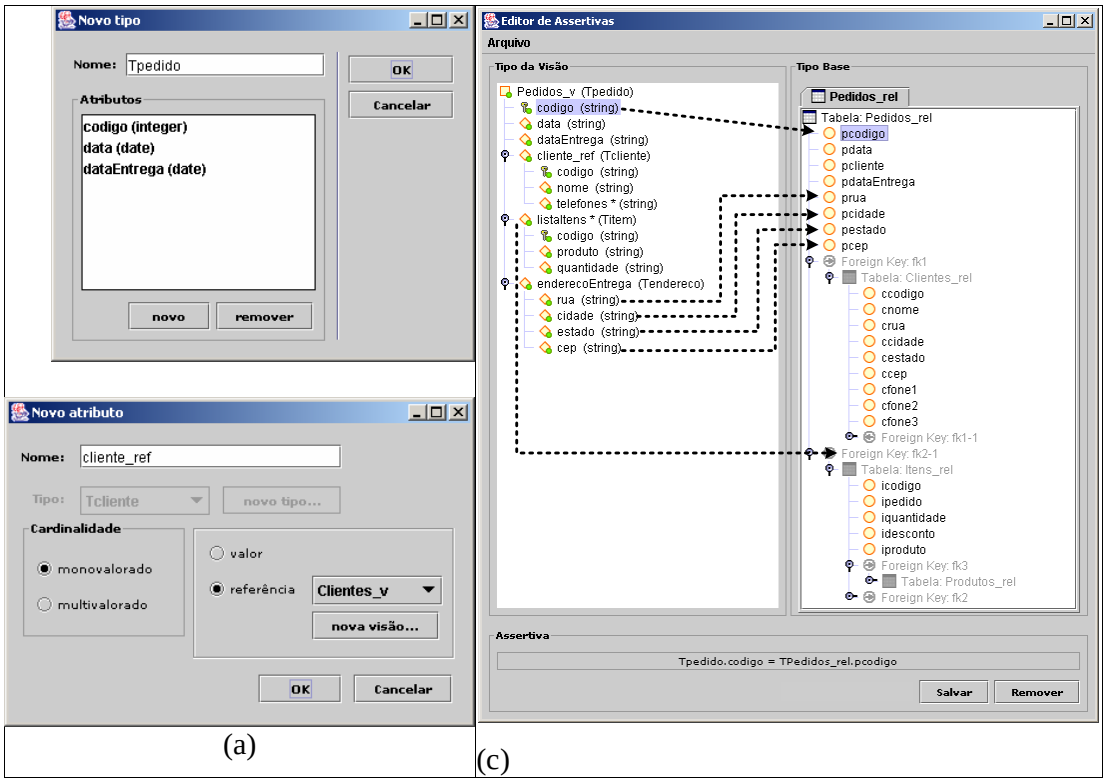

Figura 5.6: Telas do VBA: (a)Editor de Tipo; (b)Editor de Atributo; (c)Editor de Assertivas do VBA

As ACO's de **Pedidos_v** são inferidas a partir das ACC's e das chaves das tabelas. Por exemplo, ψ_2 é inferida de ψ_3 e do fato de que pcodigo é a chave primária da tabela **Pedidos_rel**. Caso um atributo da chave primária não tenha um correspondente na visão, VBA resolve o problema, adicionando o atributo à ACC correspondente, e, por fim, à ACO.

Após definir as assertivas da visão, a definição SQ:1999 da visão pode ser gerada automaticamente. A Figura 5.5 apresenta a definição da visão de objetos **Pedidos_v** gerada pelo VBA. Na Seção a seguir apresentamos o algoritmo GeraVisaoDeObjeto, o qual gera a definição de uma visão de objetos a partir das AC's da visão.

5.3 Algoritmo GeraVisaoDeObjeto

Nesta seção apresentamos o algoritmo GeraVisaoDeObjeto, o qual gera a consulta SQL:1999 que realiza o mapeamento definido pelas assertivas da visão de objetos. No resto desta seção, considere a visão de objetos **V**, cujos objetos são do tipo $\mathbf{T}_V$.

Casos	Definição da Visão
1. A extensão de **V** é definida pela ACE [**V**] ≡ [$\mathbf{R_b}$] onde $\mathbf{R_b}$ é do tipo $\mathbf{T_b}$ e [$\mathbf{T_v}$, {$\mathbf{a}_1$, $\mathbf{a}_2$,..., $\mathbf{a}_m$}] ≡ [$\mathbf{T_b}$, {$\mathbf{b}_1$, $\mathbf{b}_2$,..., $\mathbf{b}_m$}] é ACO de $\mathbf{T_v}$ & $\mathbf{T_b}$.	CREATE VIEW **V** OF $\mathbf{T_v}$ WITH OBJECT IDENTIFIER ($\mathbf{a}_1$,$\mathbf{a}_2$,...,$\mathbf{a}_m$) AS SELECT **Construtor_T_v-T_b** ($\mathbf{t_b}$) FROM $\mathbf{R_b}$ $\mathbf{t_b}$
2. A extensão de **V** é definida pela ACE [**V**] ≡[$\mathbf{R_b}$[p]], onde $\mathbf{R_b}$ é do tipo $\mathbf{T_b}$, p é um predicado condicional, e [$\mathbf{T_v}$, {$\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_m$}] ≡ [$\mathbf{T_b}$, {$\mathbf{b}_1$, $\mathbf{b}_2$, ..., $\mathbf{b}_m$}] é ACO de $\mathbf{T_v}$ & $\mathbf{T_b}$.	CREATE VIEW **V** OF $\mathbf{T_v}$ WITH OBJECT IDENTIFIER ($\mathbf{a}_1$,$\mathbf{a}_2$,...,$\mathbf{a}_m$) AS SELECT **Construtor_T_v-T_b** ($\mathbf{t_b}$) FROM $\mathbf{R_b}$ $\mathbf{t_b}$ WHERE p($\mathbf{t_b}$)

Tabela 5.1: Casos do algoritmo GeraVisaoDeObjeto

A Tabela 5.1 mostra o padrão das consultas geradas para os dois tipos de ACE's. O identificador da visão especificado na cláusula WITH OBJECT IDENTIFIER é definido pela ACO de $\mathbf{T_v}$. O construtor **Construtor_T_v-T_b** cria um objeto $\mathbf{t_v}$ do tipo $\mathbf{T_v}$, a partir de um objeto (tupla) $\mathbf{t_b}$ da tabela base $\mathbf{R_b}$, onde $\mathbf{T_b}$ é o tipo da tabela $\mathbf{R_b}$, tal que $\mathbf{t_v} \equiv \mathbf{t_b}$. Os valores dos atributos do objeto $\mathbf{t_v}$ criado são definidos de acordo com as assertivas de correspondência de caminho de $\mathbf{T_v}$ & $\mathbf{T_b}$. Como veremos na Seção 5.3.1 e na Seção 5.3.2, o construtor **Construtor_T_v-T_b** é gerado de forma automática a partir das ACC's de $\mathbf{T_v}$ & $\mathbf{T_b}$.

É importante notar que outros tipos de visões (união, intersecção e diferença), que têm mais de uma tabela base, podem ser tratadas da seguinte forma: no caso em que o tipo dos objetos das tabelas base são diferentes, primeiro, cria-se uma visão de objeto para cada tabela base; assim são criados diferentes construtores para cada tabela base. Pode-se definir, então, uma visão (de união, intersecção ou diferença) sobre as visões de objeto criadas. No caso em que as tabelas base têm o mesmo tipo, deve-se definir uma visão cujos objetos têm o tipo das tabelas base. Então, pode-se definir, sobre a visão criada, uma visão de objeto usando VBA.

Exemplo 5.2

Considere o esquema do banco de dados **Pedidos_rel** (Figura 5.2), o esquema da visão de objetos **Pedidos_v** (Figura 5.1) e as AC's da visão apresentadas na Figura 5.3.

De acordo com o algoritmo GeraVisaoDeObjetos, inicialmente a ACE ψ_1 da visão de objetos é analisada. Como ψ_1 é do tipo equivalência, a definição da visão será gerada pelo caso 1 da Tabela 5.1. A assertiva ψ_1 especifica que para cada objeto $\mathbf{t}_{pedidos}$ de **Pedidos_v**, existe um objeto $\mathbf{t}_{pedidos\_rel}$ na tabela base **Pedidos_rel** tal que $\mathbf{t}_{pedidos} \equiv \mathbf{t}_{pedidos\_rel}$ e vice-versa. A cláusula FROM na definição da visão é composta pela tabela **Pedidos_rel**. A cláusula

WITH OBJECT IDENTIFIER é montada a partir da ACO ψ_2. A Figura 5.5 apresenta a definição da visão **Pedidos_v** gerada pelo algoritmo.

O construtor **Construtor_**T_{pedido}-T_{pedido_rel} que aparece na cláusula SELECT da definição da visão gera um objeto do tipo $T_{pedidos}$ a partir de um objeto $t_{pedidos\_rel}$ do tipo T_{pedido_rel}, de forma que o objeto criado é semanticamente equivalente ao objeto $t_{pedidos\_rel}$. Para isto, os valores dos atributos do objeto criado são definidos de acordo com as AC's de T_{pedido} & T_{pedido_rel}, como veremos na Seção 5.3.1.

5.3.1 Algoritmo GeraConstrutorRelacional

Nesta seção considere a visão **V,** cujos objetos são do tipo T_v, e R_b ,a tabela pivô de **V** de tipo T_b. O algoritmo GeraConstrutorRelacional recebe como entrada o tipo T_v e o tipo base T_b e gera, com base nas ACC's de T_v & T_b, um construtor de objetos do tipo T_v, o qual cria uma instância t_v de T_v a partir de uma tupla t_b de T_b tal que t_v e t_b são semanticamente equivalentes ($t_v \equiv t_b$). Sejam C_1, ..., e C_m atributos de T_v, então, o construtor gerado tem a seguinte forma: T_v ($Q_{C1}(t_b)$, ... , $Q_{Cm}(t_b)$), onde $Q_{Ci}(t_b)$, $1 \leq i \leq m$, é o código SQL:1999 que gera o valor do atributo C_i para um objeto da visão t_v a partir da tupla t_b. Os valores dos atributos de t_v são definidos a partir das ACC's de T_v & T_b, como mostrado na Tabela 5.2. Para essa Tabela, considere **CaminhoRel** φ e **Juncao**φ, como definido a seguir:

Definição 5.8: (**CaminhoRel** φ) Considere $\varphi = \ell_1 \bullet \ell_2 \bullet ... \bullet \ell_n$ um caminho de T_b (Figura 5.7), onde:

(i) ℓ_1 é uma ligação de chave estrangeira dada por $R_b[f_1^{\ell 1}, ..., f_{m_1}^{\ell 1}] \subseteq R_{\ell 1}[g_1^{\ell 1},.., g_{m_1}^{\ell 1}]$ ou inversa da chave estrangeira dada por $R_{\ell 1}[g_1^{\ell 1}, ..., g_{m_1}^{\ell 1}] \subseteq R_b[f_1^{\ell 1}, ..., f_{m_1}^{\ell 1}]$;

(ii) ℓ_i, é uma ligação de chave estrangeira dada por $R_{\ell_{i-1}}[f_1^{\ell_i}, ..., f_{m_i}^{\ell_i}] \subseteq R_{\ell_i}[g_1^{\ell_i},.., g_{m_i}^{\ell_i}]$ ou inversa da chave estrangeira dada por $R_{\ell_i}[g_1^{\ell_i}, ..., g_{m_i}^{\ell_i}] \subseteq R_{\ell_{i-1}}[f_1^{\ell_i}, ..., f_{m_i}^{\ell_i}]$, $2 \leq i \leq n$;

Definição 5.9: (**Juncaoφ**) Considere $\varphi = \ell_1 \bullet \ell_2 \bullet ... \bullet \ell_n$ um caminho de $\mathbf{T}_b$ como definido na Figura 5.7.

Juncaoφ = $\mathbf{R}_{\ell_1}$ $\mathbf{t}_{R\ell_1}$,….., $\mathbf{R}_{\ell_n}$ $\mathbf{t}_{R\ell_n}$ WHERE $\mathbf{t}_{R_b} \bullet f_k^{\ell_1} = \mathbf{t}_{R\ell_1} \bullet g_k^{\ell_1}$ AND $\mathbf{t}_{R\ell_{j-1}} \bullet f_w^{\ell_j} = \mathbf{t}_{R\ell_j} \bullet g_w^{\ell_j}$, onde $1 \leq k \leq m_1$, $1 \leq w \leq m_j$ e $2 \leq j \leq n$. Assim, dado uma instância $\mathbf{t}_b$ de $\mathbf{T}_b$, temos que:

$\mathbf{t}_b \bullet \varphi$ = SELECT $\mathbf{t}_{R\ell_n}$ FROM **Juncaoφ** .

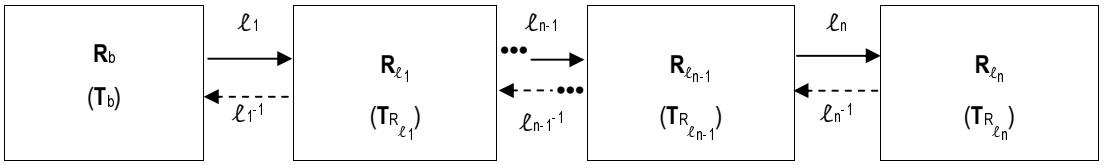

Figura 5.7: CaminhoRel φ

Casos	$\mathbf{Q}_c(\mathbf{t}_b)$
1. c é monovalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \mathbf{a}$], onde **a** é um atributo monovalorado atômico de $\mathbf{T}_b$	$\mathbf{t}_b \bullet a$
2. c é monovalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}$], onde φ é um caminho de $\mathbf{T}_b$, **a** é um atributo monovalorado atômico e $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$	(SELECT $\mathbf{t}_{R\ell_n} \bullet \mathbf{a}$ FROM **Juncaoφ**)
3. c é monovalorado de valor estruturado de tipo T_c e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $T_{R\ell_n}$ o tipo de φ e $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$.	(SELECT **Construtor_**T_c-$T_{R\ell_n}$($\mathbf{t}_{R\ell_n}$) FROM **Juncaoφ**)
4. c é monovalorado de referência para a visão $\mathbf{V}_c$ de tipo $\mathbf{T}_c$ e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $T_{R\ell_n}$ é o tipo de φ, $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$ e [T_c,{$\mathbf{c}_1$,$\mathbf{c}_2$,...,$\mathbf{c}_w$}] ≡ [$T_{R\ell_n}$,{$\mathbf{d}_1$,$\mathbf{d}_2$,..., $\mathbf{d}_w$}] é a ACO de $\mathbf{T}_c$ & $T_{R\ell_n}$	(SELECT REF ($\mathbf{t}_c$) FROM $\mathbf{V}_c$ $\mathbf{t}_c$, **Juncaoφ** AND $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_1 = \mathbf{t}_c \bullet \mathbf{c}_1$ AND ... $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_w = \mathbf{t}_c \bullet \mathbf{c}_w$)
5. c é multivalorado de valor atômico e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}$], onde φ é um caminho multivalorado de $\mathbf{T}_b$, $T_{R\ell_n}$ o tipo de φ, T_c é o tipo de **c**, $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$, e **a** um atributo monovalorado de valor atômico de $T_{R\ell_n}$.	CAST (MULTISET (SELECT $\mathbf{t}_{R\ell_n} \bullet \mathbf{a}$ FROM **Juncaoφ**) AS $\mathbf{T}_c$)
6. c é multivalorado de valor estruturado de tipo T_c, uma coleção de objetos do tipo T_{item_c} e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $T_{R\ell_n}$ é o tipo de φ.	CAST (MULTISET (SELECT **Construtor_**T_{item_c}-$T_{R\ell_n}$($\mathbf{t}_{R\ell_n}$) FROM **Juncaoφ**) AS $\mathbf{T}_c$)
7. c é multivalorado de referência de tipo T_c, uma coleção de referências para visão $\mathbf{V}_c$ de tipo T_{item_c} e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $T_{R\ell_n}$ é o tipo de φ, $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$ e [T_{v_c},{$\mathbf{c}_1$,$\mathbf{c}_2$,...,$\mathbf{c}_w$} ≡ $T_{R\ell_n}$ {$\mathbf{d}_1$,$\mathbf{d}_2$,...,$\mathbf{d}_w$}] é a ACO de $\mathbf{T}_{v_c}$ & $T_{R\ell_n}$	CAST (MULTISET (SELECT REF ($\mathbf{t}_c$) FROM $\mathbf{V}_c$ $\mathbf{t}_{item\_c}$, **Juncaoφ** AND $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_1 = \mathbf{t}_{item\_c} \bullet \mathbf{c}_1$ AND ... $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_w = \mathbf{t}_{item\_c} \bullet \mathbf{c}_w$) AS $\mathbf{T}_c$)
8. c é monovalorado de valor estruturado de tipo $\mathbf{T}_c$ e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$,{$\mathbf{c}_1$,$\mathbf{c}_2$,...,$\mathbf{c}_w$}] ≡ [$\mathbf{T}_b$,{$\mathbf{d}_1$,$\mathbf{d}_2$,..., $\mathbf{d}_w$}]	$\mathbf{T}_c$ ($\mathbf{t}_b \bullet \mathbf{d}_1$, $\mathbf{t}_b \bullet \mathbf{d}_2$, ..., $\mathbf{t}_b \bullet \mathbf{d}_w$)
9. c é multivalorado de valor atômico e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b$, {$\mathbf{d}_1$,$\mathbf{d}_2$,...,$\mathbf{d}_w$}], onde de tipo $\mathbf{T}_c$ é o tipo de **c**.	$\mathbf{T}_c$ ($\mathbf{t}_b \bullet \mathbf{d}_1$, $\mathbf{t}_b \bullet \mathbf{d}_2$,...,$\mathbf{t}_b \bullet \mathbf{d}_w$)
10. c é multivalorado de valor atômico e ψ_c: [$\mathbf{T}_v \bullet \mathbf{c}$] ≡ [$\mathbf{T}_b \bullet \varphi$,{$\mathbf{d}_1$,$\mathbf{d}_2$,...,$\mathbf{d}_w$}], onde φ é um caminho monovalorado de $\mathbf{T}_b$, $\mathbf{T}_c$ é o tipo de **c** e $\mathbf{t}_{R\ell_n}$ uma tupla da tabela $\mathbf{R}_{\ell_n}$	(SELECT $\mathbf{T}_c$ ($\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_1$, $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_2$,..., $\mathbf{t}_{R\ell_n} \bullet \mathbf{d}_w$) FROM **Juncaoφ**)

Tabela 5.2: Casos do algoritmo **GeraConstrutorRelacional**.

A seguir mostramos que o SQL:1999 gerado para cada caso da Tabela 2 realiza corretamente o mapeamento especificado pela assertiva de correspondência do atributo. No restante dessa seção, considere $\mathbf{t_v}$ uma instância de $\mathbf{T_v}$ e $\mathbf{t_b}$ uma instância de $\mathbf{T_b}$ tal que $\mathbf{t_v \equiv t_b}$.

Caso 1: De ψ_c: $[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet a}]$ temos que $\mathbf{t_v \bullet c \equiv t_b \bullet a}$. Dado que **c** é monovalorado de valor atômico, então temos que $\mathbf{t_v \bullet c = t_b \bullet a}$.

Caso 2: De ψ_c:$[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi \bullet a}]$ temos que $\mathbf{t_v \bullet c \equiv t_c \bullet \varphi \bullet a}$. Dado que **c** é monovalorado de valor atômico, então temos que:

(1) $\mathbf{t_v \bullet c = t_b \bullet \varphi \bullet a}$

Dado que, $\mathbf{t_b \bullet \varphi}$ = SELECT $\mathbf{t_{R_{\ell n}}}$ FROM **Juncao**φ, de (1) temos que:

$\mathbf{t_v \bullet c}$ = SELECT $\mathbf{t_{R_{\ell n}} \bullet a}$ FROM **Juncao**φ

Caso 3: De ψ_c: $[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi}]$ temos que:

(1) $\mathbf{t_v \bullet c \equiv t_b \bullet \varphi}$

Seja $\mathbf{t_{R_{\ell n}}}$ uma tupla da tabela $\mathbf{R_{\ell n}}$ (de tipo $\mathbf{T_{R_{\ell n}}}$) tal que $\mathbf{t_{R_{\ell n}} = t_b \bullet \varphi}$. Dado que **c** é monovalorado de valor estruturado de tipo $\mathbf{T_c}$, de (1) temos que:

(2) $\mathbf{t_v \bullet c}$ = **Construtor_**$\mathbf{T_c}$-$\mathbf{T_{R_{\ell n}}}$ ($\mathbf{t_{R_{\ell n}}}$)

Dado que $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncao**φ , de (2) temos:

$\mathbf{t_v \bullet c}$ = SELECT **Construtor_**$\mathbf{T_c}$-$\mathbf{T_{R_{\ell n}}}$ ($\mathbf{t_{R_{\ell n}}}$) FROM **Juncao**φ

Caso 4: De ψ_c: $[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi}]$ temos que:

(1) $\mathbf{t_v \bullet c \equiv t_b \bullet \varphi}$

Seja $\mathbf{t_{R_{\ell n}}}$ uma tupla da tabela $\mathbf{R_{R_{\ell n}}}$ (de tipo $\mathbf{T_{R_{\ell n}}}$) tal que $\mathbf{t_{R_{\ell n}} = t_b \bullet \varphi}$. Dado que **c** é um atributo monovalorado de referência para a visão $\mathbf{V_c}$ de tipo $\mathbf{T_c}$, de (1), temos que:

(2) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c \in V_c}$ e $\mathbf{t_c \equiv t_{R_{\ell n}}}$.[2]

Seja $\mathbf{t_c}$ uma instância de $\mathbf{T_c}$. Da ACO de $\mathbf{T_c}$ & $\mathbf{T_{R_{\ell n}}}$, $[\mathbf{T_c}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_w}\}] \equiv [\mathbf{T_{R_{\ell n}}}, \{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_w}\}]$, temos que:

(3) $\mathbf{t_c \equiv t_{R_{\ell n}}}$ sss $\mathbf{t_c \bullet c_i = t_{R_{\ell n}} \bullet d_i}$, para $1 \leq i \leq w$.

De (2) e (3) temos:

[2] REF retorna o OID de $\mathbf{t_c}$

(4) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c} \in \mathbf{V_c}$ e $\mathbf{t_c \bullet c_i = t_{R_{\ell n}} \bullet d_i}$, para $1 \leq i \leq w$.

Dado que $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncao**φ de (4) temos:

$\mathbf{t_v \bullet c}$ = SELECT REF ($\mathbf{t_c}$)

FROM $\mathbf{V_c}$ $\mathbf{t_c}$, **Juncao**φ AND $\mathbf{t_{R_{\ell n}} \bullet d_1 = t_c \bullet c_1}$ AND ... $\mathbf{t_{R_{\ell n}} \bullet d_w = t_c \bullet c_w}$

Caso 5: De ψ_c: $[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi \bullet a}]$ temos que:

(1) $\mathbf{t_v \bullet c \equiv t_b \bullet \varphi \bullet a}$

Dado que **c** é multivalorado de valor atômico, de (1) temos que:

(2) $\mathbf{t_v \bullet c} = \{ \mathbf{t_{R_{\ell n}} \bullet a} \mid \mathbf{t_{R_{\ell n}}} \in \mathbf{t_b \bullet \varphi} \}$

Dado que, $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncao**φ, e $\mathbf{T_c}$ é o tipo (*nested table*) de **c**, de (2) temos:

$\mathbf{t_v \bullet c}$ = CAST (MULTISET (SELECT $\mathbf{t_{R_{\ell n}} \bullet a}$ FROM **Juncao**φ) AS $\mathbf{T_c}$)

Como apresentando no Capítulo 3, o operador *CAST-MULTISET* indica que o resultado da consulta interna é uma coleção de objetos *(nested table)* do tipo $\mathbf{T_c}$.

Caso 6: De ψ_c:$[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi}]$ temos que:

(1) $\mathbf{t_v \bullet c \equiv t_b \bullet \varphi}$

Dado que **c** é multivalorado estruturado de tipo $\mathbf{T_c}$, uma coleção de objetos do tipo $\mathbf{T_{item\_c}}$, de (1) temos que:

(2) $\mathbf{t_v \bullet c} = \{ \mathbf{Construtor\_T_{item\_c}\text{-}T_{R_{\ell n}}(t_{R_{\ell n}})} \mid \mathbf{t_{R_{\ell n}}} \in \mathbf{t_b \bullet \varphi} \}$

Dado que, $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncao**φ, e $\mathbf{T_c}$ é o tipo (*nested table*) de **c**, de (2) temos:

$\mathbf{t_v \bullet c}$ = CAST (MULTISET (SELECT $\mathbf{Construtor\_T_{item\_c}\text{-}T_{R_{\ell n}}(t_{R_{\ell n}})}$ FROM **Juncao**φ) AS $\mathbf{T_c}$

Caso 7: De ψ_c:$[\mathbf{T_v \bullet c}] \equiv [\mathbf{T_b \bullet \varphi}]$ temos que:

(1) $\mathbf{t_v \bullet c \equiv t_b \bullet \varphi}$

Dado que **c** é um atributo multivalorado de referência para a visão $\mathbf{V_c}$ de tipo $\mathbf{T_{item\_c}}$, de (1), temos que:

(2) $\mathbf{t_v \bullet c} = \{\text{REF } (\mathbf{t_{item\_c}}) \mid \mathbf{t_{item\_c}} \in \mathbf{V_c} \wedge \exists \mathbf{t_{R_{\ell n}}} \in \mathbf{t_b \bullet \varphi} \wedge \mathbf{t_{item\_c} \equiv t_{R_{\ell n}}} \}$

Sejam $\mathbf{t_{item\_c}}$ uma instância de $\mathbf{T_{item\_c}}$ e $\mathbf{t_{R_{\ell n}}}$ uma instância de $\mathbf{T_{R_{\ell n}}}$. Da ACO de $\mathbf{T_{item\_c}}$ & $\mathbf{T_{R_{\ell n}}}$, $[\mathbf{T_{item\_c}}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_w}\}] \equiv [\mathbf{T_{R_{\ell n}}}, \{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_w}\}]$ temos que:

(3) $\mathbf{t_{item\_c} \equiv t_{R_{\ell n}}}$ sss $\mathbf{t_{item\_c} \bullet c_i = t_{R_{\ell n}} \bullet d_i}$, para $1 \leq i \leq w$.

De (2) e (3) temos:

(4) $\mathbf{t_v \bullet c}$ = {REF ($\mathbf{t_{item\_c}}$) | $\mathbf{t_{item\_c}} \in \mathbf{V_c}$ e $\mathbf{t_{R_{\ell n}}} \in \mathbf{t_b \bullet \varphi}$ e $\mathbf{t_{item\_c} \bullet c_i = t_{R_{\ell n}} \bullet d_i}$, para $1 \leq i \leq w$. }.

Dado que $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncaoφ**, e $\mathbf{T_c}$ é o tipo (*nested table*) de **c**, de (3) temos:

$\mathbf{t_v \bullet c}$ = CAST (MULTISET

(SELECT REF ($\mathbf{t_{item\_c}}$) FROM $\mathbf{V_c}$ $\mathbf{t_{item\_c}}$, **Juncaoφ**

AND $\mathbf{t_{R_{\ell n}} \bullet d_1 = t_{item\_c} \bullet c_1}$ AND ...

$\mathbf{t_{R_{\ell n}} \bullet d_w = t_{item\_c} \bullet c_w}$) AS $\mathbf{T_c}$)

Caso 8: De ψ_c: [$\mathbf{T_v \bullet c}$,{$\mathbf{c_1}$,$\mathbf{c_2}$,...,$\mathbf{c_w}$}] ≡ [$\mathbf{T_b}$,{$\mathbf{d_1}$,$\mathbf{d_2}$,..., $\mathbf{d_w}$}], onde **c** é um atributo monovalorado estruturado do tipo $\mathbf{T_c}$ e {$\mathbf{c_1}$,$\mathbf{c_2}$,...,$\mathbf{c_w}$} e {$\mathbf{d_1}$,$\mathbf{d_2}$,..., $\mathbf{d_w}$} são atributos monovalorados de valor atômico de $\mathbf{T_c}$ e $\mathbf{T_b}$, temos que:

$\mathbf{t_c \bullet c \bullet c_i = t_b \bullet d_i}$, para $1 \leq i \leq w$. Logo, $\mathbf{t_v \bullet c = T_c (t_b \bullet d_1, t_b \bullet d_2, ..., t_b \bullet d_w)}$.

Caso 9: De ψ_c: [$\mathbf{T_v \bullet c}$] ≡[$\mathbf{T_b}$, {$\mathbf{d_1}$,$\mathbf{d_2}$,...,$\mathbf{d_w}$}], onde **c** é um atributo multivalorado de tipo $\mathbf{T_c}$, e {$\mathbf{d_1}$,$\mathbf{d_2}$,...,$\mathbf{d_w}$} são atributos monovalorados de valor atômico de $\mathbf{T_b}$, temos que:

$\mathbf{t_v \bullet c = \{t_b \bullet d_1, t_b \bullet d_2,..., t_b \bullet d_w\}}$. Logo, $\mathbf{t_v \bullet c = T_c (t_b \bullet d_1, t_b \bullet d_2,..., t_b \bullet d_w)}$.

Caso 10: De ψ_c: [$\mathbf{T_v \bullet c}$] ≡ [$\mathbf{T_b \bullet \varphi}$,{$\mathbf{d_1}$,$\mathbf{d_2}$,...,$\mathbf{d_w}$}], onde **c** é um atributo multivalorado de tipo $\mathbf{T_c}$ e {$\mathbf{d_1}$,$\mathbf{d_2}$,...,$\mathbf{d_w}$} são atributos monovalorados de valor atômico, temos que: $\mathbf{t_v \bullet c = \{ t_b \bullet \varphi \bullet d_1}$, ... , $\mathbf{t_b \bullet \varphi \bullet d_w}$ }. Assim temos:

(1) $\mathbf{t_v \bullet c = T_c (t_b \bullet \varphi \bullet d_1,..., t_b \bullet \varphi \bullet d_w)}$.

Seja $\mathbf{t_{R_{\ell n}}}$ uma tupla da tabela $\mathbf{R_{\ell n}}$ (de tipo $\mathbf{T_{R_{\ell n}}}$) tal que $\mathbf{t_{R_{\ell n}} = t_b \bullet \varphi}$. De (1) temos:

(2) $\mathbf{t_v \bullet c = T_c (t_{R_{\ell n}} \bullet d_1,..., t_{\ell n} \bullet d_w)}$.

Dado que $\mathbf{t_{R_{\ell n}}}$ é obtida de **Juncaoφ**, de (2) temos:

$\mathbf{t_v \bullet c}$ = (SELECT $\mathbf{T_c(t_{R_{\ell n}} \bullet d_1,..., t_{R_{\ell n}} \bullet d_w}$) FROM **Juncaoφ**)

Exemplo 5.3

Considere o esquema do banco de dados **Pedidos_rel** (Figura 5.2), o esquema da visão **Pedidos_v** (Figura 5.1) e as AC's da visão apresentadas na **Figura 5.3**. A seguir discutimos como são gerados os valores para cada atributo do tipo $\mathbf{T_{pedido}}$ (**Construtor_**$\mathbf{T_{pedido}}$-

$T_{pedidos\_rel}$), a partir das AC's de T_{pedido} & $T_{pedidos\_rel}$. No resto desta seção, seja t_{pedido} o objeto criado pelo construtor a partir do objeto base $t_{pedidos\_rel}$.

– O atributo codigo (linha 2 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_3:[T_{pedido}•codigo] ≡ [$T_{pedidos\_rel}$•pcodigo]. Do Caso 1 da Tabela 5.2, temos que t_{pedido}•codigo = $t_{pedidos\_rel}$•pcodigo.

– O atributo data (linha 3 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_4:[T_{pedido}•data] ≡ [$T_{pedidos\_rel}$•pdata]. Do Caso 1 da Tabela 5.2, temos que t_{pedido}•data = $t_{pedidos\_rel}$•pdata.

– O atributo dataEntrega (linha 4 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_5:[T_{pedido}•dataEntrega] ≡ [$T_{pedidos\_rel}$•pdataEntrega]. Do Caso 1 da Tabela 5.2, temos que t_{pedido}•dataEntrega = $t_{pedidos\_rel}$•pdataEntrega.

– O atributo enderecoEntrega (linha 5 - Figura 5.5) é um atributo monovalorado de valor estruturado, de tipo $T_{endereco}$, definido pela AC ψ_6:[T_{pedido}•enderecoEntrega, {rua, cidade, estado, cep}] ≡ [$T_{pedidos\_rel}$,{prua, pcidade, pestado, pcep}]. Do Caso 8 da Tabela 5.2, temos que: t_{pedido}•enderecoEntrega = $T_{endereco}$ ($t_{pedidos\_rel}$•prua, $t_{pedidos\_rel}$•pcidade, $t_{pedidos\_rel}$•pestado, $t_{pedidos\_rel}$•pcep).

– O atributo cliente_ref (linha 6 - Figura 5.5) é um atributo monovalorado de referência definido pela AC ψ_7:[T_{pedido}•cliente_ref] ≡ [$T_{pedidos\_rel}$•fk_1], onde **Clientes_v** é a visão referenciada por cliente_ref e ψ_9:[$T_{cliente}$, {codigo}] ≡ [$T_{clientes\_rel}$, {ccodigo}] é a ACO de $T_{cliente}$ & $T_{clientes\_rel}$, satisfazendo, assim, o Caso 4 da Tabela 5.2. Dado que $\varphi=fk_1$, onde fk_1 é uma ligação de chave estrangeira dada por **Pedidos_rel**[pcliente] ⊆ **Clientes_rel**[ccodigo], temos:

Juncaoφ = Clientes_rel $t_{clientes\_rel}$ WHERE $t_{pedidos\_rel}$•pcliente= $t_{clientes\_rel}$•ccodigo.

Assim, do Caso 4 da Tabela 5.2 temos que:

t_{pedido}•cliente_ref= (SELECT REF($t_{cliente}$)
FROM **Clientes_v** $t_{cliente}$, **Clientes_rel** $t_{clientes\_rel}$
WHERE $t_{pedidos\_rel}$•pcliente = $t_{clientes\_rel}$•ccodigo AND
$t_{cliente}$•codigo = $t_{clientes\_rel}$• ccodigo)

– O atributo listaItens (linha 7 - Figura 5.5) é um atributo multivalorado de valor estruturado definido pela AC ψ_8:[T_{pedido}•listaItens] ≡ [$T_{pedidos\_rel}$•fk_2^{-1}], onde $T_{listaItem}$ é o tipo de listaItens, T_{item} é o tipo dos objetos em listaItens, e T_{itens_rel} é o tipo do caminho fk_2^{-1}, satisfazendo, assim, o Caso 6 da Tabela 5.2. Dado que $\varphi = fk_2^{-1}$, onde fk_2 é uma

ligação de chave estrangeira dada por **Itens_rel**[ipedido] $\subseteq$ **Pedidos_rel**[pcodigo], temos:

Juncaoφ = Itens_rel t_{itens_rel} WHERE t_{itens_rel}•pcodigo = t_{itens_rel}•ipedido).

Assim, do Caso 6 da Tabela 5.2 temos que:

t_{pedido}•listaItens= CAST (MULTISET (SELECT **Construtor_**T_{item}-T_{itens_rel}(t_{itens_rel})
FROM Itens_rel t_{itens_rel}
WHERE t_{itens_rel}•pcodigo = t_{itens_rel}•ipedido) AS $T_{listaItem}$).

Os valores dos atributos no construtor **Construtor_**T_{item}-T_{itens_rel} são definidos pelas ACC's de T_{item}&T_{Itens_rel}, como mostrado a seguir. No resto desta seção, considere t_{item} o objeto criado pelo **Construtor_**T_{Item}-T_{Itens_rel} a partir do objeto base t_{itens_rel}.

– O atributo codigo (linha 8 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_{11}:[T_{item}•codigo] ≡ [T_{itens_rel}•icodigo. Do Caso 1 da Tabela 5.2, temos que t_{item}•codigo = t_{itens_rel}•icodigo.

– O atributo produto (linha 9 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_{12}:[T_{item}•produto] ≡ [T_{itens_rel}•fk_3•pnome], onde fk_3 é uma ligação de chave estrangeira, satisfazendo, assim, o Caso 2 da Tabela 5.2. Dado que $\varphi = fk_3$, onde fk_3 é uma ligação de chave estrangeira dada por **Itens_rel**[iproduto] $\subseteq$ **Produtos_rel**[pcodigo], temos:

Juncaoφ = Produtos_rel $t_{produtos\_rel}$ WHERE t_{itens_rel}•iproduto= $t_{produtos\_rel}$•pcodigo

Assim, do Caso 2 da Tabela 5.2 temos que:

t_{item}•produto = (SELECT $t_{produtos\_rel}$•pnome
FROM Produtos_rel $t_{produtos\_rel}$
WHERE t_{itens_rel}•iproduto= $t_{produtos\_rel}$•pcodigo)

– O atributo quantidade (linha 10 - Figura 5.5) é um atributo monovalorado atômico definido pela AC ψ_{13}: [T_{item}•quantidade] ≡ [T_{itens_rel}•iquantidade]. Do Caso 1 da Tabela 5.2, temos que: t_{item}•quantidade = t_{itens_rel}•iquantidade.

5.3.2 Algoritmo GeraConstrutorObjetoRelacional

Nesta seção considere a visão **V,** cujos objetos são do tipo T_v, e R_b ,a tabela pivô de **V** de tipo T_b. O algoritmo GeraConstrutorObjetoRelacional recebe como entrada o tipo T_v e o tipo base T_b e gera, com base nas ACC's de T_v & T_b, um construtor de objetos do tipo T_v, o qual cria uma instância t_v de T_v a partir de uma instância t_b de T_b tal que t_v e t_b são semanticamente equivalentes ($t_v \equiv t_b$). Sejam C_1, ..., e C_m atributos de T_v, então, o construtor gerado tem a seguinte forma: T_v ($Q_{C1}(t_b)$, ... , $Q_{Cm}(t_b)$), onde $Q_{Ci}(t_b)$, $1 \leq i \leq m$, é o código SQL:1999 que gera o valor do atributo C_i para um objeto da visão t_v a partir do objeto t_b. Os valores dos atributos de t_v são definidos a partir das ACC's de T_v & T_b, como mostrado na Tabela 5.3. Os valores de S_φ, F_φ, e W_φ que aparecem nos casos 3, 5, 7, 8, 9, 10 e 11 da Tabela 5.3, são computados pelo procedimento GeraConsulta, apresentado na Figura 5.8. O procedimento GeraConsulta recebe como entrada um caminho φ de T_b e uma expressão de caminho **E**, e retorna uma tripla < S_φ , F_φ , W_φ > , de forma que a consulta dada por:

Select S_φ **From** F_φ **Where** W_φ,

retorna os objetos em $E \bullet \varphi$ (**Select** S_φ **From** F_φ **Where** W_φ = { t_φ | $t_\varphi \in E \bullet \varphi$}). É importante notar que isso só é necessário para os casos em que o caminho φ é virtual ou multivalorado. No caso em que o caminho φ é direto, W_φ tem valor nulo, de forma que vazio, então $Exp \bullet \varphi$ = **Select** S_φ **From** F_φ.

A seguir demonstramos que o SQL:1999 gerado para cada tipo de atributo da Tabela 5.3 realiza corretamente o mapeamento especificado pela respectiva assertiva de correspondência. No restante dessa seção, considere t_v uma instância de T_v e t_b uma instância de T_b tal que $t_v \equiv t_b$.

Caso 1: De ψ_c: [$T_v \bullet c$] ≡ [$T_b \bullet a$] temos que $t_v \bullet c \equiv t_b \bullet a$. Dado que **c** é monovalorado de valor atômico, então temos que $t_v \bullet c = t_b \bullet a$.

Caso 2: De ψ_c: [$T_v \bullet c$] ≡ [$T_b \bullet \varphi \bullet a$] temos que $t_v \bullet c \equiv t_b \bullet \varphi \bullet a$. Dado que **c** é monovalorado de valor atômico e φ é um caminho monovalorado direto, então temos que $t_v \bullet c = t_b \bullet \varphi \bullet a$.

Caso 3: De ψ_c: [$T_v \bullet c$] ≡ [$T_b \bullet \varphi \bullet a$] temos que $t_v \bullet c \equiv t_b \bullet \varphi \bullet a$. Dado que **c** é monovalorado de valor atômico e, então temos que:

(1) $t_v \bullet c = t_b \bullet \varphi \bullet a$.

Dado que φ é um caminho virtual de $\mathbf{T}_b$, suponha GeraConsulta (φ, $\mathbf{t}_b$) = < $\mathbf{S}_\varphi$, $\mathbf{F}_\varphi$, $\mathbf{W}_\varphi$ >. Então, temos:

(2) $\mathbf{t}_b \bullet \varphi$ = SELECT $\mathbf{S}_\varphi$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

Como **a** é um atributo monovalorado de valor atômico, de (2) temos que:

(3) $\mathbf{t}_b \bullet \varphi \bullet a$ = SELECT $\mathbf{S}_\varphi \bullet a$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

Sendo assim, de (1) e (3) temos:

$\mathbf{t}_v \bullet \mathbf{c}$ = SELECT $\mathbf{S}_\varphi \bullet a$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

Casos	$\mathbf{Q}_{C1}(\mathbf{t}_b)$
1. c é monovalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \mathbf{a}$], onde **a** é um atributo monovalorado de valor atômico de $\mathbf{T}_b$	$\mathbf{t}_b \bullet \mathbf{a}$
2. c é monovalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}$], onde φ é um caminho direto de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ e **a** é um atributo monovalorado de valor atômico de $\mathbf{T}_\varphi$	$\mathbf{t}_b \bullet \varphi \bullet \mathbf{a}$
3. c é monovalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}$], onde φ é um caminho virtual de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ e **a** é um atributo monovalorado de valor atômico de $\mathbf{T}_\varphi$	(SELECT $\mathbf{S}_\varphi \bullet \mathbf{a}$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$)
4. c é monovalorado de valor estruturado e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho direto de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ e $\mathbf{T}_c$ é o tipo do atributo **c**	**Construtor_**$\mathbf{T}_c$**-**$\mathbf{T}_\varphi$ ($\mathbf{t}_b \bullet \varphi$)
5. c é monovalorado de valor estruturado e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho virtual de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ e $\mathbf{T}_c$ é o tipo do atributo **c**	(SELECT **Construtor_**$\mathbf{T}_c$**-**$\mathbf{T}_\varphi$ ($\mathbf{S}_\varphi$) FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$)
6. c é monovalorado de referência e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho direto de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ é o tipo de φ, $\mathbf{V}_c$ é a visão referenciada por **c** e [$\mathbf{T}_c$, {$\mathbf{c}_1$, $\mathbf{c}_2$,..., $\mathbf{c}_w$}] ≡ [$\mathbf{T}_\varphi$, {$\mathbf{d}_1$, $\mathbf{d}_2$,..., $\mathbf{d}_w$}] é a ACO de $\mathbf{T}_c$ & $\mathbf{T}_\varphi$	MAKE_REF ($\mathbf{V}_c$, $\mathbf{t}_b \bullet \varphi \bullet \mathbf{d}_1$, $\mathbf{t}_b \bullet \varphi \bullet \mathbf{d}_2$,…, $\mathbf{t}_b \bullet \varphi \bullet \mathbf{d}_w$)
7. c é monovalorado de referência e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho virtual de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ é o tipo de φ, $\mathbf{V}_c$ é a visão referenciada por **c**, [$\mathbf{T}_c$, {$\mathbf{c}_1$, $\mathbf{c}_2$,..., $\mathbf{c}_w$}] ≡ [$\mathbf{T}_\varphi$, {$\mathbf{d}_1$, $\mathbf{d}_2$,..., $\mathbf{d}_w$}] é a ACO de $\mathbf{T}_c$ & $\mathbf{T}_\varphi$ e $\mathbf{t}_\varphi$ uma instância de $\mathbf{T}_\varphi$	(SELECT MAKE_REF ($\mathbf{V}_c$, $\mathbf{S}_\varphi \bullet \mathbf{d}_1$, ,…, $\mathbf{S}_\varphi \bullet \mathbf{d}_w$) FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$)
8. c é multivalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}$], onde φ é um caminho de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ, **a** é um atributo monovalorado de valor atômico de $\mathbf{T}_\varphi$ e $\mathbf{T}_c$ é o tipo do atributo **c**	CAST (MULTISET (SELECT $\mathbf{S}_\varphi \bullet \mathbf{a}$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$) AS $\mathbf{T}_c$)
9. c é multivalorado de valor atômico e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ e $\mathbf{T}_c$ é o tipo do atributo **c**	CAST (MULTISET (SELECT $\mathbf{S}_\varphi$•COLUMN_VALUE FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$) AS $\mathbf{T}_c$)
10. c é multivalorado de valor estruturado e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ o tipo de φ , $\mathbf{T}_c$ é o tipo do atributo **c** e $\mathbf{T}_{item\_c}$ é o tipo dos objetos de **c**	CAST (MULTISET (SELECT **Construtor_**$\mathbf{T}_{item\_c}$**-**$\mathbf{T}_\varphi$ ($\mathbf{S}_\varphi$) FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$) AS $\mathbf{T}_c$)
11. c é multivalorado de referência e ψ_c:[$\mathbf{T}_v \bullet \mathbf{c}$]≡[$\mathbf{T}_b \bullet \varphi$], onde φ é um caminho de $\mathbf{T}_b$, $\mathbf{T}_\varphi$ é o tipo de φ, $\mathbf{V}_c$ é a visão referenciada por **c**, $\mathbf{T}_c$ é o tipo do atributo **c**, $\mathbf{T}_{item\_c}$ é o tipo de $\mathbf{V}_c$ e [$\mathbf{T}_{item\_c}$, {$\mathbf{c}_1$, $\mathbf{c}_2$,..., $\mathbf{c}_w$}] ≡ [$\mathbf{T}_\varphi$, {$\mathbf{d}_1$, $\mathbf{d}_2$,..., $\mathbf{d}_w$}] é a ACO de $\mathbf{T}_{item\_c}$ & $\mathbf{T}_\varphi$.	CAST (MULTISET (SELECT MAKE_REF ($\mathbf{V}_c$, $\mathbf{S}_\varphi \bullet \mathbf{d}_1$, …, $\mathbf{S}_\varphi \bullet \mathbf{d}_w$) FROM FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$) AS $\mathbf{T}_c$)

Tabela 5.3: Casos do algoritmo **GeraConstrutorObjetoRelacional**

GeraConsulta (φ : caminho de $\mathbf{T}_b$, **E**: uma expressão de caminho)
Var
$\mathbf{S}_\varphi$: expressão de caminho
$\mathbf{F}_\varphi$: lista de tabelas ou *nested tables*
$\mathbf{W}_\varphi$: conjunções
Início
$\mathbf{S}_\varphi$:= **E**
Para cada ligação ℓ_i de φ, 1≤ i ≤ n, **Faça**
Caso 1: ℓ_i é direta
Caso 1.1: ℓ_i é monovalorada
$\mathbf{S}_\varphi$+= "•" + ℓ_i
Caso 1.2: ℓ_i é multivalorada
$\mathbf{F}_\varphi$ += "TABLE(" + $\mathbf{S}_\varphi$+ "•" + ℓ_i + ")" + $\mathbf{t}_{\ell_i}$
Se ℓ_i é de referência **Então**
$\mathbf{S}_\varphi$:= $\mathbf{t}_{\ell_i}$ + "•COLUMN_VALUE"
Senão
$\mathbf{S}_\varphi$:= $\mathbf{t}_{\ell_i}$
Fim Se
Caso 2: ℓ_i é virtual
Seja **R** a tabela de objetos (ou visão) referenciada por ℓ_i
$\mathbf{F}_\varphi$ += **R** $\mathbf{t}_{\ell_i}$
Caso 2.1: ℓ_i^{-1}é monovalorada
Se $\mathbf{S}_\varphi$contém um caminho de referência **Então**
$\mathbf{W}_\varphi$ += $\mathbf{t}_{\ell_i}$ + "•" + ℓ_i^{-1} + "=" + $\mathbf{S}_\varphi$
Senão
$\mathbf{W}_\varphi$ += $\mathbf{t}_{\ell_i}$ + "•" + ℓ_i^{-1} + "= REF(" + $\mathbf{S}_\varphi$+ ")"
Fim Se
$\mathbf{S}_\varphi$:= $\mathbf{t}_{\ell_i}$
Caso 2.2: ℓ_i^{-1} é multivalorada
Se $\mathbf{S}_\varphi$contém um caminho de referência **Então**
$\mathbf{W}_\varphi$ += $\mathbf{S}_\varphi$+ "IN " + "(SELECT * FROM TABLE (" + $\mathbf{t}_{\ell_i}$ + "•" + ℓ_i^{-1} + "))"
Senão
$\mathbf{W}_\varphi$ += "REF(" + $\mathbf{S}_\varphi$+ ") IN " +
"(SELECT * FROM TABLE (" + $\mathbf{t}_{\ell_i}$ + "•" + ℓ_i^{-1} + "))"
Fim Se
$\mathbf{S}_\varphi$:= $\mathbf{t}_{\ell_i}$
Fim Para
Retorne ⟨ $\mathbf{S}_\varphi$, $\mathbf{F}_\varphi$, $\mathbf{W}_\varphi$ ⟩
Fim

Figura 5.8: Procedimento **GeraConsulta**

Caso 4: De ψ_c: $[\mathbf{T}_v \bullet \mathbf{c}] \equiv [\mathbf{T}_b \bullet \varphi]$ temos que:

(1) $\mathbf{t}_v \bullet \mathbf{c} \equiv \mathbf{t}_b \bullet \varphi$

Seja $\mathbf{t}_\varphi$ um objeto de tipo $\mathbf{T}_\varphi$ tal que $\mathbf{t}_\varphi = \mathbf{t}_b \bullet \varphi$. Dado que **c** é monovalorado de valor estruturado de tipo $\mathbf{T}_c$ e φ é um caminho direto de $\mathbf{T}_b$, de (1) temos que:

$\mathbf{t}_v \bullet \mathbf{c}$ = **Construtor_**$\mathbf{T}_c$-$\mathbf{T}_\varphi$ ($\mathbf{t}_b \bullet \varphi$)

Caso 5: De ψ_c: $[T_v \bullet \mathbf{c}] \equiv [T_b \bullet \varphi]$ temos que:

(1) $\mathbf{t_v \bullet c} \equiv \mathbf{t_b \bullet \varphi}$

Dado que φ é um caminho virtual de T_b, suponha GeraConsulta (φ, $\mathbf{t_b}$) = < $\mathbf{S_\varphi}$, $\mathbf{F_\varphi}$, $\mathbf{W_\varphi}$ >. Então, temos:

(2) $\mathbf{t_b \bullet \varphi}$ = SELECT $\mathbf{S_\varphi}$ FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$

Dado que **c** é monovalorado de valor estruturado de tipo T_c, de (1) e (2) temos que:

$\mathbf{t_v \bullet c}$ = SELECT **Construtor_**T_c-T_φ ($\mathbf{S_\varphi}$) FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$

Caso 6: De ψ_c: $[T_v \bullet \mathbf{c}] \equiv [T_b \bullet \varphi]$ temos que:

(1) $\mathbf{t_v \bullet c} \equiv \mathbf{t_b \bullet \varphi}$

Seja $\mathbf{t_\varphi}$ um objeto de tipo T_φ tal que $\mathbf{t_\varphi} = \mathbf{t_b \bullet \varphi}$. Dado que **c** é um atributo monovalorado de referência para a visão $\mathbf{V_c}$ de tipo T_c, de (1), temos que:

(2) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c} \in \mathbf{V_c}$ e $\mathbf{t_c} \equiv \mathbf{t_\varphi}$.

Seja $\mathbf{t_c}$ uma instância de T_c. Da ACO de T_c & T_φ, $[T_c,\{\mathbf{c}_1,\mathbf{c}_2,...,\mathbf{c}_w\}] \equiv [T_\varphi,\{\mathbf{d}_1,\mathbf{d}_2,..., \mathbf{d}_w\}]$, temos que:

(3) $\mathbf{t_c} \equiv \mathbf{t_\varphi}$ sss $\mathbf{t_c \bullet c_i} = \mathbf{t_\varphi \bullet d_i}$, para $1 \leq i \leq w$.

De (2) e (3) temos:

(4) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c} \in \mathbf{V_c}$ e $\mathbf{t_c \bullet c_i} = \mathbf{t_\varphi \bullet d_i}$, para $1 \leq i \leq w$.

Dado que φ é um caminho direto de T_b, de (4) temos:

$\mathbf{t_v \bullet c}$ = MAKE_REF[3] ($\mathbf{V_c}$, $\mathbf{t_b \bullet \varphi \bullet d}_1$, $\mathbf{t_b \bullet \varphi \bullet d}_2$,…, $\mathbf{t_b \bullet \varphi \bullet d}_w$)

Caso 7: De ψ_c: $[T_v \bullet \mathbf{c}] \equiv [T_b \bullet \varphi]$ temos que:

(1) $\mathbf{t_v \bullet c} \equiv \mathbf{t_b \bullet \varphi}$

Dado que **c** é um atributo monovalorado de referência para a visão $\mathbf{V_c}$ de tipo T_c, de (1), temos que:

(2) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c} \in \mathbf{V_c}$ e $\mathbf{t_c} \equiv \mathbf{t_b \bullet \varphi}$.

Seja $\mathbf{t_c}$ uma instância de T_c e T_φ o tipo do caminho φ de $\mathbf{t_b}$. Da ACO de T_c & T_φ, $[T_c,\{\mathbf{c}_1,\mathbf{c}_2,...,\mathbf{c}_w\}] \equiv [T_\varphi,\{\mathbf{d}_1,\mathbf{d}_2,..., \mathbf{d}_w\}]$, temos que:

(3) $\mathbf{t_c} \equiv \mathbf{t_b \bullet \varphi}$ sss $\mathbf{t_c \bullet c_i} = \mathbf{t_b \bullet \varphi \bullet d_i}$, para $1 \leq i \leq w$.

De (2) e (3) temos:

(4) $\mathbf{t_v \bullet c}$ = REF ($\mathbf{t_c}$), onde $\mathbf{t_c} \in \mathbf{V_c}$ e $\mathbf{t_c \bullet c_i} = \mathbf{t_b \bullet \varphi \bullet d_i}$, para $1 \leq i \leq w$.

[3] No Oracle 9i, o operador MAKE_REF cria uma referência para uma visão de objetos a partir do nome da visão e de valores que identifiquem unicamente um objeto da visão.

Dado que φ é um caminho virtual de $\mathbf{T}_b$, temos que:

(5) $\mathbf{t}_b \bullet \varphi$ = SELECT $\mathbf{S}_\varphi$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

De (4) e (5), temos que:

$\mathbf{t}_v \bullet \mathbf{c}$ = (SELECT MAKE_REF ($\mathbf{V}_c$, $\mathbf{S}_\varphi \bullet \mathbf{d}_1$, $\mathbf{S}_\varphi \bullet \mathbf{d}_2$,…, $\mathbf{S}_\varphi \bullet \mathbf{d}_w$)

FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$)

Caso 8: De ψ_c: $[\mathbf{T}_v \bullet \mathbf{c}] \equiv [\mathbf{T}_b \bullet \varphi \bullet \mathbf{a}]$ temos que:

(1) $\mathbf{t}_v \bullet \mathbf{c} \equiv \mathbf{t}_b \bullet \varphi \bullet \mathbf{a}$

Dado que **c** é multivalorado de valor atômico, de (1) temos que:

(2) $\mathbf{t}_v \bullet \mathbf{c} = \{ \mathbf{t}_\varphi \bullet \mathbf{a} \mid \mathbf{t}_\varphi \in \mathbf{t}_b \bullet \varphi \}$

Dado que, φ é um caminho multivalorado de $\mathbf{T}_b$, suponha GeraConsulta (φ, $\mathbf{t}_b$) = < $\mathbf{S}_\varphi$, $\mathbf{F}_\varphi$, $\mathbf{W}_\varphi$ >. Então, temos:

(3) $\mathbf{t}_b \bullet \varphi$ = SELECT $\mathbf{S}_\varphi$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

Dado que $\mathbf{T}_c$ é o tipo (*nested table*) de **c**, de (2) e (3), temos que:

$\mathbf{t}_v \bullet \mathbf{c}$ = CAST(MULTISET(SELECT $\mathbf{S}_\varphi \bullet \mathbf{a}$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$) AS $\mathbf{T}_c$)

Caso 9: De ψ_c: $[\mathbf{T}_v \bullet \mathbf{c}] \equiv [\mathbf{T}_b \bullet \varphi]$ temos que:

(1) $\mathbf{t}_v \bullet \mathbf{c} \equiv \mathbf{t}_b \bullet \varphi$

Dado que **c** é multivalorado de valor atômico, de (1) temos que:

(2) $\mathbf{t}_v \bullet \mathbf{c} = \{ \mathbf{t}_\varphi \mid \mathbf{t}_\varphi \in \mathbf{t}_b \bullet \varphi \}$

Dado que, φ é um caminho multivalorado de $\mathbf{T}_b$, suponha GeraConsulta (φ, $\mathbf{t}_b$) = < $\mathbf{S}_\varphi$, $\mathbf{F}_\varphi$, $\mathbf{W}_\varphi$ >. Então, temos:

(3) $\mathbf{t}_b \bullet \varphi$ = SELECT $\mathbf{S}_\varphi$ FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

Dado que $\mathbf{T}_c$ é o tipo (*nested table*) de **c**, de (2) e (3), temos que:

$\mathbf{t}_v \bullet \mathbf{c}$ = CAST(MULTISET(

SELECT $\mathbf{S}_\varphi \bullet$COLUMN_VALUE[4] FROM $\mathbf{F}_\varphi$ WHERE $\mathbf{W}_\varphi$

) AS $\mathbf{T}_c$)

Caso 10: De ψ_c: $[\mathbf{T}_v \bullet \mathbf{c}] \equiv [\mathbf{T}_b \bullet \varphi]$ temos que:

(1) $\mathbf{t}_v \bullet \mathbf{c} \equiv \mathbf{t}_b \bullet \varphi$

Dado que **c** é multivalorado, de (1) temos que:

[4] No Oracle 9i, usamos a expressão COLUMN_VALUE para denotar os elementos de uma nested table, quando seu tipo é atômico

(2) $\mathbf{t_v \bullet c} = \{ \mathbf{t_\varphi} \mid \mathbf{t_\varphi} \in \mathbf{t_b \bullet \varphi} \}$

Seja $\mathbf{t_\varphi}$ um objeto de tipo $\mathbf{T_\varphi}$ tal que $\mathbf{t_\varphi} = \mathbf{t_b \bullet \varphi}$. Dado que φ é um caminho multivalorado de $\mathbf{T_b}$, suponha GeraConsulta (φ, $\mathbf{t_b}$) = < $\mathbf{S_\varphi}$, $\mathbf{F_\varphi}$, $\mathbf{W_\varphi}$ >. Então, temos:

(3) $\mathbf{t_b \bullet \varphi}$ = SELECT $\mathbf{S_\varphi}$ FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$

Dado que **c** é do tipo $\mathbf{T_c}$, uma coleção de objetos estruturados do tipo $\mathbf{T_{item\_c,}}$ de (2) e (3) temos que:

$\mathbf{t_v \bullet c}$ = CAST(MULTISET (

SELECT **Construtor_**$\mathbf{T_{item\_c}}$**-**$\mathbf{T_\varphi}$ ($\mathbf{S_\varphi}$) FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$

) AS $\mathbf{T_c}$)

<u>Caso 11</u>: De ψ_c:[$\mathbf{T_v \bullet c}$] ≡ [$\mathbf{T_b \bullet \varphi}$] temos que:

(1) $\mathbf{t_v \bullet c} \equiv \mathbf{t_b \bullet \varphi}$

Dado que **c** é um atributo multivalorado de referência para a visão $\mathbf{V_c}$ de tipo $\mathbf{T_{item\_c}}$, de (1) temos que:

(2) $\mathbf{t_v \bullet c}$ = {REF ($\mathbf{t_{item\_c}}$) | $\mathbf{t_{item\_c}} \in \mathbf{V_c}$ e $\exists \mathbf{t_\varphi} \in \mathbf{t_b \bullet \varphi}$ e $\mathbf{t_{item\_c}} \equiv \mathbf{t_\varphi}$ }

Seja $\mathbf{t_{item\_c}}$ uma instância de $\mathbf{T_{item\_c}}$ e $\mathbf{t_\varphi}$ uma instância de $\mathbf{T_\varphi}$. Da ACO de $\mathbf{T_{item\_c}}$ & $\mathbf{T_\varphi}$., [$\mathbf{T_{item\_c}}$,{$\mathbf{c_1}$,$\mathbf{c_2}$,...,$\mathbf{c_w}$}]≡[$\mathbf{T_\varphi}$.,{$\mathbf{d_1}$,$\mathbf{d_2}$,..., $\mathbf{d_w}$}], temos que:

(3) $\mathbf{t_{item\_c}} \equiv \mathbf{t_\varphi}$ sss $\mathbf{t_{item\_c} \bullet c_i} = \mathbf{t_\varphi \bullet d_i}$, para $1 \leq i \leq w$.

De (2) e (3) temos:

(4) $\mathbf{t_v \bullet c}$ = {REF ($\mathbf{t_{item\_c}}$) | $\mathbf{t_{item\_c}} \in \mathbf{V_c}$ e $\mathbf{t_\varphi} \in \mathbf{t_b \bullet \varphi}$ e $\mathbf{t_{item\_c} \bullet c_i} = \mathbf{t_\varphi \bullet d_i}$, para $1 \leq i \leq w$. }.

Dado que φ é um caminho multivalorado de $\mathbf{T_b}$, temos que:

(5) $\mathbf{t_b \bullet \varphi}$ = SELECT $\mathbf{S_\varphi}$ FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$

Dado que e $\mathbf{T_c}$ é o tipo (*nested table*) de **c**, De (4) e (5), temos que:

$\mathbf{t_v \bullet c}$ = CAST (MULTISET (

SELECT MAKE_REF ($\mathbf{V_c}$, $\mathbf{S_\varphi \bullet d_1}$, ..., $\mathbf{S_\varphi \bullet d_w}$)

FROM FROM $\mathbf{F_\varphi}$ WHERE $\mathbf{W_\varphi}$) AS $\mathbf{T_c}$)

Exemplo 5.4

Considere o esquema do banco de dados **Departamentos** (Figura 5.9), o esquema da visão de objetos **Departamentos_v** (Figura 5.10) e as AC's de **Departamentos_v** (Figura 5.11). O construtor **Construtor_**$\mathbf{T_{departamento\_v}}$**-**$\mathbf{T_{departamento}}$ gerado pelo algoritmo é mostrado na

Figura 5.12. A seguir, mostramos como foi gerado o código SQL:1999 para cada atributo de $\mathbf{T}_{departamento\_v}$. No resto desta seção, seja $\mathbf{t}_{departamento\_v}$ o objeto criado por esse construtor a partir do objeto base $\mathbf{t}_{departamento}$.

– O atributo codv (linha 2 - Figura 5.12) é um atributo monovalorado de valor atômico definido pelas AC ψ_3: $[\mathbf{T}_{departamento\_v}\bullet\mathbf{codv}] \equiv [\mathbf{T}_{departamento}\bullet\mathbf{sigla}]$. Do Caso 1 da Tabela 5.3, temos que $\mathbf{t}_{departamento_v}\bullet$codv = $\mathbf{t}_{departamento}\bullet$sigla.

– O atributo nome (linha 3 - Figura 5.12) é um atributo monovalorado de valor atômico definido pela AC ψ_4: $[\mathbf{T}_{departamento\_v}\bullet\mathbf{nome}] \equiv [\mathbf{T}_{departamento}\bullet\mathbf{nome}]$. Do Caso 1 da Tabela 5.3, temos que $\mathbf{t}_{departamento_v}\bullet$nome = $\mathbf{t}_{departamento}\bullet$nome.

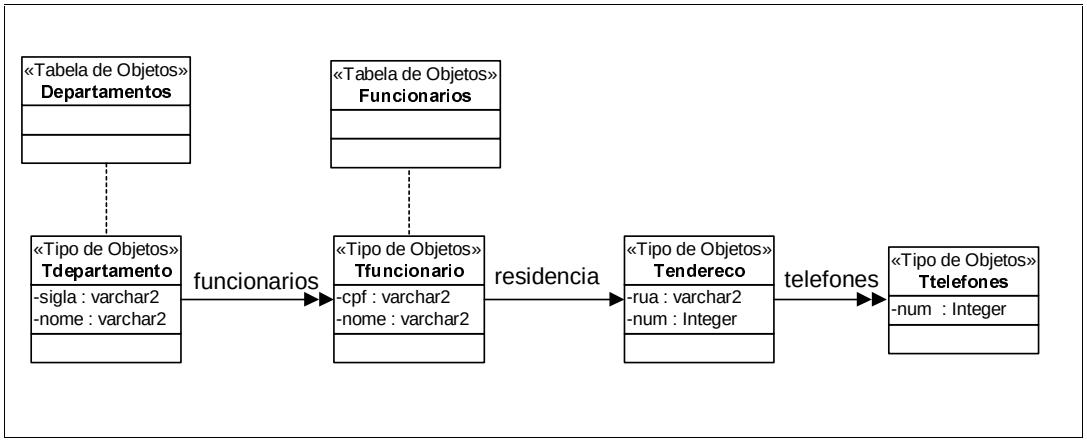

Figura 5.9: Esquema do Banco de **Departamentos**

– O atributo funcionarios (linhas 4 a 18 - Figura 5.12) é um atributo multivalorado de valor estruturado definido pela AC ψ_5:$[\mathbf{T}_{departamento\_v}\bullet\mathbf{funcionarios}] \equiv [\mathbf{T}_{departamento}\bullet\mathbf{funcionarios}]$, onde $\mathbf{T}_{funcionarios\_v}$ é o tipo do atributo funcionarios, $\mathbf{T}_{funcionario\_v}$ é o tipo dos objetos de funcionarios e $\mathbf{T}_{funcionario}$ é o tipo do caminho funcionarios, satisfazendo, assim, o Caso 10 da Tabela 5.3. Dado que φ=funcionarios é um caminho multivalorado, GeraConsulta(φ, $\mathbf{t}_{funcionario}$) retorna: $\mathbf{S}_{\varphi}$="$\mathbf{t}_{funcionario}\bullet$COLUMN_VALUE", $\mathbf{F}_{\varphi}$="TABLE($\mathbf{t}_{departamento}\bullet\mathbf{funcionarios}$) $\mathbf{t}_{funcionario}$" e $\mathbf{W}_{\varphi}$ tem valor nulo. Assim, do Caso 10 da Tabela 5.3 temos que:

$\mathbf{t}_{departamento_v}\bullet$funcionarios =

```
CAST ( MULTISET (
        SELECT Construtor_T_funcionario_v-T_funcionario (t_funcionario•COLUMN_VALUE)
        FROM TABLE( t_departamento•funcionarios ) t_funcionario )
) AS T_funcionarios_v )
```

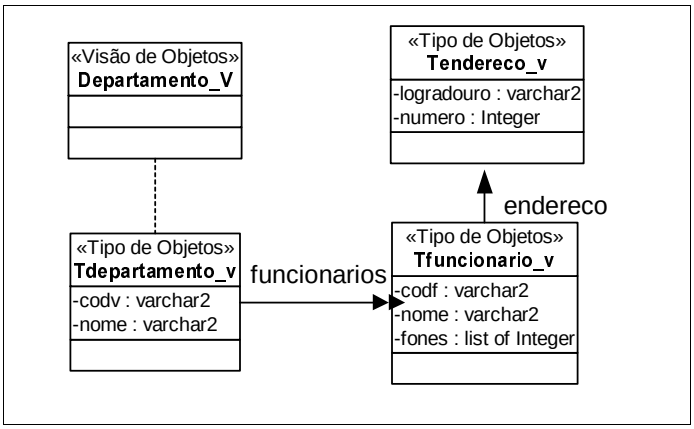

Figura 5.10: Esquema da Visão de Objetos **Departamento_v**.

ACE da visão **Departamentos_v**

ψ_1: [**Departamento_V**] ≡ [**Departamentos**]

ACO de $\mathbf{T}_{departamento\_v}$ e $\mathbf{T}_{departamento}$

ψ_2: [$\mathbf{T}_{departamento\_v}$ {**codv**}] ≡ [$\mathbf{T}_{departamento}$ {**sigla**}]

ACC's de $\mathbf{T}_{departamento\_v}$ e $\mathbf{T}_{departamento}$

ψ_3: [$\mathbf{T}_{departamento\_v}$•**codv**] ≡ [$\mathbf{T}_{departamento}$•**sigla**]

ψ_4: [$\mathbf{T}_{departamento\_v}$•**nome**] ≡ [$\mathbf{T}_{departamento}$•**nome**]

ψ_5: [$\mathbf{T}_{departamento\_v}$•**funcionarios**] ≡ [$\mathbf{T}_{departamento}$•**funcionarios**]

ACO de $\mathbf{T}_{funcionario\_v}$ e $\mathbf{T}_{funcionario}$

ψ_6: [$\mathbf{T}_{funcionario\_v}$ {**codf**}] ≡ [$\mathbf{T}_{funcionario}$ {**cpf**}]

ACC's de $\mathbf{T}_{funcionario\_v}$ e $\mathbf{T}_{funcionario}$

ψ_7: [$\mathbf{T}_{funcionario\_v}$•**codf**] ≡ [$\mathbf{T}_{funcionario}$•**cpf**]

ψ_8: [$\mathbf{T}_{funcionario\_v}$•**nome**] ≡ [$\mathbf{T}_{funcionario}$•**nome**]

ψ_9: [$\mathbf{T}_{funcionario\_v}$•**endereco**] ≡ [$\mathbf{T}_{funcionario}$•**residencia**]

ACC's de $\mathbf{T}_{endereco\_v}$ e $\mathbf{T}_{endereco}$

ψ_{11}: [$\mathbf{T}_{endereco\_v}$ •**logradouro**] ≡ [$\mathbf{T}_{endereco}$•**rua**]

ψ_{12}: [$\mathbf{T}_{endereco\_v}$•**numero**] ≡ [$\mathbf{T}_{endereco}$•**num**]

Figura 5.11: AC's da visão **Departamentos_v**

Os valores dos atributos do construtor **Construtor_**$\mathbf{T}_{funcionario\_v}$**-**$\mathbf{T}_{funcionario}$ são definidos pelas ACC's de $\mathbf{T}_{funcionario\_v}$**&**$\mathbf{T}_{funcionario}$, como mostrado a seguir. No resto desta seção, considere $\mathbf{t}_{funcionario\_v}$ o objeto criado pelo construtor a partir do objeto base em $\mathbf{t}_{funcionario}$•COLUMN_VALUE.

– O atributo codf (linha 6 - Figura 5.12) é monovalorado de valor atômico definido pela ACC ψ_7:[$T_{funcionario\_v}$•**codf**] ≡ [$T_{funcionario}$•**cpf**]. Do Caso 1 da Tabela 5.3, temos que: $t_{funcionario\_v}$• codf = $t_{funcionario}$•COLUMN_VALUE •cpf.

– O atributo nome (linha 7 - Figura 5.12) é monovalorado de valor atômico definido pela ACC ψ_8:[$T_{funcionario\_v}$•**nome**] ≡ [$T_{funcionario}$•**nome**]. Do Caso 1 da Tabela 5.3, temos que: $t_{funcionario\_v}$• nome = $t_{funcionario}$•COLUMN_VALUE •nome.

– O atributo endereco (linhas 8 a 11 - Figura 5.12) é um atributo monovalorado de valor estruturado definido pela ACC ψ_9:[$T_{funcionario\_v}$•**endereco**] ≡ [$T_{funcionario}$•**residencia**], onde $T_{endereco\_v}$ é o tipo de endereco e $T_{endereco}$ é o tipo do caminho residencia. Pelo Caso 4 da Tabela 5.3, temos que: $t_{funcionario\_v}$•endereco =

Construtor_$T_{endereco\_v}$**-**$T_{endereco}$ (**t**$_{funcionario}$•COLUMN_VALUE•residencia)

```
CREATE VIEW Departamentos_v of T departamento_v
WITH OBJECT IDENTIFIER (codv) AS

   SELECT
1.    T departamento_v(                                  Construtor_T departamento_v-T departamento
2.         t departamento•sigla,
3.         t departamento•nome,
4.         CAST ( MULTISET ( SELECT
                                                         Construtor_T funcionario_v-T funcionario
5.              T funcionario_v (
6.                   t funcionario•COLUMN_VALUE•cpf,
7.                   t funcionario•COLUMN_VALUE•nome,
                                                         Construtor_T endereco_v-T endereco
8.                   T endereco_v (
9.                      t funcionario•COLUMN_VALUE•residencia•rua,
10.                     t funcionario•COLUMN_VALUE•residencia•num
11.                  ),
12.                  CAST ( MULTISET (
13.                       SELECT t telefone•num
14.                       FROM TABLE (t funcionario•COLUMN_VALUE•residencia•telefones) t telefone
15.                  ) AS T telefones_v )
16.             FROM TABLE( t departamento•funcionarios ) t funcionario
17.        )
18.   ) AS T funcionarios_v )

19.  )
      FROM Departamentos t departamento
```

Figura 5.12 : Definição da Visão de Objetos **Departamentos_v**

Os valores dos atributos do construtor **Construtor_**$T_{endereco\_v}$**-**$T_{endereco}$ são definidos pelas ACC's de $T_{endereco\_v}$ & $T_{endereco}$. No resto desta seção, considere $t_{endereco\_v}$ o objeto criado por esse construtor a partir do objeto em $t_{funcionario}$•COLUMN_VALUE•residencia.

– O atributo logradouro (linha 9 - Figura 5.12) é monovalorado de valor atômico definido pelas ACC ψ_{11}:[$T_{endereco\_v}$ •**logradouro**] ≡ [$T_{endereco}$•**rua**]. Pelo Caso 1 da Tabela 5.3, temos que: $t_{endereco\_v}$•logradouro = $t_{funcionario}$•COLUMN_VALUE•residencia •rua.
– O atributo numero (linha 10 - Figura 5.12) é monovalorado de valor atômico definido pelas ACC ψ_{12}:[$T_{endereco\_v}$•**numero**] ≡ [$T_{endereco}$•**num**]. Pelo Caso 1 da Tabela 5.3, temos que: $t_{endereco\_v}$•numero = $t_{funcionario}$•COLUMN_VALUE•residencia•num.

Exemplo 5.5

Considere o esquema do banco de dados **Deptos** apresentado na Figura 5.13, o esquema da visão de objetos **Projetos_v** (Figura 5.14) e as AC's dessa visão (Figura 5.15). O construtor **Construtor_**$T_{projeto\_v}$**-**$T_{projeto}$ gerado pelo algoritmo é mostrado na Figura 5.12. A seguir, mostramos como foi gerado o código SQL:1999 para cada atributo de $T_{projeto\_v}$. No resto desta seção, seja $t_{projeto\_v}$ o objeto criado por esse construtor a partir do objeto base $t_{projeto}$.

– O atributo codigo (linha 2 - Figura 5.16) é monovalorado de valor atômico definido pela ACC ψ_3:[$T_{projeto\_v}$•**codigo**] ≡ [$T_{projeto}$•**cod**]. Do Caso 1 da Tabela 5.3, temos que: $t_{projeto\_v}$•codigo = $t_{projeto}$•cod.
– O atributo titulo (linha 3 - Figura 5.16) é monovalorado de valor atômico definido pela ACC ψ_4: [$T_{projeto\_v}$•**titulo**] ≡ [$T_{projeto}$•**titulo**]. Do Caso 1 da Tabela 5.3, temos que: $t_{projeto\_v}$•titulo = $t_{projeto}$•titulo.
– O atributo departamento (linhas 5 e 6 - Figura 5.16) é monovalorado de valor atômico definido pela ACC ψ_5: [$T_{projeto\_v}$•**departamento**] ≡ [$T_{projeto}$•**gerente**•**funcionarios**$^{-1}$•**nome**], onde φ = gerente•funcionarios^{-1} é um caminho virtual, satisfazendo, assim, o Caso 3 da Tabela 5.3. Dado que GeraConsulta(φ, $t_{projeto}$) retorna:

S_φ="ℓ_2", F_φ="**Depto** ℓ_2" e W_φ="$t_{projeto}$•**gerente** in (SELECT * FROM TABLE(ℓ_2•**funcionarios**))"; então, do Caso 3 da Tabela 5.3 temos que:

$\mathbf{t}_{projeto\_v}$•departamento =

(SELECT ℓ_2•**nome** FROM **Depto** ℓ_2

WHERE $\mathbf{t}_{projeto}$•**gerente** in (SELECT * FROM TABLE (ℓ_2•**funcionarios**)))

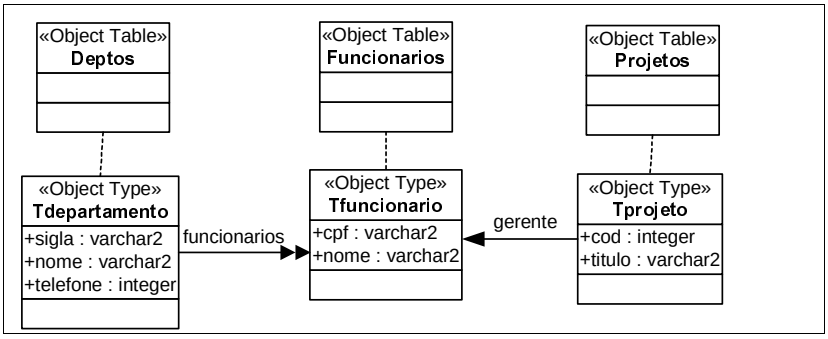

Figura 5.13 : Esquema do Banco de Dados **Deptos**

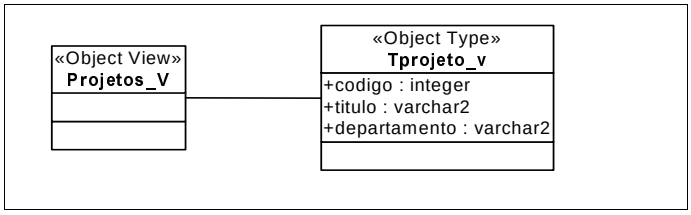

Figura 5.14: Esquema da Visão de Objetos **Projetos_v**

ACE da visão **Projetos_V**

ψ_1: [**Projetos_V**] ≡ [**Projetos**]

ACO de $\mathbf{T}_{projeto\_v}$ e $\mathbf{T}_{projeto}$

ψ_2: [$\mathbf{T}_{projeto\_v}$ {**codigo**}] ≡ [$\mathbf{T}_{projeto}$ {**cod**}]

ACC's de $\mathbf{T}_{projeto\_v}$ e $\mathbf{T}_{projeto}$

ψ_3 : [$\mathbf{T}_{projeto\_v}$•**codigo**] ≡ [$\mathbf{T}_{projeto}$•**cod**]

ψ_4 : [$\mathbf{T}_{projeto\_v}$•**titulo**] ≡ [$\mathbf{T}_{projeto}$•**titulo**]

ψ_5 : [$\mathbf{T}_{projeto\_v}$•**departamento**] ≡ [$\mathbf{T}_{projeto}$•**gerente**•**funcionarios**$^{-1}$•**nome**]

Figura 5.15: Assertivas de Correspondências da Visão **Projetos_v**

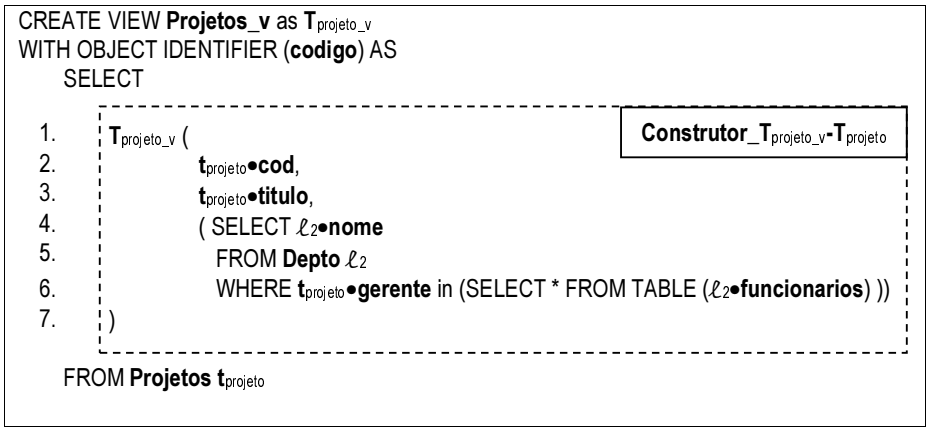

Figura 5.16: Definição da visão **Projetos_v**

Exemplo 5.6

Cosidere o esquema do banco de dados **Funcionarios** (Figura 5.17), o esquema da visão de objetos **Chefes_v** (Figura 5.18) e as AC's dessa visão (Figura 5.19). O construtor **Construtor_**T_{chefe_v} **-**$T_{funcionario}$ gerado pelo algoritmo é mostrado na Figura 5.20. A seguir, mostramos como foi gerado o código SQL:1999 para cada atributo de T_{chefe_v}. No resto desta seção, seja t_{chefe_v} o objeto criado pelo construtor a partir do objeto base $t_{funcionario}$.

- O atributo cpf (linha 2 - Figura 5.20) é monovalorado de valor atômico definido pela ACC ψ_3: $[T_{chefe\_v}\bullet\mathbf{cpf}] \equiv [T_{funcionario}\bullet\mathbf{cpf}]$. Do Caso 1 da Tabela 5.3 temos que: $t_{chefe_v}\bullet$cpf = $t_{funcionario}\bullet$cpf.
- O atributo nome (linha 3 - Figura 5.20) é monovalorado de valor atômico definido pela ACC ψ_4 $[T_{chefe\_v}\bullet\mathbf{nome}] \equiv [T_{funcionario}\bullet\mathbf{nome}]$. Do Caso 1 da Tabela 5.3 temos que: $t_{chefe_v}\bullet$nome = $t_{funcionario}\bullet$nome.
- O atributo projetos (linhas 4 a 9 - Figura 5.20) é multivalorado de referência, definido pela ACC ψ_5: $[T_{chefe\_v}\bullet\mathbf{projetos}] \equiv [T_{funcionario}\bullet\mathbf{chefe}^{-1}\bullet\mathbf{projetos}]$, onde $T_{projetos\_v}$ é o tipo do atributo projetos, Projetos_v (de tipo $T_{projeto\_v}$) é a visão de objetos referenciada por projetos, $T_{projeto}$ é o tipo dos objetos de chefe^{-1}•projetos e ψ_6 : $[T_{projeto\_v}\{\mathbf{codigo}\}] \equiv [T_{projeto}\{\mathbf{cod}\}]$ a ACO de $T_{projeto\_v}$ & $T_{projeto}$, satisfazendo assim o Caso 11 da Tabela 5.3. Dado que φ= chefe^{-1}•projetos, onde chefe^{-1} é uma ligação virtual e projetos uma ligação multivalorada, GeraConsulta(φ, $t_{funcionario}$) retorna: $\mathbf{S}_\varphi$="ℓ_2•COLUMN_VALUE", $\mathbf{F}_\varphi$="**Departamentos** ℓ_1, TABLE (ℓ_1•**projetos**) ℓ_2" e $\mathbf{W}_\varphi$="ℓ_1•**chefe** = REF ($t_{funcionario}$)".
  Assim, do Caso 11 da Tabela 5.3, temos que:

$\mathbf{t}_{chefe\_v}$•projetos =

```
CAST ( MULTISET (
        SELECT
                MAKE_REF ( Projetos_v, ℓ2•COLUMN_VALUE•cod )
        FROM Departamentos ℓ1, TABLE (ℓ1•projetos) ℓ2
        WHERE ℓ1•chefe = REF (tfuncionario )
) AS Tprojetos_v )
```

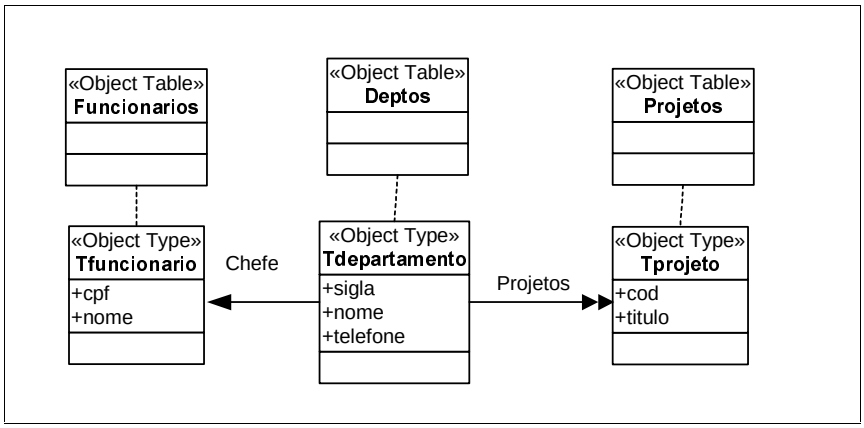

Figura 5.17: Esquema do Banco de Dados **Funcionarios**

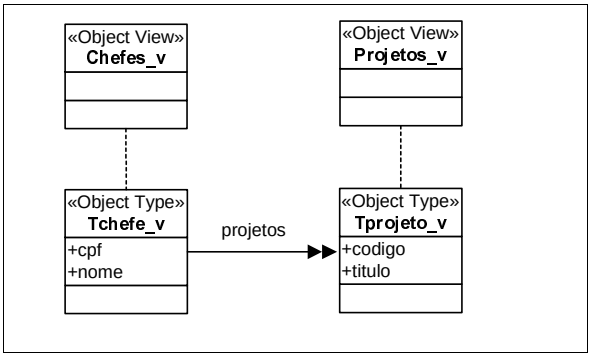

Figura 5.18: Esquema da Visão de Objetos **Chefe_v**

ACE da visão **Chefes_v**

ψ_1: **Chefes_v** ≡ **Funcionarios**

ACO de $\mathbf{T}_{chefe\_v}$ e $\mathbf{T}_{funcionario}$

ψ_2 : [$\mathbf{T}_{chefe\_v}$ { **cpf** }] ≡ [$\mathbf{T}_{funcionario}$ { **cpf** }]

ACC's de $\mathbf{T}_{chefe\_v}$ e $\mathbf{T}_{funcionario}$

ψ_3 : $\mathbf{T}_{chefe\_v}$•**cpf** ≡ $\mathbf{T}_{funcionario}$•**cpf**

ψ_4 : $\mathbf{T}_{chefe\_v}$•**nome** ≡ $\mathbf{T}_{funcionario}$•**nome**

ψ_5 : $\mathbf{T}_{chefe\_v}$ •**projetos** ≡ $\mathbf{T}_{funcionario}$•**chefe**$^{-1}$•**projetos**

ACO de $\mathbf{T}_{projeto\_v}$ e $\mathbf{T}_{projeto}$

ψ_6 : [$\mathbf{T}_{projeto\_v}$ { **codigo** }] ≡ [$\mathbf{T}_{projeto}$ { **cod** }]

ACC's de $\mathbf{T}_{projeto\_v}$ e $\mathbf{T}_{projeto}$

ψ_7 : $\mathbf{T}_{projeto\_v}$•**codigo** ≡ $\mathbf{T}_{projeto}$•**cod**

ψ_8 : $\mathbf{T}_{projeto\_v}$•**titulo** ≡ $\mathbf{T}_{projeto}$•**titulo**

Figura 5.19: Assertivas de Correspondênca da Visão **Chefe_v**

```
CREATE VIEW Chefes_v of T_chefe_v
WITH OBJECT IDENTIFIER ( cpf ) AS
    SELECT
1.      T_chefe_v (                                   Construtor_T_chefe_v-T_funcionario
2.          t_funcionario•cpf,
3.          t_funcionario•nome,
4.          CAST ( MULTISET (
5.                  SELECT MAKE_REF ( Projetos_v, ℓ2•COLUMN_VALUE•cod )
6.                  FROM Departamentos ℓ1,
7.                      TABLE (ℓ1•projetos) ℓ2
8.                  WHERE ℓ1•chefe = REF (t_funcionario )
9.          ) AS T_projetos_v )
10.     )
    FROM Funcionarios t_funcionario
```

Figura 5.20: Definição da Visão **Chefe_v**

Capítulo 6
XQueryTranslator: O Processador de Consultas *d*o XML *Publisher*

Neste capítulo, discutimos a implementação do *XQueryTranslator*, o processador de consultas *XQuery* do XML *Publisher*. Iniciamos com uma breve discussão sobre o processamento de consultas em sistemas gerenciadores de bancos de dados (SGBD's) convencionais. Em seguida, descrevemos as fases do processamento de consultas no XML *Publisher*, estabelecendo um paralelo com o processamento de consultas em SGBD's convencionais. Na Seção 6.3, abordamos os tipos de consultas *XQuery* que podem ser processadas no *XQueryTranslator*. Nas demais seções, discutimos os módulos que compõem a arquitetura do *XQueryTranslator* e apresentamos os algoritmos que implementam suas funcionalidades.

6.1 Processamento de Consultas em SGBD's Convencionais

Em um SGBD relacional, objeto-relacional ou orientado a objeto, o interesse dos usuários em obter dados é especificado através de consultas descritas em uma linguagem de alto nível, tal como SQL. Consultas descritas numa linguagem de alto nível são chamadas de consultas declarativas, pois expressam apenas o resultado a ser obtido, não expressando a forma como extrair esse resultado. Dessa forma, fica para o SGBD a tarefa de produzir os resultados de uma consulta da maneira mais eficiente possível. Ao conjunto de atividades envolvidas na extração de dados de um SGBD dá-se o nome de processamento de consultas [48].

As atividades envolvidas no processamento de consultas são: *parsing,* tradução, otimização e execução. Na fase de *parsing*, ocorre a análise léxica e sintática da

consulta. Em seguida, a consulta deve ser analisada de uma forma a verificar se todos os atributos e nomes de tabelas (ou visões) são válidos e semanticamente significativos no esquema do banco de dados consultado [17]. Para que o processamento de uma consulta possa continuar, o SGBD precisa traduzi-la numa forma intermediária mais adequada de manipulação, que normalmente é uma representação algébrica da consulta. A representação intermediária torna fácil explorar formas alternativas de uma consulta (planos de execução) para recuperar seus resultados nos arquivos do SGBD da forma mais eficiente possível. O processo de escolha de um plano de execução que deverá exigir menos tempo para a execução da consulta é conhecido como otimização de consulta. Finalmente, na fase de execução, o plano de consulta escolhido na fase de otimização é executado e o resultado da consulta é então gerado.

6.2 Processamento de Consultas no *XQueryTranslator*

Como discutido anteriormente, as visões XML publicadas no XML *Publisher* podem ser consultadas usando a linguagem *XQuery* [67]. O processamento de consultas *XQuery* no XML *Publisher* é realizado pelo *XQueryTranslator*. Basicamente, o *XQueryTranslator* é um software para tradução de consultas *XQuery* em SQL:1999 e transformação de objetos em XML. Como ilustrado na Figura 6.1, a arquitetura do *XQueryTranslator* é composta pelos módulos Validador, Simplificador, Tradutor e Gerador, sendo que as fases envolvidos no processamento de consultas *XQuery* no *XQueryTranslator* são: Parsing, Tradução, Execução e Transformação.

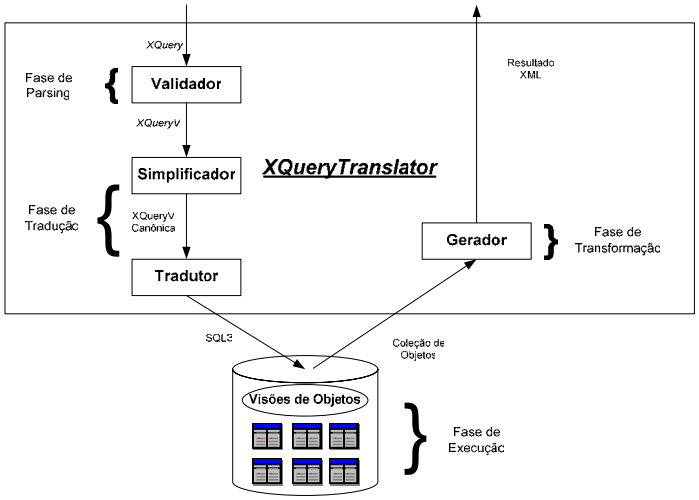

Figura 6.1: Arquitetura do ***XQueryTranslator***

6.2.1 Fase de *Parsing*

No *XQueryTranslator*, assim como no processamento de consultas em SGBD's convencionais, na fase de *parsing* ocorre a análise léxica e sintática da consulta *XQuery*. Em seguida, ocorre a análise semântica, ou validação, onde são verificados se todas as expressões de caminhos (*XPath*) da consulta são válidas e semanticamente significativas para o XML *Schema* da respectiva visão XML. A validação nos permite avaliar com mais precisão erros de sintaxe e semântica na consulta *XQuery* entrada. Se a validação não existisse, a consulta *XQuery* seria traduzida erroneamente em SQL:1999 durante a fase de tradução. Sendo assim, os erros seriam detectados durante a fase de execução da consulta SQL:1999 e a mensagem de erro gerada pelo SGBD talvez não fosse suficiente para depurar o erro na *XQuery*.

A fase de *parsing* no *XQueryTranslator* acontece no módulo Validador. Sua implementação não é discutida nesta dissertação. Maiores detalhes sobre as análises léxica, sintática e semântica podem ser encontrados em livros especializados em compiladores [2]. Para este trabalho, é suficiente saber que toda consulta *XQuery* validada corretamente pelo módulo Validador é denominada consulta *XQuery* Válida (XQueryV). Na Seção 6.3 descrevemos em que condições uma consulta *XQuery* é válida.

6.2.2 Fase de Tradução

Na fase de tradução a consulta é traduzida numa consulta SQL:1999, que é a linguagem de consulta para bancos de dados objeto-relacionais. A fase de tradução no *XQueryTranslator*, dá-se em duas etapas. A primeira etapa é executada no módulo Simplificador, que traduz uma consulta *XQueryV* numa forma intermediária mais adequada de manipulação. Chamamos essa representação intermediária de consulta *XQueryV Canônica* (*XQueryVC*). Como veremos mais adiante, a representação *XQueryVC* torna a tradução para SQL:1999 simples e direta. Na Seção 6.4.1 apresentamos um dos algoritmos implementados nesse módulo. Esse algoritmo traduz consultas *XPath* em consultas *XQueryVC*.

A etapa seguinte de tradução é executada no módulo Tradutor, no qual a consulta *XQueryVC* é diretamente traduzida numa consulta SQL:1999 sobre a VOD. Nessa tradução, as cláusulas *for*, *where* e *return* da consulta *XQueryVC* são diretamente mapeadas nas cláusula *FROM, WHERE e SELECT*, respectivamente, da consulta SQL:1999. Na Seção 6.5 discutimos detalhadamente o algoritmo de tradução.

6.2.3 Fase de Execução

A fase de execução da consulta *XQuery* no *XQueryTranslator* corresponde então ao processamento da consulta SQL:1999 no SGBD. Observe que antes dessa fase não existe uma fase de otimização, como no processamento de consultas em SGBD's convencionais, e a consulta SQL:1999 gerada na fase de tradução é diretamente executada no SGBD. Cabe ao SGDB, então, otimizar a consulta SQL:1999, quando possível.

6.2.4 Fase de Transformação

Finalmente, na fase de transformação, o resultado da consulta SQL:1999 é transformado no documento ou fragmento XML especificado pela consulta *XQuery* de entrada. Após a execução da consulta SQL:1999 gerada pelo módulo Tradutor, seu resultado é manipulado no módulo Gerador. Nesse módulo, a coleção de objetos extraída do banco de dados é transformada em XML a partir da consulta *XQueryVC* e do XML *Schema* da respectiva visão XML. Na Seção 6.6 apresentamos o algoritmo utilizado por esse módulo para a geração do resultado XML final.

6.3 Consultas *XQuery* Válidas

A linguagem de consulta aceita pelo XML *Publisher* é a *XQuery*. Essa linguagem foi escolhida por ser largamente aceita, além do que está se tornando um

padrão para consulta a dados XML. No entanto, várias características de *XQuery* são difíceis ou impossíveis de traduzir em SQL [31], devido às diferenças semânticas entre XML e o modelo relacional ou objeto-relacional. Nesse trabalho não identificamos nem discutimos todas as características de *XQuery* que podem ou não ser traduzidas em SQL:1999. No entanto, limitamos a classe de consultas *XQuery* aceitas pelo *XQueryTranslatror*.

As consultas que podem ser processadas no *XQueryTranslator* são chamadas de consultas *XQuery* Válidas (*XQueryV*), ou seja, consultas que podem ser validadas pelo módulo Validador; e sua gramática é apresentada na Figura 6.2. Observe que a *XQueryV* abrange desde simples consultas *XPath* abreviadas, ou expressões de caminho absoluto (AbsolutePath), até expressões *for-where-return* (FWRExpr) com vários níveis de aninhamento.

```
[1] Query ::= AbsolutePath | InitialFWR | "<"ElmtName ">" "{" AbsolutePath "}" "</" ElmtName ">"
                | "<" ElmtName ">" "{" InitialFWR "}" "</" ElmtName ">"
[2] AbsolutePath ::= "view" "(" StringLiteral ")" CPath ( "/" "text" "(" ")" )?
[3] CPath ::= ("/" ElmtName ( "[" LogicExpr "]" )? )+
[4] RPath ::= "/" ElmtName ( "/" ElmtName )*
[5] VPath ::= "$" Var RPath
[6] PathExpr ::= VPath ( "/" "text" "(" ")" )?
[7] Var ::= ElmtName
[8] InitialFWR ::= "for" "$" Var "in" "view" "(" StringLiteral ")" RPath ( "," "$" Var "in" VPath )*
                   WhereExpr? ReturnExpr
[9] WhereExpr ::= "where" LogicExpr
[10] ReturnExpr ::= "return" ElmtConstructor
[11] ElmtConstructor ::= "<" ElmtName ">" "{" ElmtContent "}" "</" ElmtName ">"  |  ElmtContent
[12] ElmtContent ::=  ElmtConstructor | Expr | "("Expr ( ","Expr )* ")"
[13] Expr ::= FWRExpr | PathExpr
[14] FWRExpr ::= "for" "$" Var "in" VPath ( "," "$" Var "in" VPath )*
                   WhereExpr? ReturnExpr
[15] LogicExpr ::= OrExpr
[16] OrExpr ::= AndExpr ( "or" AndExpr )*
[17] AndExpr ::= ComparasionExpr ( "and" ComparasionExpr )*
[18] ComparasionExpr ::= ValueExpr GeneralComp ValueExpr
[18] ValueExpr ::= VPath | PrimaryExpr
[19] PrimaryExpr ::= Literal | ParenthesizedExpr
[20] ParenthesizedExpr ::= "(" LogicExpr? ")"
[21] GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[22] Literal ::= IntegerLiteral | DoubleLiteral | StringLiteral
[23] ElmtName ::= /* nome válido para elemento na sintaxe XML */
[24] StringLiteral ::= /* corresponde ao tipo string de XML Schema */
[25] IntegerLiteral ::= /* corresponde ao tipo interger de XML Schema*/
[26] DoubleLiteral ::= /* corresponde ao tipo double de XML Schema */
```

Figura 6.2: Gramática da *XQueryV*

Nesse subconjunto de *XQuery*, definimos uma nova função de entrada chamada "**view**" (produções [2] e [8] da Figura 6.2). Essa função recebe como parâmetro o nome de uma VOD e retorna a instância da respectiva visão XML. ***View*** é a única função de entrada permitida em consultas *XQueryV*, e deve ocorrer exatamente uma vez em cada consulta. Dessa forma, não é possível escrever consultas *XQueryV* para extrair dados de duas ou mais visões XML, o que impossibilita a realização de operações como a junção entre visões XML no *XQueryTranslator*.

A seguir, apresentamos alguns exemplos de consultas *XQueryV*. Nos exemplos, usamos a visão XML Jornais cujo esquema é apresentado na Figura 6.3.

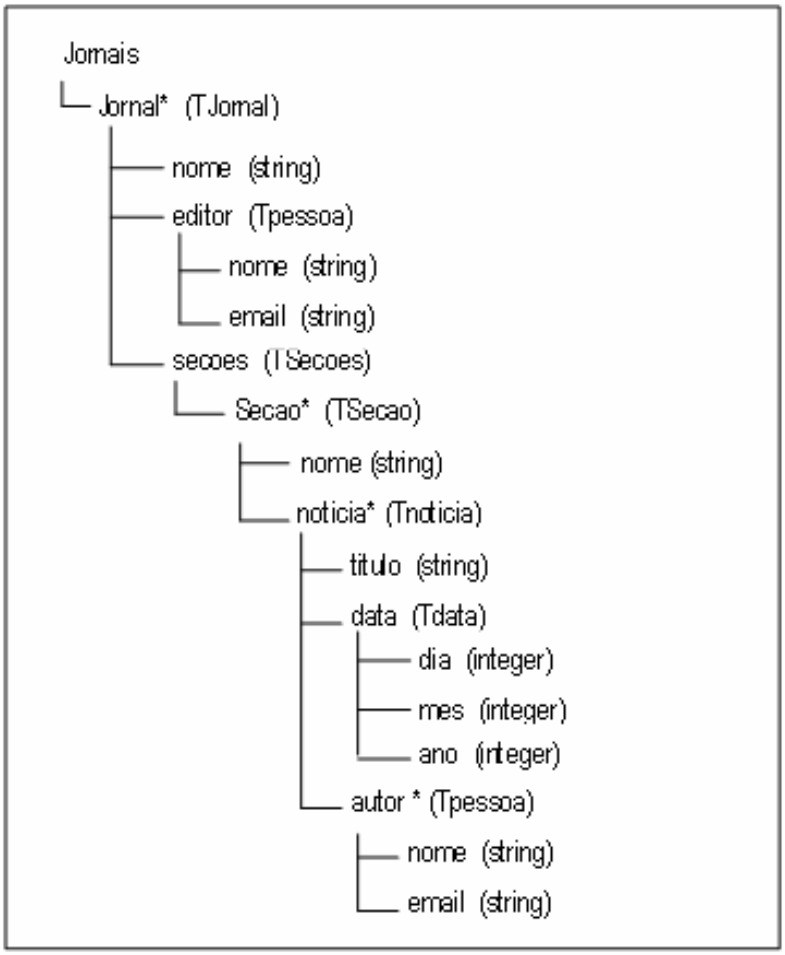

Figura 6.3 : Esquema da visão XML Jornais

Exemplo 6.1

A consulta *XQueryV* abaixo, demonstra como utilizar uma expressão de caminho absoluto para extrair da visão Jornais os editores dos jornais "O Povo" e "O Globo".

```
view("Jornais")/Jornais/Jornal [nome = "O Povo" or nome = "O Globo"]/editor
```

Figura 6.4: Consulta *XQueryV* Q_1

Como apresentado abaixo, o resultado de Q_1 não é um documento XML bem-formado, e sim uma coleção de elementos XML (fragmento XML).

```
<editor>
        <nome>Danilo</nome>
        <email>danilo@opovo.com.br</email>
</editor>
<editor>
        <nome>Maria</nome>
        <email>mariazinha@globo.com.br</email>
</editor>
```

Figura 6.5: Resultado de Q_1.

Exemplo 6.2

A expressão *XQueryV* a seguir, constrói um documento XML bem-formado, composto pelas notícias da seção "policial" do jornal "Meio Norte", onde o elemento raiz é Policiais.

```
<Policiais>{
    view("Jornais")/Jornais/Jornal [nome = "Meio Norte"]/Secoes/Secao [ nome= "policial"]/noticia
}</Policiais>
```

Figura 6.6: Consulta *XQueryV* Q_2

```
<Policiais>
        <noticia>
                <titulo>Assalto a Banco</titulo>
                <data>
                        <dia>30</dia>
                        <mes>01</mes>
                        <ano>2003</ano>
                </data>
                <autor>
                        <nome>Francisca</nome>
                        </email>chica@jmn.com.br</email>
                </autor>
        </noticia>
        ...
</Policiais>
```

Figura 6.7: Resultado de Q_2

Exemplo 6.3

A consulta *XQueryV* a seguir é uma expressão *for-where-return* que retorna o nome dos jornais e os títulos de suas respectivas notícias publicadas no ano de 2004.

```
for $i in view("Jornais")/Jornais/Jornal
return
    <Jornal>{
        $i/nome,
        for $j in $i/Secoes/Secao/noticia
        where $j/data/ano = "2004"
        return
                <Noticia> { $j/titulo/text() } </Noticia>
    }</Jornal>
```

Figura 6.8: Consulta *XQueryV* Q_3

Como apresentado a seguir, o resultado é um conjunto de elementos Jornal, cada um contendo o nome do jornal e um conjunto de Noticia.

```
<Jornal>
        <nome>Meio Norte</nome>
        <Noticia>Salário Mínimo Aumenta R$ 10,00 </Noticia>
        <Noticia>A vista do Presidente Lula a China</Noticia>
</Jornal>
<Jornal>
        <nome>Pasquim</nome>
</Jornal>
...
```

Figura 6.9: Resultado de Q_3

Exemplo 6.4

A consulta *XQueryV* abaixo é uma expressão *for-where-return* que gera um documento XML bem-formado. Esse documento contém as notícias (manchete) do jornal "O Globo" publicadas no ano de "2004" e seus respectivos autores (reporter). E o elemento raiz do documento gerado é Globo:

```
<Globo>{
      for $i in view("Jornais")/Jornais/Jornal
          $j in $i/Secoes/Secao/noticia,
          $l in $j/autor
      where $j/data/ano = "2004" and $i/nome = "O Globo"
      return
            <manchete>{
                  <reportagem>{ $j/nome/text() }</reportagem>
                  <reporter>{ $l/nome/text() } </reporter>
            }</manchete>
}</Globo>
```

Figura 6.10: Consulta *XQueryV* Q_4

```
<Globo>
      <manchete>
            <reportagem>Brasil ganha da Argentina </reportagem>
            <reporter>Reinaldo</reporter>
      </manchete>
      <manchete>
            <reportagem> Brasil ganha da Argentina </reportagem>
            <reporter>Joaquim</reporter>
      </manchete>
      <manchete>
            <reportagem> Rock in Rio Lisbôa </reportagem>
            <reporter>Manoel</reporter>
      </manchete>
</Globo>
```

Figura 6.11: Resultado de Q_4

6.4 Módulo Simplificador

No *XQueryTranslator*, a primeira etapa da fase de tradução ocorre no módulo Simplificador. Sua função é preparar uma consulta *XQueryV*, sintaticamente correta e semanticamente válida, para a etapa de tradução. Essa preparação consiste na transformação sintática da consulta em uma outra consulta *XQueryV* equivalente e de tradução mais fácil para SQL:1999.

(a)

```
for $i in view("Jornais")/Jornais/Jornal/Secoes/Secao/noticia
where $i/data/ano = "2003"
return
($i/titulo, $i/autor/nome)
```

(b)

```
for $i in view-("Jornais")/Jornais/Jornal,
      $j in $i/Secoes/Secao,
      $y in $j/noticia
where $y/data/ano = "2003"
return
    ($y/titulo,
     for $x in $y/autor
     return
          $x/nome
)
```

Figura 6.12: Consultas *XQueryV* equivalentes

Essa etapa de preparação justifica-se pela diversidade de características redundantes da linguagem *XQuery* que possibilitam a escrita de consultas a documentos XML de uma forma mais simples e flexível [66]. Ou seja, devido às características de *XQuery*, é possível escrever uma mesma consulta de várias formas diferentes. Por exemplo, a Figura 6.12 ilustra duas consultas *XQueryV* sintaticamente distintas, mas que têm o mesmo significado (extrair o titulo das notícias publicadas no ano de 2003 e o nome dos seus respectivos autores).

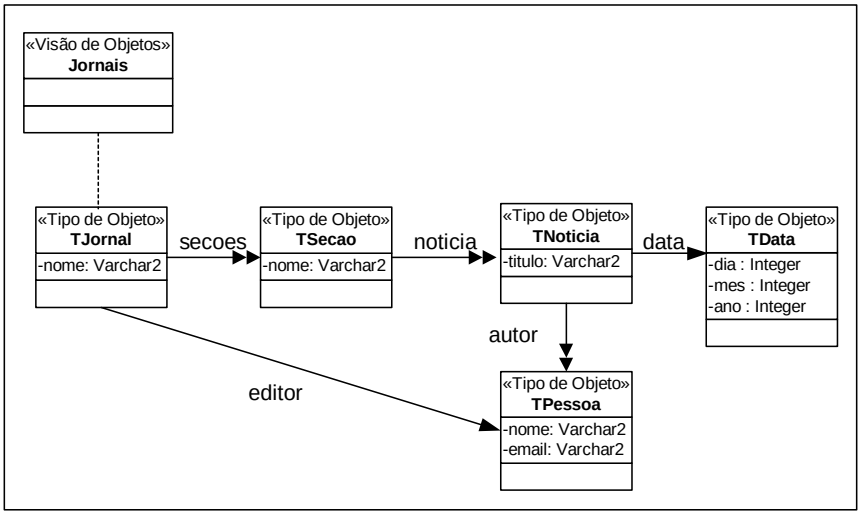

Figura 6.13: Esquema da VOD da Visão XML Jornais

A consulta *XQueryV* resultante dessa etapa de preparação é chamada de consulta *XQueryV* Canônica (*XQueryVC*). Uma consulta desse tipo torna o processo de tradução mais simples, pois possui o mesmo comportamento que uma consulta SQL:1999, onde cada coleção de objetos (tabela de objetos, visão de objetos, tabela aninhada ou vetor de objetos) é tratada de forma separada, ou seja, em expressões de caminho multivalorado de SQL:1999 existe apenas uma única ligação multivalorada, e essa ligação é a última do caminho.

Por exemplo, considere o esquema da VOD (Figura 6.13) da visão XML Jornais (Figura 6.3). Se quisermos extrair dessa VOD o titulo das notícias publicadas no ano de 2003 e o nome dos seus respectivos autores, poderíamos pensar na consulta SQL:1999 (a) da Figura 6.14. No entanto, essa consulta está sintaticamente errada, pois todas as ligações do caminho Jornais•secoes•noticia (cláusula FROM) são multivaloradas, e em SQL:1999 isso não é permitido. Somando-se a isso, na expressão de caminho multivalorada h•autor•nome a ligação multivalorada não é a última. A consulta

SQL:1999 (b) da Figura 6.14 apresenta uma forma correta. Observe que todas as coleções que aparecem na consulta são declaradas separadamente na cláusula FROM:

(a)
```
SELECT h•titulo ,
       h•autor•nome
FROM Jornais•secoes•noticia h
WHERE h•data•ano = 2003
```

(b)
```
SELECT a•titulo ,
       CURSOR ( SELECT a•nome
                FROM TABLE(n•autor) a)
FROM Jornais j, TABLE (j•secoes) s, TABLE
(s•noticia) n
WHERE n•data•ano = 2003
```

Figura 6.14: Consulta SQL:1999 para Jornais

Ao contrário de SQL:1999, com *XQueryV* é possível escrever consultas contendo expressões de caminho com mais de um passo multiocorrência. Sendo assim, antes de traduzir uma consulta *XQueryV* em SQL:1999, devemos transformar cada passo multiocorrência de uma expressão de caminho com essa característica, numa outra expressão de caminho apropriada. Considere novamente as consultas *XQueryV* da Figura 6.12. Como foi dito anteriormente, essas consultas possuem o mesmo significado. No entanto, observe que a consulta *XQueryV* (b) possui o mesmo comportamento de uma consulta SQL:1999, o que a torna mais apropriada para tradução. Podemos dizer então que a consulta (b) é a forma canônica da consulta (a), ou seja, a consulta (b) é a representação *XQueryVC* da consulta (a).

Definição 6.1: (Consulta *XQueryV* Canônica) uma consulta *XQueryV* está na forma canônica se esta é uma expressão *for-where-return*, para a qual todas as expressões de caminho multiocorrência que a compõem existem em um único passo multiocorrência, sendo esse passo o último.

A seguir, apresentamos o algoritmo para transformar uma expressão *XQueryV* do tipo caminho absoluto (expressão *XPath* abreviada), numa expressão *XQueryVC*. Esse não é o único algoritmo necessário para a implementação do módulo Simplificador. No entanto, para simplificar este trabalho discutiremos apenas esse algoritmo.

6.4.1 Algoritmo para Traduzir *XPath* em *XQueryVC*

Nesta seção apresentamos o algoritmo *XPath2XQueryVC* que é implementado no Módulo Simplificador para traduzir consultas *XPath* em consultas *XQueryVC*, a partir do esquema da visão XML. Sejam, V_X uma visão XML e EC uma expressão de caminho absoluto da forma **view**(V_X)/C_1/C_2/.../C_{n-1}/C_n, onde V é a respectiva VOD da visão XML V_X, n é o número de passos de EC, $C_i = E_i$ [$Cond_{E_i}$] ($1 \leq i \leq n$) é um passo de EC, E_i corresponde ao nome dos elementos alcançados pelo passo i e $Cond_{E_i}$ uma expressão condicional sobre o elemento E_i, então a consulta *XQueryVC* equivalente a EC terá a seguinte forma:

```
for $i1 in view(VX)/E1/E2,
    $i2 in $i1/E3/.../Ea,
    $ia in $i2/Ea+1/.../Eb,
    ... ,
    $iy in $iw/Ez+1/.../Eδ
where Cond_E2 ($i1) and
      Cond_Ea ($i2) and
      Cond_Eb ($ia) and
      ... and
      Cond_En-1 ($iy) and
return $iy/ Eδ/ Eδ+1/.../En
```

onde, $2 < a < b < y < \delta < n$, $Cond_{E_j}$ ($\$i_m$) é a expressão condicional $Cond_{E_j}$ reescrita com a variável $\$i_m$ e E_a, E_b, ..., E_{n-1} são passos multiocorrência.

Na Figura 6.15, apresentamos o algoritmo que realiza essa transformação. Nesse algoritmo, cada sub-expressão de caminho multiocorrência de EC gera uma declaração de variável (linhas 6 e 15) para a cláusula *for* da consulta *XQueryVC* resultante. Predicados condicionais de EC são reescritos a partir das variáveis declaradas na cláusula *for* (linhas 9 e 18) e colocados na cláusula *where*. A última sub-expressão de caminho de EC será reescrita na cláusula *return*, a partir da última variável declarada na cláusula *for* (linhas 22 a 25).

```
01  Var
02          f, w : vetor de String
03          r : String
04          j,p,i : Inteiro
05  início
06          f [1] = { view (V)/E1 /E2 }
07          t = 1
08          Se C2 possui um predicado condicional Então
09                  w [t] =  { Cond_E2 ($i1) }
10                  t = t + 1
11          j = 2
12          p = 3
13          Para todo passo Ci de C3/C4/.../C_φn-1 Faça
14                  Se Ci é um passo multiocorrência Então
15                          f [j] =  { Ep/.../Ei }
16                          Se Ci possui um predicado condicional Cond_Ei Então
17
18                                  w [t] = { Cond_Ei ($im) }
19                                  t = t + 1
20                          j = j+1
21                          p = i +1
22          Se p ≠ n Então
23              r = { Ep/.../En }
24          Senão
25              r = { En }
26
27          Retorne     for $i1 in f [1], ..., $ip in f [p-1]
28                      where w [1] and ... w [t-1]
29                      return
30                              $ip/r
31  Fim
```

Figura 6.15: Algoritmo *XPath2XqueryVC*

Exemplo 6.5

Considere a expressão de caminho absoluto abaixo, que retorna o título das notícias do jornal "O Globo", publicadas no mês de outubro ("10") de "2003", da visão XML Jornais (Figura 6.3).

```
view("Jornais")/Jornais/Jornal[nome= "O Globo"]/secoes/Secao/noticia[data/ano="2003" and data/mes="10"]/titulo
```

Figura 6.16: Consulta *XQueryV* Q_5

Essa consulta é transformada na consulta *XQueryVC* da Figura 6.17 pelo algoritmo *XPath2XQueryVC*. Inicialmente, o algoritmo verifica a sub-expressão view("Jornais")/Jornais/Jornal[nome= "O Globo"] e gera a declaração da variável $\$i_1$ para a cláusula for (linha 1 da Figura 6.17) e uma expressão condicional reescrita com $\$i_1$ para a cláusula where (linha 5 da Figura 6.17). Em seguida, o algoritmo adiciona uma declaração de variável na cláusula *for* cada vez que encontra um passo multiocorrência e, havendo predicado condicional nesse passo, adiciona o predicado na cláusula *where*.

Sendo assim, as sub-expressões secoes/Secao e noticia[data/ano="2003" and data/mes="10"] geram variáveis para a cláusula for (linhas 2 e 3 da Figura 6.17, respectivamente) e uma expressão condicional para a cláusula where (linha 6 da Figura 6.17). A sub-expressão restante (titulo) gera a cláusula *return* da consulta *XQueryVC* (linha 8 da Figura 6.17).

```
01      for $i1 in view("Jornais")/Jornais/Jornal,
02          $i2/secoes/Secao,
03          $i3/noticia
04      where
05          $i1/nome = "O Globo" and
06          ($i3/data/ano="2003" and $i3/data/mes="10")
07      return
08          $i3/titulo
```

Figura 6.17: Consulta *XQueryVC* de Q_5

6.5 Módulo Tradutor

A última etapa da fase de tradução no *XQueryTranslator* ocorre no Módulo Tradutor. Sua função é traduzir cada consulta *XQueryVC* gerada no Módulo Simplificador em uma consulta SQL:1999 sobre a respectiva visão de objetos *default*. Essa tradução é realizada de uma forma simples e direta, na qual as cláusulas *for*, *where* e *return* da consulta *XQueryVC* são diretamente transformadas nas cláusulas *from*, *where* e *select*, respectivamente, da consulta SQL:1999 correspondente.

A seguir apresentamos o algoritmo para realizar essa tradução. Para simplificar a discussão, ignoramos os construtores de elementos da consulta *XQueryVC* de entrada, uma vez que eles não influenciam no resultado dessa fase. Os construtores de elementos estão associados com a formatação do resultado esperado e não com a extração desses dados.

6.5.1 Algoritmo para Traduzir *XQueryVC* em SQL:1999

Nesta seção apresentamos o algoritmo implementado no módulo Tradutor do *XQueryTranslator* para transformar consultas *XQueryVC* de uma visão XML V_x, em consultas SQL:1999 sobre a respectiva visão de objetos *default* V, a partir do esquema

de V_x. Como apresentado na Figura 6.18, o algoritmo GeraSQL recebe como entrada uma consulta *XQueryVC* e o XML *Schema* da respectiva visão XML:

```
01  GeraSQL (Q: Expressão XQueryVC)
02      Para cada variável $v definida na cláusula for de Q Faça
03          Seja p a expressão de caminho associada a $v
04          Se p é da forma view("V")/E1/E2 Então
05              Adicione { Visao v } à cláusula FROM
06          Senão
07              Seja p da forma $w / Ea / Ea+1/.../ Eb-1/ Eb, e TEi (a ≤ i ≤ b) o tipo de Ei
08              Se TEb-1 é tipo coleção Então
09
10                  Adicione { Table (w●Ea●Ea+1●...●Eb-1) v } à cláusula FROM
11              Senão
12                  Adicione { Table (w●Ea●Ea+1●...●Eb-1●Eb) v } à cláusula FROM
13      Se existe cláusula where em Q Então
14          Adicione as condições da cláusula where de Q na cláusula WHERE
15      Para cada expressão Expr contida na cláusula return Faça
16          Se Expr é uma expressão de caminho Então
17              Seja $v o passo inicial de Expr e Ea / Ea+1/.../ Eb-1/ Eb a expressão de caminho associada a $v na
18                  cláusula for
19              Se Expr é da forma $v / Eb+1/ Eb+2/ ... / Ec-1/ Ec Então
20                  Adicione { v● Eb+1●Eb+2●...●Ec-1●Ec } à cláusula SELECT
21              Senão Se Expr é da forma $v / Eb+1/ Eb+2/ ... / Ec-1/ Ec / text() Então
22                  Adicione { v● Eb+1●Eb+2●...●Ec-1●Ec } à cláusula SELECT
23          Senão Se Expr é uma expressão FWR Então
24              Adicione { CURSOR ( GeraSQL (Expr) ) } à cláusula SELECT
```

Figura 6.18: Algoritmo GeraSQL

O algoritmo trata da tradução da cláusula *for* nas linhas de 2 a 11. Cada variável declarada na cláusula *for* é mapeada num objeto (*alias*) da cláusula *from*. Nas linhas 4 e 5 tratamos a primeira variável declarada na consulta *XQueryVC*. A expressão de caminho associada a essa variável sempre nos fornece o nome da respectiva visão de objetos *default* da visão XML consultada. As demais variáveis declaradas são tratadas nas linhas 7 a 11, e correspondem às tabelas aninhas da VOD. O nome da respectiva tabela aninhada corresponde ao passo multiocorrência da expressão de caminho associada à variável.

Se a consulta *XQueryVC* de entrada possui uma cláusula *where*, sua tradução é processada na linha 13 do algoritmo. A cláusula *where* da SQL:1999 gerada é praticamente uma cópia da cláusula *where* da *XQueryVC*, guardadas algumas diferenças sintáticas.

Finalmente, nas linhas 14 a 24 o algoritmo monta a cláusula Select da SQL:1999 a partir da cláusula *return*. Na cláusula *return* podemos encontrar expressões *for-where-return* e/ou expressões de caminho. No primeiro caso, é feita uma chamada recursiva ao algoritmo, passando a expressão *for-where-return* (linha 24) para montar uma

subconsulta SQL:1999 na cláusula *select*. No segundo caso, uma simples expressão de caminho SQL:1999 é adicionada à cláusula Select (linhas 16 a 22).

A seguir, apresentamos alguns exemplos que ilustram a execução do algoritmo. Nos exemplos, considere o esquema da visão XML Jornais representado na Figura 6.3 e o esquema da sua respectiva VOD, mostrado na Figura 6.13. Para essa discussão o esquema do banco de dados pode ser ignorado uma vez que a execução da consulta SQL:1999 da VOD é feita pelo próprio SGBD.

Exemplo 6.6

A consulta SQL:1999 abaixo corresponde à tradução da consulta (b) da Figura 6.12. Observe que as expressões de caminho da cláusula *for* mais externa foram mapeadas na visão de objetos (linha 04), e nas tabelas aninhadas secoes (linha 05) e noticia (linha 06), respectivamente, para a cláusula *from* da SQL:1999 resultante. A tabela aninhada (*nested table*) secoes foi mapeada pela linha 9 do algoritmo e a tabela aninhada noticias pela linha 11.

```
01.  SELECT y•titulo,
02.              CURSOR ( SELECT x•nome
03.                                FROM TABLE (y•autor) x)
04.  FROM Jornais i,
05.           TABLE ( i•secoes ) j,
06.           TABLE ( j•noticia ) y
07.  WHERE y•data•ano = 2003
```

Figura 6.19: Tradução SQL:1999

Em seguida, a cláusula *where* da consulta *XQueryVC* foi mapeada na cláusula *where* da consulta SQL:1999 (linha 07). Observe que nesse mapeamento foram feitas pequenas modificações sintáticas, ou seja, aspas foram retiradas, "/" foram substituídas por "•" e o delimitador de variável "$"foi descartado.

Uma parte da cláusula Select da SQL:1999 (linha 1) foi gerada pela linha 21 do algoritmo e contém uma expressão de caminho SQL:1999 correspondente à expressão $y/titulo de return. A outra parte (linhas 2 e 3) foi gerada de forma recursiva na linha 25 do algoritmo e corresponde à consulta *for-return* aninhada na *XQueryVC*.

Exemplo 6.7

A consulta *XQuery* Q_6 (Figura 6.20) será traduzida na consulta SQL:1999 da Figura 6.21. A única expressão de caminho da cláusula for da linha 2 (Figura 6.20) é mapeada na cláusula FROM da linha 9 (Figura 6.21), pela linha 5 do algoritmo. Como não existe cláusula *where* relacionada a essa cláusula for, o algoritmo segue mapeando as expressões da cláusula *return*. A expressão de caminho da linha 4 (Figura 6.20), é mapeada para a linha 1 (Figura 6.21) da cláusula SELECT, pela linha 20 do algoritmo. A expressão de caminho da linha 5 (Figura 6.20), é mapeada para a linha 2 da Figura 6.21 da cláusula SELECT, pela linha 22 do algoritmo. A expressão *fwr* das linhas 6 a 17 (Figura 6.20), é mapeada na consulta SQL:1999 aninhada das linhas 3 a 7 da Figura 6.21, pela linha 24 do algoritmo. Observe que nesse ponto o algoritmo faz uma chamada recursiva para montar todas as cláusulas da consulta aninhada, passando como parâmetro à expressão *fwr*.

```
01    <jornais>{
02        for $i in view("Jornais")/Jornais/Jornal
03        return
04            <jornal>{  $i/nome,
05                <editor> { $i/editor/nome/text() }</editor>,
06                for $j in $i/Secoes/Secao
07                return
08                    <secao>{ $j/nome,
09                        for $z in $j/noticia
10                        return
11                        <noticia>{
12                            $z/titulo/text(),
13                            for $w in $z/autor
14                            return
15                                <autor>{ $w/nome/text() }</autor>
16                        }</noticia>
17                    }</secao>
18            }</jornal>
19    }</jornais>
```

Figura 6.20: Consulta *XQueryVC* Q_6

```
01    SELECT  i•nome,
02            i•editor•nome,
03            CURSOR ( SELECT   j•nome,
04                        CURSOR ( SELECT   z•titulo,
05                                    CURSOR ( SELECT w•nome
06                                             FROM TABLE (z•autor) w)
07                                 FROM TABLE (j•noticia) z)
08                        FROM TABLE(i•secoes) j)
09    FROM Jornais i
```

Figura 6.21: Tradução SQL:1999 de Q_6

6.6 Módulo Gerador

Nesta seção apresentamos o algoritmo GeraXML, o qual transforma uma coleção de objetos extraídos de um banco de dados objeto-relacional em XML. Esse algoritmo é implementado pelo módulo Gerador, sendo então responsável pela última fase da execução de uma consulta *XQuery* no *XqueryTranslator*: a transformação. Ele recebe como entrada o esquema da visão XML consultada, a respectiva consulta *XQueryVC*, e uma coleção de objetos extraídos do banco de dados. Para a figura 6.22, considere S um banco de dados, V uma visão XML que publica dados de S e Expr uma consulta *XQuery* sobre V. Considere Q a consulta SQL:1999 gerada pelo algoritmo GeraSQL (Seção 6.6.1) a partir de Expr e R a coleção de objetos extraídos de S resultantes da execução de Q sobre **S**.

Casos	**Resultado**
1. Expr é da forma <Elem>{ SubExpr } </Elem>	<Elem> GeraXML (SubExpr, R) </Elem>
2. Expr é da forma for...where..return(SubExpr) e R é a coleção de objetos { o_1, o_2, ..., o_n }, onde o_i, $1 \leq i \leq n$, é um objeto do tipo T.	GeraXML (SubExpr, o_1) GeraXML (SubExpr, o_2) ... GeraXML (SubExpr, o_n)
3. Expr é da forma ($SubExpr_1$, $SubExpr_2$, ..., $SubExpr_n$) e R é um objeto complexo composto pelos atributos {a_1, a_2, ..., a_n }, onde a_i, $1 \leq i \leq n$, é do tipo T_i.	GeraXML ($SubExpr_1$, a_1) GeraXML ($SubExpr_2$, a_2) ... GeraXML ($SubExpr_n$, a_n)
4. Expr é da forma $v/$p_1$/$p_2$/.../text() e R é um objeto do tipo T_R	R
5. Expr é da forma $v/$p_1$/$p_2$/.../$p_n$, onde p_n é um passo monocorrência, T_n é o tipo de p_n, T_n é simples e R é um literal.	<p_n> R </p_n>
6. Expr é da forma $v/$p_1$/$p_2$/.../$p_n$, onde p_n é um passo monocorrência, T_n é o tipo de p_n, T_n é complexo e R é um objeto estruturado.	<p_n> GeraElementoComplexo (T_n, R) </p_n>
7. Expr é da forma $v/$p_1$/$p_2$/.../$p_n$, onde p_n é passo multiocorrência, T_n é o tipo de p_n, T_n é simples e R é a coleção de objetos {o_1, o_2, ..., o_n}, onde o_i, $1 \leq i \leq n$, é um literal.	<p_n> o_1 </p_n> <p_n> o_2 </p_n> ... <p_n> o_n </p_n>
8. Expr é da forma $v/$p_1$/$p_2$/.../$p_n$, onde p_n é passo multiocorrência, T_n é o tipo de p_n, T_n é complexo e R é a coleção de objetos {o_1, o_2, ..., o_n }, onde o_i, $1 \leq i \leq n$, é um objeto complexo	< p_n> GeraElementoComplexo (T_n, o_1) </p_n> ... <p_n> GeraElementoComplexo (T_n, o_n) </p_n>

Figura 6.22: Algoritmo GeraXML.

Observe que nos casos 1, 2, 3, 4, 5 e 7 a estrutura da resultado XML é inferida da própria expressão *XQueryV* corrente. No entanto, nos casos 6 e 8 o esquema da visão XML precisa ser analisado para determinarmos que elementos XML gerar. Sendo assim, os casos 6 e 8 fazem chamadas ao algoritmo GeraElementoComplexo. Esse algoritmo recebe como entrada um objeto complexo O composto pelos atributos b_1, b_2, ..., b_m, e o tipo T_E do elemento complexo E, onde T_E é composto pelos elementos E_1, E_2, ..., E_m. Então, o algoritmo gera o conteúdo do elemento E. O conteúdo XML gerado tem a seguinte forma: Construtor_E_1-b_1 Construtor_E_2-b_2 ... Construtor_E_m-b_m, onde Construtor_E_i-b_i é a coleção de elementos XML E_i geradas a partir do objeto(s) b_i, $1 \leq i \leq m$. A figura a seguir mostra como esses elementos XML são gerados:

Casos	Construtor_E-b
1. E é mononocorrência e T_E é simples	<E> b </E>
2. E é mononocorrência e T_E é complexo	<E> GeraElementoComplexo (T_E, b) </E>
3. E é multiocorrência e T_E é simples, onde b é a coleção de literais {c_1, c_2, ..., c_w}	<E> c_1 </E> <E> c_2 </E> ... <E> c_w </E>
4. E é multiocorrência e T_E é complexo, onde b é uma coleção de objetos (d_1, d_2, ..., d_w)	<E> GeraElementoComplexo (T_E, d_1) </E> <E> GeraElementoComplexo (T_E, d_2) </E> ... <E> GeraElementoComplexo (T_E, d_w) </E>

Figura 6.23: Algoritmo para gerar elementos complexos.

Exemplo 6.8

Considere a visão XML Jornais, cujo esquema é mostrado na Figura 6.3, o esquema da sua respectiva VOD apresentado na Figura 6.13, a consulta *XQueryVC* da

Figura 6.12(b) e sua respectiva tradução SQL:1999 mostrada na Figura 6.19. Suponha que a figura abaixo apresente o resultado da execução da consulta SQL:1999 sobre a VOD em um determinado estado do banco de dados.

titulo	CURSOR
Alta no Dolár	nome Pedro Bial Fátima Bernardes
Lula Ganha Prêmio Nobel da Paz	nome Bussunda Hélio

Figura 6.24: Resultado da consulta SQL:1999 R_1

Para esse exemplo, o algoritmo GeraXML deverá gerar o resultado XML apresentado na Figura 6.25. O algoritmo recebe como entrada a consulta *XQueryVC* apresentada na Figura 6.12(b) e a coleção de objetos R_1, mostrado na figura 6.24. Como a consulta *XQueryVC* é da forma fwr**,** R_1 é tratada pelo caso 2 do algoritmo**.** No caso 2, para cada objeto em R_1 é feita uma chamada recursiva do GeraXML, passando como parâmetro de entrada o conteúdo da cláusula *return* e o objeto de $\mathbf{R_1}$.

Como o conteúdo da cláusula *return* é uma seqüência de expressões, cada objeto de R_1 é tratado pelo caso 3. Considere a primeira chamada recursiva do algoritmo. No caso 3, cada expressão é associada ao valor de um atributo da estrutura do objeto de entrada e para cada associação é feita uma chamada recursiva do GeraXML. Neste exemplo, a estrutura dos objetos de R_1 é composta por um atributo monovalorado e um multivalorado, assim, a expressão da cláusula *return* $y/titulo é associada ao valor do atributo monovalorado titulo e com esses parâmetros é feita uma chamada do GeraXML. De forma análoga, a outra expressão da cláusula *return* é associada ao atributo multivalorado e é feita mais uma chamada recursiva a GeraXML.

Na chamada que recebe a expressão $y/titulo como parâmetro, o objeto de entrada é tratado pelo caso 5 do algoritmo, pois a expressão $y/titulo é monocorrência de tipo simples, gerando desta forma a linha 01 da (Figura 6.25).

Na chamada que recebe como entrada a expressão fwr aninhada da *XQueryVC*, o objeto multivalorado de entrada é tratado pelo caso 2 do algoritmo, onde para cada objeto da coleção é feita uma chamada do GeraXML passando como parâmetro o

conteúdo da cláusula *return* e o objeto. Em cada uma dessas chamadas, o objeto de entrada é tratado pelo caso 5, pois a cláusula *return* é composta de apenas uma expressão e, assim, são geradas as linhas 02 e 03 da (Figura 6.25).

```
01.  <titulo>Alta no Dólar</titulo>
02.  <nome>Pedro Bial</nome>
03.  <nome>Fátima Bernardes</nome>
04.
05.  <titulo>Lula Ganha Prêmio Nobel da Paz</titulo>
06.  <nome>Bussunda</nome>
07.  <nome>Hélio</nome>
```

Figura 6.25: Resultado XML para R_1

Capítulo 7
Conclusão

Neste trabalho abordamos o problema de publicação de dados objeto-relacionais através de visões XML. No nosso enfoque, usamos visões de objeto *default* (VOD) como interface entre visões XML e o banco de dados.

Desenvolvemos XML *Publisher*, um *framework* para publicar dados objeto-relacionais de SGBD's *Oracle* 9i, através de visões XML. XML *Publisher* define o seguinte conjunto de operações de acesso: *GetCapabilities*, *GetSchema*, *ExecuteQuery* e *ExecuteUpdate*. Para definir uma visão XML no *framework*, o projetista deve criar uma VOD através de uma consulta SQL:1999 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados. Em seguida, a visão XML pode ser consultada usando *XQueryV*, um subconjunto da linguagem *XQuery* definido neste trabalho. Caso atualizações sobre a visão XML sejam permitidas, devem ser definidos tradutores (*instead of triggers*) os quais especificam como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados. O problema de atualização de visões no XML *Publisher* não é discutido neste trabalho.

O processamento de consultas *XQueryV* no XML *Publisher* é feito no *XQueryTranslator* nas seguintes fases: *parsing*, tradução, execução e transformação. A fase de *parsing* compreende a analise léxica, sintática e semântica da consulta *XQueryV*, e acontece no Módulo Validador do *XQueryTranslator*. Na fase de tradução, a consulta *XQueryV* validada é traduzida em uma consulta SQL:1999 pelos Módulos Simplificador e Tradutor. No Módulo Simplificador a consulta *XQueryV* é traduzida em uma consulta *XQueryVC*, que é uma consulta *XQueryV* de mais fácil tradução para SQL:1999. No Módulo Tradutor, a consulta XQueryVC é traduzida em uma consulta SQL:1999 sobre a respectiva VOD, usando o algoritmo **GeraSQL** apresentado no Capitulo 6. Essa tradução é

simples e direta, pois o esquema da VOD possui os mesmos tipos e a mesma estrutura do esquema da visão XML. Na fase de execução a consulta SQL:1999 é processada pelo SGBD. Finalmente, na fase de transformação, o resultado XML é gerado a partir dos objetos retornados pelo SGBD, da consulta *XQueryV* e do XML *Schema* da visão XML. Essa fase ocorre no Módulo Gerador, que implementa o algoritmo **GeraXML** apresentado no Capítulo 6.

Outra contribuição deste trabalho foi VBA (*View-By-Assertion*), uma ferramenta para auxiliar a tarefa de criação de visões de objeto. A novidade do enfoque proposto é que as visões de objetos são especificadas através de assertivas de correspondência que especificam formalmente o relacionamento entre o esquema da visão e o esquema do banco de dados. O formalismo permite especificar várias formas de correspondência, inclusive casos onde os esquemas possuem estruturas diferentes. A ferramenta oferece uma interface gráfica para apoiar a criação do tipo da visão e a edição de suas assertivas de correspondência; e gera, de forma automática, a consulta SQL:1999 que realiza o mapeamento definido pelas assertivas da visão. O algoritmo **GeraVisaoDeObjeto** apresentado no Capitulo 5 recebe como entrada o esquema da visão, o esquema do banco de dados e as assertivas da visão de objetos, e gera, de forma automática, a consulta SQL:1999 que mapeia a base de dados no esquema da visão. No caso em que o banco de dados é relacional, o procedimento **GeraConstrutorRelacional** é usado para gerar a cláusula SELECT da consulta SQL:1999. No caso em que o banco de dados é objeto-relacional, a cláusula SELECT da consulta SQL:1999 é gerada pelo procedimento **GeraConstrutorObjetoRelacional**. A cláusula SELECT especifica como os objetos da visão são construídos a partir dos dados na base de dados, os quais são obtidos nas cláusulas *FROM* e *WHERE*. Neste trabalho, para cada caso dos procedimentos **GeraConstrutorRelacional** e **GeraConstrutorObjetoRelacional**, mostramos que o código SQL:1999 gerado, o qual define o valor de cada atributo do tipo dos objetos da visão, realiza corretamente o mapeamento especificado pelas assertivas de correspondência do tipo da visão com o tipo da tabela-base.

Desenvolvemos ainda um ambiente que dá suporte ao processo de definição de visões XML no XML *Publisher*. O processo de definição de visões compreende os seguintes passos: (i) o projetista especifica, através de uma interface gráfica, o tipo dos

elementos da visão XML; (ii) a ferramenta DSG (***D****efault Object View* ***S****chema* ***G****enerator*) gera automaticamente o esquema da VOD; (iii) usando a ferramenta VBA o projetista deve então carregar o esquema do banco e, através de uma GUI, definir as assertivas de correspondência do esquema VOD com o esquema do banco de dados; (iv) com base no conjunto de assertivas de correspondência da visão, VBA gera automaticamente a consulta SQL:1999 da visão de acordo o mapeamento definido pelas assertivas; e (iv) os arquivos de configuração do XML *Publisher* são atualizados, de forma que a visão XML é finalmente publicada.

Como trabalhos futuros pretendemos:

a) Verificar o desempenho do *XQueryTranslator* no processamento de consultas *XQueryV* submetidas ao XML *Publisher*;
b) Estender o *XQueryTranslator* para suportar um conjunto maior de consultas *XQuery*;
c) Estender VBA para:

- Suportar novos tipos de assertivas de correspondência de forma a permitir a especificação de outros tipos de visões de objetos;
- Gerar páginas XSQL. Como foi visto no Capítulo 1, uma página XSQL é definida através de consultas SQL:1999, podendo a página ser parametrizada ou não. Como VBA gera consultas SQL:1999, basta adaptá-lo para gerar consultas SQL:1999 parametrizadas. Além disso, devemos criar um algoritmo para gerar uma folha de estilo XSLT que transforma o resultado Canônico da SQL:1999 no formato definido pelas assertivas de correspondência;
- Gerar consultas *XQuery* que definem visões XML públicas no *SilkRoute* e no *XPeranto*. Nesses sistemas, uma consulta *XQuery* pública geralmente é complexa, pois transforma dados relacionais em estruturas de dados XML aninhados. Através de VBA, seria possível então gerar as correspondências entre a visão XML Canônica e as visões XML públicas de forma gráfica. A partir das assertivas de correspondência entre essas visões, a consulta *XQuery* pública poderia ser gerada automaticamente.
- Usar o mecanismo de definições de visões de objetos proposto em [49] para criar visões de objetos sobre qualquer SGBD relacional.

Referências Bibliográficas

[1] ABITEBOUL, S.; BUNEMAN, P.; SUCIU, D.. Gerenciando Dados na WEB. Rio de Janeiro: Campos, 2000.

[2] AHO, A. V.; SETHI, R.; ULLMAN, J.D. Compiladores: Princípios, Técnicas e Ferramentas**.** LTC, 1995.

[3] BARU, C.; GUPTA, A.; LUDASCHER, B.; MARCIANO, R.; PAPAKOSTATINOU, Y.; VELIKHOV, P.; CHU, V.. XML-Based Information Mediation with MIX. In: *Proceddings of ACM SIGMOD Conf. on Management of Data*, p. 597-599, Philadelphia, Pennsylvania, USA, Junho 1999.

[4] Boas, R. M. F. V.. XMLS+ Matcher: Um Método para Matching de XML Schemas Semânticos. Fortaleza. Dissertação /Mestrado em Ciência da Computação/ - Universidade Federal do Ceará - UFC, 2002.

[5] Bohannon, P.; Freire, J.; Roy, P.; Simeon, J.. From XML schema to relations: A cost-based approach to XML storage. In: *Proceddings of 18th International Conference on Data Engineering (ICDE'02)*. San Jose, California, 2002, pages 64–75.

[6] BSML - Bioinformatic Sequence Markup Language. Disponível em: <http://www.bsml.org/>. Acesso em: 20 abril 2004.

[7] CAREY, M., et al. XPERANTO: Publishing object-relational data as XML. In: *Proceedings Third International Workshop on the Web and Databases*. Dallas, Texas: May 2000, pages 105-110.

[8] CAREY, M.J.; KIERNAN, J.; SHANMUGASUNDARAM, J.; SHEKITA, E. J.; SUBRAMANIAN, S. N.. XPERANTO: A Middleware for Publishing Object-

Relational Data as XML Documents. In: *Proceedings InternationalConference Very Large Data Bases*. Cairo, Egypt: September 2000, pages 646–648.

[9] CHAMBERLIN, D.; ROBIE, J.; FLORESCU, D.. Quilt: An XML Query Language for Heterogeneous Data Source. In: *Proceedings of the Workshop on Web and Databases* (WebDb). In Conj. with SIGMOD'00. Dallas, Texas: Addison-Wesley, May, 2000.

[10] CHAUDHRI, A. B.; RASHID, A.; ZICARI, R.. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[11] CHENG, J.M.; XU, J.. XML and DB2. In: *Proceedings of the International Conference on Data Engineering*. California, USA: March 2000, pages 569-576.

[12] COSTA, J. P.. Atualização de Bancos de Dados Objeto Relacionais Através de Visões de Objetos. Fortaleza. Dissertação /Mestrado em Computação/ - Universidade Federal do Ceará – UFC, Dezembro de 2002.

[13] COX, Simon; DAISEY, Paul; LAKE, Ron; PORTELE, Clemens; WHITESIDE, Arliss. OpenGIS Consortium, Schema for Geography Markup Language (GML) 3.0. 2003-01-29. Disponível em: <http://www.opengis.org/docs/02-023r4.pdf>. Acesso em: 16 fevereiro 2004.

[14] DEITEL, H. M.; DEITEL, P. J.. *Java How to Program*. Prentice Hall, 3ª edição, 2000.

[15] DEUTSCH, A.; FERNANDEZ, M.; FLORESCU, D.; LEVY, A.; SUCIU, D.. *XML-QL:* A Query Language for XML. 8 th International WWW Conference, Toronto, May 1999.

[16] DUCKETT, J.; GRIFFIN, O.; MOHR, S.; NORTON, F.; STOKES-REES, I.; WILLIAMS, K.; CAGLE, K.; OZU, N.; TENNISON, J.. *Professional XML Schemas*. Wrox Press Ltd. Birmingham, 2001.

[17] ELMASRI, R.; NAVATHE, S.. Fundamentals of Database Systems. Addison-Weslley, 3ª ed., 2000.

[18] El-SAYED, M.; WANG, L.; DING, L.; RUNDENSTEINER, E. A.. An Algebraic Approach for Incremental Maintenance of Meterialized XQuery Views. /Technical Report/ Worcester Polytechnic Institute, February 2003.

[19] FAGIN, R.; KOLAITIS, P.G.; MILLER, R.J.; POPA, L.: Data Exchange: Semantics and Query Answering. In ICDT, pages 207–224, 2003.

[20] FAHL, Gustav; RISCH, Tore. Query processing over object views of relational data. In: *The VLDB Journal*. 1997, pages 261-281.

[21] FERNÁNDEZ, M.; KADIYSKA, Y.; SUCIU, D.; MORISHIMA, A.; TAN, W.. SilkRoute: A Framework for Publishing Relational Data in XML. In: *ACM Transactions on Database Systems*, Vol. 27, Nº. 4, December 2002, Pages 438-493.

[22] FERNÁNDEZ, M.; TAN, W.; SUCIU, D.. Silkroute: Trading between relations and XML. In: *Proceedings of the Ninth International World Wide Web Conference, (WWW'9)*. Amsterdam: May 2000.

[23] FLORESCU, D.; DEUTSCH, A.; LEVY, A.; SUCIU, D.; FERNANDEZ, M.. A Query Language for XML. In: *Proceedings of Eighth International Conference on World Wide Web*. Toronto, Canadá, 1999, pages 1155-1169.

[24] HAYASHI, L. S.; HATTON, J.. Combining UML, XML and Relational Database Tecnologies – The Best of All Worlds for Robust Linguistic Databases. In: *Proceedings of the IRCS Workshop on Linguistic Databases*. Philadelphia, USA: University of Pennsylvania, December 2001, 115-124 pp.

[25] *IBM XML for Tables*. Disponível em:

<http://www.alphaworks.ibm.com/tech/xtable>. Acesso em: 03 março 2004.

[26] LEE, D.; CHU, W.. Comparative Analysis of Six XML Schema Language. In: *ACM SIGMOD Record*, p.76-87, vol. 29, nº 3, September 2000.

[27] LEHTI, P.. Designing and Implemention of a Data Manipulation Processor for a XML Query Processor, Technical University of Darmstadt, Darmstadt, Germany, Diplomarbeit, Master Thesis, August 2001.

[28] LIMA, T. P.. *ProDIWA:* Um Processo Automatizável para Geração e Manutenção de Visões de Contexto de Navegação para Aplicações DIWA. Dissertação / Mestrado em Ciência da Computação/ - Universidade Federal do Ceará - UFC, 2004.

[29] LÓSCIO, B. F.. Atualização de múltiplas bases de dados através de mediadores. Fortaleza. Dissertação /Mestrado em Ciência da Computação/ - Universidade Federal do Ceará - UFC, 1998.

[30] MADHAVAN, J.; BERNESTEIN, P.; RAHM, E.. Generic Schema Matching with Cupid. In: *Proceddings of VLDB*. 2001, pages 49-58.

[31] MANOLESEU, I.; FLORESEN, D.; KOSSMANN, D.. Answering XML Queries over Heterogeneous Data Sources. In: *Proccedings of the 27th VLDB Conference*. Roma, Italy, 2001.

[32] MELLO, Ronaldo dos Santos; HEUSER, Carlos Alberto. A Bottom-Up Approach for Integration of XML Sources. In: *WIIW International Workshop on Data Integration on the Web*. Rio de Janeiro, Brazil, 2001.

[33] MUENCH, S.. Building Oracle XML Applications. O'Reilly, September 2000.

[34] O. Corporation, "PL/SQL - User's Guide and Reference", vol. Release 2 (9.2) - A96624-01, March, 2002. Disponível em: <http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96624.pdf>. Acesso em: 12 novembro 2004.

O. Corporation, Oracle9i - Application Developer's Guide - Object-Relational

[35] Features, vol. Release 2 (9.2) - A96594-01, Março de 2002. Disponível em: <http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96594.pdf >. Acesso em: 27 março 2004.

[36] OLIVEIRA, W. G. C.. Atualizando banco de dados objeto-relacional através de visões XML. Dissertação /Mestrado em Ciência da Computação/ - Universidade Federal do Ceará - UFC, 2004.

[37] Oracle *Corporation*. Disponível em: <http://technet.oracle.com>. Acesso em: 19 maio 2004.

[38] PATEL, J. M; RUNAPONGSA, K.. Storing and Querying XML Data in Object-Relational DBMS. In: *XML-BASED DATA MANAGEMENT AND MULTIMEDIA ENGINEERING - EDBT Workshop*, 2002, Prague, Czech Republic.

[39] POPA, L.; VELEGRAKIS, Y.; MILLER, R. J.; HERNANDEZ, M. A.; FAGIN, R. Translating Web Data. In: *VLDB*. August 2002, pages 598–609.

[40] RAHM, E.; BERNSTEIN, P. A. A Survey of Approaches to Automatic Schema Matching. In: *VLDB Journal*. 10(4), 2001, pages 334–350.

[41] RYS, M.. Bringing the Internet to Your Database: Using SQL Server 2000 e XML to Build Loosely-Coupled Systems. In: *Proceedings of the International Conference on Data Engineering*. Heidelberg, Germany: 2001, pp. 465-472.

[42] SAHUGUET, A.. Everything you ever wanted to know about dtds, but were afraid to ask. International Workshop on the Web and Databases. In: *Proceedings Of WebDB'2000*, pages 69-74.

[43] SCHMIDT, A.; KERSTEN, M.; WINDHOUWER, M.; WAAS, F.. Efficient Relational Storage and Retrieval of XML Documents. In: *Proceddings International Workshop on the Web and Databases (In conjunction with ACM SIGMOD)*. Dallas, USA, 2000, pages 47-52.

[44] SHANMUGASUNDARAM, J., et al. Querying XML Views of Relational Data. In: *Proceeding of 27th VLDB Conference*. Roma, Italy: 2001, pages 261-270.

[45] SHANMUGASUNDARAM, J., et al. XPERANTO: Bridging Relational Tecnology and XML. In: *IBM Research Report*. June, 2001.

[46] SHANMUGASUNDARAM, J.; TUFTE, K.; HE, H.; ZHANG, C.; DeWITT D., NAUGHTON, J.. Relational database for querying XML documents: limitations and opportunities. In: *Proceedings of the 25th VLDB Conference*. Edinburgh, Scotland, 1999, pages 302-314.

[47] SHIMURA, Takeyuki; YOSHIKAWA, Masatoshi; UEMURA, Shunsuke. Storage and Retrieval of XML Documents Using Object-Relational Databases. DEXA, 1999, pages 206-217.

[48] SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S.. Sistema de Banco de Dados. Makron Books, 3ª ed..

[49] TAKAHASHI, T.; KELLER, A.M.. Implementation of object view query on relational database. *In: Int'l Conf. on Data and Knowledge Systems for Manufaturing and Engineering - DKSME*. Hong Kong, May 1994.

[50] TATARINOV, I.; VIGLAS, S.; BEYER, K. S.; SHANMUGASUNDARAM, J.; SHEKITA, E. J.; ZHANG, C.. Storing and querying ordered XML using a relational database system. In: *Proceedings of the SIGMOD International Conference On Management of Data.* Wisconsin, USA, 2002.

[51] TATARINOV, Igor; ZACHARY, Ives G.; HALEVY, Alon Y.; WELD, Daniel S.. Updating XML*, ACM SIGMOD Record*. v.30 n.2, June 2001, p.413-424.

[52] VIDAL, V.M.P, LÓSCIO, B.F.: Solving the Problem of Semantic Heterogeneity in Defining Mediator Update Translators. In Proc. of 18th Intern. Conf. on Conceptual Modeling, Paris, France, p.293-308, 1999

[53] VIDAL, V. M. P.; LÓSCIO, B. F., SALGADO, A. C. Using correspondence assertions for specifying the semantics of xml-based mediators. In: WIIW 2001 - International Workshop on Information Integration on the Web - Technologies and Applications, Rio de Janeiro, 2001. *Proceedings of the International Workshop on Information Integration on the Web*. Rio de Janeiro: 2001, p.3 – 11.

[54] VIEIRA JÚNIOR, H. J.. XVerter: Armazenamento e Consulta de Dados XML em SGBDs. Rio de Janeiro. Dissertação /Mestrado em Ciência da Computação/ - Universidade Federal do Rio de Janeiro - UFRJ. COPPE, 2002.

[55] Widerhold, G.; Barsalou, T.; Lee, B. S.; Siambela, N.; Sujansky, W.. Use of relational storage and a semantic model to generate objects: The Peguin Project. Database'91: Merging Policy, Standards and Technology, 1991, pp. 503-515.

[56] WIEDERHOLD, Gio. Views, Objects, and Databases. In: *IEEE Computer*. Vol.19 No.12, December 1986, pages 37-44.

[57] World Wide Web Consortium. Disponível em: <http://www.w3.org>. Acesso em: 7 janeiro 2004.

[58] World Wide Web Consortium. *Extensible Markup Language (XML) Version 1.1*. W3C Recommendation, 4th February 2004. Disponível em: <http://www.w3.org/TR/2004/REC-xml11-20040204/>. Acesso em: 25 março 2004.

[59] World Wide Web Consortium. *HTML Version 4.0.1*. W3C Recommendation, 24 December 1999. Disponível em: <http://www.w3.org/TR/html4/>. Acesso em: 4 dezembro 2003.

[60] World Wide Web Consortium. *MathML 2.0*. W3C Recommendation, 21 February 2001. Disponível em: <http://www.w3.org/TR/MathML2/>. Acesso em: 14 março 2004.

World Wide Web Consortium. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004.

[61] Disponível em: <http://www.w3.org/TR/rdf-concepts/>. Acesso em: 20 março 2004.

[62] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999. Disponível em: <http://www.w3.org/TR/xpath>. Acesso em: 21 abril 2004.

[63] World Wide Web Consortium. XML Schema Part 1: Structures. W3C Recomendation, 02 May 2001. Disponível em: <http://www.w3.org/TR/xmlschema-1/>. Acesso em: 15 outubro 2003.

[64] World Wide Web Consortium. XML Schema Part 2: Datatypes. W3C Recomendation, 02 May 2001. Disponível em: <http://www.w3.org/TR/xmlschema-2/>. Acesso em: 17 agosto 2003.

[65] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model – Full Text. W3C Working Draft, 9 July 2004. Disponível em: <http://www.w3.org/TR/2004/WD-xquery-full-text-20040709/>. Acesso em: 11 julho 2004.

[66] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, 20 February 2004. Disponível em: <http://www.w3.org/TR/xquery-semantics/>. Acesso em: 30 março 2004.

[67] World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft, 23 July 2004. Disponível em: <http://www.w3.org/TR/xquery/>. Acesso em: 8 abril 2004.

[68] World Wide Web Consortium. XSL Transformation (XSLT) Version 1.0. W3C Recommendation, 16 November 1999. Disponível em: <http://www.w3.org/TR/xslt>. Acesso em: 22 setembro 2003.

[69] ZEDULKA, J.. Object-relational modeling in UML. In: *Proceedings of the Conference*. Information Systems Modelling, Ostrava, CZ, MARCH, 2001, p. 17-24.