FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

MARCELO SOARES DE MEIRELLES

INTEGRAÇÃO DE FONTES DE DADOS HETEROGÊNEAS BASEADAS NO MODELO DE DADOS XML

Fortaleza
Maio / 2004

FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

MARCELO SOARES DE MEIRELLES

INTEGRAÇÃO DE FONTES DE DADOS HETEROGÊNEAS BASEADAS NO MODELO DE DADOS XML

DISSERTAÇÃO APRESENTADA AO CURSO DE MESTRADO EM INFORMÁTICA APLICADA – MIA DA UNIVERSIDADE DE FORTALEZA COMO REQUISITO PARA A OBTENÇÃO DO TÍTULO DE MESTRE EM INFORMÁTICA APLICADA, NA LINHA DE PESQUISA DE BANCO DE DADOS.

ORIENTADOR: PROF. DR. ANGELO RONCALLI ALENCAR BRAYNER

Fortaleza
Maio / 2004

INTEGRAÇÃO DE FONTES DE DADOS HETEROGÊNEAS BASEADAS NO MODELO DE DADOS XML

BANCA EXAMINADORA:

_____

Prof. Orientador Ângelo Roncalli Alencar Brayner, D.-Ing. – UNIFOR

_____

Prof (a). Vânia Maria Ponte Vidal, D.Sc. – UFC

_____

Prof. Arnaldo Dias Belchior, D.Sc. – UNIFOR

MARCELO SOARES DE MEIRELLES, Integração de Fontes de Dados Heterogêneas Baseadas no Modelo de Dados XML. Fortaleza: Universidade. Dissertação de Mestrado. 2004.

PERFIL DO AUTOR: Bacharel em Análise de Sistemas pela Pontifícia Universidade Católica de Campinas

RESUMO

Esta dissertação propõe a utilização da arquitetura MDBS (*Multidatabase System*) para integrar fontes de dados heterogêneas disponibilizadas em ambientes flexíveis e dinâmicos, como a Web ou redes *ad hoc*. Uma das principais propriedades de um MDBS é garantir uma maior autonomia para as fontes de dados locais. Essa maior autonomia decorre do fato da tecnologia de MDBS utilizar uma linguagem de consulta como mecanismo de integração, sem a necessidade de especificação de um esquema global de integração.

PALAVRAS CHAVES: Integração de Fontes de Dados – Fontes de Dados Heterogêneas – Banco de Dados Múltiplos – Linguagens de Consulta – Banco de Dados

DEDICATÓRIA

Aos meus pais que sempre me ensinaram o valor da perseverança.

AGRADECIMENTOS

Ao meu orientador Prof. Dr. Angelo Brayner que compartilhou comigo seus conhecimentos.

Aos companheiros do MIA que não desistiram, me forçando a prosseguir.

Aos professores de ontem, hoje e amanhã que de forma abnegada trabalham em prol do nosso país.

ABSTRACT

Over the last few years, the **Web** (World Wide Web) has been used to publish databases, however the user must consider the features of such environment before to establish the database integration. This work proposes using the MDBS (Multidatabase System) architecture to integrate heterogeneous databases in flexible and dynamic environment, such as Web or ad hoc networks.

Flexible and dynamic environments are characterized by high independence from connection participants, low control over available services and high tolerance to communication failures. Integrating databases published on such environments requires an integration strategy that guarantees local databases autonomy.

MDBS technology uses a query language as integration mechanism, which is responsible for solving integration conflicts. Thus, the query language must provide constructs to perform queries over several different data sources. Furthermore, it should be capable for solving integration conflicts.

This work proposes an extension to the XQuery language, called **MXQuery**. The key feature of the proposed language is to provide mechanisms, which support the capability to jointly manipulate data in different data sources based on an XML data model.

RESUMO

Cada vez mais, a Web tem sido utilizada como um ambiente para disponibilizar bancos de dados, porém, utilizar esta arquitetura impõe ao usuário as vantagens e desvantagens desse ambiente. Esta dissertação propõe a utilização da arquitetura MDBS (*Multidatabase System*) para integrar bancos de dados heterogêneas disponibilizadas em ambientes flexíveis e dinâmicos, como a Web ou redes *ad hoc*.

Ambientes flexíveis e dinâmicos são caracterizados pela alta independência dos participantes da conexão, pelo baixo controle sobre os serviços solicitados e disponibilizados e pela necessidade de uma alta tolerância às falhas de comunicação. Integrar bancos de dados baseados nesses ambientes requer uma estratégia de integração que garanta uma maior autonomia para os bancos de dados locais. Este é o principal motivo para a utilização da arquitetura MDBS.

Na arquitetura MDBS, a linguagem de consultas é responsável por mapear e resolver todos os conflitos de integração. Entretanto, considerando as várias linguagens de consulta propostas para documentos XML, inclusive a XQuery, ainda não se pode identificar nessas linguagens mecanismos que possibilitem o acesso a múltiplas fontes de dados XML distribuídas e heterogêneas.

Portanto, essa dissertação propõe, ainda, uma extensão à linguagem XQuery, denominada MXQuery, que apresenta suporte necessário à especificação de consultas que acessam fontes de dados heterogêneas e distribuídas. Assim, a MXQuery pode ser incorporada a um MDBS para integrar fontes de dados heterogêneas.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

GLOSSÁRIO DE ABREVIATURAS

B

BDR – Banco de Dados Relacional

BDOO – Banco de Dados Orientado a Objetos

C

CDM – Common Data Model

CML – Chemical Markup Language

CORBA – Common Object Request Broker Architecture

cXML – Commerce XML

D

DBMS – Database Management System

DBA – Database Administrator

DTD – Document Type Definition

E

EBNF – Extended Backus-Naur Form

ECG – Esquema Conceitual Global

ECL – Esquema Conceitual Local

EE – Esquema Externo

EFLWOR – Each FLWOR

EIL – Esquema Interno Local

F

FDBS – Federated Database System

FLWOR – For Let Where Order by Return

FRAQL – Federation Query Language

H

HTML – Hypertext Markup Language

I

IDL – Interoperable Database Language

L

LAN – Local Area Network

LDB – Local Database

LDBS – Local Database System

LP – Location Processor

M

MANET – Mobile Ad hoc Networking

MD – Mapping Definition

MDL – Multidatabase Language

MDBS – Multidatabase System

MIND – METU Interoperable Database System

MSQL – Multidatabase SQL

MR – MDBS Repository

O

OQL – Object Query Language

Q

QM – Query Manager

QP – Query Processor

QPp – Query Postprocessor

QUI – Query User Interface

QT – Query Translator

R

RT – Result Translator

S

SINGAPORE – Single Access Point for Heterogeneous Data Repositories

SOQL – SINGAPORE Object Query Language

SQL – Structured Query Language

SR – Source Register

U

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

W

W3C – World Wide Web Consortium

WAN –Wide Area Network

X

XFDL – Extensible Forms Description Language

XML – Extended Markup Language

XML-I – XML Interface

XSL – Extensible Stylesheet Language

Y

YAT – Yet Another Tree-based

YATL – YAT Language

CAPÍTULO 1

INTRODUÇÃO

1.1 Motivação

Cada vez mais, a **Web** (*World Wide Web*) tem sido utilizada como um ambiente para disponibilizar bancos de dados, porém utilizar esta arquitetura impõe ao usuário as vantagens e desvantagens desse ambiente. A Web pode ser vista como uma grande rede constituída a partir da união de várias redes locais (**LAN** – *Local Area Network*) espalhadas pelo mundo inteiro, dessa forma, é possível acessar um banco de dados localizado em uma rede local distinta através da malha de rede formada pelo ambiente Web. Contudo, as redes locais, que integram a Web, são independentes entre si, podendo conectar-se ou desconectar-se da rede mundial a qualquer momento.

Recentemente, alguns trabalhos têm abordado a conexão de redes locais sem fio e de forma dinâmica, essas redes são conhecidas como **MANET** (*Mobile Ad hoc Networking*) [Men02]. As redes MANET possuem um raio de cobertura e os equipamentos que entram nesse raio de cobertura passam automaticamente a fazer parte da rede estabelecida. Portanto, equipamentos móveis podem trafegar por diversas áreas de cobertura de redes MANET distintas, e embora esse ambiente esteja limitado a uma pequena área de abrangência, ele possui algumas características semelhantes ao ambiente Web, tais como:

1. Alta independência dos participantes da conexão;
2. Baixo controle sobre os serviços solicitados e disponibilizados;
3. Alta tolerância às falhas de comunicação.

Esses ambientes garantem uma maior disponibilidade de bancos de dados interconectados, onde o modelo Cliente/Servidor tradicional não é adequado. Esse cenário caracteriza-se pela disponibilidade de inúmeros bancos de dados, muito provavelmente, heterogêneos, onde as estratégias de integração de bancos de dados

heterogêneos assumem um papel fundamental na concretização de todo o potencial dessa alta disponibilidade.

Um banco de dados representa conceitos do mundo real através de um modelo de dados próprio, porém essa modelagem representa apenas uma parte desse mundo real, denominada mini-mundo. Conseqüentemente, diferentes visões do mundo real podem originar mini-mundos distintos, ainda que os mesmos possam ter uma área de intersecção. Além disso, técnicas de modelagem distintas podem resultar representações diferentes dos mini-mundos.

Apesar dessas diferenças, muitas vezes os conceitos do mundo real representados por bancos de dados distintos podem estar relacionados entre si de alguma forma (por exemplo, referem-se a um mesmo objeto livro). Esse trabalho denomina esse tipo de relacionamento de relacionamento conceitual. Por esse motivo, é natural que informações relevantes para o cliente (ou usuário) devam ser extraídas de bancos de dados distintos e, em seguida, combinadas para apresentar um resultado consolidado. Esse processo de extração e combinação é conhecido como integração de bancos de dados. Podem ser identificadas várias situações onde a integração desses bancos de dados seria vital, tais como: apoio à tomada de decisões, fusões ou parcerias entre empresas, sondagem de clientes e fornecedores.

A integração de bancos de dados heterogêneos é uma questão complexa dentro da tecnologia de banco de dados, pois, para fornecer uma resposta consolidada ao usuário, é necessário eliminar as diferenças existentes (resolução de conflitos) entre os bancos de dados heterogêneos participantes. Várias abordagens têm sido propostas para viabilizar essa integração, cada uma delas sendo mais adequada a uma determinada situação. Por esse motivo, a definição da estratégia de integração mais apropriada deve considerar as características dos bancos de dados a serem integrados e o contexto em que a integração é necessária.

Uma abordagem utilizada para a integração de bancos de dados heterogêneos é a de Sistema de Bancos de Dados Federados (**FDBS** – *Federated Database System*). A implementação de um FDBS impõe a existência de um esquema global de integração, denominado esquema federado [Elm98, Özs99, Sel01], que é definido com base nos esquemas locais dos bancos de dados que compõem a federação. O esquema federado fornece, portanto, uma visão de um grande e único banco de dados. Essa abordagem possui um alto grau de transparência, pois problemas de integração já estão resolvidos no esquema federado, contudo, limita a autonomia dos bancos de dados locais. Essa

limitação acontece porque modificações nos esquemas locais podem implicar em alterações no esquema federado.

Uma outra estratégia para integrar múltiplos bancos de dados heterogêneos é a de Sistema de Bancos de Dados Múltiplos (**MDBS** – *Multidatabase System*). A tecnologia de MDBS garante um maior grau de autonomia para os bancos de dados participantes da integração, denominados de Sistemas de Bancos de Dados Locais (**LDBS** – *Local Database System*) [Elm98, Özs99, Dom00]. Entretanto, utilizar essa abordagem obriga o usuário a definir questões de localização e resolução de problemas de integração, como, por exemplo, conflitos semânticos e estruturais entre os dados armazenados nos diversos LDBS's. Em um MDBS, não é necessária a definição de um esquema global de integração, pois a linguagem de consulta é utilizada como mecanismo de integração. A idéia é que os usuários sejam capazes de "enxergar" cada esquema dos LDBS's representados em um modelo comum de dados (**CDM** – *Common Data Model*) [Elm98]. Os esquemas locais representados nesse modelo comum são denominados de esquema conceitual.

Uma abordagem para integração de bancos de dados, que estejam baseados na Web ou em redes *ad hoc*, deve oferecer suporte às características desses ambientes. Portanto, a arquitetura de integração deve conferir um alto grau de autonomia para os participantes. Além disso, deve ser capaz de resolver conflitos de integração originados, principalmente, da grande heterogeneidade dos bancos de dados a serem integrados. Finalmente, deve permitir uma manipulação de dados distribuídos em diferentes locais de forma eficiente.

A figura 1.1 apresentada em [Özs91, Özs99] traz uma classificação dos sistemas de banco de dados, onde são considerados três fatores: distribuição, heterogeneidade e autonomia. Como pode ser observada na figura, a abordagem de MDBS permite maior grau de distribuição, heterogeneidade e autonomia entre os bancos de dados locais. A diferença apresentada entre as abordagens MDBS e FDBS, em relação à autonomia, pode ser explicada em função da necessidade que a abordagem FDBS tem em manter o esquema federado a partir dos bancos de dados locais.

Outra característica de ambientes como a Web e redes *ad hoc*, que favorece a adoção da abordagem de MDBS, é a dificuldade em implementar uma administração central capaz de manter o esquema federado. Essa administração pode tornar-se inviável, devido às decisões estratégicas que visam preservar a independência da administração dos bancos de dados locais.

Nos últimos anos, a utilização do **XML** (*eXtensible Markup Language*) para representar dados semi-estruturados tem-se mostrado ideal para disponibilizar dados na Web. Isto se deve à propriedade do XML de ser capaz de disponibilizar dados e uma descrição desses dados (metadados) [BrP00]. Desse fato, decorre a facilidade em construir programas (*parsers*) capazes de validar e interpretar os documentos XML. Essa característica é essencial para possibilitar a troca e integração de dados. Outra vantagem da utilização do XML é poder representar fontes de dados que não podem ser caracterizadas como bancos de dados. Dessa forma, é possível representar dados desprovidos de uma estrutura regular rígida (por exemplo, documentos HTML).

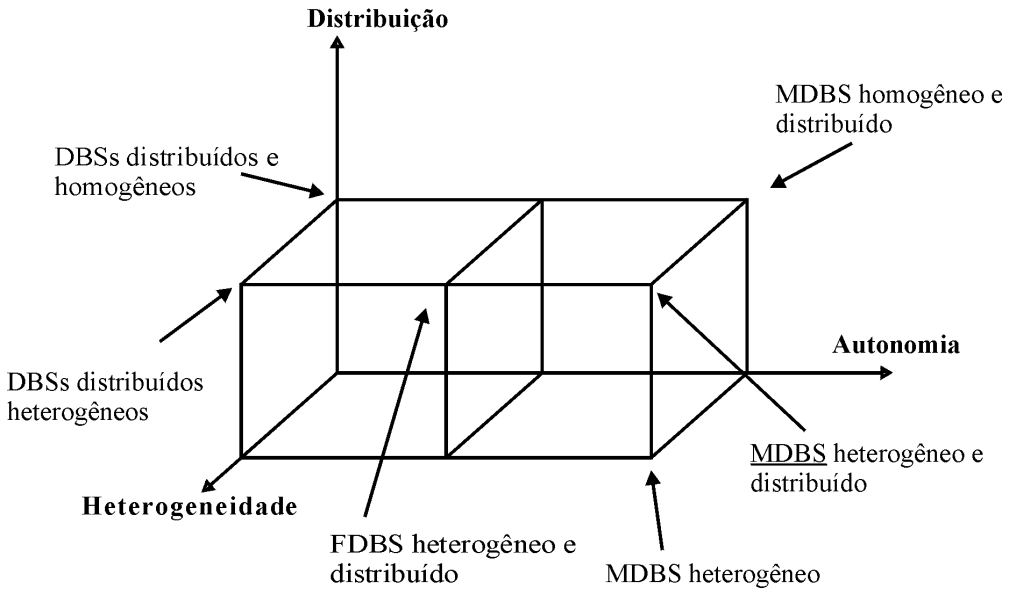

Figura 1.1. Classificação de Arquiteturas de Bancos de Dados.

Alguns trabalhos discutem a utilização do XML como meio de possibilitar a integração através da abordagem de banco de dados federados [Man01, Sel01]. Embora a presença de um esquema global de integração (esquema federado) imponha restrições de autonomia às fontes de dados, ainda não existem propostas para integrar fontes de dados disponibilizadas na Web com base na tecnologia de MDBS. Portanto, este trabalho propõe uma estratégia de integração de fontes de dados Web através da tecnologia de sistema de bancos de dados múltiplos (MDBS).

A idéia básica desta estratégia é utilizar o XML como modelo comum (esquema conceitual) dos dados armazenados nas múltiplas fontes de dados. Conseqüentemente, deve-se utilizar uma linguagem de consultas que seja capaz de manipular bancos de dados múltiplos de forma integrada e baseada em um modelo de dados XML.

1.2 Objetivos do Trabalho

Algumas linguagens de consulta para dados XML têm sido propostas. Dentre elas, a XQuery tem-se destacado, pois está sendo analisada pelo **W3C** (*World Wide Web Consortium*) e deve tornar-se uma recomendação desse órgão [Boa03]. Contudo, ainda não se pode identificar nessas linguagens mecanismos que possibilitem manipular eficientemente bancos de dados múltiplos. Por esse motivo, este trabalho apresenta, como contribuição principal, uma extensão para a linguagem XQuery, denominada MXQuery que oferece suporte à integração de múltiplas fontes de dados sem a necessidade da utilização de esquema de integração.

Portanto, o objetivo deste trabalho é descrever uma extensão à linguagem XQuery que permita o acesso a múltiplas fontes de dados de modo integrado e de acordo com a abordagem MDBS, e uma estratégia para o processamento de consultas MXQuery em uma arquitetura MDBS. A relevância desta dissertação está na necessidade de integrar fontes de dados disponibilizadas na Web ou em redes *ad hoc*, onde a autonomia deve ser preservada e a manutenção de um esquema global é muito cara ou inviável. Além disso, esse trabalho está baseado em tecnologias de grande interesse de estudos e pesquisas, sem, contudo, ser penalizado pelo uso de padrões pouco consolidados.

1.3 Organização da Dissertação

O restante dessa dissertação está organizado como se segue.

- Capítulo 2. Esse capítulo apresenta conceitos básicos sobre a interação entre bancos de dados distintos, destacando alguns fatores que servem para classificá-los. Além disso, aborda algumas estratégias para integração de bancos de dados.
- Capítulo 3. Discute alguns aspectos sobre dados semi-estruturados e as características do padrão XML que aderem aos conceitos referentes à representação de dados semi-estruturados. Além disso, algumas linguagens de consulta para documentos XML são comentadas, onde as deficiências dessas linguagens para integração de bancos de dados são destacadas.
- Capítulo 4. O capítulo 4 apresenta a linguagem MXQuery, implementações de consultas nessa linguagem, resolução de conflitos e apresenta uma estratégia para contornar problemas de disponibilidade das fontes de dados.

- Capítulo 5. Esse capítulo explora os detalhes da arquitetura de integração utilizada nesse trabalho. Os aspectos principais de um MDBS são abordados e uma sugestão para a solução do problema é apresentada.
- Capítulo 6. São apresentadas as considerações finais sobre o trabalho desenvolvido, contribuições e sugestões para trabalhos futuros.

CAPÍTULO 2

INTEGRAÇÃO DE BANCOS DE DADOS: TAXONOMIA E ESTRATÉGIAS

2.1 Introdução

Vários fatores devem ser considerados para a definição da estratégia de integração de bancos de dados a ser adotada, onde a escolha de uma estratégia inadequada pode comprometer a viabilidade do processo de integração e, conseqüentemente, redundar em prejuízo para o gestor desse processo. Por esse motivo, torna-se essencial caracterizar a arquitetura de comunicação, a localização e distribuição dos dados, o nível de acoplamento desejável (ou possível) entre os LDBS's e as diferentes concepções do mundo real representadas nos projetos dos LDBS's.

Esse trabalho está direcionado à integração de fontes de dados baseadas em ambientes cujas arquiteturas de comunicação são flexíveis e dinâmicas. Ambientes flexíveis são caracterizados por possibilitarem a interconexão de equipamentos tecnologicamente distintos, utilizando sistemas operacionais e protocolos de comunicação, também, distintos. Além disso, esses ambientes podem abranger áreas de cobertura de tamanho variado, desde redes locais até redes de longa distância (**WAN** – *Wide Area Network*).

Ambientes dinâmicos caracterizam-se por permitir a conexão e desconexão de seus participantes de forma dinâmica [Pap95], onde um equipamento (ou nó de rede) possui total autonomia para entrar na comunidade, disponibilizar e solicitar serviços e, de forma análoga, deixar a comunidade e suspender os serviços disponibilizados. De acordo com a concepção dessa arquitetura, o participante desses ambientes dinâmicos pode ser um nó de uma rede local. Conseqüentemente, esse nó funciona como um *gateway* entre a rede local e a rede externa. Esse tipo de arquitetura é o fundamento do funcionamento da Web e das redes MANET.

Os sistemas de bancos de dados foram concebidos originalmente como centralizadores de acesso e controle de dados [Lit90], porém a evolução tecnológica dos ambientes de comunicação possibilitou a distribuição de funcionalidade dos sistemas gerenciadores de bancos de dados e, posteriormente, o armazenamento distribuído desses mesmos dados. São alguns benefícios da distribuição: redução da carga de trabalho dos servidores, proximidade entre os dados e os clientes, maior disponibilidade dos dados e menor tempo de acesso [She90, Özs91, Özs99, Elm98].

Com o amadurecimento das técnicas empregadas em sistemas de bancos de dados distribuídos, surgiu a possibilidade de tornar sistemas de bancos de dados independentes inter-operáveis. O grau de dependência existente entre esses bancos de dados determina a autonomia de cada banco de dados [Özs91, Özs99], onde um banco de dados totalmente autônomo deve ser capaz de gerenciar seus dados sem interferência de qualquer componente externo, ou seja, tal banco de dados não necessita de nenhum tipo de permissão ou consulta a um outro banco de dados para realizar operações sobre o seu conjunto de dados. Pode-se dizer que um banco de dados totalmente autônomo desconhece a existência de qualquer outro banco de dados (isolamento total) [Özs99].

A heterogeneidade entre bancos de dados locais é definida como a falta de um padrão em seus projetos de bancos de dados ou modelos de dados ou linguagens de consulta ou, ainda, equipamentos e protocolos adotados [Özs91, Özs99, Dog95]. A heterogeneidade de bancos de dados é uma decorrência direta da existência de sistemas de bancos de dados independentes, onde, normalmente, os bancos de dados locais são projetados sem considerar a necessidade de uma futura integração com outros bancos de dados.

As características apontadas até aqui são descritas, mais detalhadamente, na seção 2.2 desse capítulo. As seções 2.3 e 2.4 abordam alguns aspectos de estratégias de integração de bancos de dados.

2.2 Taxonomia

Os sistemas de bancos de dados podem ser classificados sob diversos aspectos diferentes. Porém, esse trabalho apresenta uma classificação baseada em 3 (três) propriedades de um processo de integração: distribuição, autonomia e heterogeneidade [She90, Özs91, Özs99, Elm98].

2.2.1 Distribuição

O primeiro aspecto a ser analisado sobre a integração entre bancos de dados é a distribuição. A distribuição se refere ao modo como os dados são acessados, controlados e armazenados pelos equipamentos que fazem parte do ambiente de comunicação. Existem, portanto, 3 (três) formas básicas de distribuição [Sil99]:

- Função. Esse tipo de distribuição determina que parte da funcionalidade executada pelo DBMS será executada em máquinas distintas. A distribuição de função caracteriza a arquitetura Cliente/Servidor;
- Controle. Esse tipo de distribuição especifica que as atividades do DBMS serão executadas por mais de um processador. Os sistemas de bancos de dados paralelos utilizam a distribuição de controle;
- Dados. A distribuição de dados implica na fragmentação (ou cópia) dos dados por mais de um DBMS. Os sistemas de bancos de dados distribuídos implementam a distribuição de dados.

Distribuição de Dados

A distribuição de dados, citada acima, está diretamente ligada aos aspectos de integração de bancos de dados. Os dados podem ser distribuídos em diversos bancos de dados, que podem estar centralizados em uma máquina ou distribuídos geograficamente, desde que partilhem algum meio de comunicação. Vários critérios podem ser utilizados para promover a distribuição de dados [Özs99, Sil99]:

- Fragmentação Vertical. Considerando a álgebra relacional, esse tipo de distribuição utiliza o resultado de uma projeção para distribuir colunas (ou atributos) por diversos bancos de dados ($S' \leftarrow \pi_{\text{<lista de atributos>}}(S)$);
- Fragmentação Horizontal. Considerando a álgebra relacional, esse tipo de distribuição utiliza o resultado de uma seleção para distribuir linhas (ou tuplas) por diversos bancos de dados ($S' \leftarrow \sigma_{\text{<condição>}}(S)$);
- Replicação. Nessa forma de distribuição, cópias idênticas (ou parciais) de um banco de dados são mantidas em mais de uma localização. Assim, uma mesma consulta pode ser realizada em mais de um banco de dados e de acordo com a conveniência do usuário ou do projeto de distribuição implementado.

Para exemplificar a utilização de um desses critérios, pode-se imaginar uma empresa multinacional que opte por manter os dados referentes às suas filiais em bancos de dados situados em suas próprias filiais. Considerando que um determinado atributo de uma relação a ser fragmentada identifique a filial, o critério de fragmentação horizontal deve ser utilizado. Nesse caso, a condição (ou critério) de seleção a ser adotado será o atributo de identificação da filial.

Em um ambiente de dados distribuídos, existe uma etapa adicional no processamento de consultas que é a localização dos dados. Uma consulta, ao ser submetida a um banco de dados distribuído, é decomposta de acordo com os critérios utilizados para a distribuição e as sub-consultas geradas nesse processo de decomposição são encaminhadas para os diversos bancos de dados. Finalmente, os resultados são consolidados e entregues aos usuários. De acordo com o tipo de arquitetura de distribuição de dados utilizada, o processo de localização de dados e, conseqüentemente, a integração desses dados, é mais ou menos complexo. Assim, a distribuição de dados pode ser classificada de acordo com a complexidade e estratégia utilizada para a localização dos dados, como se segue.

2.2.1.1 Arquitetura Centralizada

A arquitetura centralizada (ou não distribuída) não possibilita qualquer tipo de distribuição de dados [Özs99], pois, geralmente, os equipamentos que compõe esse ambiente não possuem capacidade de processamento e armazenamento de dados (memória ou disco). O sistema centralizado é encarregado de gerenciar o banco de dados e suprir os equipamentos conectados com os aplicativos de acesso e apresentação dos dados.

Embora um sistema composto por um computador central (*mainframe*) e vários terminais seja a representação mais obvia desse tipo de arquitetura, a arquitetura centralizada pode ser adotada em uma rede de computadores. Nesse caso, as máquinas conectadas à rede local não utilizariam sua capacidade de processamento para manipular ou armazenar os dados.

A arquitetura centralizada possui apenas uma única máquina detentora dos dados e responsável por executar o DBMS. Geralmente, os usuários utilizam-se de terminais que se conectam diretamente ao computador principal. Nesse tipo de arquitetura, não é

necessário nenhum tipo de processo de localização ou definição (decisão) de qual equipamento (*host*) deve-se solicitar o serviço de banco de dados.

Considerando as atividades executadas por um DBMS como controle de concorrência, concessão de bloqueios e permissões de acesso, a arquitetura centralizada implementa tais operações mais facilmente. Contudo, esse tipo de arquitetura tende a possuir um custo de implementação mais elevado, além de exigir maior capacidade de processamento da máquina central, uma vez que todas as atividades do DBMS e demais serviços estão concentradas nela. Por isso, esse tipo de arquitetura tende a ser menos eficiente, estando, dessa forma, obsoleta.

2.2.1.2 Arquitetura Cliente/Servidor

A arquitetura cliente/servidor pode ser implementada de várias maneiras, onde a mais simples consiste na conexão de vários clientes a um único servidor. Esse tipo configuração da arquitetura Cliente/Servidor não utiliza a distribuição de dados, portanto, em relação à localização dos dados, é similar à arquitetura centralizada, já que os dados são armazenados em um único computador central (servidor). Entretanto, como visto anteriormente, existe uma distribuição na funcionalidade executada pelo sistema de banco de dados.

Contudo, a arquitetura cliente/servidor pode permitir, ainda, a conexão de vários clientes a vários servidores, onde nesse caso, existe uma distribuição de dados do ponto de vista do cliente. Por esse motivo, o processo de localização é mais complexo e deve ser utilizado algum mecanismo para a resolução desse problema.

Existem duas estratégias de gerenciamento de conexão para a resolução do problema de localização [Özs99]: a primeira confere ao cliente a responsabilidade por gerenciar a sua conexão com o servidor apropriado, portanto o cliente deve ser responsável por localizar o servidor (cliente robusto); a segunda adota um servidor local, onde cada cliente envia suas solicitações a um único servidor e, após receber as solicitações, o servidor (ou servidores) local encaminha as solicitações ao servidor apropriado.

Nesse último tipo de implementação da arquitetura cliente/servidor, o cliente não percebe a existência da distribuição dos dados, o que caracteriza a transparência de localização [Özs99].

2.2.1.3 Arquitetura Não Hierárquica (*Pier-to-Pier*)

A arquitetura não hierárquica está voltada, exclusivamente, para a solução do problema da distribuição de dados. Os dados são distribuídos em diversos bancos de dados de acordo com regras de fragmentação e/ou replicação e, em seguida, é utilizado o modelo da arquitetura não hierárquica para representar os esquemas gerados e as regras de sua formação. A principal finalidade da utilização da arquitetura não hierárquica é garantir a transparência de localização.

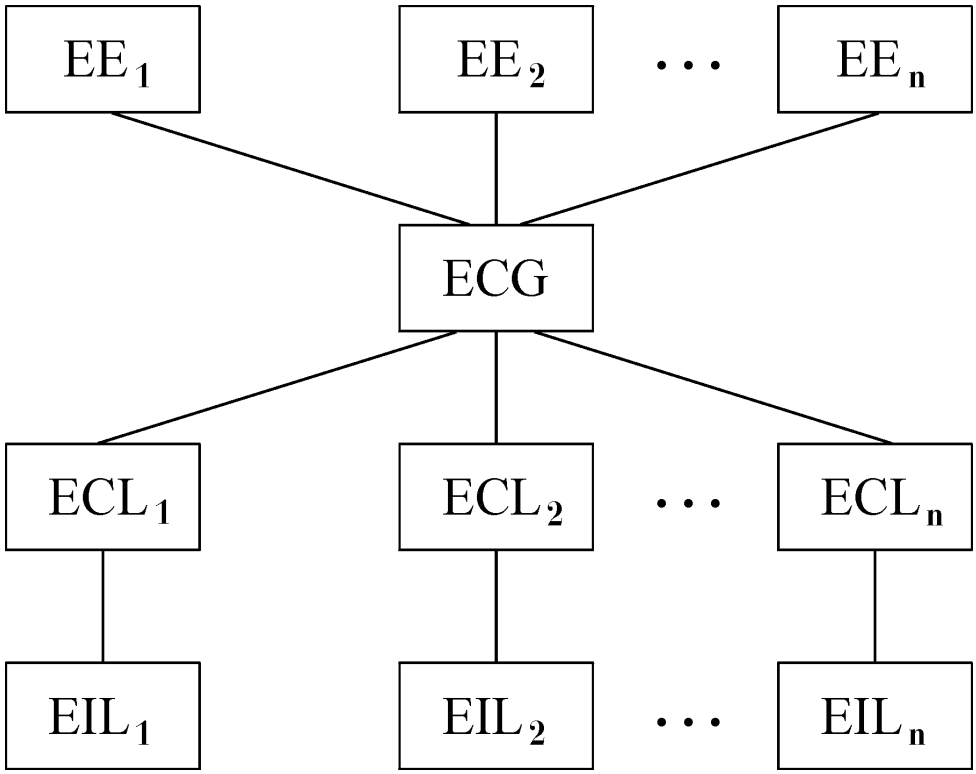

Figura 2.1. Arquitetura de Banco de Dados Distribuído de Referência.

Segundo esse modelo apresentado na figura 2.1 [Özs99], cada banco de dados possui um esquema denominado esquema interno local (**EIL**). Existe um esquema conceitual local (**ECL**) associado a cada EIL, com a função de definir as estratégias de fragmentação que devem ser utilizadas para obtenção de cada banco de dados. Acima dos ECL's existe um esquema conceitual global (**ECG**), cuja finalidade é representar de maneira integrada todos os dados existentes nos bancos de dados distribuídos. Finalmente, cada usuário acessa o ECG por meio de um esquema externo (**EE**), o qual define permissões de acesso concedidas para cada usuário.

A transparência de localização é obtida nesse modelo, através dos esquemas conceituais locais e do esquema conceitual global. Dessa forma, o usuário submete suas consultas independentemente da localização dos dados e, posteriormente, a implementação do modelo não hierárquico deve resolver o problema de localização consultando as regras de fragmentação ou replicação estipuladas nos esquemas conceituais locais.

Nesse tipo de arquitetura, a complexidade do processo de localização é resolvida através da manutenção dos esquemas conceituais. Assim, o cliente não precisa realizar qualquer tipo de processamento para definir (ou decidir) de qual servidor serão solicitados os dados de sua consulta. O processo de distribuição de dados adota a estratégia *top-down*, onde os dados, inicialmente integrados, são divididos de acordo com as regras de fragmentação.

2.2.2 Autonomia

Basicamente, autonomia local reflete o fato dos LDBS's terem sido projetados e implementados independentemente, sem ter conhecimento de um processo de integração futuro [Bra01]. De acordo com essa definição, a dimensão de autonomia só tem sentido ser avaliada na existência de dados distribuídos, ainda que os dados distribuídos residam na mesma máquina.

A autonomia define o nível de controle exercido pelos bancos de dados locais sobre os seus dados e metadados. Considerando as atividades exercidas no gerenciamento de um banco de dados, é possível identificar 4 (quatro) tipos de autonomia [She90, Bou95, Elm98, Özs99]:

- Autonomia de Projeto. A autonomia de projeto garante ao banco de dados local a liberdade de determinar as características de sistema gerenciador de banco de dados que deseja (ou pode) implementar e de que forma essa implementação será feita. Portanto, estabelecer padrões de modelos de dados ou nomenclatura de objetos viola a autonomia de projeto. Esse tipo de autonomia é responsável pelos problemas de heterogeneidade semântica (detalhada mais à frente);
- Autonomia de Comunicação. Uma das funções do DBMS é garantir a consistência do banco de dados através do controle de transações. O conceito de transações também existe na integração de banco de dados

(transação global), porém a manutenção da consistência global e local é uma tarefa complexa [Bra99, Bra01]. Para manter a consistência global seria necessário obter informações dos bancos de dados locais, tais como: escalonamento de operações e grafos de bloqueio. Contudo, a autonomia de comunicação garante aos bancos de dados locais a liberdade de decidir [She90, Elm98] se vai responder, o que vai responder e quando vai responder. Conseqüentemente, as estratégias de controle de transação e detecção de *deadlocks* devem utilizar recursos que não necessitem da cooperação dos LDBS's, pois eles podem decidir não enviar as informações solicitadas;

- Autonomia de Execução. A autonomia de execução determina que o LDBS pode executar suas operações sem interferência de qualquer componente externo. Dessa forma, o LDBS pode decidir a ordem de escalonamento de operações e abortar transações que violem a consistência local. Além disso, o LDBS não precisa informar a qualquer componente externo que transações abortou ou qual a ordem das operações executadas [She90]. Basicamente, um LDBS com autonomia de execução não distingue as transações externas das internas. Dessa forma, todas as transações executadas são consideradas transações locais.
- Autonomia de Associação. Esse tipo de autonomia refere-se à possibilidade de um LDBS poder associar-se ou desassociar-se de um grupo de bancos de dados, sem a necessidade de notificar sua decisão. Por esse motivo, é esperado que esse grupo de bancos de dados não tenha seu funcionamento dependente de nenhum dos LDBS's integrantes [Elm98]. Essa característica é fundamental em ambientes dinâmicos. A autonomia de associação também garante a liberdade dos LDBS's de decidir quais funções (operações suportadas) e recursos (dados gerenciados) compartilhar com os demais integrantes [Elm98].

A autonomia define até que ponto os bancos de dados locais podem operar o seu conjunto de dados de forma independente, além de definir o grau de acoplamento existente entre os LDBS's. Considerando o grau de acoplamento possível entre os LDBS's, pode-se classificar a autonomia como [Özs91, Özs99]: integração estreita, semi-autônomos e totalmente autônomos.

2.2.2.1 Integração Estreita

Esse tipo de autonomia, também conhecida como fortemente acoplada, disponibiliza uma única imagem do banco de dados inteiro para os usuários. Os administradores da federação possuem total controle sobre a criação e manutenção dos esquemas federados e sobre o acesso aos esquemas de exportação (ver figura 2.3) [Elm98]. O objetivo dessa estratégia é prover transparência de localização, replicação e distribuição.

A criação dos esquemas federados envolve uma negociação entre os administradores da federação e os administradores locais e, por vezes, essa negociação viola a autonomia local dos bancos de dados. Além disso, o esquema federado, uma vez estabelecido, raramente é modificado e, portanto, possui um caráter mais estático. Qualquer modificação nos esquemas de exportação implica em redesenhar todo o esquema federado.

Em contrapartida, nesse tipo de estratégia, é mais fácil manter uma uniformidade na interpretação semântica dos diversos objetos integrados. Isso porque o administrador da federação é responsável por determinar essa interpretação semântica.

2.2.2.2 Semi-autônomos

Esse tipo de autonomia, conhecida como fracamente acoplada, permite que o usuário acesse os esquemas de exportação e defina dinamicamente seus esquemas de integração. Por esse motivo, os usuários devem possuir conhecimento suficiente sobre os esquemas de exportação [Elm98]. A flexibilidade transferida ao usuário pode levar a diferentes interpretações semânticas a respeito da mesma informação. Além disso, é possível que usuários distintos manipulem o mesmo conjunto de dados e, ainda assim, utilizem esquemas federados diferentes.

Essa estratégia lida melhor com modificações nos esquemas de exportação, pois os esquemas definidos pelos usuários, normalmente, utilizam apenas uma parcela da informação e, como conseqüência, redefinir esses esquemas é mais fácil.

2.2.2.3 Totalmente Autônomos

Os bancos de dados locais não têm conhecimento da existência de outros bancos de dados locais e, conseqüentemente, executam suas operações sem sofrer interferência e

sem comunicar as operações realizadas a esses bancos de dados locais. Esse tipo de autonomia, conhecida como isolamento total [Özs99], geralmente, não é garantido ao implementar-se uma estratégia que utilize integração de esquemas, pois nenhum tipo de negociação com os bancos de dados locais pode ser realizado.

Além disso, não existe nenhum controle sobre a conexão e desconexão dos bancos de dados locais e, portanto, a simples manutenção de um esquema de integração não oferece nenhuma garantia de funcionamento para o usuário. Contudo, bancos de dados disponibilizados em ambientes dinâmicos como a Web ou redes *ad hoc*, normalmente, requerem esse tipo de autonomia.

2.2.3 Heterogeneidade

A última dimensão da taxonomia apresentada refere-se às diferenças no padrão adotado por cada banco de dados local, onde tais diferenças são uma decorrência da liberdade de implementação dos DBMS's e de seus projetos de bancos de dados. Geralmente, um DBMS é implantado isoladamente e um projetista é responsável por desenvolver o projeto de banco de dados. Posteriormente, caso haja a necessidade de integrar esse banco de dados a um outro, muito provavelmente, os dois bancos de dados apresentarão características heterogêneas.

É importante notar que heterogeneidade é independente da distribuição física dos dados discutida na seção 2.2.1. Sistemas de informação ou bancos de dados podem estar em locais geograficamente remotos e ainda serem homogêneos [Elm98]. Pode-se identificar 3 (três) tipos distintos de heterogeneidade: plataforma, fonte de dados e semântica.

2.2.3.1 Heterogeneidade de Plataforma

A heterogeneidade de plataforma endereça os problemas ligados ao ambiente de execução dos bancos de dados (ou plataforma), que pode ser entendido como os recursos computacionais utilizados para implantar um DBMS. Fazem parte da plataforma de um DBMS: os equipamentos (*hardware*), os sistemas operacionais e os protocolos de comunicação.

Considerando a diversidade de plataformas existentes, é natural que os responsáveis (ou administradores) pela área de banco de dados optem por plataformas distintas em

seus projetos independentes. A opção por uma plataforma, geralmente, é norteada por características técnicas, adequação com aplicação, representatividade no mercado e preço do produto. Sistemas concebidos originalmente independentes, muito provavelmente, possuem uma plataforma heterogênea.

Uma das etapas no processo de integração de fontes de dados é a resolução de problemas advindos da heterogeneidade de plataforma. Nesse caso, bancos de dados locais implantados em plataformas heterogêneas devem utilizar um meio comum para comunicação e troca de informações. Basicamente, várias soluções existentes no mercado podem ser utilizadas para a resolução desse problema, tais como, *gateways* e banco de dados e linguagens multiplataforma. De certa forma, a Web pode ser considerada uma solução para heterogeneidade de plataforma, onde várias redes são interligadas formando a "grande rede".

Outra proposta para resolver problemas de heterogeneidade de plataforma é a utilização de camadas de transparência (*middleware*) que são responsáveis por fornecer uma interface comum aos processos requisitantes. As requisições encaminhadas ao *middleware* são traduzidas e repassadas para o nível de execução (plataforma). O projeto MIND [Dog95, Dog96] utilizou objetos **CORBA** (*Common Object Request Broker Architecture*) para exercer o papel da camada de transparência. Dessa forma, o banco de dados utilizado no projeto MIND foi construído totalmente independente de plataforma.

2.2.3.2 Heterogeneidade de Fonte de Dados

Enquanto a heterogeneidade de plataforma refere-se às características da arquitetura sobre a qual o DBMS está implantado, a heterogeneidade de fonte de dados refere-se às características do DBMS em si. Um DBMS possui uma série de características, que são responsáveis por implementar suas funcionalidades. De um modo geral, a heterogeneidade de fonte de dados é uma decorrência da adoção de padrões diferentes para implementar as características dos DBMS's.

Uma das características de um DBMS, que pode gerar a heterogeneidade de fonte de dados, é o modelo de dados adotado. Dessa forma, os primeiros DBMS's eram baseados em um modelo de dados hierárquico, posteriormente surgiram os DBMS's relacionais e orientados a objetos. O modelo de dados é uma característica fundamental, pois é a forma como os dados são representados e manipulados pelo DBMS.

Outra característica que causa heterogeneidade de fonte de dados é a linguagem de banco de dados utilizada. Essa característica está muito ligada ao modelo de dados adotado; no entanto, para um mesmo modelo de dados podem existir linguagens de consulta distintas. Conseqüentemente, um DBMS pode adotar linguagens de consulta diferente de outro DBMS, ainda que utilizando um mesmo modelo de dados.

Finalmente, as fontes de dados podem implementar funcionalidades diferentes. Por exemplo, pode-se considerar uma planilha como sendo uma fonte de dados, no entanto, essa fonte de dados não implementa nenhuma funcionalidade de um DBMS convencional.

O modo mais comum de resolver os problemas advindos da heterogeneidade de fonte de dados é através da utilização do modelo de dados comum. Os esquemas dos bancos de dados são representados através de um modelo de dados comum, conseqüentemente, as consultas podem ser expressas através de uma linguagem comum. A utilização de um CDM impõe a existência de componentes de *software* capazes de traduzir requisições baseadas no modelo de dados comum, para o modelo de dados das fontes de dados locais e vice-versa.

2.2.3.3 Heterogeneidade Semântica

Esse último tipo de heterogeneidade refere-se ao padrão utilizado nos projetos de bancos de dados, tais como nomenclatura dos objetos, restrições utilizadas, estrutura de tabelas (ou objetos) utilizada. A heterogeneidade semântica ocorre quando existe um desacordo a respeito do significado ou da interpretação de um mesmo dado em bancos de dados distintos [She90].

O grande problema da heterogeneidade semântica é que esse tipo de heterogeneidade é de difícil constatação, principalmente, de constatação automática (por exemplo, utilizando programas de computador). Geralmente, é necessária a intervenção humana, para identificação da ocorrência da heterogeneidade semântica.

A ocorrência da heterogeneidade semântica entre dois bancos de dados independe da incidência de outros tipos de heterogeneidade, ou seja, um mesmo DBMS implantado sobre arquiteturas similares pode apresentar bancos de dados semanticamente heterogêneos. Nesse caso, não seria necessário utilizar um *software* para obtenção do CDM, porém técnicas de resolução de problemas (conflitos) originados das discrepâncias semânticas devem ser utilizadas para integração dessas

fontes de dados. Como decorrência da heterogeneidade semântica, podem surgir vários tipos de conflitos entre classes pertencentes a fontes de dados distintas, que devem ser resolvidos durante o processo de integração. Alguns tipos de conflitos decorrentes da heterogeneidade semântica são:

- Conflito de Nome;
- Conflito de Domínio;
- Conflito Semântico;
- Conflito de Esquema;
- Conflito de Dados.

Os **conflitos de nome** podem ser subdivididos em dois tipos distintos: sinônimos e homônimos. Os conflitos de sinônimo ocorrem quando classes (ou atributos de classes) de fontes de dados distintas representam o mesmo conceito (semântica) do mundo real, mas apresentam nomes diferentes [Dog95, Elm98, Dom00, Law01]. Por outro lado, duas ou mais classes pertencentes a fontes de dados distintas podem representar conceitos diferentes no mundo real, no entanto apresentarem o mesmo nome [Dog95, Elm98, Dom00, Law01]. Esse fenômeno é conhecido como conflito de nome do tipo homônimo.

Os **conflitos de domínio** são caracterizados pela ocorrência do seguinte fenômeno: classes pertencentes a fontes de dados distintas representam o mesmo conceito do mundo real, porém suas propriedades são definidas através de tipos de dados (incompatibilidade de tipos) ou escala de medidas ou ainda restrições de integridade diferentes [Dog95, Dom00]. A ocorrência de um conflito de domínio implica que objetos de fontes de dados distintas, mesmo representando o mesmo conceito do mundo real, não possam ser comparados diretamente. Por isso, alguma função de conversão deve ser empregada, para possibilitar a comparação entre esses objetos. As funções de conversão devem padronizar unidades de medidas, tipos de dado e estabelecer critérios para o tratamento de restrições de integridades diferentes.

Os **conflitos semânticos** ocorrem devido à possibilidade de diferentes percepções do mundo real originarem projetos diferentes de fontes de dados [Dog95, Law01]. Essa diferença de percepção pode originar um conjunto diferente de propriedades para classes representando o mesmo conceito no mundo real. Os conflitos semânticos podem ser classificados como: classes idênticas, intersecção de classes, classes contidas e classes disjuntas [Dog95].

Conflito semântico do tipo classes idênticas ocorre quando classes pertencentes a fontes de dados distintas representam o mesmo conceito do mundo real. Na realidade, pode-se afirmar que existe um conflito semântico do tipo classes idênticas quando as propriedades de classes de fontes de dados distintas são iguais. Nesse caso, as classes que apresentam este conflito são consideradas semanticamente equivalentes. Para resolver esse tipo de conflito é necessário combinar as classes originais em uma única classe. Além disso, regras de comparação entre objetos pertencentes a essas classes, com a finalidade de especificar quando tais objetos são idênticos, devem ser especificadas. Por exemplo, objetos que possuem o mesmo identificador universal de objeto (OID) podem ser considerados idênticos, pois representam um mesmo objeto do mundo real. Nesse caso, esses objetos devem ser representados através de um único objeto.

Conflito semântico do tipo intersecção de classes acontece quando apenas algumas propriedades de classes, pertencentes a fontes distintas, são coincidentes. Para resolver este conflito, deve-se construir uma nova classe, que contenha apenas propriedades comuns às duas classes originais e, em seguida, definir especializações dessa classe recém construída contendo as propriedades não comuns.

Conflito semântico do tipo classes contidas ocorre quando uma classe está contida em outra, ou seja, suas propriedades representam um subconjunto das propriedades da outra classe. Nesse caso, a segunda classe deve ser mapeada como uma subclasse da primeira classe.

Classes disjuntas são classes que não possuem propriedades comuns, porém estão relacionadas de alguma forma, por exemplo, estudantes formandos e formados. Para resolver esse tipo de conflito, uma terceira classe, contendo a generalização dessas duas classes, pode ser criada [Dog95].

Os **conflitos de esquema**, também conhecidos como estruturais, são os mais difíceis de serem resolvidos [Kris91], pois inúmeras variações de estrutura de esquema podem existir [Kris91, Elm98, Dom00, Law01]. Por exemplo, um mesmo dado do mundo real pode representar uma tupla em uma fonte de dados, enquanto, em uma outra fonte de dados, pode representar apenas um atributo.

Em algumas situações de integração, pode-se ter objetos pertencentes a classes de fontes de dados distintas, mas que representam uma mesma entidade do mundo real (por exemplo, no caso de conflito semântico do tipo classes idênticas). Nesse caso, os objetos idênticos devem sofrer um processo, denominado fusão (ver capítulo 4),

originando um único objeto como resultado do processo de integração [Pap96]. Contudo, é possível que os objetos idênticos apresentem propriedades iguais, mas com conteúdos divergentes. Este fenômeno é conhecido como **conflito de dados** [Par98, Sat00].

2.3 Estratégias para Integração de Bancos de Dados

Existem várias abordagens para promover a integração de bancos de dados, que podem ser analisadas sob diferentes enfoques. Assim, esse trabalho classifica as arquiteturas de integração em 2 (duas) categorias principais: bancos de dados federados e bancos de dados múltiplos.

2.3.1 Bancos de Dados Federados

A arquitetura de bancos de dados federados pode ser vista como uma comunidade de bancos de dados, que cooperam entre si para constituir a federação. O nível de cooperação existente entre os participantes da federação é variável, de tal forma, que os bancos de dados participantes podem ser mais ou menos autônomos. De acordo com a classificação de autonomia apresentada na seção 2.2.2, os bancos de dados locais podem ser fracamente ou fortemente acoplados [Elm98].

Uma federação de bancos de dados fracamente acoplados especifica que os bancos de dados participantes possuem autonomia de projeto, comunicação e execução. A autonomia de associação não é totalmente satisfeita, pois a decisão de deixar a federação não pode ser tomada unilateralmente. Entretanto, cada participante da federação pode ser responsável por definir e manter seu próprio esquema federado.

Uma federação constituída de bancos de dados fortemente acoplados denota uma relação maior de interdependência dos participantes. Nesse tipo de federação, as autonomias de projeto, de comunicação e de execução continuam garantidas, no entanto, a autonomia de associação é totalmente violada. Nesse tipo de arquitetura existe um administrador da federação que é responsável por manter o esquema federado. Esse administrador acessa os esquemas de exportação e determina as regras de integração para formação do esquema federado.

Embora não seja uma regra, muitas vezes as autonomias de comunicação e de execução também são sacrificadas. Esse tipo de concessão deve-se ao fato de possibilitar um controle mais eficiente sobre as transações globais definidas sobre a federação, e conseqüentemente, um melhor controle sobre a consistência dos bancos de dados participantes dessa federação.

2.3.1.1 Esquema Federado

O esquema federado é uma representação do esquema global de integração, onde são especificadas todas as informações disponibilizadas pelas fontes de dados através da federação de bancos de dados para o usuário. O usuário recorre ao esquema federado (ou esquema global) para propor consultas, sendo que, tais consultas são expressas como se existisse apenas um único e homogêneo banco de dados. Portanto, os problemas de heterogeneidade foram completamente solucionados antes da obtenção do esquema federado.

Analisando o modelo abstrato apresentado na figura 2.6 (ver seção 2.3.3), e de acordo com a arquitetura de bancos de dados federados, o componente de acesso ao esquema conceitual é um outro mediador, que é responsável por especificar o esquema federado. Esse mediador deve resolver os problemas de heterogeneidade semântica remanescentes e, em seguida, apresentar o esquema federado utilizando o modelo de dados comum adotado pela arquitetura de integração. Acima desse último mediador pode existir uma interface de consulta do usuário, porém, nem a interface, nem a linguagem de consultas utilizada nessa arquitetura deve prover mecanismos de resolução de conflitos.

O esquema federado pode ser estático ou dinâmico. O primeiro é uma característica dos bancos de dados fortemente acoplados, onde apenas um único esquema federado global é constituído e acessado pelos usuários. O objetivo dessa característica é garantir uma transparência de localização, distribuição e replicação [Elm98]. Contudo, a evolução dos esquemas dos bancos de dados locais implica na redefinição do esquema federado.

O esquema federado dinâmico é definido a qualquer momento pelo usuário que deve conhecer os esquemas de exportação, violando assim a transparência de localização e distribuição. Por outro lado, os participantes da federação são mais autônomos (fracamente acoplados).

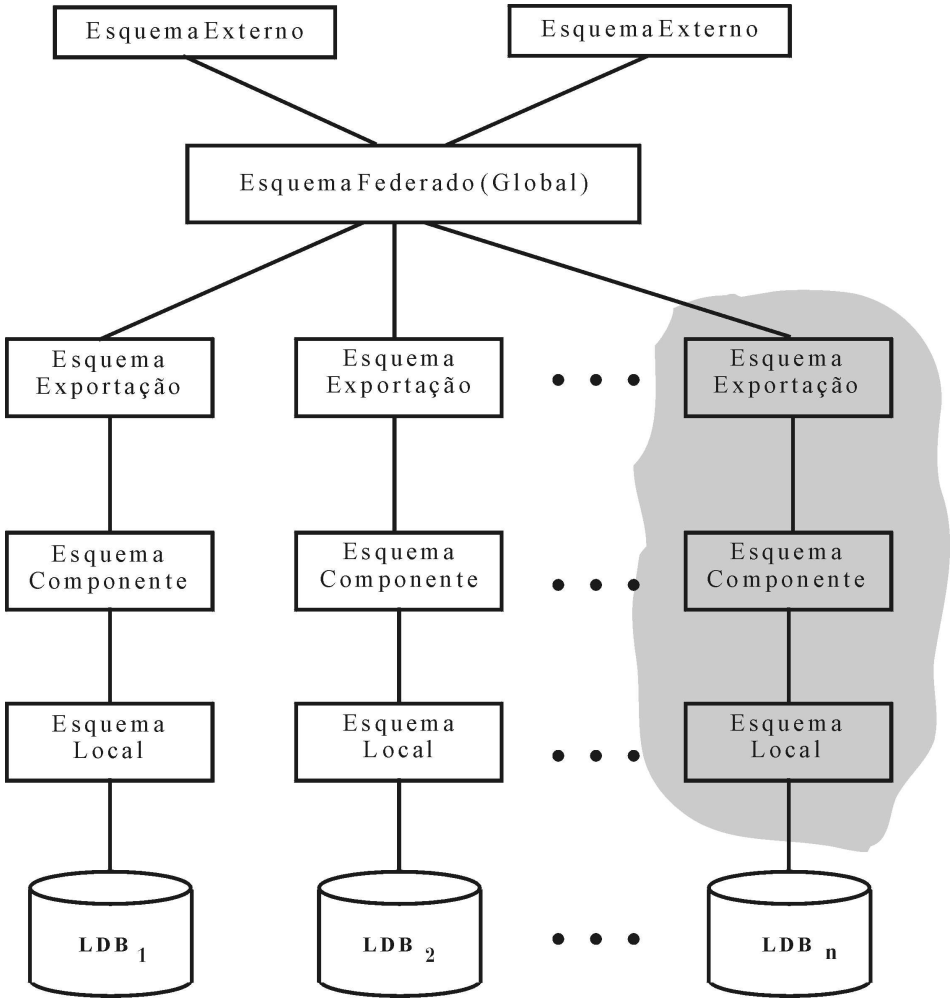

Figura 2.2. Esquema de Integração de Banco de Dados Federados.

2.3.1.2 Arquitetura de Integração

A figura 2.2 apresenta o esquema de integração utilizando a estratégia de bancos de dados federados, onde é possível identificar 5 (cinco) camadas de esquemas distintas:

- Esquema Local. Esse esquema representa a descrição dos dados de cada fonte de dados, onde é utilizado o modelo de dados adotado por cada uma dessas fontes de dados;
- Esquema Componente. O esquema componente representa a descrição dos dados de cada fonte de dados, porém, utilizando um modelo de dados comum a todos os esquemas componentes. Dessa forma, pode ser necessário implementar regras de transformação entre o esquema local e o esquema componente de cada fonte de dados;
- Esquema Exportação. No esquema exportação é definida a parte do esquema local acessível à federação;

- Esquema Federado. O esquema federado apresenta um esquema integrado de todos os esquemas exportação. Para tanto, devem ser resolvidos todos os conflitos de integração e redundâncias de informação provenientes dos esquemas exportação;
- Esquema Externo. Esse esquema apresenta visões sobre o esquema federado;

Os esquemas locais, componentes e exportações devem ser definidos por um administrador de banco de dados (**DBA** – *DataBase Administrator*) local. Como esse DBA local está limitado à administração do seu banco de dados local, ele não necessita conhecer detalhes de implementação de outros bancos de dados. Em contrapartida, o esquema federado e os esquemas externos devem ser definidos pelo DBA da federação, pois é necessário que o mesmo conheça todos os esquemas de exportação das fontes de dados a serem integradas.

2.3.2 Bancos de Dados Múltiplos

A principal diferença entre a arquitetura de bancos de dados federados e a arquitetura de bancos de dados múltiplos é que a arquitetura dos FDBS's supõe que os bancos de dados locais têm conhecimento da existência do esquema global integrado. Isso se reflete na necessidade de cada banco de dados local em exportar seu esquema e partilhar informações sobre a evolução de seus esquemas. Em contrapartida na abordagem MDBS isso não acontece, pois não é gerado nenhum esquema de integração. Portanto, existe em termos de autonomia, o que foi classificado na seção 2.2.2 como isolamento total. Os bancos de dados locais não têm qualquer conhecimento a respeito da existência de outros bancos de dados locais, conseqüentemente, nenhuma restrição em suas operações locais é feita.

A arquitetura de MDBS satisfaz a autonomia de associação, pois cada banco de dados local decide, unilateralmente, conectar-se ou desconectar-se do ambiente de comunicação. Além disso, as autonomias de projeto, comunicação e execução continuam sendo satisfeitas. Por ser uma característica fundamental em ambientes dinâmicos, a garantia da autonomia de associação confere à arquitetura de bancos de dados múltiplos uma importante vantagem sobre os bancos de dados federados. Por outro lado, a transparência de localização deve ser sacrificada. Conseqüentemente, o

usuário deve indicar quais fontes acessar e, sobretudo, como resolver os conflitos resultantes dessa integração.

2.3.2.1 Linguagens de Consulta

A arquitetura de banco de dados múltiplos não utiliza mediadores para resolver os problemas de integração, portanto algum outro mecanismo deve ser utilizado para tratar tais problemas, em especial os problemas resultantes da heterogeneidade semântica. Nessa arquitetura, a linguagem de consultas desempenha o papel feito pelos mediadores da arquitetura dos bancos de dados federados, assim sendo, a consulta deve conter informações a respeito da resolução de conflitos e localização de fontes de dados. Desse modo, a linguagem de consultas para bancos de dados múltiplos (**MDL** – *Multidatabase Language*) deve possuir, em sua sintaxe, declarações ou construções que não sejam necessárias em linguagens de consulta para bancos de dados. Essas construções adicionais definem a constituição dos bancos de dados múltiplos, a resolução de conflitos de integração e execução de funções agregadas sobre os dados integrados. Naturalmente, uma consulta expressa através de uma MDL é mais complexa, pois deve ser capaz de definir a relação de integração existente entre fontes de dados distribuídas, heterogêneas e autônomas.

MSQL – Multidatabase SQL

A manipulação de dados de forma integrada requer funções que não existem nas linguagens de manipulação de bancos de dados clássicas [Lit89]. Por esse motivo, uma extensão à linguagem **SQL** (*Structured Query Language*), denominada MSQL, foi proposta, onde são definidas construções para manipulação de bancos de dados múltiplos baseados em um modelo de dados relacional. A linguagem SQL e o modelo de dados relacional foram responsáveis pela grande aceitação do MSQL como linguagem de banco de dados múltiplos. Conseqüentemente, a linguagem MSQL foi empregada em diversos projetos de integração de bancos de dados [Elm98].

Por definição, qualquer função presente na sintaxe da linguagem SQL compõe a sintaxe da linguagem MSQL, além da introdução de novas funções não procedurais para manipulação de dados armazenados em bancos de dados relacionais distintos. Resumidamente, essas novas funções de manipulação possuem as seguintes funcionalidades [Lit89]:

1. Utilização de uma única declaração para criar ou alterar qualquer número de bancos de dados;
2. Recuperação ou modificação envolvendo a junção de dados em diferentes esquemas de bancos de dados;
3. Requisição ou modificação transmitidas por difusão (*broadcasting*) sobre qualquer número de bancos de dados, de acordo com regras de resolução de conflitos;
4. Transformação dinâmica do significado dos atributos e unidades de medida, de acordo com as definições do usuário;
5. Consultas de bancos de dados para definir o fluxo de dados entre os bancos de dados;
6. Agregação dinâmica de dados pertencentes a diferentes bancos de dados;
7. Criação de visões de bancos de dados múltiplos;
8. Criação de objetos auxiliares, tais como *triggers* e consultas pré-armazenadas.

```
USE banks
LET x BE br branch
SELECT *
FROM x
WHERE street=‘Av. Champs Elysées’
```

Figura 2.3. Exemplo de Utilização da Linguagem MSQL.

A figura 2.3 apresenta um exemplo da utilização da linguagem MSQL extraído de [Lit89]. A primeira linha identifica qual MDBS deve ser utilizado, em seguida, especifica que a variável semântica “x” representará as tabelas “br” ou “branch” pertencentes a algum dos bancos de dados locais participantes do MDBS “banks”. A cláusula de seleção utiliza a variável “x” e seleciona as tuplas de todos os bancos participantes do MDBS “banks”, de acordo com a condição “street=‘Av. Champs Elysées’”.

Em [Gra93] foi apresentada uma extensão à álgebra relacional, onde algumas das funções adicionais introduzidas pela linguagem MSQL foram formalizadas. Embora a álgebra multirelacional proposta não tenha formalizado totalmente a linguagem MSQL, sua importância reside na apresentação de regras para reescrever consultas MSQL em consultas SQL.

IDL – Interoperable Database Language

A linguagem IDL também adota o modelo de dados relacional, porém utiliza uma sintaxe diferente da linguagem SQL e MSQL. A linguagem proposta permite endereçar os problemas abordados pela linguagem MSQL, porém a principal característica da linguagem IDL é resolver problemas de conflito de esquemas (ver seção 2.2.3.3), denominados de discrepâncias esquemáticas [Kri91]. Embora parte desses conflitos possa ser resolvida através da linguagem MSQL, alguns problemas não encontram em MSQL uma solução, como, por exemplo, integrar informações representadas como dado em um esquema e metadado em outro esquema.

Além de permitir a resolução de problemas de discrepâncias esquemáticas, a linguagem IDL possui operações de atualização de bancos de dados que possibilitam a obtenção e manipulação de conjuntos de dados heterogêneos [Kri91]. Esses conjuntos heterogêneos podem ser definidos como dados de estruturas irregulares ou dados semi-estruturados. Embora dados semi-estruturados sejam incompatíveis com os bancos de dados que adotam o modelo de dados relacional, essa característica demonstra a flexibilidade da linguagem IDL.

```
?.chwab.r(.date=D,.S=P),
   .ource.S(.date=D,.clsPrices=P)
```

Figura 2.4. Exemplo de Utilização da Linguagem IDL.

A figura 2.4 traz um exemplo da utilização da linguagem IDL, que foi extraído de [Kri91]. Essa consulta se baseia em um universo que possui dois bancos de dados “chwab” e “ource” e busca identificar estoques pertencentes a bancos de dados distintos que tiveram o mesmo preço de fechamento. O banco “chwab” possui uma relação “r” composta do atributo “date” e vários atributos de preço de fechamento, o nome do atributo do preço de fechamento é igual ao nome do estoque que ele representa. O banco “ource” possui várias relações e cujo nome é igual ao estoque que representa. Essas relações são compostas pelos atributos “date” e “clsPrice”.

No exemplo, “S”, “D” e “P” são variáveis que ora representam um dado e ora representam um metadado. Assim, a consulta especifica que o atributo “date” das relações “chwab.r” e “ource.S” deve possuir o mesmo valor. Além disso, determina que o atributo de preço de fechamento em “chwab” a ser utilizado deve possuir o mesmo nome da relação de “ource” a ser utilizada. Finalmente, determina que o valor do preço de fechamento deve ser igual nas duas relações.

A principal contribuição da linguagem IDL foi identificar o problema da discrepância esquemática não resolvida pela linguagem MSQL, além de apontar uma possível forma de solução.

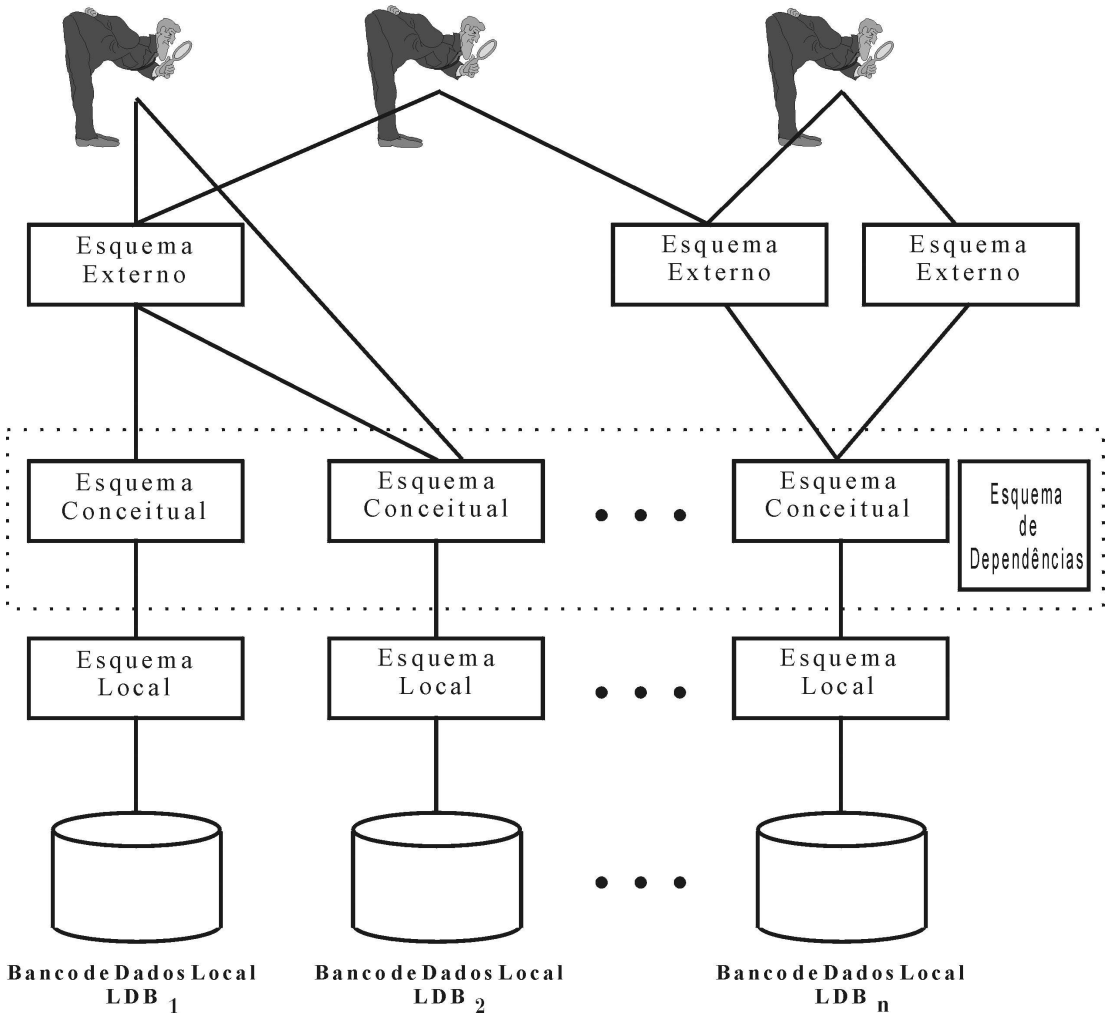

Figura 2.5. Esquema de Integração de Banco de Dados Múltiplos.

2.3.2.2 Arquitetura de Integração

A figura 2.5 apresenta o esquema de integração utilizando a estratégia de bancos de dados múltiplos, onde se podem identificar 4 (quatro) componentes distintos:

- Esquema Local. Esse esquema representa a descrição dos dados de cada fonte de dados, utilizando o modelo de dados específico de cada fonte de dados;
- Esquema Conceitual. O esquema conceitual representa a descrição dos dados de cada fonte de dados, porém utilizando um modelo de dados

comum a todos os esquemas conceituais. Dessa forma, pode ser necessário implementar regras de transformação entre o esquema local e o esquema conceitual de cada fonte de dados. O esquema conceitual pode, ainda, representar apenas a parte do esquema local acessível para o MDBS;

- Esquema Externo. Esse esquema apresenta visões de integração sobre os esquemas conceituais, onde conflitos de integração são resolvidos;
- Esquema de Dependências. O esquema de dependências representa as restrições de integridade globais, onde são especificadas regras para manutenção da consistência envolvendo mais de uma fonte de dados.

Embora a representação do esquema externo possa ser comparada ao esquema federado presente na arquitetura FDBS, sua utilização não é obrigatória, ou seja, o usuário pode acessar as fontes de dados diretamente através dos esquemas conceituais. A finalidade dos esquemas externos é armazenar visões integradas de esquemas conceituais, simplificando, dessa forma, a formulação de consultas. Em contrapartida, tais visões são dependentes das evoluções dos esquemas locais e conceituais, pois alterações nesses esquemas podem repercutir em alterações nos esquemas externos.

2.3.3 Modelo Abstrato de uma Arquitetura de Integração

As várias arquiteturas, que podem ser utilizadas para promover a integração de bancos de dados, devem ter como objetivo principal a resolução eficiente dos problemas advindos das formas de integração descritas na seção 2.2. Idealmente, a arquitetura de integração deve viabilizar o máximo de distribuição, autonomia e heterogeneidade, entretanto, comumente, algum tipo de concessão é feito [She90, Bou95]. Assim, a arquitetura de integração pode optar por violar a autonomia de comunicação e estabelecer que os participantes devem informar decisões tomadas por eles.

As arquiteturas de integração são compostas por componentes, cujo principal objetivo é transformar dados das fontes de dados locais em dados mais próximos do resultado esperado pelo usuário. Conseqüentemente, os componentes dessas arquiteturas promovem refinações sucessivas dos dados até a obtenção do resultado final. A figura 2.6 apresenta um modelo abstrato de uma arquitetura de integração, onde os componentes apresentados podem ser utilizados na implementação de várias arquiteturas de integração, inclusive nas abordagens que foram descritas nas seções 2.3.1 e 2.3.2.

2.3.3.1 Modelo de Dados Comum

A figura 2.6 representa o modelo de dados comum (CDM), cujo principal objetivo é fornecer um modelo único sobre o qual se possam realizar consultas. O CDM não é propriamente um componente do modelo abstrato, na realidade, ele funciona como uma especificação de apresentação de resultados. Dessa forma, os componentes da arquitetura devem produzir resultados que sejam compatíveis com a especificação do modelo de dados comum.

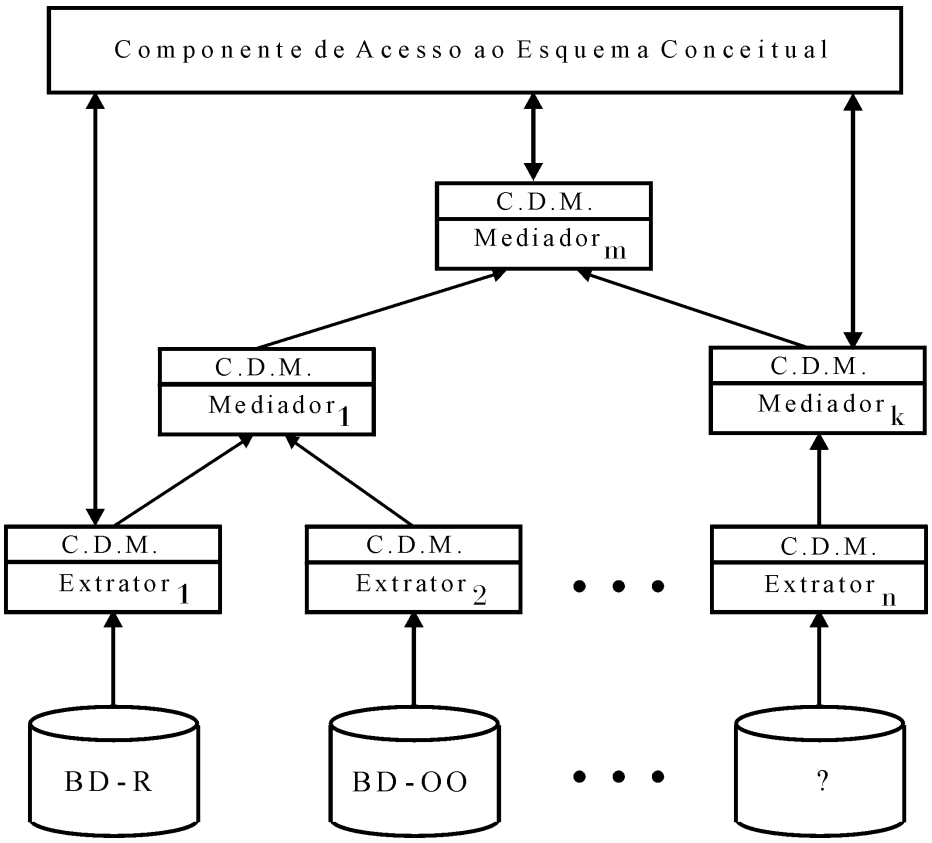

Figura 2.6. Modelo Abstrato de uma Arquitetura de Integração.

A primeira responsabilidade do administrador do processo de integração é definir o modelo de dados comum a ser utilizado. Se todas as fontes de dados participantes adotarem o mesmo modelo de dados, esse modelo de dados será um forte candidato a tornar-se o CDM. Da mesma forma, se todas as fontes de dados utilizarem modelos de dados estruturados, então optar por um modelo de dados estruturado como CDM será uma boa estratégia. Portanto, a definição do CDM depende dos tipos de fontes de dados que participarão do processo de integração.

Existem vários modelos de dados, que podem ser classificados como modelos estruturados, modelos semi-estruturados e modelos não estruturados. Um modelo de dados estruturado é caracterizado por possuir uma estrutura de dados regular e rígida, onde qualquer dado a ser inserido no banco de dados deve respeitar a estrutura previamente definida.

Um modelo de dados semi-estruturado também define uma estrutura para inserção de dados, porém essa estrutura é menos rígida. É possível inserir dados que não obedeçam à estrutura definida, ou ainda, essa estrutura é modificada de acordo com os dados inseridos. O modelo de dados semi-estruturado será abordado no capítulo 3.

Um modelo de dados não estruturado não possui uma estrutura de dados definida. Dessa forma, os dados inseridos são constituídos por textos (geralmente longos) e figuras, que não possuem qualquer referência ao seu conteúdo ou significado. Portanto, para identificar o significado de uma informação é necessário analisar o dado em si.

Devido às características das fontes de dados disponibilizadas na Web, um modelo de dados semi-estruturado mostra-se mais adequado. Esse foi um dos motivos que determinou a escolha do XML como modelo de dados para representar as fontes de dados nesse trabalho. O próximo capítulo discutirá as vantagens na utilização do XML como modelo de dados comum.

2.3.3.2 Extratores

Os extratores, representados na figura 2.6, possuem a função de mapear o esquema das fontes de dados locais em um esquema conceitual utilizando o modelo de dados comum. Algumas arquiteturas de integração utilizam os tradutores para executar essa função; entretanto, o papel dos extratores é um pouco mais complexo, pois os extratores podem recuperar informações de fontes de dados não estruturados.

Cada fonte de dados diferente deve possuir um extrator de dados correspondente, pois a função executada pelo extrator depende do conhecimento dos modelos de dados das fontes de dados locais. O produto gerado pelo extrator é uma representação dos esquemas das fontes de dados locais em um modelo de dados comum, onde nenhum outro tipo de transformação é realizado. Os extratores resolvem, portanto, o problema da heterogeneidade de fonte de dados, sem, contudo, abordar os problema de heterogeneidade semântica.

2.3.3.3 Mediadores

Os mediadores são programas que têm por finalidade refinar, através da transformação de dados ou resolução de conflitos semânticos, os esquemas conceituais recebidos. É importante observar que, tanto a entrada quanto a saída de um mediador, baseia-se em um esquema conceitual, que adere ao padrão especificado pelo modelo de dados comum. Por esse motivo, podem ser utilizados vários níveis de mediadores, possibilitando a construção de mediadores mais simples e, ao mesmo tempo, eficientes.
O refinamento sucessivo de dados extraídos das fontes locais, que é obtido através dos múltiplos níveis de mediadores, foi utilizado em [Cha94].

Embora a utilização de mediadores não signifique, necessariamente, a resolução de todos os conflitos decorrentes da heterogeneidade semântica, os mediadores produzem um esquema conceitual mais fácil de ser integrado, pois parte dos problemas já foi solucionado.

A arquitetura de integração MDBS não utiliza mediadores, pois eles podem ser programados para resolver conflitos entre fontes de dados distintas. Isso limita a autonomia das fontes de dados visto que uma alteração em sua definição pode implicar em alterações na definição dos mediadores.

2.3.3.4 Componente de Acesso ao Esquema Conceitual

O componente de acesso ao esquema conceitual é o último componente do modelo abstrato de integração. Na realidade, seu funcionamento é determinado pela arquitetura de integração implementada. Dessa forma, esse componente pode ser uma aplicação utilizando uma linguagem de consultas ou uma interface de consulta de usuários ou um novo mediador responsável por definir um esquema global de integração.

A característica do componente de acesso comum a qualquer implementação do modelo abstrato é a utilização do CDM como parâmetro de entrada. Por esse motivo, pode-se observar na figura 2.6 que o componente de acesso pode conectar-se a qualquer outro componente que produza um modelo de dados expresso em CDM. O grau de dificuldade da construção do componente de acesso depende do nível de refinamento utilizado (ou níveis de mediadores utilizados).

2.4 Linguagens de Consulta como Suporte para Integração de Bancos de Dados

Inúmeros trabalhos apresentam linguagens de consultas para bancos de dados onde é possível manipular fontes de dados distintas. Em [Lit89, Gra93], é apresentada uma linguagem MDL para manipulação de bancos de dados múltiplos. A linguagem MSQL possui mecanismos para integrar bancos de dados relacionais, que aderem ao padrão SQL de forma a permitir junções entre tais bancos de dados. Embora essa proposta confira maior expressividade à linguagem SQL, ela é limitada no tratamento de incompatibilidades estruturais. Apenas conflitos de nome (sinônimos e homônimos) são resolvidos através da utilização de variáveis semânticas e a junção entre bancos de dados é limitada a operações com atributos de domínios compatíveis [Mis95]. Além disso, Krishnamurthy et al. apontam a incapacidade da linguagem MSQL em resolver discrepâncias esquemáticas ou conflitos de esquema (ver seção 2.2.3.3) [Kri91].

Outro fator limitante da linguagem MSQL é permitir apenas a integração de esquemas relacionais. Conseqüentemente, para utilizar essa linguagem, seria necessário adotar como CDM o modelo de dados relacional. A tarefa de mapear modelos de dados pós-relacionais, em especial os baseados em dados semi-estruturados, para um modelo de dados tão rígido quanto o modelo de dados relacional é uma tarefa bastante complexa. Portanto, a MDL MSQL não é indicada para integrar fontes de dados baseadas na Web ou em redes *ad hoc*.

A proposta apresentada por Missier e Rusinkiewiez define atributos globais, que representarão o resultado integrado de uma consulta [Mis95]. Em seguida, utilizam instruções declarativas de mapeamento entre os atributos dos bancos de dados locais e os atributos globais. Após a definição dos mapeamentos ou contexto de uma consulta, as consultas, que são expressas em uma linguagem derivada do SQL, podem ser submetidas. A utilização de junções implícitas possibilita expressar consultas mais flexíveis e orientadas ao resultado.

As características da linguagem IDL apresentada por Krishnamurthy et al. já foram discutidas nesse capítulo. Tanto a proposta apresentada por Krishnamurthy et al. [Kri91] quanto à apresentada por Missier e Rusinkiewiez [Mis95] possuem mais expressividade comparada à linguagem MSQL, contudo continuam limitadas pelo modelo de dados relacional.

Em [Clu98], foi apresentado o sistema **YAT** (*Yet Another Tree-based*) para integrar fontes de dados heterogêneas. O sistema YAT utiliza extratores para representar os dados das fontes de dados locais, com modelos de dados distintos, em um modelo de dados comum baseado em árvore. Por esse motivo, o sistema é suficientemente flexível para representar dados semi-estruturados. Além disso, é possível especificar regras de conversão para definição de mediadores através da linguagem **YATL** (YAT *Language*) baseada no modelo de dados YAT. A primeira versão da YATL foi proposta com a intenção de ser uma linguagem de especificação de mediadores e, portanto, não possuía as características de uma linguagem de consultas para MDBS.

Em [Chr00, Clu00], uma nova versão da linguagem YATL foi apresentada, onde é possível especificar consultas a fontes de dados XML. Essa nova versão apresentou mais características de linguagem de consultas MDL, além de introduzir construtores capazes de mapear fontes de dados distintas em um único documento resultado. Contudo, esses trabalhos não discutiram problemas de heterogeneidade semântica e resolução de conflitos de integração, especialmente, como resolver conflitos de esquema através da linguagem YATL.

Domenig e Dittrich apresentaram uma linguagem baseada em **OQL** (*Object Query Language*), onde os dados podem ser visualizados em diferentes níveis conceituais. Tais níveis de visões conceituais organizam os dados de fontes de dados distintas de acordo com características comuns. Esse tipo de apresentação foi chamado pelos autores de *data space*, que é utilizado para integrar dados estruturados, semi-estruturados e não estruturados. Apesar da linguagem OQL apresentar mecanismos para consultar conjuntos de dados, esses mecanismos não podem ser utilizados para consultar dados não estruturados e semi-estruturados. Por esse motivo, os autores apresentaram uma extensão à linguagem OQL, denominada **SOQL** (SINGAPORE OQL) [Dom00].

A abordagem proposta pelos autores define duas etapas distintas para integração de fontes de dados. Na etapa de pré-integração, os administradores identificam similaridades e conflitos entre as fontes de dados e, em seguida, definem regras de correspondência entre esquemas (contexto da consulta). Na etapa de integração, o esquema integrado é automaticamente construído com base nas regras previamente estabelecidas. Embora a abordagem proposta pelos autores ofereça um certo grau de flexibilidade com a utilização das regras de correspondência, esse processo tem que ser executado antes da proposição de consultas. Além disso, é necessário redefinir o

esquema integrado cada vez que uma regra é modificada. Caso a quantidade de regras de correspondência seja muito grande pode degradar a performance do sistema.

A proposta apresentada por Sattler et al. assemelha-se à estratégia adotada por [Mis95]. A linguagem **FRAQL** (*Federation Query Language*) possui instruções declarativas que possibilitam mapear os objetos de um esquema local em objetos de um esquema global [Sat00]. Inicialmente, o esquema global, que estará acessível às consultas dos usuários, é definido. Em seguida, os esquemas locais são mapeados segundo o esquema global, onde é possível utilizar funções de conversão criadas pelos próprios usuários. As correspondências criadas pelo usuário ou pelo administrador formam o contexto de execução de consultas. As consultas são submetidas considerando o esquema global e, em seguida, são transformadas em sub-consultas utilizando os mesmos critérios estabelecidos no mapeamento.

Diferentemente da proposta adotada por [Dom00], essa abordagem não necessita que o esquema global seja refeito cada vez que uma nova fonte de dados entre na comunidade, sendo necessário, apenas, que essa nova fonte de dados estabeleça sua correspondência com o esquema já definido. Embora essa proposta seja mais flexível, o fato de a etapa de mapeamento estar dissociada da consulta propriamente dita, torna a especificação de consultas menos dinâmica que a abordagem utilizada nessa dissertação. Além disso, a abordagem proposta em [Sat00] está direcionada ao modelo de dados objeto-relacional e, embora seja mais flexível que o modelo relacional para representar dados semi-estruturados, ainda necessita de algum esforço para representar fontes de dados semi-estruturadas e não estruturadas.

A linguagem XQuery [ChF00, Rob00, Boa03] possui várias características que permitem consultar documentos XML e apresentar como resultado dessa consulta novos documentos XML. Os documentos XML resultados podem ser sub-árvores do documento XML original ou podem apresentar estrutura diferente, inclusive com a utilização de novos elementos obtidos através de construtores de elementos (ver seção 3.4).

Essas características tornam a linguagem bastante expressiva e poderosa, sendo possível, inclusive, consultar múltiplas fontes de dados XML. Embora seja possível exprimir operações de união e junção entre essas fontes de dados, a linguagem não apresenta mecanismos para resolução de conflitos. Dessa forma, as consultas XQuery assumem que as fontes de dados, embora distintas, foram concebidas dentro de um único projeto.

A XQuery permite a construção de elementos, o que torna possível contornar a falta de alguns mecanismos para resolução de conflitos. Por exemplo, é possível resolver o problema de sinônimos modificando o nome do elemento de uma das fontes de dados para que o mesmo coincida com o nome do elemento da outra fonte de dados. Embora os construtores de elementos forneçam alguns mecanismos para contornar o problema da heterogeneidade semântica (ver seção 2.2.3.3), não existe um mapeamento formal entre atributos de fontes de dados distintas e, portanto, tais construtores não expressam de forma clara o relacionamento conceitual existente entre as fontes de dados que estão sendo integradas.

Conseqüentemente, as consultas XQuery tendem a ser mais complexas que o necessário. Além disso, a linguagem XQuery não considera problemas de conexão e desconexão de fontes de dados e essa característica é fundamental em ambientes dinâmicos. A linguagem XQuery será apresentada com mais detalhes no capítulo 3.

Linguagem Proposta	Modelo de Dados	Esquema Global	Pré-Integração	Resolução de Conflitos	Funções de Usuário / Reconciliação
MSQL	Relacional (estruturado)	Não	Não	N – D – A	Sim / Não
IDL	Relacional (estruturado)	Não	Não	N – E – A	Não / Não
MSQL Extensão	Relacional (estruturado)	Sim	Não	N – D – S – A	Sim / Não
YATL*	XML (semi-estruturado)	Sim	Sim	N – D – A	Sim / Não
SOQL	Orientado a Objeto (semi-estruturado)	Sim	Sim	N – D – E – A	Sim / Não
FRAQL	Objeto-Relacional (estruturado)	Sim	Não	N – D – S – E – A	Sim / Sim
XQuery	XML (semi-estruturado)	N/A	N/A	N/A	Sim / Não

Legenda: **N** – Nome **D** – Domínio **S** – Semântico **E** – Esquema **A** – Dados
N/A – Não se aplica

(*) – A classificação apresentada refere-se a 2ª versão da linguagem YATL.

Tabela 2.1. Linguagens de Consulta como Mecanismo de Integração.

A tabela 2.1 apresenta um comparativo entre algumas linguagens de consulta para banco de dados que permitem acesso a múltiplas fontes de dados. A coluna modelo de dados apresenta o modelo de dados utilizado por cada linguagem e uma especificação do tipo de dados para o qual essa linguagem está orientada. Embora a linguagem SOQL utilize o modelo de dados orientado a objetos, a definição dos níveis de visões

conceituais (*data space*) permite que essa linguagem trabalhe com dados estruturados, não estruturados e semi-estruturados.

A coluna esquema global refere-se à necessidade, que algumas linguagens têm, de definir um esquema para o resultado das consultas, para que, em seguida, esse esquema tenha seus atributos mapeados para os atributos das fontes de dados locais. Embora todas as linguagens listadas, com exceção da linguagem XQuery, utilizem um processo de mapeamento, apenas aquelas que definem um esquema global realizam o mapeamento dissociado da consulta.

A coluna pré-integração refere-se à necessidade de executar algum processo de submissão de regras de integração antes de submeter consultas globais. A linguagem YATL identifica e resolve conflitos de integração através de regras de conversão, porém é necessário construir um esquema global com base nas regras de conversão antes de propor consultas. No caso da linguagem SOQL, é necessário definir as regras de conversão antes de utilizar a linguagem; tais regras não fazem parte da sintaxe da linguagem e são definidas e submetidas ao sistema SINGAPORE antes da proposição de qualquer consulta.

A coluna funções de usuário / reconciliação refere-se à possibilidade do usuário definir funções que possam ser utilizadas em consultas, possivelmente para resolver conflitos e utilizar essas funções para executar a reconciliação de dados, ou seja, utilizar funções para resolver um conflito de dados. Por exemplo, a linguagem FRAQL permite que uma função definida pelo usuário seja utilizada caso ocorra um conflito de dados, enquanto as demais linguagens resolvem esse tipo de conflito preservando todos os valores. Portanto, nesse aspecto, a abordagem utilizada pela linguagem FRAQL é mais flexível.

Como a linguagem XQuery não está orientada à integração de dados, as colunas esquema global, pré-integração e resolução de conflitos não se aplicam ao seu objetivo. Contudo, como destacado anteriormente, alguns tipos de conflitos podem ser resolvidos por essa linguagem através da construção de elementos.

CAPÍTULO 3

DADOS XML: DEFINIÇÃO E ACESSO

3.1 Introdução

A natureza das fontes de dados disponibilizadas na Web ou em redes *ad hoc* difere daquelas representadas por bancos de dados convencionais. Por esse motivo, o processo de integração baseado nesses ambientes deve considerar os seguintes aspectos [Pap95]:

- Parte da informação disponibilizada não possui uma estrutura (não-estruturada) ou a estrutura apresentada é irregular (semi-estruturada);
- O ambiente é dinâmico (ver capítulo 2), onde o número de fontes de dados, seus conteúdos e significados podem mudar freqüentemente;
- O processo de acessar uma informação está entrelaçado ao processo de integração de informação. Como não existe uma estrutura regular e a estrutura apresentada pode mudar, faz pouco sentido dissociar a etapa de integração, na qual os esquemas são combinados da etapa de acesso, onde os dados são consultados;
- A integração nesses ambientes requer maior participação humana, pois nem sempre é possível automatizar o processo de resolução de conflitos, mesmo utilizando dicionários semânticos.

Essas características apontam a necessidade de utilizar um modelo de dados que propicie maior flexibilidade à representação de tais fontes de dados. Conseqüentemente, deve-se considerar a possibilidade de representar essas informações como um conjunto de dados semi-estruturado e de adotar uma linguagem de consultas que explore eficientemente esse tipo de representação.

Dados semi-estruturados podem ser definidos como uma coleção de dados que não possui uma estrutura rígida ou regular e, muitas vezes, essa estrutura está implícita nos próprios dados. Freqüentemente a estrutura dos dados semi-estruturados deve ser

extraída dos próprios dados. Abiteboul apontou, como principais aspectos de uma coleção de dados semi-estruturados, as seguintes características [Abi97a]:

- Estrutura irregular. A coleção de elementos mantidos, geralmente, é heterogênea, com alguns elementos estão, ou ainda, possuem informações adicionais. Tal característica pode ser simulada em um banco de dados convencional, através da utilização de campos/propriedades não obrigatórios (por exemplo, campos que admitem valor "null" em bancos de dados relacionais), porém nem sempre é possível determinar quais campos/propriedades são obrigatórios;
- Estrutura implícita. Muitas vezes, embora exista uma estrutura precisa, tal estrutura está implícita nos dados. Em um documento XML, os marcadores especificam essa estrutura, mas é necessário algum esforço computacional para extrair essa informação;
- Estrutura parcial. Enquanto parte dos dados possui uma estrutura, outra parte desses dados pode ser vista como texto não estruturado. A análise de documentos de texto (ex., HTML) pode extrair uma estrutura básica, de acordo com padrões previamente definidos, no entanto, parte do documento permanece sem definição de uma estrutura formal;
- Pouca restrição estrutural. Um banco de dados convencional possui um esquema com tipos de dados rígidos, que forçam um preenchimento em conformidade com a estrutura. Por outro lado, os dados semi-estruturados possuem um esquema menos rígido e, por vezes, tal esquema é alterado para suportar os dados em si;
- Definição do esquema *a-posteriori*. Os sistemas de bancos de dados convencionais trabalham com a hipótese de esquemas fixos (pouco mutáveis), que devem ser definidos antes de serem introduzidos os dados. Enquanto as aplicações que utilizam dados semi-estruturados, quando definem o esquema, normalmente, o fazem *a-posteriori*, ou seja, depois de serem introduzidos os dados. Muitas vezes, isso só acontece porque surge algum tipo de pressão para padronização de informações correlatas;
- Esquema muito grande. As aplicações de dados semi-estruturados devem acomodar uma grande heterogeneidade de dados, por isso, geralmente, os esquemas são muito grandes em comparação com os dados. Em

contrapartida, em um banco de dados convencional espera-se que os esquemas sejam bem menores que os dados;

- Esquema ignorado. Freqüentemente, algumas pesquisas sobre dados semi-estruturados ignoram a existência de um esquema definido. Por exemplo, uma pesquisa pode definir a busca de um texto sobre dados semi-estruturados sem, contudo, especificar onde tal conjunto de dados deve ocorrer;
- Evolução constante de esquema. O esquema extraído de dados semi-estruturados evolui rapidamente para refletir novas estruturas de dados a serem representadas. Por vezes, a natureza das aplicações que utilizam dados semi-estruturados muda constantemente, obrigando uma adaptação do esquema a essa nova realidade;
- Mudança de tipos de dados. Muitas vezes, o esquema dos dados semi-estruturados é extraído do próprio conjunto de dados. Por esse motivo, é comum técnicas de extrações distintas mudarem a classificação de uma determinada informação, ou ainda, é possível a obtenção de novas informações a respeito de um determinado dado que determinem uma reclassificação de seu tipo;
- Distinção confusa entre esquema e dados. As características dos dados semi-estruturados dificultam a distinção entre dados e esquema. Por exemplo, um esquema muda tão freqüentemente quanto os dados; o tamanho do esquema e dos dados é semelhante; consultas submetidas a uma fonte de dados, usualmente, investigam o esquema e os dados indistintamente. Tais características aproximam dados e metadados, onde, na realidade, tudo é informação.

Além das características descritas acima, outros aspectos devem ser considerados para garantir o sucesso do processo de integração. Por isso, deve-se considerar o modelo de dados adotado, os recursos da linguagem de consultas utilizada e a extração e manipulação das estruturas (esquema implícito) presentes nas informações representadas [Abi97a].

As linguagens de marcação XML e XML *Schema* são apresentadas na seção 3.2. Algumas linguagens de consulta para documentos XML são descritas na seção 3.3. Finalmente, a seção 3.4 aborda o processamento de consultas expressas em XQuery.

3.2 XML

A linguagem XML nasceu como uma metalinguagem de marcação, com a qual é possível definir diferentes linguagens de marcação [Hol01]. Essa característica torna possível utilizar essa linguagem para representar e descrever dados. Os marcadores são utilizados para descrever o conteúdo do documento XML, por esse motivo não podem ser pré-definidos.

Inicialmente, a linguagem XML foi utilizada para construir linguagens de marcação personalizadas (ou vocabulários) que propiciavam o intercâmbio de dados. Dessa forma, foram definidos vocabulários XML para diversas áreas, como: **CML** (*Chemical Markup Language*), **XFDL** (*Extensible Forms Description Language*) e **cXML** (*CommerceXML*). Esses vocabulários, bem como os programas que os interpretam (*browsers* ou *parsers*), são compartilhados, oferecendo bases de pesquisa mais fáceis de serem compartilhadas, visualizadas e manipuladas.

Um documento XML possui como conteúdo dados e uma descrição desses dados. Fundamentalmente, um documento XML é composto por elementos e valores (ou conteúdo) associados a esses elementos. Um elemento XML é identificado através da utilização de marcadores (*tags*).

A figura 3.1 apresenta um exemplo de um documento XML, onde se pode identificar o marcador *biblioteca*. O marcador *biblioteca* possui um delimitador de início identificado por *<biblioteca>* e um delimitador de fim identificado por *</biblioteca>*. Nesse exemplo, o marcador *biblioteca* é o elemento raiz do documento apresentado.

Os elementos XML são constituídos por valores, atributos e sub-elementos. Um sub-elemento é um elemento XML que possui seus delimitadores de início e fim entre os delimitadores de início e fim de um outro elemento mais externo. Caso todos os elementos de um documento XML, com exceção do elemento raiz, possam ser descritos como sub-elemento de um elemento mais externo, esse documento XML possui seus elementos perfeitamente aninhados.

Um elemento pode ser vazio e, nesse caso, pode-se representar o elemento com apenas um delimitador, que deve trazer uma barra antes do sinal de “>”. Na figura 3.1, o elemento *<edicao tipo=“normal”/>* é um elemento vazio, contudo, pode-se identificar nesse elemento um atributo *tipo*.

Um documento XML que possua um único elemento raiz e que tenha seus elementos perfeitamente aninhados é denominado documento XML bem formado (a figura 3.1 é um exemplo de documento XML bem formado). Entretanto, por vezes, é necessário definir regras adicionais para utilização de elementos. Essa definição de regras é feita através de documentos de validação de documentos XML. Um documento XML que obedece as regras estipuladas em um documento de validação é denominado documento XML válido [And00, Hol01].

```
<?xml version="1.0" encoding="UTF-8"?>
<biblioteca>
   <livro isbn="393">
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <autor>Özsu</autor>
      <autor>Valduriez</autor>
      <ano>1999</ano>
      <edicao tipo="normal"/>
      <preco moeda="R$">89.00</preco>
   </livro>
   <livro isbn= "352">
      <titulo>Implementação de Sistemas de BDs</titulo>
      <autor>Garcia-Mollina</autor>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <ano>2001</ano>
      <edicao tipo="capa dura">Campus</edicao>
      <preco moeda="R$">110.00</preco>
   </livro>
</biblioteca>
```

Figura 3.1. Exemplo de um Documento XML bem formado.

O **DTD** (*Document Type Definition*) foi o primeiro padrão a especificar regras de validação para documentos XML [Gol99, And00, Hol01]. Esse tipo especial de documento utiliza uma gramática própria, através da qual são definidos quais elementos são permitidos nos documentos XML. Além disso, especifica a obrigatoriedade ou não de elementos, a ordem em que esses elementos devem aparecer dentro do documento XML e algumas outras características. Um documento XML, para ser considerado válido, deve ser associado a algum DTD (ou XML *Schema*, definido mais adiante) e, obviamente, obedecer às regras de validações contidas no DTD especificado.

O XML *Schema* é um outro padrão de especificação de regras de validação para documentos XML [Fal01]. Um XML *Schema* possui, basicamente, duas vantagens sobre o DTD:

1. Uma especificação escrita em XML *Schema* é um documento XML bem formado, portanto os programas utilizados para interpretar ou analisar um

documento XML podem ser utilizados para verificar uma especificação escrita em XML *Schema*;

2. Uma especificação escrita em XML *Schema* é mais poderosa, pois permite definir regras mais específicas e até tipos de dados mais complexos.

Por esses motivos, o XML *Schema* tem sido utilizado, cada vez mais, para validar documentos XML.

Embora o DTD e o XML *Schema* imponham restrições aos documentos XML, eles não podem ser vistos como um esquema de um banco de dados convencional. Esses documentos de validação não são obrigatórios e podem possuir uma estrutura apenas parcial, ou seja, parte dos documentos XML não é validada pelos DTD's ou XML *Schemas*. Além disso, os documentos de validação, freqüentemente, são definidos a partir dos próprios documentos XML, isto é, após a criação dos documentos XML.

3.2.1 XML *Schema*

A definição de esquemas XML através de DTD's era considerada muito complexa, por isso, o W3C formou um grupo para o desenvolvimento de uma nova linguagem de definição de esquemas. Entretanto, a especificação apresentada pelo W3C tornou-se mais complexa que os DTD's [Hol01]. Apesar disso, um XML *Schema* permite especificações mais completas e precisas a respeito de documentos XML.

A figura 3.2 apresenta uma descrição do esquema do documento XML apresentado na figura 3.1 utilizando XML *Schema*. Como se pode observar, o próprio XML *Schema* é um documento XML bem formado, cujo elemento raiz define um *namespace* padrão "xmlns" que aponta para uma **URI** (*Uniform Resource Identifier*) que possui as definições sobre XML *Schema*. A utilização de *namespaces* garante a definição de documentos XML livres de conflitos de nomes de *tags*. Por esse motivo, a definição de um elemento através de XML *Schema* geralmente é precedida por um *namespace*. Por exemplo, o elemento raiz "schema" está definido para o *namespace* "xsd".

Após a definição do elemento raiz, o XML *Schema* começa a descrever o esquema dos documentos XML que serão validados por esse XML *Schema*. Desse modo, é declarado um elemento do tipo complexo e que contém um atributo "name" que especifica qual nome do elemento raiz deve ser utilizado pelo documento XML.

Cada elemento definido pela linguagem XML *Schema* pode possuir atributos que especificam o número mínimo e máximo de ocorrências para aquele elemento dentro do documento XML. A omissão da ocorrência mínima implica em um elemento de ocorrência mínima igual a 1 (um). A omissão da ocorrência máxima indica que a ocorrência máxima é igual a ocorrência mínima, logo, se os dois atributos forem omitidos, o elemento terá ocorrência mínima e máxima igual a 1 (um). Além disso, os elementos definidos em XML *Schema* podem possuir um atributo indicando o tipo de dado esperado.

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
  <xsd:element name=“biblioteca”>
    <xsd:complexType>
      <xsd:element name=“livro” minOccurs=“0” maxOccurs=“unbounded”>
        <xsd:complexType>
          <xsd:attribute name=“isbn” type=“xsd:ID” use=“required”/>
          <xsd:element name=“titulo” type=“xsd:string”/>
          <xsd:element name=“ano” type=“xsd:string”/>
          <xsd:element name=“autor” type=“xsd:string”
                maxOccurs=“unbounded”/>
          <xsd:element name=“edicao” minOccurs=“0” maxOccurs=“1”>
            <xsd:complexType content=“mixed”>
              <xsd:attribute name=“tipo” type=“xsd:string” use=“required”/>
            </xsd:complexType>
          </xsd:element>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figura 3.2. Exemplo de uma Definição de Esquemas através de XML *Schema*.

A definição dos atributos é bastante similar, porém como um atributo não pode se repetir dentro de um elemento, apenas deve ser indicado se o atributo é obrigatório ou não. Além disso, a especificação de atributos aceita alguns tipos especiais como: xsd:ID, xsd:IDREF e xsd:IDREFS. O tipo xsd:ID especifica um atributo que identifica unicamente o elemento ao qual ele pertence, os atributos xsd:IDREF e xsd:IDREFS são atributos que referenciam outros elementos através de seus atributos xsd:ID.

Existem outras características de documentos XML que podem ser especificadas através de XML *Schema*, porém a informação apresentada até aqui é suficiente para a compreensão do restante do texto e em especial do capítulo 4. Maiores referências sobre a definição de esquemas através de XML *Schema* podem ser encontradas em [Fal01, Hol01].

3.3 Linguagens de Consulta para XML

As linguagens para dados semi-estruturados devem ser mais flexíveis que linguagens para bancos de dados convencionais [Pap95, Bun96, Abi97b]. Segundo Abiteboul, elas devem possuir as seguintes características [Abi97a]:

1. Primitivas de consulta similares a um banco de dados convencional;
2. Navegação ao estilo de hipertexto;
3. Busca por modelos (*pattern*);
4. Consulta temporal, incluindo consulta a versões e modificações e
5. Acesso aos dados e ao esquema na mesma consulta.

Embora existam programas (*parsers* e *browsers*) que possibilitem a visualização de documentos XML, geralmente, esses programas não oferecem recursos suficientes para buscar uma determinada informação de forma eficiente, como, por exemplo, através de um filtro baseado em uma condição.

Baseando-se nas características enumeradas acima, é necessária a implementação de recursos adicionais, tais como:

- Filtros. Expressões que extraem partes de um documento de acordo com uma condição ou que especificam um padrão que deve ocorrer no documento;
- Expressões de caminho. Estruturas que permitem percorrer em profundidade uma árvore de nós, especificando uma hierarquia entre os nós do grafo que representam o documento;
- Construtores de elementos. Um recurso que possibilita a transformação de um documento XML em outro documento XML, através da criação de novos elementos;
- Expressões de iteração. Essas expressões atribuem objetos a variáveis e, em seguida, utilizam essas variáveis para construir novos elementos. Essas expressões exploram a regularidade de uma estrutura representada em XML, para promover iterações nos elementos de um documento XML.

O principal objetivo de uma linguagem de consulta é extrair e apresentar uma determinada informação utilizando parâmetros fornecidos pelo usuário. Desse modo, cada um dos recursos adicionais descritos acima cumpre, pelo menos parcialmente, essa tarefa.

Esses recursos adicionais são implementados através das linguagens de consulta para documentos XML. As diversas linguagens apresentadas possuem objetivos e características diferentes e, por esse motivo, nem todas implementam todos os recursos adicionais descritos acima.

Inúmeras linguagens têm sido propostas para consultar documentos XML [Deu98, Rob98, Cla99, Gol99, Clu00, Pan02, Boa03, Erw03]. Esse trabalho destacou 3 (três) propostas que são particularmente interessantes, por terem identificado e resolvido alguns problemas no processo de consultar documentos XML. Como, por exemplo, preservação da hierarquia dos documentos XML e a construção de novos elementos XML.

3.3.1 XML–QL

A linguagem XML-QL foi apresentada para apreciação do W3C em 1998 [Deu98]. Sua principal característica é transformar elementos XML, porém, diferentemente da linguagem **XSL** (*eXtensible Stylesheet Language*), seu objetivo principal não é estilo e *layout* de apresentação. A linguagem XML-QL utiliza a transformação de elementos XML para a construção de novos elementos que representarão a resposta à consulta XML-QL proposta.

A linguagem possui elementos que propiciam a extração de dados, transformação e integração de diferentes fontes de dados. XML-QL utiliza uma estrutura modelo (*element patterns*) para comparar com o documento XML consultado e, em seguida, extrai as informações que combinam com o modelo utilizado.

```
WHERE <book>
            <publisher><name>Addison-Wesley</name></publisher>
            <title> $t</title>
            <author> $a</author>
        </book> IN "www.a.b.c/data.xml",
CONSTRUCT $a
```

Figura 3.3. Exemplo de uma Consulta Expressa em XML-QL.

O exemplo apresentado na figura 3.3 (extraído de [Deu98]) procura no documento identificado pela URI *"www.a.b.c/data.xml"* uma estrutura que se assemelhe à informada, onde um elemento *book* possua três sub-elementos. O sub-elemento *publisher* deve possuir um sub-elemento *name*, cujo conteúdo deve ser *Addison-Wesley*.

Os sub-elementos *title* e *author* devem ocorrer pelo menos uma vez e seus conteúdos devem ser atribuídos às variáveis *$t* e *$a*, respectivamente.

Finalmente, o resultado é construído através da expressão *"CONSTRUCT $a"*, que retorna todos os autores que satisfizeram o modelo de comparação informado na consulta. A consulta XML-QL pode utilizar uma notação simplificada, onde os delimitadores de fim dos elementos XML são representados apenas através do símbolo "</>". Dessa forma, as expressões *"<title> $t</title>"* e *"<title> $t</>"* são equivalentes e, portanto, produzem o mesmo resultado.

A linguagem XML-QL permite a realização de operações de junção e união e possui funções de agregação. Além disso, é possível consultar marcadores através da utilização de variáveis de marcadores que permitem a realização de consultas sem conhecer a estrutura exata de um documento XML. Esse recurso é importante devido à natureza, geralmente, irregular de documentos XML, além de permitir que usuários com pouco conhecimento da estrutura de documento consigam extrair informações desse documento.

As expressões de caminho podem ser utilizadas em consultas XML-QL através da seguinte sintaxe: *"<publisher><name>Addison-Wesley</name></publisher>"* (veja figura 3.3). Nesse exemplo, apenas o sub-elemento *"name"* pertencente ao elemento *"publisher"* importa; por isso, apenas essa informação é representada, mesmo que o elemento *"publisher"* possua conteúdo ou outros sub-elementos. As expressões de caminho em XML-QL permitem a utilização de curingas (*****), símbolos de alternância – ou lógico (**|**) e concatenação (**.**). Esses símbolos conferem maior flexibilidade às expressões de caminho e podem ser utilizados onde for permitida a utilização de elementos.

Finalmente, a linguagem XML-QL permite consultas sobre diferentes fontes de dados, onde é possível especificar condições de junção e montar uma visão integrada. A figura 3.4 apresenta uma consulta em XML-QL sobre dois documentos XML distintos [Deu98], onde o resultado apresentado é obtido através da junção dos dois documentos especificados na consulta. Embora a condição de junção não esteja especificada explicitamente, isso é feito utilizando o mesmo nome de variável para o elemento que deve servir como condição de junção. A variável *"$ssn"* é a condição de junção do exemplo apresentado na figura 3.4 onde, nesse caso, apenas elementos que possuírem o mesmo conteúdo para "ssn" farão parte do resultado (*equijoin*). Além do tipo de junção

"equijoin", é possível construir consultas em XML-QL onde se obtenha a junção do tipo *"outer join"*.

```
WHERE <person>
            <name></> ELEMENT_AS $n
            <ssn> $ssn</>
        </> IN "www.a.b.c/data.xml",
        <taxpayer>
            <ssn> $ssn</>
            <income></> ELEMENT_AS $i
        </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

Figura 3.4. Consulta Expressa em XML-QL sobre Múltiplas Fontes de Dados.

A linguagem XML-QL apresenta todos os recursos citados no início da seção 3.3. Embora permita especificar consultas sobre fontes de dados distintas e apresentar uma visão integrada dessas mesmas fontes, o processo de especificação de consultas de integração é muito complexo. Além disso, não são discutidos conflitos de integração que, geralmente, decorrem da integração de fontes de dados heterogêneas.

A atribuição de variáveis na cláusula *WHERE* e a transformação de elementos na cláusula *CONSTRUCT* são recursos poderosos, porém os modelos de elementos utilizados na linguagem tendem a ser, desnecessariamente, prolixos [Rob00]. Além disso, não é possível preservar toda a informação sobre a hierarquia e relações entre os elementos do documento original, porque o resultado das expressões XML-QL retorna valores escalares [Rob00]. Por esse motivo, mesmo que a saída deva ser um documento com a mesma estrutura do documento original, a cláusula *"CONSTRUCT"* deverá especificar toda a hierarquia novamente.

3.3.2 XPath

A linguagem XPath possui uma sintaxe poderosa para navegar em uma árvore de nós [Cla99]. Essa sintaxe permite que a hierarquia de nós contidas em um documento XML seja explorada de modo mais natural que a linguagem XML-QL. Além disso, os resultados de consultas expressas através do XPath preservam a hierarquia do documento XML original. A linguagem XPath é uma recomendação do W3C e tem sido utilizada em várias aplicações relacionadas ao XML, tais como XPointer e XSLT [Cla99, ChF00, Rob00].

Basicamente, uma consulta em XPath é expressa através da especificação de elementos e sub-elementos dispostos hierarquicamente e separados por uma barra (/). Assim, a expressão *"/biblioteca/livro/titulo"* pode ser utilizada para localizar um elemento *titulo* em um documento XML, onde o referido elemento deve possuir como ancestrais os elementos *livro* e *biblioteca*.

A linguagem XPath permite a especificação de filtros associados ao caminho de localização. Esses filtros selecionam uma determinada sub-árvore do documento original, ou o conteúdo de um elemento ou atributo. A figura 3.6 apresenta uma expressão de caminho em XPath que ao ser aplicada ao documento exemplo apresentado na figura 3.1 produz uma sub-árvore. A sub-árvore resultado também está representada na forma de um documento XML na figura 3.5.

```
/biblioteca/livro[position()=1]

-----------------------------------------------------------------

   <livro isbn="393">
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <autor>Özsu</autor>
      <autor>Valduriez</autor>
      <ano>1999</ano>
      <edicao tipo="normal"/>
      <preco moeda="R$">89.00</preco>
   </livro>
```

Figura 3.5. Expressão de Caminho em XPath, com Possível Resultado.

Na expressão exemplo é utilizado um caminho de localização para determinar que o elemento raiz *biblioteca* deve possuir um sub-elemento *livro*. Em seguida, é aplicado um filtro que utiliza a função *"position"* para selecionar o primeiro sub-elemento *livro*. O resultado dessa expressão é uma sub-árvore contendo o primeiro sub-elemento *livro*, bem como todos os sub-elementos e atributos pertencentes a esse sub-elemento *livro*.

Embora as expressões de caminho definidas através da sintaxe da XPath sejam muito poderosas para percorrer uma árvore de nós em profundidade e, diferentemente da linguagem XML-QL, preservem informações hierárquicas do documento original, não existe nessa linguagem uma forma de construir novos elementos. Os resultados de consultas expressas através de XPath são sempre sub-árvores da árvore original. Além disso, a linguagem não possui nenhum tipo de mecanismo para promover a iteração entre elementos do documento consultado.

Finalmente, a utilização da linguagem XPath padrão não permite a consulta a múltiplas fontes de dados, portanto, não é possível utilizá-la para integração de fontes de dados. Atualmente, o W3C está trabalhando em uma nova versão da linguagem XPath. A versão 2.0 da linguagem XPath é uma extensão da versão 1.0 acrescida de alguns recursos extraídos da linguagem XQuery [Ber03].

3.3.3 XQuery

A linguagem XQuery está atualmente sendo analisada pelo W3C e deve tornar-se uma recomendação de linguagem de consultas para XML. Essa linguagem é baseada na linguagem Quilt, que adaptou em sua sintaxe expressões ou recursos presentes em outras linguagens [ChF00, Rob00].

O projeto da linguagem Quilt utilizou a sintaxe da linguagem XPath [Cla99] e XQL [Rob98] para percorrer hierarquicamente um documento. A atribuição de variáveis e construção de novas estruturas foi herdada da linguagem XML-QL [Deu98]. A linguagem SQL contribuiu com uma estrutura baseada em cláusulas (*SELEC-FROM-WHERE*), que originou as expressões **FLWOR** (*FOR-LET-WHERE-ORDER BY-RETURN*). Finalmente, a noção de linguagem funcional foi retirada da linguagem OQL, que pode ser composta por vários tipos expressões aninhadas ou não [ChF00, Rob00].

A abordagem utilizada para projetar a linguagem Quilt/XQuery definiu 3 (três) construções básicas para expressar consultas a documentos XML: expressões de caminho, construtores de elementos e expressões FLWOR [Boa03]. Essas três construções podem ser utilizadas isoladamente ou em conjunto, onde é permitido inclusive utilizar o resultado de uma construção como parâmetro para uma construção mais externa. A seguir, as construções básicas de uma consulta XQuery são descritas e analisadas.

3.3.3.1 Expressões de Caminho

As expressões de caminho presentes na sintaxe da linguagem XQuery possuem uma notação bastante similar à utilizada pela linguagem XPath. Geralmente, as consultas expressas através da XQuery utilizam as expressões de caminho para identificar o início de uma sub-árvore do documento para, em seguida, utilizando as outras expressões da linguagem, extrair a informação requerida.

Entretanto, como toda a flexibilidade das expressões de caminho da linguagem XPath foi preservada, é possível formular consultas e obter resultados apenas utilizando tais expressões. A expressão de caminho apresentada na figura 3.5 produziria o mesmo resultado se analisada por um interpretador de consultas XQuery.

A principal diferença entre as expressões de caminho da linguagem XQuery e a linguagem XPath é a possibilidade de se utilizar variáveis para representar o caminho ou parte do caminho. Tais variáveis devem ter sido associadas a um determinado caminho por outras estruturas da linguagem XQuery. Dessa forma, a expressão de caminho *"$a/livro/titulo"* tem sua avaliação dependente do valor atribuído à variável *$a* que pode significar um ponto fixo de entrada na floresta de nós do documento consultado ou pode possuir um valor variável dependente de uma iteração especificada por uma expressão FLWOR.

As expressões de caminho podem ser vistas como o nome qualificado de um elemento, pois nem sempre é possível distinguir um determinado elemento sem especificar sua hierarquia completa (caminho).

3.3.3.2 Construtores de Elementos

Além de procurar elementos em um documento, geralmente, uma consulta necessita gerar novos elementos [Boa03]. Os construtores de elementos permitem que o resultado de uma consulta seja uma árvore diferente da árvore fornecida inicialmente. Isso é possível através da criação de novos elementos e modificação da ordem dos elementos originalmente fornecidos.

Os construtores de elementos utilizam a mesma sintaxe dos documentos XML para gerar os novos elementos. Desse modo, para criar um novo elemento deve-se inserir um marcador para representá-lo e especificar o delimitador de início e fim, além do conteúdo, caso seja necessário. De modo similar às expressões de caminho, pode-se utilizar variáveis para representar elementos complexos ou valores escalares. Tais variáveis devem ter sido associadas a algum valor em outras construções da linguagem XQuery.

A utilização mais comum dos construtores de elementos é na produção do resultado de uma consulta. Por isso, é comum aparecer construtores de elementos após a cláusula *RETURN* das expressões FLWOR.

3.3.3.3 Expressões FLWOR

As expressões FLWOR são expressões compostas por uma série de cláusulas, que são: *FOR*, *LET*, *WHERE*, *ORDER BY* e *RETURN*. Tais cláusulas devem ser declaradas em uma ordem específica, podendo ou não, serem aninhadas. Uma expressão FLWOR associa valores a uma ou mais variáveis e então usa essas variáveis para construir o resultado [Boa03].

A cláusula *FOR* é utilizada para executar uma iteração sobre a variável associada. Assim, para cada passo executado na iteração da cláusula *FOR*, a variável é associada a um novo valor. Por exemplo, a variável declarada na cláusula *FOR* pode ser associada a uma expressão de caminho que retorne uma seqüência de nós. Durante o processamento da consulta, o valor de cada nó é atribuído individualmente a variável da cláusula *FOR*.

A cláusula *LET* também é utilizada para associar variáveis, entretanto, não executa nenhum tipo de iteração. Dessa forma, caso uma seqüência de nós seja atribuída a uma variável da cláusula *LET*, essa variável representa a seqüência de nós completa e de uma única vez.

A expressão *"FOR $x IN /library/book"* resulta tantas associações quantos forem os sub-elementos *book* pertencentes ao elemento *library*. Por outro lado, a expressão *"LET $x := /library/book"* resulta uma única associação contendo todos os sub-elementos *book* [Boa03].

O conteúdo das variáveis associadas pelas cláusulas *FOR* e *LET* pode ser filtrado através da utilização da cláusula opcional *WHERE*. A cláusula *WHERE* deve conter uma condição que, ao ser analisada, determina se aquela variável associada deve fazer parte do resultado da consulta.

A cláusula *ORDER BY* contém uma ou mais especificações de ordenamento, denominadas *orderspecs* [Boa03]. Para cada linha do documento resultado, as especificações de ordenamento são avaliadas e a construção do resultado obedece às suas especificações.

Finalmente, a cláusula *RETURN* é utilizada para determinar o resultado que será apresentado ao usuário. Normalmente, são utilizadas as variáveis que foram associadas pelas cláusulas *FOR* e *LET* e filtradas pela cláusula *WHERE*. Além disso, é comum utilizar construtores de elementos para apresentar os resultados através da cláusula *RETURN*.

A linguagem XQuery apresenta outras funcionalidades, como: operadores aritméticos e lógicos, funções e expressões condicionais. Além disso, é possível consultar múltiplas fontes de dados e utilizar operações de união e junção para produção de um resultado único. Entretanto, a linguagem não apresenta operadores que especifiquem relacionamentos conceituais entre fontes de dados distintas. Tais operadores são úteis para identificar conflitos de integração, por isso, utilizar a linguagem XQuery para integrar fontes de dados heterogêneas é uma tarefa bastante complicada.

A figura 3.6 apresenta um exemplo de consulta utilizando a linguagem XQuery, que foi extraído de [Boa03]. A consulta procura identificar os livros que possuem seu preço maior que o preço médio de todos os livros. O resultado deve apresentar o título do livro e a diferença entre o preço do livro e o preço médio calculado.

```
<result>
  {
    LET $a := avg(document("bib.xml")//book/price)
    FOR $b IN document("bib.xml")//book
    WHERE $b/price > $a
    RETURN
        <expensive_book>
            {$b/title}
            <price_difference>
                {$b/price – $a}
            </price_difference>
        </expensive_book>
  }
</result>
```

Figura 3.6. Exemplo de uma Consulta Expressa em XQuery.

O exemplo apresenta todas as construções abordadas anteriormente. A expressão *"LET $a := avg(document("bib.xml")//book/price)"* utiliza a função "document()" para especificar o nome do documento consultado e, em seguida, é identificada, através de uma expressão de caminho, a posição hierárquica do sub-elemento *"price"* (preço). A função "avg()" extrai o preço médio de todos os livros que são associados à variável *$a* pela cláusula *LET*.

A expressão *"FOR $b IN document("bib.xml")//book"* executa uma iteração em todos os elementos *book* pertencentes ao documento *"bib.xml"*. Cada ocorrência de um novo elemento *book* tem seu preço comparado ao preço médio através da cláusula *WHERE*. Caso o preço encontrado seja maior que o preço médio, um novo elemento

expensive_book é criado. Serão criados tantos elementos *expensive_book* quantos forem os livros que possuírem um preço maior que o preço médio.

A figura 3.7, extraída de [Boa03], apresenta uma cláusula *FOR*, onde são associadas 3 (três) variáveis de iteração *$i, $p* e *$s*. As variáveis *$p* e *$s* são obtidas através de uma junção, onde a condição de junção é especificada entre colchetes. Dessa forma, o documento *"parts.xml"* possui um elemento raiz *part* e um sub-elemento *partno*, onde o valor do sub-elemento *partno* deve ser igual ao valor do sub-elemento *partno* pertencente ao documento *"catalog.xml"*. Caso esse critério seja satisfeito ocorre a junção entre os elementos de *"catalog.xml"* e *"parts.xml"*. De maneira análoga se obtém a junção com o documento *"suppliers.xml"*.

```
<descriptive-catalog>
  {
    FOR $i IN document("catalog.xml")//item,
        $p IN document("parts.xml")//part[partno = $i/partno],
        $s IN document("suppliers.xml")//supplier[suppno = $i/suppno]
    RETURN
        <item>
            {
                $p/description, $s/suppname, $i/price
            }
        </item>
    SORTBY(description, suppname)
  }
</descriptive-catalog >
```

Figura 3.7. Consulta Expressa em XQuery sobre Múltiplas Fontes de Dados.

Esse exemplo especifica uma consulta sobre múltiplas fontes de dados, entretanto, tais fontes de dados não apresentam nenhum tipo de conflito de integração. O documento [Boa03] do W3C não aborda questões de conflitos de integração, conseqüentemente, nenhum exemplo de como resolvê-los utilizando a linguagem é apresentado.

3.4 Processamento de Consultas sobre Dados XML

Uma das etapas mais importantes no processamento de consultas é a simplificação/otimização de consultas. Essa importância deve-se ao fato de que consultas bem escritas, definição de índices adequados, planos lógicos corretamente selecionados e algoritmos de execução de consultas eficientes diminuem

significativamente o tempo de resposta de uma consulta. Obviamente, um bom processador de consultas deve implementar um eficiente processo de otimização.

Inúmeros trabalhos abordam a otimização de consultas baseadas em dados semi-estruturados. McHugh e Widom discutiram as etapas do processamento de consultas realizadas em seu projeto Lore, em especial, a otimização de consultas [McH99a, McH99b]. Lore apresenta várias estruturas de indexação que permitem encontrar valores atômicos e caminhos dentro de um grafo (hierarquia de objetos). Além disso, definiu um gerador de planos lógicos de consulta e de planos físicos de consulta. Finalmente, utilizou um modelo de custos para identificar o plano lógico e físico mais eficiente para uma determinada consulta.

Em [Coo01, Li01] foi abordada a definição de estruturas de indexação mais eficientes para consulta de dados representados em documentos XML. A abordagem utilizada nesses trabalhos foi mais genérica e não foi direcionada a nenhuma linguagem de consultas para documentos XML. Conseqüentemente, o processo de otimização e a elaboração de planos lógicos e físicos de consulta não foram discutidos.

A importância da elaboração do plano lógico e físico de consulta está relacionada ao fato de uma mesma consulta poder ser escrita de várias maneiras distintas. Conseqüentemente, deve-se identificar qual a maneira mais eficiente de se expressar uma determinada consulta. Por esse motivo, o processador de consultas, ao receber uma consulta, procura identificar outras consultas equivalentes à consulta originalmente proposta. Contudo, geralmente, esse processo é baseado em uma representação formal da linguagem de consultas.

A utilização de uma linguagem formal permite representar as consultas através de fatores (tabela, atributo) e operações (projeção, seleção, fusão). De forma análoga às operações matemáticas, cada operação da linguagem formal possui um conjunto de propriedades. Assim sendo, uma operação que possua a propriedade comutativa, significa que seu resultado independe da ordem de seus fatores. As transformações executadas sobre a consulta original consideram as propriedades das operações envolvidas nessa consulta.

O otimizador de consultas deve considerar uma série de variáveis para poder determinar o melhor plano lógico, entretanto, em um banco de dados local o fator determinante de performance de um banco de dados são as operações de leitura e escrita em disco. Por esse motivo, o otimizador de consultas considera como melhor plano lógico aquele que necessitar menos acesso a disco.

Por outro lado, em ambientes de bancos de dados distribuídos, o fator crítico de performance do sistema é o tráfego imposto à rede. Assim, o otimizador de consultas global tenta gerar sub-consultas nas quais os resultados gerem o menor tráfego na rede possível. Considerando o ambiente Web e de redes *ad hoc* tem-se um grau de distribuição muito grande, conseqüentemente, o otimizador de consultas global deve minimizar o tráfego imposto ao ambiente de comunicação.

Atualmente, o W3C tem trabalhado em um projeto para definir formalmente as linguagens XQuery e XPath [Dra03]. Essa formalização da linguagem XQuery vai permitir construir processadores de consulta para linguagem XQuery que possuam otimizadores de consulta mais eficientes. Esse trabalho especifica um modelo de processamento dividido em 5 (cinco) etapas, onde cada etapa consome o produto da etapa anterior e resulta o produto consumido pela etapa seguinte:

1. Análise (*parsing*). A consulta especificada é comparada à gramática da linguagem XQuery. Essa etapa pode gerar erros de sintaxe, caso contrário, uma árvore sintática é criada a partir da consulta analisada;
2. Processamento de Contexto. O processamento de uma consulta XQuery depende do contexto da consulta que é gerado antes do processamento da consulta. O contexto de uma consulta é constituído por uma série de declarações e definições que afetam o processamento de consultas. Como por exemplo, a definição de variáveis de ambiente ou de funções definidas pelo usuário;
3. Normalização. O processo de normalização de uma consulta consiste em reescrever a consulta utilizando apenas expressões que fazem parte do núcleo (*core*) da linguagem XQuery. Por exemplo, uma expressão *for* complexa pode ser reescrita como um grupo de várias expressões *for* simples;
4. Análise de Tipo Estático. As expressões e funções utilizadas pela XQuery são tipadas e a combinação de expressões e funções sempre resulta tipos compatíveis com os tipos originais. Essa etapa do processamento verifica se essa compatibilidade foi mantida. Isso não significa que o conteúdo do documento XML deve possuir um tipo fixo e rígido. A linguagem XQuery suporta tipos flexíveis (por exemplo, para representar o conteúdo de um elemento);

5. Avaliação Dinâmica. Essa etapa utiliza os documentos XML consultados para avaliar as expressões e variáveis, onde um erro de avaliação dinâmica pode ser gerado.

A especificação apresentada pelo W3C utiliza regras de mapeamento para reescrever as consulta XQuery e utiliza regras de inferência para avaliar o resultado de expressões compostas, tipos de expressões e erros.

A equação 3.1 apresenta um exemplo de como as expressões XQuery são reescritas (normalizadas) utilizando apenas expressões do núcleo da linguagem XQuery. A figura define uma equivalência entre uma expressão que utiliza a cláusula WHERE e uma expressão que utiliza uma cláusula IF. Assim, ao encontrar uma expressão com a cláusula WHERE, o processador de consultas XQuery altera a expressão, substituindo a cláusula WHERE original por uma cláusula IF. Maiores referências sobre a formalização de consultas XQuery podem ser encontradas em [Dra03].

$$\left[\text{where } Expr_1\right]_{FLWR}(Expr)$$
$$==$$
$$\text{if } (\ \left[Expr_1\right]_{Expr}\ ) \text{ then } Expr \text{ else } ()$$

Equação 3.1. Normalização da Cláusula “WHERE”.

CAPÍTULO 4

MXQUERY

4.1 Introdução

A abordagem de Sistemas de Banco de Dados Múltiplos para integração de bancos de dados heterogêneos requer a utilização de uma linguagem de consultas que seja capaz de expressar relacionamentos conceituais entre fontes de dados distintas. Isso significa que a linguagem de consultas não deve apenas ter a capacidade de acessar fontes de dados distintas em uma mesma consulta, mas deve, também, identificar equivalência entre objetos de fontes distintas, atributos correspondentes, valores ambíguos para atributos correspondentes, incompatibilidade de domínios entre outras características.

Portanto, uma linguagem de consultas para banco de dados múltiplos (MDL) deve identificar conflitos de integração originados da heterogeneidade semântica existente entre as fontes de dados a serem integradas e determinar uma forma de resolver tais conflitos. O resultado de uma consulta expressa através da MDL deve ser consolidado e homogêneo, não possibilitando identificar, a menos que seja necessária, a origem da informação contida nesse resultado.

O restante desse capítulo está organizado da seguinte forma. A Seção 4.2 discute os objetivos da definição da linguagem MXQuery. As expressões e funções introduzidas pela linguagem MXQuery são apresentadas nas seções 4.3 e 4.4. A Seção 4.5 aborda alguns aspectos sobre formalização e normalização de consultas MXQuery. Finalmente, a Seção 4.6 analisa algumas consultas definidas através da MXQuery.

4.2 Objetivos da Linguagem MXQuery

A linguagem MXQuery é uma extensão da linguagem XQuery, onde foram acrescentadas características que permitem manipular e integrar fontes de dados

heterogêneas, tendo XML como modelo de dados comum para representação de esquemas conceituais (ver Figura 2.5). A linguagem XQuery é, naturalmente, orientada a consultar dados semi-estruturados e, portanto, apresenta estruturas flexíveis para identificar as informações requeridas e arranjá-las de acordo com as especificações do usuário. A linguagem MXQuery aproveitou-se dessas características e implementou declarações de mapeamento entre fontes de dados distintas, para representar o relacionamento conceitual existente entre essas fontes de dados.

A idéia por trás da linguagem MXQuery é utilizar declarações de mapeamento entre os elementos originais das fontes de dados e um ou mais elementos de um documento XML que conterá o resultado da consulta. O processo de mapeamento está inserido na própria consulta, tornando, dessa forma, a MXQuery mais flexível que as propostas apresentadas em [Mis95] e [Sat00]. Além disso, é possível utilizar variáveis de resultado (associadas aos elementos do documento XML resultado) e variáveis das fontes de dados (associadas aos elementos das fontes de dados) em especificações de condições na cláusula "*WHERE*". Isso permite que sejam inseridos filtros (condições) sobre dados integrados e dados das fontes de dados locais.

Por esse motivo, as condições existentes na consulta são processadas em dois momentos distintos. As condições sobre dados das fontes de dados locais são inseridas nas sub-consultas encaminhadas às fontes de dados, limitando, portanto, o resultado da sub-consulta sobre essas fontes de dados. As condições sobre os dados integrados são processadas após a etapa de construção do resultado global, ou seja, elementos inicialmente inseridos no resultado podem ser excluídos ou modificados de acordo com as condições sobre os dados integrados.

Por exemplo, a Figura 4.6 apresenta a condição sobre dados integrados ***WHERE $p/livro/ano > "1999"***. Essa expressão só será avaliada após a integração dos dados, por isso, poderá ser gerado um tráfego desnecessário pela rede. Elementos "livro" que possuam sub-elementos "ano" menores que 1999 são enviados para o gerenciador de consultas e, só após a etapa de integração, esses elementos são descartados.

Obviamente, durante a etapa de simplificação, a arquitetura MDBS tenta transformar as condições baseadas em dados integrados em condições baseadas nas fontes de dados locais, pois essa simplificação reduz a quantidade de dados trafegada pela rede. Por exemplo, a condição acima deve ser reescrita pelo gerenciador de consultas, como ***WHERE $d1/livro/ano > "1999" AND $d3/book/year > "1999"***.

Diferentemente das propostas [Mis95], [Dom00] e [Sat00], a linguagem MXQuery não requer a definição de contextos de uma consulta. Por esse motivo, as consultas expressas através da linguagem MXQuery são mais dinâmicas e, conseqüentemente, sofrem menor interferência da evolução dos esquemas locais, conexão e desconexão de novas fontes de dados. Isso significa que, no momento em que uma consulta MXQuery é proposta, toda a informação sobre como integrar as fontes de dados consultadas é expressa na própria consulta. Dessa forma, a evolução dos esquemas locais não compromete as definições do ambiente de integração, apenas exige novas formas de expressar as consultas MXQuery.

Uma grande dificuldade em processar consultas, sobre múltiplas fontes de dados heterogêneas que merece destaque, é o tratamento da disponibilidade das fontes de dados a serem consultadas [Tom96, Abi01]. Para ilustrar esse problema, considere a seguinte consulta: “Retorne todos os livros de um determinado autor com as respectivas críticas, considerando que tais livros devem estar disponíveis em livrarias da cidade de Fortaleza no próximo final de semana”. Suponha que esta consulta deva acessar dados distribuídos em diversas fontes de dados. Por exemplo, a primeira fonte de dados conteria a informação de livros e autores, a segunda apresentaria as críticas e a terceira a localização das livrarias próximas de Fortaleza.

Para o cenário descrito acima, considere que a indisponibilidade da fonte de dados contendo as críticas dos livros pode ser considerada contornável, ou seja, essa informação não é essencial para o usuário. A resposta à consulta seria incompleta, porém seria satisfatória. Por outro lado, a possibilidade da fonte de dados com a localização das livrarias estar indisponível tornaria uma resposta incompleta inútil para o usuário.

A estratégia adotada pela linguagem MXQuery para resolver esse problema é especificar as diversas fontes de dados de uma consulta *C* e, em seguida, classificá-las de acordo com sua relevância para *C*. Se uma fonte de dados é imprescindível para a consulta *C*, então ela é classificada como obrigatória, caso contrário, ela é classificada como opcional. As fontes de dados opcionais, não disponíveis durante a execução de uma consulta, não interrompem o processamento dessa consulta. Com isso, fornece-se ao processador de consultas do MDBS a capacidade de alterar a consulta original para retirar qualquer referência a essa fonte de dados opcional. Em contrapartida, as fontes de dados classificadas como obrigatórias não podem estar indisponíveis no momento da consulta, caso isso ocorra, a consulta não é executada.

4.3 Expressões EFLWOR

A idéia básica da MXQuery é estender a estrutura FLWOR presente na linguagem XQuery. Na MXQuery poderão existir expressões do tipo **EFLWOR** (*Each* – FLWOR), além de expressões FLWOR. Uma consulta baseada em uma expressão EFLWOR percorre todos os elementos das árvores referentes aos documentos XML que representam as fontes de dados participantes da consulta. O resultado da consulta é uma árvore, que é denominada árvore resultado. A árvore resultado é construída através da união, junção e/ou fusão de elementos pertencentes a documentos (fontes de dados) distintos.

A forma menos restritiva de obtenção da árvore resultado de uma integração é através da **união** das fontes de dados especificadas na consulta. Como dito anteriormente, as fontes de dados representadas através do XML não possuem uma estrutura regular rígida e, dessa forma, a operação de união não requer nenhum tipo de compatibilidade entre as fontes de dados (como a requerida, por exemplo, no modelo de dados relacional). Além disso, não requer a especificação de uma condição como é o caso das operações de junção e fusão. No caso de uma operação de união, todos os elementos das fontes de dados referenciadas na consulta são incluídos na árvore resultado como filhos da raiz desta árvore. Embora possam ser especificadas restrições através da cláusula “WHERE”, tais restrições, não devem envolver a comparação entre objetos pertencentes a fontes de dados distintas. Na Seção 4.6.1, a consulta 3 apresenta um exemplo de operação de união.

A operação de **junção** pode ser especificada em uma consulta MXQuery de forma implícita ou explícita. A *junção implícita* ocorre quando o operador de dereferência (=>) é utilizado na especificação de alguma expressão de caminho na consulta (ver consulta 6). A *junção explícita* utiliza a cláusula “WHERE” para especificar condições de junção, definindo que elementos pertencentes a documentos (ou fontes de dados) distintos devem ser comparados (veja consulta 5, na Seção 4.6.1). É importante observar que o operador de dereferência faz parte da sintaxe das expressões de caminho definidas na linguagem XQuery e adaptadas da linguagem XPath, por isso não é possível utilizar esse operador em fontes de dados distintas. Conseqüentemente, apenas a junção explícita pode ser utilizada para realizar a junção entre elementos pertencentes a fontes de dados distintas.

A operação de **fusão** é semelhante a um tipo especial de junção da álgebra relacional, o *full outer join*. A árvore resultado obtida através da operação de fusão apresenta todos os elementos pertencentes a documentos (fontes de dados) distintos que satisfazem uma condição de junção mais os que não satisfazem esta condição. A consulta 2 (Seção 4.6.1) ilustra o funcionamento da operação de fusão.

4.3.1 Gramática das Expressões EFLWOR

Por se tratar de uma extensão da linguagem XQuery, parte da gramática da linguagem MXQuery já está descrita em [Boa03]. A tabela 4.1 apresentada abaixo descreve a sintaxe das expressões EFLWOR onde a notação **EBNF** (*Extended Backus-Naur Form*) foi utilizada. A primeira coluna da tabela na gramática XQuery original apresenta um número seqüencial para todos as cláusulas definidas pela gramática. A tabela 4.1 inseriu elementos numerados de 144 a 147. O elemento 42 deve substituir o elemento da gramática original e os outros elementos presentes na tabela 4.1 foram repetidos por uma questão de clareza.

[42]	EFLWORExpr	::=	EachClause (ForClause | LetClause) * WhereClause? OrderByClause? “return” ExprSingle
[144]	EachClause	::=	“each” “$” VarName “full”? (“,” “$” VarName “full”?)* DefClause
[145]	DefClause	::=	AliasClause (AliasClause)* (EqualClause)*
[146]	AliasClause	::=	“alias” PathExpr “$” VarName “null”? (“,” “$” VarName “null”?)*
[147]	EqualClause	::=	“equal” RelativePathExpr* “is” RelativePathExpr “key”? “hide”?
[43]	ForClause	::=	“for” “$” VarName TypeDeclaration? PositionalVar? “in” ExprSingle (“,” “$” VarName TypeDeclaration? PositionalVar? “in” ExprSingle)*
[45]	LetClause	::=	“let” “$” VarName TypeDeclaration? “:=” ExprSingle (“,” “$” VarName TypeDeclaration? “:=” ExprSingle)*
[122]	TypeDeclaration	::=	“as” SequenceType
[44]	PositionalVar	::=	“at” “$” VarName
[46]	WhereCaluse	::=	“where” Expr
[47]	OrderByCaluse	::=	(“order” “by” | “stable” “order” “by”) OrderSpecList
[48]	OrderSpecList	::=	OrderSpec (“,” OrderSpec)*
[49]	OrderSpec	::=	ExprSingle OrderModifier
[50]	OrderModifier	::=	(“ascending” | “descending”)? ((“empty” “greatest”) | (“empty” “least”))? (“collation” StringLiteral)?

Tabela 4.1. Sintaxe das Expressões EFLWOR.

O elemento FLWORExpr (42), presente na gramática original da linguagem XQuery, foi substituído pelo elemento EFLWORExpr. Dessa forma, qualquer referência ao elemento 42 na gramática original da linguagem XQuery deve ser relacionada ao elemento EFLWORExpr. Além disso, foram introduzidos novos elementos com a finalidade de completar a definição das expressões EFLWOR. Os elementos adicionados foram: EachClause (144), DefClause (145), AliasClause (146) e EqualClause (147). A definição dos elementos referenciados e não descritos na tabela 4.1 pode ser encontrada em [Boa03]. Por exemplo, o elemento VarName, referenciado pelo elemento EachClause, corresponde ao elemento n° 20 apresentado na gramática completa da linguagem XQuery [Boa03].

4.3.2 Definição das Expressões EFLWOR

A Figura 4.1 apresenta uma consulta MXQuery genérica, onde as cláusulas introduzidas pela extensão aparecem com letras maiúsculas. Na consulta, a variável "$p" é um documento XML bem-formado, que representa o resultado da consulta. As variáveis "$axml" e "$bxml" são documentos XML, que representam as fontes de dados consultadas. As variáveis "$a" e "$b" são documentos XML, que representam o ponto de entrada na floresta dos documentos "$axml" e "$bxml", respectivamente.

Após a etapa inicial de definição dessas variáveis, são introduzidas na consulta expressões de caminhos, que especificam um mapeamento entre elementos e sub-elementos de documentos distintos com o documento resultado. Finalmente, a variável "$p" pode ser utilizada para construir um resultado.

```
EACH $p FULL
  ALIAS document($axml)/raiz_a $a
  ALIAS document($bxml)/raiz_b $b NULL
  EQUAL $a/elemento_a $b/elemento_b IS $p/elemento_p
  EQUAL $a/elemento_a/../sub1_a $b/elemento_b/../sub1_b IS $p/elemento_p/../sub1_p KEY
  EQUAL $a/elemento_a/../sub2_a $b/elemento_b/../sub2_b IS $p/elemento_p/../sub2_p HIDE
return
  <nova_raiz>
    $p
  </nova_raiz>
```

Figura 4.1. Uma Consulta MXQuery Genérica.

As expressões EFLWOR introduzidas pala linguagem MXQuery originaram cláusulas não definidas para a linguagem XQuery. Tais cláusulas são as seguintes: "EACH", "ALIAS", "EQUAL", "FULL", "NULL", "HIDE" e "KEY". Essas cláusulas são utilizadas para identificar as fontes de dados consultadas, especificar mapeamentos entre atributos locais (originados das fontes de dados) e atributos globais (representantes

dos documentos resultado). As cláusulas introduzidas pela linguagem MXQuery serão descritas a seguir.

EACH

Essa cláusula associa valores a uma ou mais variáveis, denominadas variáveis de resultado. Essas variáveis são utilizadas para a construção de um resultado que poderá representar, na realidade, elementos XML gerados a partir da integração das fontes de dados participantes do mecanismo de integração. A declaração ***EACH $p*** define que uma variável "$p" conterá o resultado de um processo de integração de elementos pertencentes a fontes de dados distintas. Regras para o processo de integração (por exemplo, regras para a resolução de conflitos) deverão ser definidas nas demais declarações da linguagem MXQuery. A variável declarada na cláusula "EACH" é um documento XML bem formado e pode ser referenciada por qualquer declaração da estrutura EFLWOR que aceite como parâmetro uma árvore de nós.

ALIAS

Essa cláusula especifica as árvores (ou sub-árvores) XML que serão utilizadas na consulta onde cada árvore referenciada pode representar uma fonte de dados distinta. Portanto, essa cláusula é utilizada para definir o escopo da consulta. Um documento referenciado na declaração "ALIAS" é associado a uma ou mais variáveis, denominadas variáveis de fonte de dados. De forma análoga às variáveis definidas pela cláusula "EACH", essas variáveis podem ser utilizadas nas cláusulas que aceitem uma árvore de nós como parâmetro. Por exemplo, a declaração ***ALIAS document("d1.xml")/bib $d1*** define uma variável "$d1" que representa a fonte de dados especificada pelo documento "d1.xml"; a função "document" definida em [Boa03] identifica o documento XML que representa uma das fontes de dados participantes da consulta, além de indicar o ponto de entrada na floresta de nós do documento XML. Portanto, uma variável definida através da cláusula "ALIAS" pode representar qualquer sub-árvore da fonte de dados referenciada. A ordem de declaração de cláusulas "ALIAS" em uma consulta MXQuery especifica a ordem de prioridade entre as fontes de dados a serem acessadas. Dessa forma, a fonte de dados, especificada na primeira declaração "ALIAS", é a fonte de dados com maior ordem de prioridade e assim por diante. Em casos de conflito de dados, a ordem de prioridade será utilizada para resolver esse tipo de conflito segundo o seguinte critério: serão considerados válidos os valores de elementos pertencentes a

fontes de dados com maior ordem de prioridade. Na consulta 2 (Seção 4.6.1) é ilustrada a utilização da ordem de prioridade para a resolução de um conflito de dados.

EQUAL

Essa cláusula identifica a relação entre os elementos de documentos distintos e, em seguida, um novo elemento contendo o resultado da integração é gerado. A construção especifica regras de integração de elementos de documentos distintos onde grande parte dos conflitos é resolvida (ver Seção 4.6.2). Na especificação ***EQUAL $d1/livro $d3/book IS $p/livro***, um elemento (no caso, livro) é inserido na árvore resultado. A função desse novo elemento é integrar os elementos "livro" e "book" pertencentes a fontes de dados distintas. Nesse exemplo, os dois elementos originais são especificados e caracterizados, portanto, como sinônimos. Contudo essa especificação pode ser abreviada através da omissão do elemento da fonte de dados local identificada pela variável "$d1". Essa abreviação é possível porque, em uma declaração de cláusulas "ALIAS", existe um documento (no caso $d1) que possui um elemento "livro". A declaração abreviada seria ***EQUAL $d3/book IS $p/livro***. Na consulta 2 (Seção 4.6.1) pode ser observada a utilização do mecanismo de abreviação em uma cláusula "EQUAL". Normalmente, apenas os elementos referenciados em alguma declaração da cláusula "EQUAL" comporão a variável especificada na cláusula "EACH".

FULL

A cláusula opcional "FULL" pode ser utilizada em associação com a cláusula "EACH". Essa cláusula especifica, que todos os elementos de todas as fontes de dados declaradas na cláusula "ALIAS" comporão a árvore resultado associada à variável de resultado especificada na cláusula "EACH", mesmo que não sejam referenciados na cláusula "EQUAL". Se nenhuma forma de restrição for especificada, esse tipo de declaração ocasiona a UNIÃO de todas as fontes de dados referenciadas. Diferentemente do modelo de dados estruturados (bancos de dados convencionais), onde a operação de união necessita que as fontes de dados sejam compatíveis, documentos XML não oferecem nenhuma restrição para operações de união.

NULL

A cláusula opcional "NULL" pode ser utilizada em conjunto com a cláusula "ALIAS". Essa cláusula especifica que a fonte de dados ou o documento referenciado não é

imprescindível para a consulta. Dessa forma, caso a fonte de dados não esteja acessível ou não seja declarada corretamente, qualquer declaração referente a essa fonte de dados (ou algum elemento pertencente à mesma) será ignorada. Por exemplo, se uma consulta especificar a declaração ***ALIAS document("d3.xml")/lib $d3 NULL***, então o documento "d3.xml" não será considerado imprescindível para o resultado dessa consulta. Nesse caso, mesmo que a fonte de dados, contendo o documento, não esteja acessível, a consulta será processada. Da mesma forma, os elementos pertencentes ao documento "d3.xml" que aparecerem em outras declarações serão suprimidos. A declaração ***EQUAL $d1/livro $d3/book IS $p/livro*** é processada como se o elemento "$d3/book" não fizesse parte da expressão. A supressão do elemento "$d3/book" também ocorreria caso esse elemento fosse declarado incorretamente. Isso possibilita que alterações nos esquemas das fontes de dados não inviabilizem uma consulta. Esta propriedade é fundamental para ambientes como a Web e redes *ad hoc*.

HIDE

A cláusula opcional "HIDE" pode ser utilizada em conjunto com a cláusula "EQUAL". A cláusula indica que o resultado da relação declarada através da cláusula "EQUAL", mesmo compondo a variável especificada na cláusula "EACH", não deve aparecer na árvore resultado apresentada pela cláusula "RETURN". Por exemplo, na declaração ***EQUAL $d3/book/year $d1/livro/ano IS $p/livro/ano HIDE*** o elemento "livro/ano" (resultado da resolução de um conflito de sinônimos) pode ser referenciado em outras cláusulas da consulta, no entanto, ele não faz parte dos elementos apresentados pela cláusula "RETURN". O elemento "livro/ano" é um elemento virtual e seu ciclo de vida termina após a avaliação de todas as expressões condicionais presentes na consulta. A cláusula "HIDE" tem precedência sobre a cláusula "FULL", ou seja, um elemento declarado que utilize a cláusula "HIDE" não comporá a árvore resultado apresentada, mesmo tendo sido utilizada a cláusula "FULL".

KEY

A cláusula opcional "KEY" pode ser utilizada em conjunto com a cláusula "EQUAL", onde indica que os elementos especificados na declaração devem servir como identificador de equivalência entre objetos. Conforme descritos na Seção 2.2.3.3, os objetos que representam o mesmo conceito do mundo real devem sofrer uma fusão, onde uma única representação do objeto deve existir. Contudo, é necessário especificar

quando dois objetos são idênticos. Isso é feito através da comparação entre propriedades de fontes de dados distintas que são indicadas pela cláusula “KEY”. Considerando a declaração ***EQUAL $d1/livro/@isbn $d3/book/isbn IS $p/livro/isbn KEY***, os documentos “d1.xml” e “d3.xml” são processados originando um novo elemento “livro/isbn” que integra os elementos das fontes de dados originais. O conteúdo das fontes de dados sofrerá uma união para formar a árvore resultado, no entanto, caso o valor associado a algum elemento “$d1/livro/@isbn” seja igual ao valor de algum elemento “$d3/book/isbn”, então esses dois objetos (considerados idênticos) sofrerão uma fusão.

4.4 Funções Embutidas

A linguagem XQuery apresenta uma série de funções que podem ser utilizadas em suas expressões. Algumas funções foram herdadas das funções existentes na linguagem XPath, enquanto outras foram definidas na linguagem XQuery. Essas funções realizam operações de agrupamento, manipulação de nós entre outras. Todas as funções embutidas apresentadas pela linguagem XQuery podem ser utilizadas em expressões da linguagem MXQuery, em especial, as funções de agrupamento introduzidas pela linguagem XQuery. As funções *avg*, *sum*, *count*, *max* e min podem ser utilizadas para agrupar dados de uma mesma fonte ou de fontes distintas, pois podem ser utilizadas para modificar o resultado armazenado na variável da fonte de dados ou na variável de resultado. O funcionamento dessas e de outras funções está descrito em [Boa03] e, portanto, não será analisado nesse trabalho. Além das funções definidas pela linguagem XQuery, foi necessário modificar o comportamento de uma dessas funções para que a mesma oferecesse maior flexibilidade na resolução de conflitos de esquema.

Função NAME(n)

A função NAME(n), definida inicialmente em [Lit89], já foi incorporada à linguagem XQuery [Boa03]. Contudo, na linguagem XQuery é responsável apenas por identificar a origem de um dado. Em outras palavras, a função retorna o nome do dado designado por n. Adicionalmente, a extensão MXQuery incorpora ainda a função NAME(.n). Esta função retorna o ***container*** do dado especificado por n. O termo *container* representa o objeto hierarquicamente superior a um determinado objeto. Por exemplo, em um banco

de dados relacional tem-se tabelas como *container* de atributos, enquanto o banco de dados comporta-se como *container* de tabelas**.** Essa função pode ser utilizada de forma aninhada, onde NAME(.NAME(.n)) retorna o *container* do *container* de n. A necessidade dessa função está no fato de operações relacionais utilizarem não apenas os dados, mas também a origem desses dados [Lit89]. Essa função pode ser utilizada em lugar de um valor (ou constante) em qualquer um dos construtores presentes na estrutura EFLWOR. O resultado apresentado por essa função é um conjunto de caracteres (ou *string*).

4.5 Normalização das Expressões EFLWOR

O capítulo 3 destacou a importância da etapa de simplificação/otimização no processamento de consultas. Considerando que o objetivo da linguagem MXQuery é produzir consultas sobre fontes de dados distintas, muito provavelmente, tais fontes de dados encontram-se distribuídas e interconectadas através de um ambiente de rede. Conseqüentemente, os esforços em termos de otimização de consultas, estarão concentrados na produção do menor tráfego de rede possível. Mesmo considerando que a tendência é que esses ambientes possuam uma largura de banda cada vez maior, ainda assim, o tráfego imposto à rede continuará sendo o "gargalo" no processamento de uma consulta distribuída.

O processo de simplificação de uma consulta MXQuery pode ser dividido em 2 (duas) etapas: processo de extração de documentos e processo de simplificação de condições. O processo de extração de documentos será discutido com mais detalhes no capítulo 5. Basicamente, a idéia é extrair da consulta MXQuery todas as regras de mapeamento, regras de construção e anotações sobre as fontes de dados e sobre os elementos.

Antes de serem definidas as sub-consultas XQuery, o processo de simplificação de condições é executado. O processador tentará transformar todas as condições baseadas em variáveis de resultado em condições baseadas em variáveis de fontes de dados. A equação 4.1 ilustra essa tentativa, onde a $Expr_1$ referencia uma expressão baseada em uma variável de resultado. Essa expressão deve possuir pelo menos uma expressão de comparação entre dois termos, onde um deles apresenta uma expressão de caminho que utiliza a variável de resultado especificada pela cláusula "EACH". O segundo termo não

deve apresentar nenhuma expressão de caminho que utilizem variáveis de resultado ou variáveis de fontes de dados.

$$\left[\text{where } Expr_1\right]_{\text{GLOBAL}}(Expr) \;==\; \left[\text{where } Expr_{21}\right]_{\text{LOCAL1}}) \,(Expr) \text{ and } \ldots \text{ and } \left[\text{where } Expr_{2n}\right]_{\text{LOCALn}}) \,(Expr)$$

Equação 4.1. Normalização da Cláusula "WHERE" Baseada em Variável Resultado.

Caso a expressão condicional completa possua mais de uma expressão de comparação separadas por operadores lógicos, cada expressão de comparação deve ser analisada e, se possível, simplificada separadamente. A regra de normalização especificada na equação 4.1 foi utilizada na Seção 4.2 para transformar a condição ***WHERE $p/livro/ano > "1999"***.

As expressões condicionais das sub-consultas XQuery só conterão as condições baseadas em variáveis de fontes de dados e, mais especificamente, referente a fonte de dados para a qual a sub-consulta será encaminhada. Antes de encaminhar as sub-consultas XQuery para as fontes de dados, um outro processo de simplificação é realizado, onde devem ser utilizadas as regras estabelecidas em [Dra03].

4.6 Discussão sobre a Linguagem Proposta

Nos tópicos anteriores, a linguagem MXQuery foi apresentada através da descrição de suas cláusulas. A seguir, serão propostas algumas consultas a fontes de dados distintas e será analisada uma forma de escrever tais consultas através da MXQuery. Além disso, será apresentado com maiores detalhes como a linguagem MXQuery resolve os conflitos de integração descritos no capítulo 2.

4.6.1 Implementação de Consultas

Preparação

A Figura 4.2 apresenta uma típica consulta expressa através da linguagem MXQuery, onde se pode observar 4 (quatro) estruturas básicas. A primeira especifica a variável de resultado que será responsável por conter o resultado integrado das fontes de dados. A

segunda estrutura especifica as fontes de dados consultadas, onde cada fonte de dados é associada a uma variável de fonte de dados. A terceira estrutura é responsável por especificar o mapeamento entre as fontes de dados consultadas e a variável de resultado. Finalmente, a quarta estrutura é responsável por especificar o formato do documento resultado, onde, geralmente, deve ser utilizada a variável de resultado.

A seguir será apresentado um cenário no qual serão baseadas as consultas propostas nesta seção. Os apêndices A, B e C apresentam algumas informações complementares

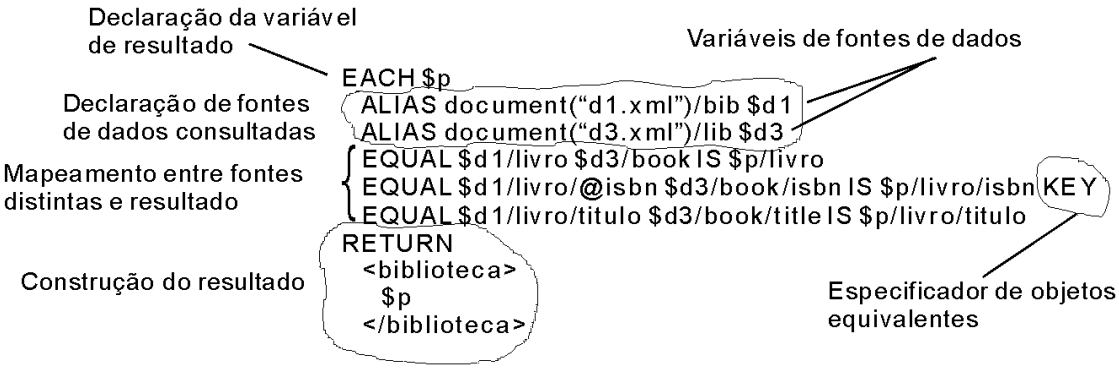

Figura 4.2. Uma Típica Consulta MXQuery.

para compreensão das consultas propostas. Assim, o apêndice A apresenta documentos XML que representam os dados existentes nas fontes de dados. Na realidade, nenhum documento similar aos documentos apresentados no apêndice A é gerado pela arquitetura de integração, pois os dados existem apenas em suas origens, não sendo materializados após a definição do CDM baseado em XML.

O apêndice C apresenta os esquemas conceituais expressos em XML *Schema*, que correspondem aos esquemas locais transformados pelas *interfaces* XML (ver capítulo 5). Esses XML *Schemas* são armazenados pela arquitetura de integração e servem como base para a proposição de consultas. O apêndice B apresenta documentos XML que representam o resultado das consultas propostas. Esses documentos podem ser persistidos ou simplesmente apresentados aos usuários como resposta às suas consultas.

Cenário

A utilidade e aplicabilidade da linguagem MXQuery serão ilustradas com exemplos descritos a seguir. A idéia é especificar consultas de acordo com o seguinte cenário. Considere que cinco grandes livrarias resolvem fundar um consórcio, onde seus títulos seriam disponibilizados na Web. Cada uma delas gerencia sua própria fonte de dados. Muito provavelmente, tais fontes de dados são heterogêneas. Essa parceria, no entanto,

requer que os dados referentes aos seus acervos sejam “vistos” de forma integrada. Porém, nenhuma das livrarias deseja perder o controle sobre seus dados locais. Dessa forma, devem-se integrar fontes de dados heterogêneas, preservando a autonomia local de cada uma delas.

Suponha que a integração dos dados das diversas livrarias é realizada através da abordagem de MDBS e considere que os esquemas conceituais correspondentes a cada fonte de dados local serão representados em XML. Em Apêndice C são apresentados os esquemas conceituais das fontes de dados de cada livraria. Por questão de simplicidade, considere que os dados referentes às fontes de dados locais serão disponibilizados na forma de documentos XML. Por exemplo, o documento “d1.xml” (veja apêndice A) pode representar parte do esquema de um banco de dados relacional que deve ser visível para o consórcio. O documento “d2.xml”, por sua vez, pode representar o mapeamento do esquema de um banco de dados orientado a objeto em XML. Finalmente, os demais documentos podem representar outros tipos de fontes de dados que não podem ser caracterizadas como sistemas de bancos de dados (por exemplo, documentos HTML, XML, planilhas). No capítulo 5, será mostrado que os dados não precisam ser armazenados como documentos XML.

Os esquemas conceituais apresentados em XML *Schema* foram gerados automaticamente a partir das fontes de dados locais. Por este motivo, nenhum conflito semântico foi resolvido durante a geração destes esquemas. Considere, ainda, que o MDBS, responsável pela integração das diversas fontes de dados, tem a funcionalidade de processar consultas expressas em MXQuery. Suponha agora que as consultas apresentadas a seguir foram enviadas ao MDBS para acessarem dados referentes às livrarias participantes do consórcio.

Consultas

Consulta 1: Retornar títulos e autores dos livros existentes nas livrarias, cujos esquemas conceituais estão representados pelos documentos XML “d1.xml” e “d3.xml”.

Uma possível expressão MXQuery referente à consulta 1 é apresentada na Figura 4.3. A execução da consulta 1 retornará uma variável “$p” contendo elementos “livro” dos documentos d1.xml e d3.xml (especificados nas cláusulas “ALIAS”), cada um contendo os sub-elementos “titulo” e “autor”. Conforme a declaração da cláusula “RETURN”, o

resultado deve ser atribuído a um elemento com raiz “biblioteca”. O resultado dessa consulta é mostrado no Apêndice B.

```
EACH $p
  ALIAS document(“d1.xml”)/bib $d1
  ALIAS document(“d3.xml”)/lib $d3
  EQUAL $d1/livro $d3/book IS $p/livro
Sinônimos
  EQUAL $d1/livro/titulo $d3/book/title IS $p/livro/titulo
  EQUAL $d1/livro/autor $d3/book/author IS $p/livro/autor
RETURN
  <biblioteca>
    $p
  </biblioteca>
```

Figura 4.3. Consulta 1 Expressa em MXQuery.

Observe que o atributo “isbn” e os sub-elementos “ano”, “editora”, “isbn”, “year” e “press” dos elementos originais não fazem parte da árvore resultado, pois não foram especificados em nenhuma das cláusulas “EQUAL”. Vale ressaltar que, através da expressão ***EQUAL $d1/livro/titulo $d3/book/title IS $p/livro/titulo***, está sendo especificado que “livro/titulo” e “book/title” são sinônimos, representando, portanto, um mesmo atributo do mundo real. Isto demonstra que a MXQuery fornece suporte necessário para que usuários especifiquem diretivas para resolução de conflitos por parte do processador de consultas de um MDBS.

Consulta 2: Retornar títulos, autores e preços dos livros existentes nas livrarias, cujos esquemas conceituais estão representados pelos documentos “d1.xml” e “d3.xml”. Utilize o ISBN para especificar a equivalência entre os objetos de fontes de dados distintas. Considere ainda que os preços representados nos documentos em questão estão expressos em moedas distintas, onde U$ 1,00 equivale à R$ 2,98.

Uma possível expressão MXQuery é apresentada na Figura 4.4. Observe que a consulta 2 é semelhante à anterior, contudo existem algumas diferenças. Uma diferença reside no fato de que foi inserida uma linha para recuperar a informação de preço, sendo necessário efetuar uma conversão de moeda (neste exemplo de Dólar para Real). Essa função de conversão resolve, portanto, um conflito de domínio existente entre as duas fontes de dados especificadas.

Outra diferença da consulta 2 em relação à consulta 1 é que foi inserida uma linha para comparar o ISBN dos documentos declarados. Observe que a cláusula “KEY” especifica a propriedade que deve ser utilizada para identificar objetos idênticos, onde

objetos considerados idênticos devem sofrer uma fusão no processo de integração. Ainda nessa declaração, pode ser identificada a resolução de um conflito de esquema, visto que se pode comparar diretamente o atributo “livro/@isbn” com o sub-elemento “book/isbn”. Essa declaração insere o sub-elemento “isbn” na árvore resultado, como resultado da integração.

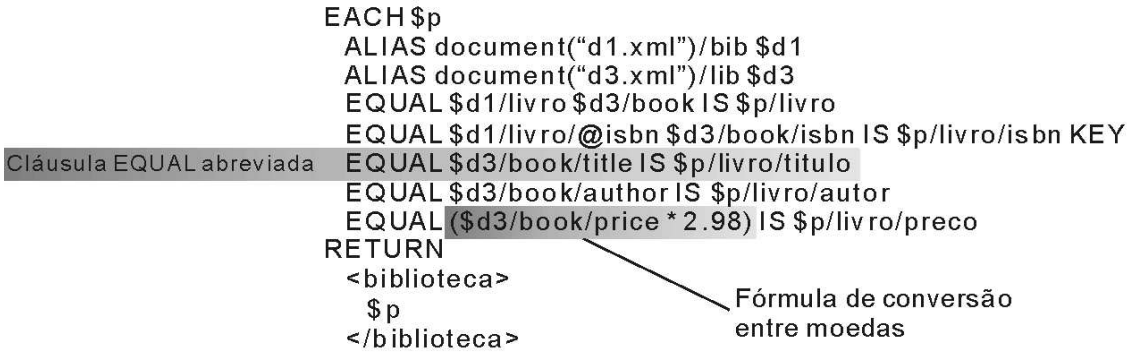

Figura 4.4. Consulta 2 Expressa em MXQuery.

Observe que, na consulta 2, as declarações de equivalência de título, autor e preço estão na sua forma abreviada, contudo, esse fato não causa nenhum impacto no resultado da consulta. O resultado dessa consulta corresponde a um elemento composto por 4 sub-elementos do tipo livro (veja apêndice B).

As duas primeiras consultas apresentadas efetuam a integração em duas fontes de dados distintas, porém a utilização da cláusula “KEY” modifica o modo como essa integração é realizada. A primeira consulta não especifica a cláusula “KEY”, portanto a arquitetura MDBS produz um resultado através da **união** das duas fontes de dados especificadas. Como dito anteriormente, não é necessário existir uma compatibilidade entre as fontes de dados para que elas possam sofrer uma união. Por isso, mesmo que a consulta não especificasse a ocorrência de sinônimos, ainda assim, a operação de união seria realizada.

A segunda consulta utiliza a cláusula “KEY” em um de seus mapeamentos. Isso significa que na consulta o atributo/elemento “isbn” servirá com o identificador de objetos equivalentes. O resultado não pode apresentar mais de um objeto contendo um mesmo valor para o atributo/elemento “isbn”, conseqüentemente, o processador de consultas MXQuery deve fazer uma **fusão** entre objetos equivalentes. A cláusula “KEY” determinou a diferença entre o número de elementos “livro” apresentado no resultado das consultas 1 e 2.

Consulta 3: Retornar a união de todos os elementos representados nos documentos “d1.xml” e “d2.xml”, suprimindo o sub-elemento “autor” da árvore resultado.

A consulta expressa utilizando a linguagem MXQuery é ilustrada na Figura 4.5. Observe que essa consulta realiza uma operação de união entre as fontes de dados especificadas nas cláusulas “ALIAS” e que a cláusula “FULL” determina que todos os elementos dessas fontes de dados devem ser representados. Caso fosse necessário eliminar duplicidades entre as fontes de dados participantes, algum elemento poderia ser indicado para ser comparado. Dessa forma, elementos equivalentes seriam identificados e os objetos por ele especificados sofreriam uma fusão. As duplicidades seriam eliminadas acrescentando uma declaração de equivalência seguida da cláusula “KEY”.

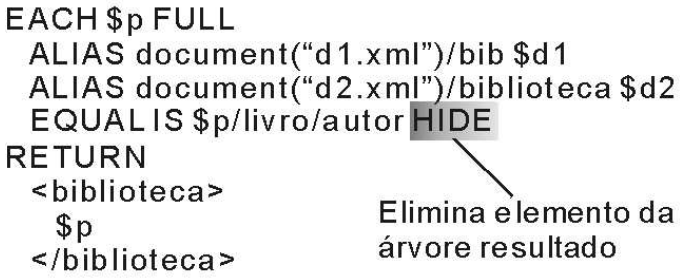

Figura 4.5. Consulta 3 Expressa em MXQuery.

A declaração ***EQUAL IS $p/livro/autor HIDE*** é utilizada para suprimir a apresentação do sub-elemento “autor”, pois a cláusula “HIDE” tem precedência sobre a cláusula “FULL”. Note que a cláusula “EQUAL” está abreviada e mesmo não apresentando explicitamente nenhum elemento “autor” das fontes de dados consultadas, ainda assim, o processador de consultas é capaz de identificá-los. O resultado da consulta é apresentado no apêndice B.

Consulta 4: Retornar os títulos dos livros existentes nas livrarias cujos esquemas conceituais estão representados pelos documentos XML “d1.xml” e “d3.xml”. Utilize o ISBN para suprimir duplicidades. Os livros devem ter sido publicados após 1999.

A expressão MXQuery mostrada na Figura 4.6 pode ser utilizada para expressar esta consulta. A consulta 4 é similar à consulta 2, porém foi acrescentada a declaração de equivalência do sub-elemento “ano” com “year”, resolvendo dessa forma um conflito de sinônimos. Observe que, diferentemente das declarações de equivalência de sub-elementos especificadas nas consultas anteriores, a declaração especificada na consulta

4 não determina que o sub-elemento “ano” componha o resultado dessa consulta, devido a utilização da cláusula “HIDE”. A idéia aqui é utilizar esse sub-elemento (após a resolução do conflito de sinônimo) apenas para o filtro especificado na cláusula “WHERE”.

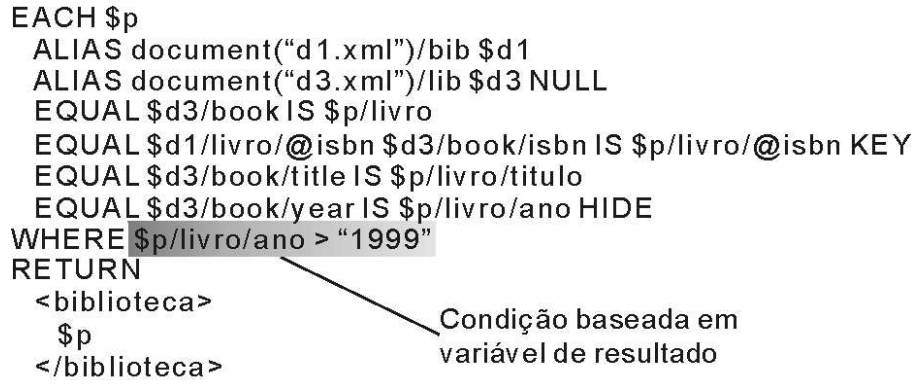

Figura 4.6. Consulta 4 Expressa em MXQuery.

Conforme os documentos apresentados no apêndice A, existe um conflito de dados entre dois dos elementos que comporão a árvore resultado dessa consulta. Dois elementos “livro” que apresentam o mesmo ISBN possuem o sub-elemento “ano” e “titulo” com valores distintos, onde o sub-elemento “ano” possui um dos valores igual a 2001 e o outro igual a 1999. No caso do sub-elemento “titulo”, o conteúdo de sub-elemento é apresentado em português no documento “d1.xml” e em inglês no documento “d3.xml”.

Esse conflito de dados é resolvido através da indicação da fonte de dados cujo valor de seus elementos deverá prevalecer. Essa indicação é feita através da ordem das declarações das fontes de dados consultadas. Na consulta 4, a fonte de dados especificada pela variável “$d1” é priorizada, significando que, em caso de conflito de dados o seu conteúdo será preservado. Devido a esse fato, o resultado da integração entre as fontes de dados “d1.xml” e “d3.xml” determina que o valor do sub-elemento “ano” deve ser igual a 2001 e o título em português deve constar no resultado. Nesse exemplo, se a ordem das fontes de dados na cláusula “ALIAS” fosse invertida, o resultado da consulta seria um elemento “livro” vazio. Observe que, na consulta 2, também ocorreram conflitos de dados (sub-elementos “titulo” e “preço”), que foram resolvidos da mesma forma.

A utilização da cláusula “NULL” na consulta determina que a indisponibilidade do documento “d3.xml” durante a execução da mesma não inviabiliza o resultado. Coincidentemente, o resultado apresentado é o mesmo, independente da disponibilidade

de “d3.xml”, no entanto, sua indisponibilidade ocasionaria um erro se a cláusula “NULL” não fosse especificada.

A condição utilizada na cláusula “WHERE” da quarta consulta é baseada em uma variável de resultado. Isso significa que, a princípio, para que essa condição seja avaliada, é necessário realizar a integração entre as duas fontes de dados e, em seguida, verificar a condição. Contudo, no processo de otimização de consultas, essa condição pode ser substituída por uma condição baseada em fonte de dados. A expressão equivalente seria: ***WHERE $d1/livro/ano > “1999” AND $d3/book/year > “1999”***. Pode-se notar que caso o processo de otimização execute essa substituição, então a fonte de dados “d3.xml” não enviará nenhuma resultado parcial (eliminando tráfego de rede desnecessário). Além disso, nenhum conflito de dados ocorrerá, pois apenas o documento “d1.xml” resultará um elemento cujo ISBN será igual a “352”.

Consulta 5: Retornar os títulos e editoras de publicações em inglês e títulos e editoras de suas traduções. Os documentos “d1.xml” e “d3.xml” devem ser utilizados e o ISBN deve identificar os elementos correspondentes.

Uma expressão MXQuery, que pode ser utilizada para expressar a consulta acima pode ser a apresentada na Figura 4.7. Analisando a consulta 5, deve-se ter uma priorização da fonte de dados “d3.xml” sobre “d1.xml”, já que os títulos originais controlam a consulta. Devem ser geradas duas linhas para título e duas linhas para editoras, pois as quatro informações devem ser preservadas. Devido a esse fato, os sub-elementos “title” e “titulo” e “press” e “editora” não são considerados sinônimos.

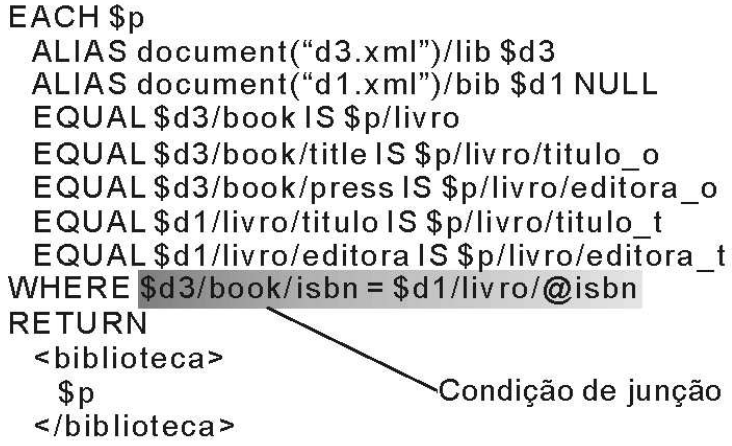

Figura 4.7. Consulta 5 Expressa em MXQuery.

A cláusula “WHERE” especifica as regras da junção, onde apenas os livros que satisfizerem a condição comporão o resultado. Nesse caso, apenas livros com títulos originais e traduzidos comporão o resultado (veja o apêndice B). Os elementos inseridos na árvore resultado devem apresentar nomes diferentes dos elementos das fontes de dados participantes, pois, caso contrário, o processador de consultas MXQuery “entederia” que a cláusula de equivalência (*EQUAL*) estava abreviada e, dessa forma, apresentaria um resultado diferente do esperado.

Nessa consulta não é possível utilizar a cláusula “KEY” como na cláusula ***EQUAL $d1/livro/@isbn $d3/book/isbn IS $p/livro/isbn KEY***, pois o resultado dessa fusão é diferente do resultado da junção.

Consulta 6: Retornar os títulos, autores e editoras dos livros existentes nas livrarias, cujos esquemas conceituais estão representados pelos documentos “d1.xml”, “d4.xml” e “d5.xml”. Utilize o ISBN para suprimir as duplicidades. Apenas autores com obras disponíveis em mais de uma livraria (das especificadas) devem ter suas obras listadas.

A Figura 4.8 apresenta uma possível expressão MXQuery para representar essa consulta. Observando mais atentamente a expressão MXQuery abaixo, pode-se identificar dois componentes distintos nessa consulta. O primeiro componente tem a responsabilidade de construir um documento contendo todos os livros (títulos, editoras e autores) das fontes de dados (documentos) participantes. Deve-se destacar a obtenção dos autores através do operador de dereferência sobre o atributo “autref” do documento “d4.xml” e a utilização do curinga “*” para representar as editoras do documento “d5.xml”.

A função NAME(n) é utilizada na consulta 6 para determinar o nome das editoras do documento “d5.xml” e o *container* dos elementos “livro”. No exemplo, ela foi aplicada para determinar qual documento originou cada elemento na árvore resultante armazenando essa informação no sub-elemento “origem”.

A segunda parte da consulta atribui às variáveis “$q” e “$r” iterações sobre valores (elementos “livro”) associados à variável “$p”. A idéia é aninhar dois laços para comparar cada elemento associado a “$p” com todos os outros elementos também associados a “$p”. Dentro do laço mais interno, os sub-elementos “autor” são comparados, assim como os sub-elementos criados pela função NAME(.n) com o objetivo de identificar origens diferentes.

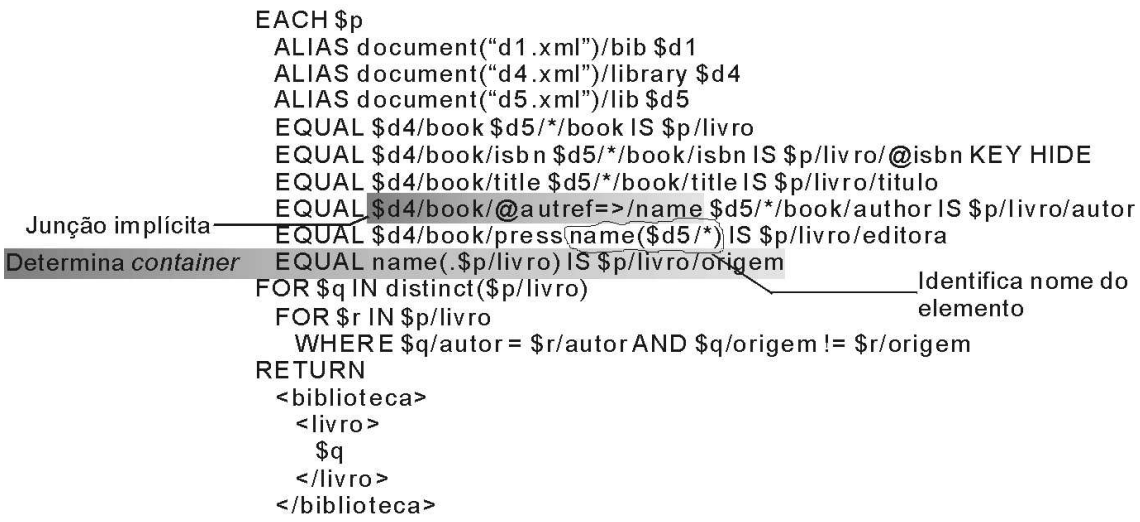

Figura 4.8. Consulta 6 Expressa em MXQuery.

A comparação dos autores é feita para cada sub-elemento "autor" pertencente ao elemento "livro". Essa forma de comparação poderia gerar elementos "livro" iguais para cada autor que satisfizesse a condição imposta pela construção "WHERE", porém os elementos duplicados são suprimidos da árvore resultante devido à utilização da função "*distinct*" definida em [Boa03].

4.6.2 Resolução de Conflitos

O capítulo 2 apontou alguns problemas que devem ser resolvidos para possibilitar a integração de dados. A linguagem MXQuery concentra-se nos problemas referentes à heterogeneidade semântica. Nesta seção, será apresentada uma discussão sobre como a solução de tais problemas é suportada pela linguagem MXQuery proposta.

Os **conflitos de nome** do tipo sinônimo e homônimo são resolvidos através do mapeamento do esquema das fontes de dados, onde a cláusula "EQUAL" tem a função de fazer esse mapeamento. Alguns **conflitos de domínio**, como conversão de medidas, também são resolvidos utilizando a cláusula "EQUAL". Nesse caso, um dos elementos deve ser expresso através de uma fórmula de conversão (por exemplo, *EQUAL ($d3/book/price*2.98) IS $p/livro/preco*) ou através de funções embutidas ou definidas pelo usuário (por exemplo, *EQUAL dolartoreal($d3/book/price) IS $p/livro/preco*). As funções definidas pelo usuário utilizam a sintaxe adotada pela linguagem XQuery [Boa03].

No caso de **conflitos de domínio** provocados por incompatibilidade de tipos de dado, a linguagem MXQuery utiliza coerção de tipos para resolver tais conflitos. Na

coerção de tipos, os dados têm seus tipos convertidos automaticamente para outros tipos de dados. Existem dois critérios para realizar a coerção de tipos. O primeiro é o contexto, no qual a coerção precisa ser realizada. Na expressão exemplo do parágrafo anterior, considerando que o tipo de dado do elemento "price" é string, esse tipo de dado é convertido para ponto flutuante e a operação de multiplicação é realizada. Em seguida, o resultado é novamente convertido para string, pois a declaração "EQUAL" deve processar a integração e construir o sub-elemento "preco". O segundo critério é converter o tipo de dado menos restritivo para o tipo de dado mais restritivo, segundo a seguinte ordem: ***String → Pt. Flutuante → Inteiro → Booleano***. Caso a conversão não seja possível, o processador realiza a conversão de tipos no sentido inverso.

Os **conflitos semânticos** são resolvidos através do mapeamento entre os elementos originais em novos elementos e/ou da construção de novos elementos, conforme analisado na Seção 2.2.3.3. Por exemplo, naquela seção, o conflito semântico do tipo intersecção de classes é resolvido através da integração das propriedades comuns pertencentes às fontes de dados distintas e da definição de subclasses contendo as propriedades não comuns. Esse tipo de conflito é resolvido através da cláusula "EQUAL", onde é possível especificar o relacionamento (integração) entre propriedades comuns e definir a construção de novos elementos contendo as propriedades não comuns (definição de subclasses).

Alguns tipos de **conflitos de esquema** (estrutural) podem ser resolvidos através do mapeamento de elementos com a cláusula "EQUAL". A consulta 6 utiliza esse mapeamento para resolver o conflito de autores dos livros, onde em um documento essa informação é representada como um sub-elemento de livro e em outro documento a informação de autores é expressa como um elemento independente. Ainda utilizando a consulta 6, um outro problema de heterogeneidade estrutural é resolvido com relação à "editora", pois este dado é representado como elemento em uma fonte de dados (metadado) e como conteúdo em outra fonte de dados (dado). Nesse caso, a função "NAME" foi utilizada para extrair o nome do elemento que representava a informação "editora" na fonte de dados "d5.xml".

Os **conflitos de dados** são resolvidos através da priorização de fontes de dados, que é estabelecida através da ordem das declarações dessas fontes na cláusula "ALIAS". A ordem das declarações das fontes de dados participantes indica a ordem de confiabilidade dessas fontes de dados e, conseqüentemente, é estabelecida uma hierarquia para resolução de conflitos de dados. Outra forma de resolver esse tipo de

conflito é preservando as duas informações em conflito das fontes de dados distintas. Nesse caso, os elementos devem ser mapeados para elementos distintos da árvore resultado.

CAPÍTULO 5

UMA ARQUITETURA PARA PROCESSAR CONSULTAS MXQUERY EM SISTEMAS DE BANCOS DE DADOS MÚLTIPLOS

5.1 Introdução

Como afirmado anteriormente, a arquitetura MDBS é a mais adequada para integrar múltiplas fontes de dados heterogêneas em ambientes cujo controle sobre as conexões e desconexões das fontes de dados é impraticável. As fontes de dados publicadas nesses ambientes não possuem qualquer tipo de compromisso ou subordinação com uma administração central, que pudesse ser responsável por manter um esquema global de integração. Conseqüentemente, adotar a abordagem de bancos de dados federados é bastante difícil. Por outro lado, quando a responsabilidade de especificar as regras de integração é transferida para a consulta, em última instância para o usuário, tal personagem pode verificar a disponibilidade das fontes de dados e definir dinamicamente os critérios de integração.

A abordagem de bancos de dados múltiplos viabiliza essa flexibilidade e, embora seja penalizada pela falta de transparência na integração das diversas fontes de dados, garante uma maior autonomia para tais fontes de dados. Assim, nenhum esforço adicional é requerido quando uma fonte de dados deixa a comunidade ou altera as especificações de seus dados publicados.

Dois aspectos são fundamentais para a adoção da estratégia de integração baseada em um MDBS. O primeiro aspecto é a definição de um modelo de dados comum que deve representar os dados extraídos das fontes de dados locais. O segundo aspecto é a linguagem de consultas para bancos de dados múltiplos que, além de apresentar as características de uma linguagem de bancos de dados convencional, deve apresentar mecanismos que permitam relacionar e integrar dados de fontes distintas.

Atualmente, o processo de integração de fontes de dados deve considerar a necessidade de atuar sobre dados não estruturados e estruturados conjuntamente. Por esse motivo, ao adotar um modelo de dados comum, deve-se considerar a possibilidade de representar tais tipos de dados. Portanto, é necessário optar por um modelo de dados flexível que seja capaz de representar dados não estruturados e ao mesmo tempo permita representar restrições de dados originados de bancos de dados convencionais.

O capítulo 3 discutiu a representação de dados semi-estruturados, onde o padrão XML vem ganhando notoriedade por sua flexibilidade, independência de plataforma e simplicidade de utilização [BrP00, Chr00, Sel01, Pan02]. Esse trabalho adotou o XML como modelo de dados comum para representar as fontes de dados a serem integradas, possibilitando, dessa forma, representar qualquer tipo de informação com um mínimo de esforço. Além disso, as ferramentas desenvolvidas para consultar dados representados em XML podem ser utilizadas e a linguagem de consultas para bancos de dados múltiplos pode basear-se em linguagens orientadas a XML.

O restante desse capítulo está organizado como se segue. A seção 5.2 analisa os diversos componentes de uma arquitetura MDBS. A seção 5.3 explora algumas técnicas de mapeamento entre modelos de dados e linguagens de consulta. A seção 5.4 discute a função dos documentos auxiliares extraídos de uma consulta MXQuery.

5.2 Componentes de uma Arquitetura MDBS

Esse trabalho propõe utilizar a tecnologia de MDBS como a estratégia de integração de fontes de dados heterogêneas. A Figura 5.1 apresenta um modelo abstrato da arquitetura utilizada para integrar dados armazenados em fontes de dados heterogêneas, distribuídas e autônomas.

Na Figura 5.1, a interface de consulta de usuário (**QUI** – *Query User Interface*) faz a intermediação entre o usuário e a arquitetura de integração, sendo de responsabilidade deste componente receber as consultas e apresentar os resultados. Um usuário que desejar acessar múltiplas fontes de dados integradas através da arquitetura, deverá enviar ao QUI uma consulta MXQuery, denominada consulta global. As consultas globais são enviadas, então, ao gerenciador de consultas (**QM** – *Query Manager*). A principal funcionalidade deste componente é dividir uma consulta global em sub-consultas e, em seguida, enviá-las às fontes de dados correspondentes.

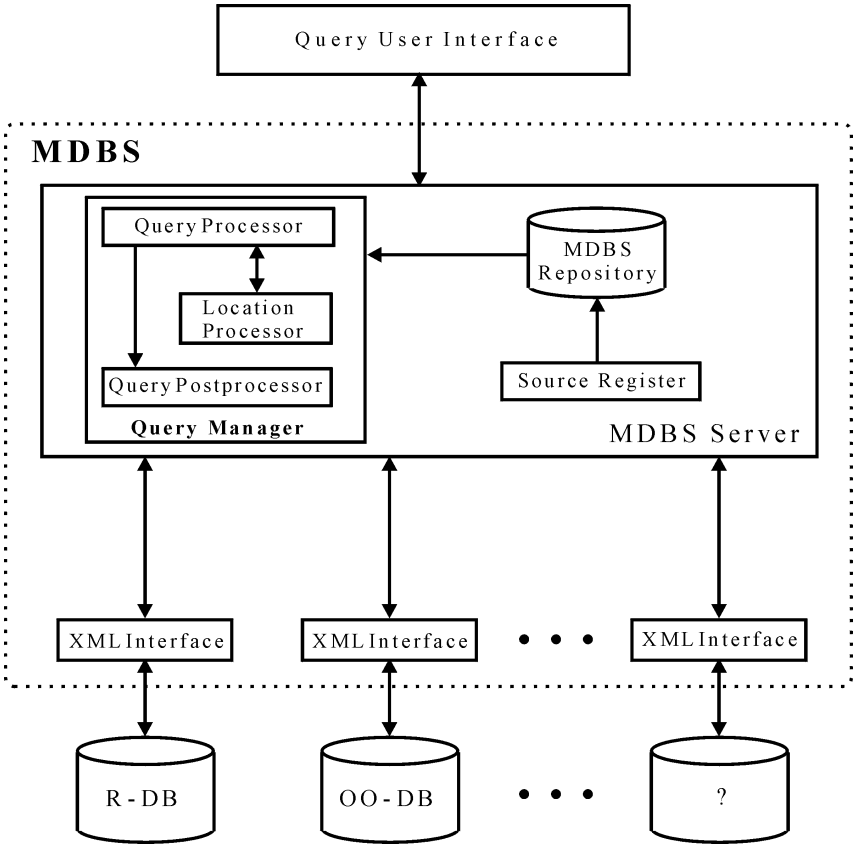

Figura 5.1. Arquitetura do Mecanismo de Integração (MDBS).

O gerenciador de consulta está dividido, por sua vez, em três componentes principais, que são: processador de consulta (**QP** – *Query Processor*), processador de localização (**LP** – *Location Processor*) e pós-processador de consulta (**QPp** – *Query Postprocessor*). O processador de consulta deve receber as consultas expressas em MXQuery, identificar as fontes de dados (ou interfaces das fontes de dados) referenciadas, simplificar e otimizar as consultas globais, gerar as sub-consultas e, finalmente, encaminhar as sub-consultas para as fontes de dados correspondentes.

O processador de localização desempenha um papel auxiliar para o processador de consulta, pois ele deve identificar e localizar as fontes de dados referenciadas e indicar erros de localização. Os erros de localização podem significar a indisponibilidade de alguma fonte de dados referenciada, sendo, portanto, úteis no processo de simplificação de consultas, uma vez que as fontes de dados inacessíveis devem ser excluídas da consulta.

O processo de exclusão de uma fonte de dados de uma consulta deve obedecer às regras estabelecidas na própria consulta. Dessa forma, nem sempre será possível excluir a fonte de dados da consulta e, nesse caso, a consulta inteira é rejeitada. Caso seja

possível excluir a fonte de dados, todas as expressões que referenciam a fonte de dados excluída devem ser excluídas. As fontes de dados que podem ser excluídas são identificadas na sintaxe do MXQuery através da utilização da cláusula “NULL”. O pós-processador de consulta será explicado mais à frente.

As sub-consultas são enviadas para as interfaces XML (**XML-I** – *XML Interface*), onde são traduzidas para linguagem de consulta nativa e direcionadas para as fontes de dados locais correspondentes às interfaces XML. Portanto, há uma interface XML associada a cada fonte de dados local. O papel desempenhado pela XML-I será discutido posteriormente.

Depois de processada localmente, o resultado de uma sub-consulta é enviado à XML-I, onde é traduzido para XML e encaminhado ao pós-processador de consulta. Esse componente deve utilizar as regras estabelecidas na consulta original para resolução de conflitos. Por exemplo, pode-se estabelecer uma regra para resolver problemas de sinônimo utilizando a expressão “EQUAL”. Assim, através da MXQuery, o usuário é capaz de fornecer informação para que o pós-processador de consulta resolva conflitos.

Após a resolução dos conflitos, o QPp deve combinar o resultado das diversas sub-consultas recebidas de acordo com as regras de construção de resultados também estabelecidas na consulta original. O resultado combinado das sub-consultas é, finalmente, encaminhado à interface de consulta de usuário.

Dois outros componentes são necessários para o funcionamento do mecanismo de integração proposto. O registrador de fonte (**SR** – *Source Register*) é encarregado de controlar a conexão e desconexão das fontes de dados participantes do MDBS. Esse componente recebe as requisições de participação no MDBS enviadas pelas fontes de dados locais, registra a localização física das fontes de dados (**URL** – *Uniform Resource Locator*) e publica o esquema disponibilizado pelas fontes de dados locais.

O repositório MDBS (**MR** – *MDBS Repository*) deve armazenar informações de controle do MDBS. Essas informações incluem os metadados das fontes de dados participantes, definições do MDBS, como, por exemplo, as fontes de dados participantes ou consultas pré-formatadas e a localização física das fontes de dados participantes.

Uma interface XML é responsável por mapear o esquema das fontes de dados locais em um modelo de dados comum (esquema conceitual), que é apresentado na arquitetura de integração. Na arquitetura proposta, os esquemas conceituais devem ser

representados através de esquemas XML, definidos através da XML *Schema*, uma linguagem de definição de esquemas para dados XML [Fal00] (veja capítulo 3 e apêndice C).

Várias propostas de mapeamento de dados armazenados em bancos de dados convencionais (por exemplo, relacionais e orientados a objeto) em documentos XML têm sido publicadas e implementadas, como, por exemplo, a ferramenta XDK for PL/SQL da Oracle [XDK03]. De qualquer forma, pode-se utilizar a seguinte estratégia genérica de mapeamento: representar os objetos (ou tabelas) como elementos XML e as propriedades (ou atributos) como sub-elementos. Essa forma de mapeamento é mais transparente e garante uma maior autonomia para as fontes de dados, porém o processo de integração de dados através da linguagem de consultas é sobrecarregado com as atividades de resolução de conflitos.

A Figura 5.2 apresenta a interface XML com mais detalhes. A atividade de mapeamento descrita anteriormente é executada pelo componente de definição de mapeamento (**MD** – *Mapping Definition*).

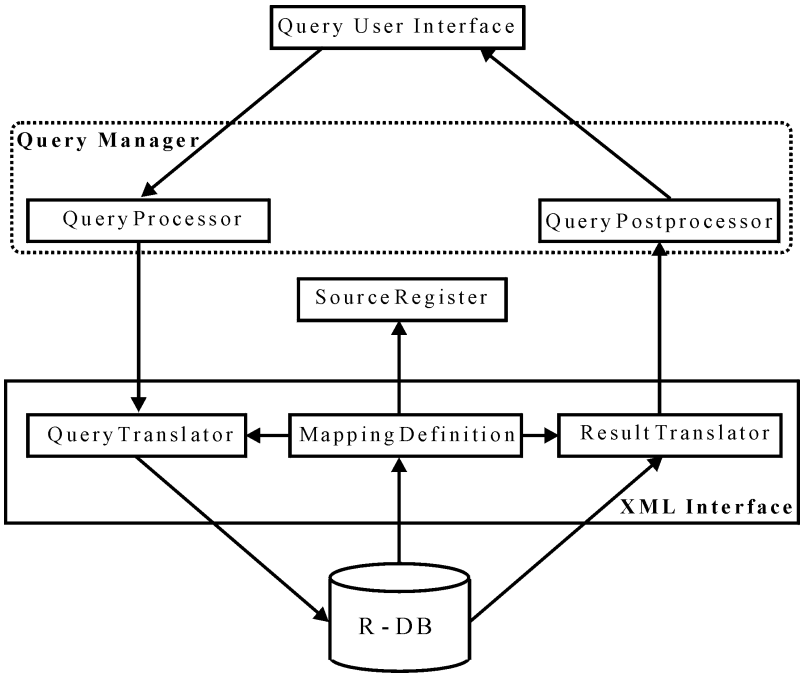

Figura 5.2. Detalhe da Interface XML.

Além da atividade de mapeamento, a interface XML deve traduzir consultas expressas em uma linguagem orientada ao modelo de dados XML para a linguagem nativa da fonte de dados local. Essa tarefa é realizada pelo tradutor de consultas (**QT** – *Query Translator*).

O último componente da interface XML é o tradutor de resultados (**RT** – *Result Translator*), que executa o papel inverso do tradutor de consultas. Esse componente

deve apresentar os resultados expressos de acordo com o modelo de dados da fonte de dados local em um formato de documento XML. Por exemplo, um banco de dados relacional retorna um conjunto de tuplas para uma determinada consulta e, em seguida, o tradutor de resultados deve criar elementos e sub-elementos XML que identifiquem a tabela consultada, as diversas tuplas obtidas e o conteúdo dos atributos de cada tupla. A saída deste componente é, portanto, um documento XML, como os apresentados no apêndice B.

5.2.1 Interface de Consulta de Usuário

A interface de consulta de usuário não é propriamente um componente da arquitetura de integração, portanto, nenhum processamento é executado por ela. Ela foi representada para simbolizar a forma como uma consulta MXQuery é enviada à arquitetura de integração e como as respostas são devolvidas por essa arquitetura.

Pode-se pensar na QUI como um aplicativo que apresenta uma interface de entrada de consultas MXQuery e possibilita a visualização de documentos XML resultados, ou como uma biblioteca (API ou classe Java) que confere a uma linguagem hospedeira possibilidade de especificar consultas expressas através da MXQuery, ou ainda, como uma agente móvel que transporta consultas MXQuery para serem executadas pela arquitetura de integração [Bra03].

O importante na especificação da QUI é garantir uma forma amigável de interagir com a arquitetura MDBS e as fontes de dados participantes do ambiente de integração. Além disso, a ferramenta deve prover acesso direto às informações armazenadas no repositório MDBS, possibilitando ao usuário explorar os esquemas publicados pelas fontes de dados. Essa característica é fundamental, pois como as consultas e regras de integração são definidas dinamicamente pelo próprio usuário, ele deve possuir a maior quantidade de informação possível a respeito dos esquemas das fontes de dados a serem consultadas.

5.2.2 Processador de Consultas

O primeiro passo executado pelo QP, ao receber uma consulta MXQuery, é checar a sintaxe e semântica da consulta recebida, onde uma árvore de expressões que simboliza

a consulta original é gerada. Observando-se a figura 5.3, podem-se identificar dois componentes de análise de validade de consultas.

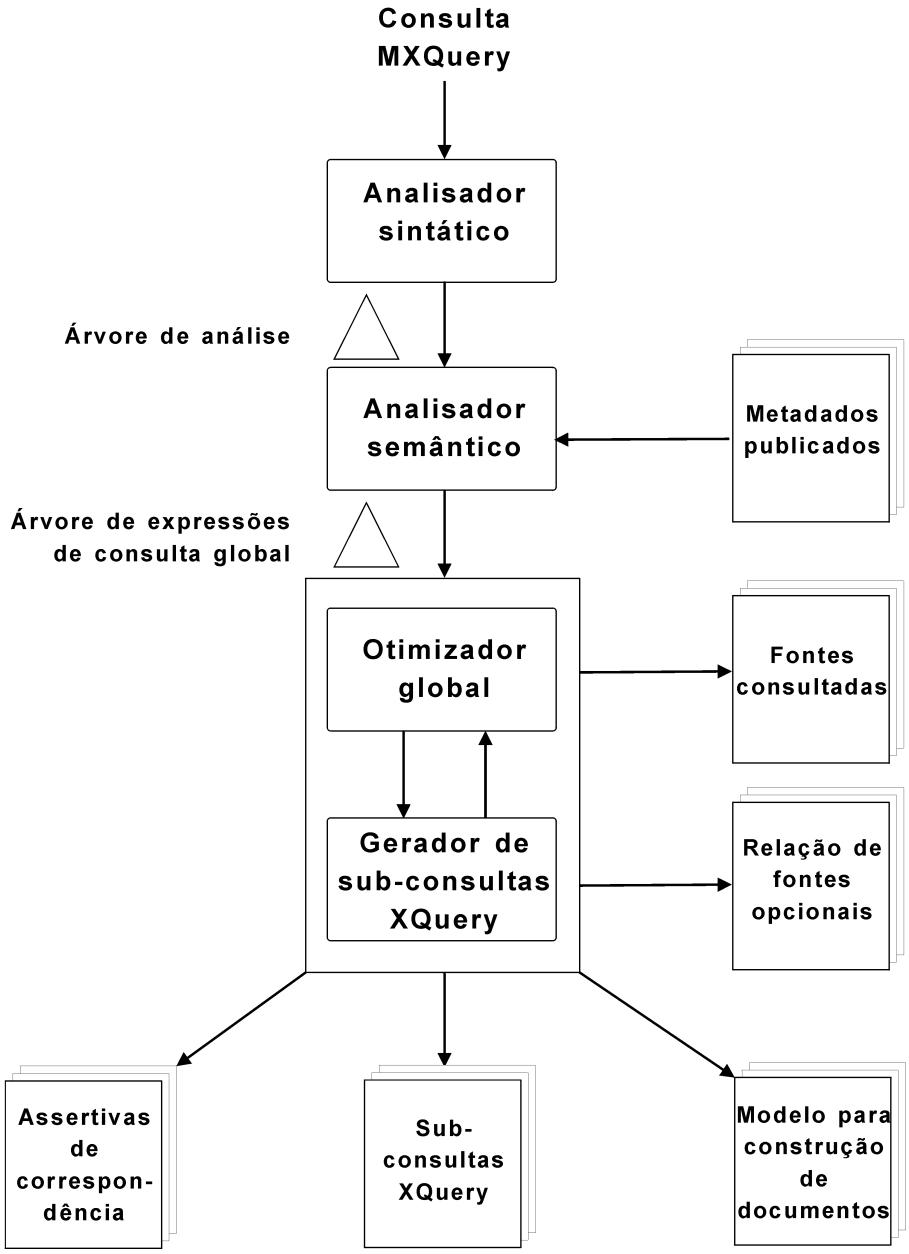

Figura 5.3. Produtos Gerados pelo Processador de Consultas (QP).

O primeiro faz a verificação sintática da consulta, onde uma árvore de análise obtida a partir da consulta original é comparada à gramática da linguagem MXQuery. Caso a consulta seja sintaticamente válida, o segundo componente verifica a existência dos objetos apontados na consulta no catálogo de metadados publicados. Finalmente, se todos os objetos existirem e forem declarados corretamente, é gerada a árvore de expressões de consulta global.

Um analisador semântico convencional deveria, também, verificar a correta utilização dos atributos de acordo com seus tipos de dados declarados. Entretanto, essa tarefa não é realizada no modelo proposto, pois a linguagem MXQuery utiliza a coerção de tipos para manipulação de dados. A coerção de tipos é uma característica bastante explorada por linguagens orientadas a dados semi-estruturados e foi discutida, mais detalhadamente, no capítulo 4.

O otimizador global tenta simplificar a consulta original avaliando diferentes planos lógicos de consulta equivalentes, que são obtidos através da aplicação de regras de transformação especificadas na álgebra que define a linguagem de consultas. Quanto melhor elaborada for a álgebra da linguagem, maior será a possibilidade de se obter um processo de otimização mais eficiente.

Em um banco de dados convencional, o processador de consultas deve determinar o plano físico a ser executado a partir do plano lógico selecionado durante a etapa de otimização de consultas, porém, ao integrar bancos de dados múltiplos, só se conhece a disponibilidade dos resultados parciais no momento da integração. Essa informação pode alterar a seleção do plano físico [Ozc97]. Além disso, o MDBS não possui todas as informações estatísticas necessárias a respeito das fontes de dados, conseqüentemente, é difícil optar pelo melhor plano físico antes de obter os resultados parciais.

O gerador de sub-consultas utiliza a árvore de expressões simplificada para produzir várias sub-consultas expressas em XQuery puro. Será produzida pelo menos uma sub-consulta XQuery para cada fonte de dados que conste na relação de fontes de dados consultadas. Como as próprias expressões XQuery podem ser simplificadas utilizando a especificação formal apresentada em [Dra03], existe um fluxo nos 2 (dois) sentidos entre o otimizador global e o gerador de sub-consultas.

Assim, a interação entre esses dois processos determina a geração de 5 (cinco) produtos:

1. Fontes consultadas. Uma lista de todas as fontes que foram referenciadas pela consulta global. Essa informação será utilizada pelo processador de localização e pelos processos de envio de consulta e recebimento de resultados;
2. Relação de fontes opcionais. A linguagem MXQuery permite a classificação das fontes de dados em obrigatórias e não obrigatórias (opcionais). Resumidamente, uma fonte de dados opcional indisponível não inviabiliza uma consulta, apenas altera parcialmente seu resultado. É produzida uma

lista de todas as fontes opcionais, que será consultada quando uma fonte de dados estiver indisponível. Caso essa fonte de dados faça parte dessa relação, tal evento não interromperá a consulta;

3. Sub-consultas XQuery. É gerada pelo menos uma sub-consulta XQuery simplificada para cada fonte de dados consultada;
4. Assertivas de correspondência. É produzida uma relação com todas as assertivas de correspondência entre os elementos das fontes de dados consultadas e os elementos do documento que representará o resultado global integrado [Spa92, Los98];
5. Modelo para construção de documentos. É gerado um documento que especifica as transformações/construções que devem ser realizadas sobre os elementos originais para a produção do documento resultado.

Antes de enviar as sub-consultas às fontes de dados correspondentes é necessário substituir a referência lógica utilizada para identificar a fonte de dados com a localização física da referida fonte. Essa tarefa é realizada pelo processador de localização que consulta o repositório MDBS, extrai a informação necessária e substitui as referências lógicas.

Finalmente, o QP envia as sub-consultas para as interfaces XML correspondentes e aguarda o sinal de recebimento. Se decorrido um intervalo de tempo $\triangle T$ e o sinal de reconhecimento não chegar, o QP reenvia a consulta. O QP continua tentando enviar a sub-consulta durante um intervalo de tempo $\triangle T'$, porém, decorrido esse intervalo de tempo e se, ainda assim, alguma interface XML não enviar o sinal de reconhecimento o QP executa a operação de descarte ou interrompe a consulta de acordo com os seguintes critérios:

1. Descartar. Caso a referida fonte de dados conste na lista de fontes de dados opcionais, todas as suas referências são excluídas. Assim, as regras de integração e de construção de elementos que referenciam a fonte de dados indisponível, são descartadas, bem como, a lista de fontes consultadas é atualizada. Além disso, a sub-consulta é eliminada e o processo de envio é interrompido.
2. Cancelar. Caso a fonte de dados não conste na lista de fonte de dados opcionais, o QP envia uma mensagem de erro para o QPp, envia mensagens de "*abort*" para as demais interfaces XML e descarta as tabelas temporárias.

5.2.3 Processador de Localização

Cada fonte de dados participante da comunidade MDBS reside em um local físico, muito provavelmente, distinto dos demais participantes. Por esse motivo, ao registrar uma fonte de dados por intermédio de sua interface XML, a localização física da referida fonte de dados é armazenada no repositório MDBS. Em seguida, cada fonte de dados é associada a um nome lógico que será referenciado nas consultas expressas em MXQuery. Portanto, o usuário utiliza apenas esse nome lógico para consultar as informações desejadas.

O processador de localização é responsável por substituir as referências lógicas utilizadas nas consultas pela localização física das fontes de dados. Para tanto, o LP verifica a lista de fontes consultadas para identificar no repositório MDBS a localização física de cada uma das fontes da lista. Após a geração das sub-consultas, as referências lógicas podem ser substituídas pela localização física das fontes de dados.

Entretanto, antes que esse processo seja realizado, o LP checa a disponibilidade das fontes de dados referenciadas. O processador de localização só altera as sub-consultas quando essas referenciam fontes de dados disponíveis. Essa informação é periodicamente atualizada pelo registrador de fontes de dados no repositório MDBS e, embora tal informação possa estar desatualizada no momento da consulta, esse processo pode evitar tráfego desnecessário pela rede.

5.2.4 Pós-processador de Consultas

O QPp recebe documentos XML como resultado de sub-consultas enviadas à XML–I. Cada documento recebido é associado à sub-consulta original até que todos os documentos resultado, pertencentes a uma consulta global *C*, sejam recebidos. Decorrido um intervalo de tempo $\triangle T$ após o recebimento do primeiro resultado parcial pertencente a *C* e, caso alguma resposta não chegue, o QPp consulta o repositório MDBS a fim de detectar se a referida fonte de dados está disponível.

Se a fonte de dados não estiver disponível, o QPp verifica a lista de fontes de dados opcionais e, caso a fonte de dados conste na lista, o QPp executa a operação de descarte de forma análoga a realizada pelo QP. Caso a fonte de dados indisponível não conste na lista de fontes opcionais, a consulta é interrompida, todos os resultados parciais recebidos são descartados, uma mensagem de *"abort"* é enviada para as fontes de dados

que ainda não responderam e uma mensagem de erro é enviada ao usuário através da QUI.

Se a fonte de dados que não respondeu estiver disponível, o QPp espera um intervalo de tempo $\triangle T'$. Após esse intervalo e se a resposta ainda não tiver chegado, o QPp pode finalizar a consulta (excluindo fonte de dados, se opcional, ou interrompendo a consulta *C*, se fonte de dados obrigatória) ou pode solicitar ao QP que reenvie as sub-consultas das fontes de dados que já responderam. Esse processo é necessário para que informações antigas não sejam integradas às informações mais recentes.

Após receber todos os resultados parciais referentes à consulta *C*, o pós-processador de consultas utilizará as assertivas de correspondência (seção 5.2.2) para montar o resultado global. Existem 2 (dois) processos embutidos na combinação dos resultados das sub-consultas:

- Integração. Esse processo especifica as operações que devem ser realizadas com o conjunto de dados. Por exemplo, pode ser necessário executar uma operação de junção entre dois conjuntos de dados distintos, como se tais conjuntos de dados representassem tabelas distintas em um SGBD relacional. Além disso, o QPp deve decidir qual algoritmo (plano físico) deve ser executado para realizar a referida operação. Finalmente, pode ser necessário resolver conflitos de integração utilizando as assertivas de correspondência extraídas da consulta *C*;
- Construção. Além de integrar os conjuntos de dados, pode ser necessário transformar os elementos de acordo com as regras de construção extraídas da consulta *C*. O processo de transformação produz um documento resultado diferente dos documentos originais.

Como a especificação do plano físico a ser adotado não foi realizada pelo otimizador global de consultas pelos motivos apontados na seção 5.2.2, essa tarefa cabe ao QPp. Uma última consideração a respeito do processo de otimização é que o processador de consultas e o pós-processador de consultas realizam a otimização de consultas apenas no nível da consulta global. As fontes de dados locais, quando possuem características de um Sistema Gerenciador de Banco de Dados, realizam um processo de otimização local onde é determinado o plano lógico e físico mais adequado para responder as sub-consultas propostas, como se essas fossem consultas locais enviadas por usuários locais.

5.2.5 Registrador de Fonte de Dados

Esse componente é encarregado de controlar a conexão e desconexão das fontes de dados participantes da comunidade. Esse processo é uma tarefa bastante complexa, pois a arquitetura utilizada deve preservar a autonomia de associação e, portanto, uma fonte de dados não informa à comunidade sua decisão de abandonar (ou interromper temporariamente) sua participação.

A primeira tarefa realizada pelo registrador de fonte de dados é receber uma solicitação de participação na comunidade. Isso é feito através do recebimento de um esquema (esquema de exportação) enviado pela interface XML. Esse esquema é definido através da linguagem XML *Schema* e contém todas as informações (metadados) que serão disponibilizadas a respeito da fonte de dados participante. A partir da publicação do esquema de uma fonte de dados participante é possível referenciar essa fonte de dados nas consultas MXQuery.

Várias informações são armazenadas durante o processo de publicação de uma fonte de dados, tais como: esquema disponibilizado, localização física e referência lógica para a fonte de dados.

Após a tarefa de publicação de uma fonte de dados, o registrador de fonte de dados executa periodicamente um processo de reconhecimento de fonte de dados. Esse processo tem por objetivo identificar mudanças no esquema publicado, além de detectar desconexões temporárias de uma fonte de dados participante. Essas informações são armazenadas no repositório MDBS e são, posteriormente, utilizadas pelo processador de localização e pelo pós-processador de consultas. Portanto, quanto menor for a periodicidade de execução do processo de reconhecimento mais precisa será a informação utilizada por esses componentes.

Ao contrário do processo de conexão de uma fonte de dados, o processo de desconexão não é formalizado. Conseqüentemente, o registrador de fonte de dados deve decidir quando excluir a definição de uma fonte de dados do repositório MDBS. Essa tarefa deve ser executada por uma questão de performance, pois a arquitetura de integração acessa (ou tenta acessar) qualquer fonte de dados que esteja registrada no repositório MDBS durante o processo de reconhecimento. Por isso, caso uma fonte de dados esteja constantemente indisponível, tal fonte de dados é candidata a ter sua definição retirada do repositório MDBS.

5.2.6 Repositório MDBS

Todas as informações recolhidas pelo registrador de fonte de dados são armazenadas pelo repositório MDBS. Assim, o MR mantém as informações sobre as fontes de dados publicadas e sua localização física, tais informações são periodicamente atualizadas para refletir as modificações ocorridas nas fontes de dados participantes.

Além de armazenar informações sobre as fontes de dados, é possível utilizar o repositório MDBS para manter informações sobre a comunidade MDBS. É possível, também, armazenar consultas MXQuery pré-formatadas, ou ainda, restrições de acesso global aos recursos disponibilizados pela comunidade. Dessa forma, podem-se representar esquemas externos, que são acessados como se não houvesse múltiplas fontes de dados (ver figura 2.7, capítulo 2). Naturalmente, os esquemas externos são definidos sobre uma versão das fontes de dados participantes, caso tais fontes de dados sofram modificações em seus esquemas isso pode tornar os esquemas externos incorretos.

5.2.7 Interface XML

A interface XML dever representar o esquema (ou parte do esquema) de uma fonte de dados utilizando uma linguagem de definição de esquemas XML (XML *Schema*). Após realizar esse mapeamento, o esquema obtido é enviado ao SR que será responsável por publicá-lo, armazenando suas definições no repositório MDBS. O esquema XML publicado é uma referência aos dados existentes nas fontes de dados, portanto, fornece uma descrição dos dados disponibilizados.

Segundo a estratégia adotada, a representação dos dados das fontes de dados em XML é apenas virtual, ou seja, nenhum documento XML representando uma fonte de dados é armazenado em um repositório de dados. A geração de documentos XML só se dá como resultado a uma consulta proposta, por exemplo, após a transformação de uma relação resultante obtida a partir de consulta enviada a uma fonte de dados relacional. Conseqüentemente, caso uma consulta MXQuery solicite que todos os elementos (objetos ou tuplas) de uma determinada fonte de dados sejam retornados, então, apenas nesse momento, um documento XML representando tal fonte de dados é produzido como resposta à consulta proposta.

De acordo com a figura 5.1, pode-se identificar um componente XML–I exclusivo associado a cada fonte de dados local, portanto, a implementação desse componente depende da fonte de dados a qual a interface XML está associada. Até mesmo fontes de dados que utilizam o mesmo modelo de dados para representar seus esquemas são mapeadas por interfaces XML distintas.

5.3 Técnicas de Mapeamento

Nesta seção, são descritas as técnicas para mapear dados nativos em XML e para mapear consultas XQuery em consultas das linguagens nativas dos bancos de dados locais que compõem um MDBS.

Para exemplificar as transformações realizadas pela XML–I, será utilizada uma fonte de dados local baseada em um sistema gerenciador de banco de dados relacional. O esquema disponibilizado pela fonte de dados está inicialmente expresso através de um modelo de dados relacional que é convertido para um documento XML *Schema*. As consultas expressas em XQuery enviadas à interface XML são convertidas para a linguagem nativa do DBMS, ou seja, SQL. Finalmente, a relação resultante da consulta SQL enviada pela fonte de dados é transformada e apresentada como um documento XML.

5.3.1 Gerando o XML *Schema*

A figura 5.4 apresenta uma série de instruções expressas através da linguagem SQL, cuja finalidade dessas instruções é definir o esquema do banco de dados “biblio”. Na figura 5.4, são definidas 4 tabelas, 4 chaves primárias e 3 chaves estrangeiras. Existem inúmeras formas de representar essas mesmas informações através do XML *Schema*, onde uma possível representação é apresentada na figura 5.5. Essa tarefa é executada por um dos componentes da XML–I que é responsável pela definição de mapeamento.

Inicialmente, deve-se definir a partir de quais objetos (tabela ou visão) será obtido o XML *Schema*. Embora não seja obrigatória, a utilização de visões confere maior flexibilidade à definição do esquema a ser publicado. Isso porque é possível implementar, através de visões, restrições de atributos, condições de seleção e junções e uniões entre relações distintas.

Além disso, a criação de visões para implementar a técnica proposta não viola a autonomia das fontes de dados, pois essa tarefa é executada como um processo local. A consulta SQL apresentada na figura 5.5 cria uma visão “livro” contendo:

```
use biblio

create table editoras (
codeditora      smallint not null constraint pk_editora1 primary key,
nome            varchar(35) not null,
endereco        varchar(40) null,
fone            varchar(20) null
)

create table autores (
codautor        smallint not null constraint pk_autor1 primary key,
nome            varchar(35) not null,
fone            varchar(20) null
)

create table livros (
isbn            char(13) not null constraint pk_livro1 primary key,
idioma          varchar(20) not null,
ano             char(4) not null,
pais            varchar(20) not null,
titulo          varchar(40) not null,
subtitulo       varchar(30) null,
editora         smallint not null constraint fk_livro2 references editoras (codeditora),
moeda           varchar(15) not null,
preco           decimal(7,2) null
)

create table obras (
livro           char(13) not null constraint fk_obras1 references livros (isbn),
autor           smallint not null constraint fk_obras2 references autores (codautor)
)

alter table obras
add constraint pk_obras1 primary key (livro,autor)
```

Figura 5.4. Definição de um Esquema de Banco de Dados Utilizando SQL.

1. os atributos “isbn”, “titulo”, “ano” e “preço” originados da relação “livros”;
2. o atributo “editora” obtido da relação “editoras” e
3. o atributo “autor” originado da relação “autores”.

Embora seja importante, a definição do objeto (tabela ou visão) não é suficiente para gerar o XML *Schema*. É necessário analisar os metadados associados às relações (visões ou tabelas) exportadas, onde se observam os seguintes aspectos: chave primária, chaves estrangeiras, atributos obrigatórios, tipos de dados e número de ocorrências distintas de cada atributo em relação à chave primária (quando houver).

Com base nessas informações pode-se montar o XML *Schema* correspondente aos dados a serem publicados. Cada documento XML possui um elemento raiz cujo nome é extraído do nome do banco de dados consultado. Assim, tem-se a linha **“<xsd:element**

name=“biblio”>” definindo que o banco de dados “biblio” origina um elemento raiz de mesmo nome. O elemento raiz é composto por um único elemento do tipo complexo que é definido a partir das tabelas (visões) que comporão o esquema publicado.

```
create view livro as (
select isbn,titulo,ano,preco,editoras.nome as editora,autores.nome as autor
from obras
join livros on livro=isbn
left outer join editoras on editora=codeditora
join autores on autor=codautor
)

       ---------------------------------------------------------------------

<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
 <xsd:element name=“biblio”>
  <xsd:complexType>
   <xsd:element name=“livro” minOccurs=“0” maxOccurs=“unbounded”>
     <xsd:complexType>
       <xsd:attribute name=“isbn”     type=“xsd:ID” use=“required”/>
       <xsd:element  name=“titulo”    type=“xsd:string”/>
       <xsd:element  name=“ano”       type=“xsd:string”/>
       <xsd:element  name=“preco”     type=“xsd:decimal”
            minOccurs=“0” maxOccurs=“1”/>
       <xsd:element  name=“editora”   type=“xsd:string”
            minOccurs=“0” maxOccurs=“1”/>
       <xsd:element  name=“autor”     type=“xsd:string”
            maxOccurs=“unbounded”/>
     </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figura 5.5. Possível XML *Schema* Obtido a Partir de uma Consulta SQL.

De acordo com a consulta SQL apresentada na figura 5.5, o tipo complexo que compõe o elemento raiz é composto de um único elemento “livro” representando a única visão definida no momento. Na definição do elemento livro, deve-se informar o número mínimo e máximo de linhas pertencentes à visão “livro”, portanto tem-se **“<xsd:element name=“livro” minOccurs=“0” maxOccurs=“unbounded”>”** representando a visão “livro”. O elemento “livro”, por sua vez, é composto por um elemento complexo, que engloba os atributos expressos na visão “livro”.

A linha **“<xsd:attribute name=“isbn” type=“xsd:ID” use=“required”/>”** representa o atributo chave “isbn”, onde é especificado o tipo de atributo como “xsd:ID” e de ocorrência obrigatória. O tipo de atributo “xsd:ID” especifica que só pode existir um único elemento identificado com o valor atribuído ao atributo “isbn”, portanto esse atributo serve como chave (OID) de identificação do elemento.

[1]	Schema	::=	RootClause
[2]	RootClause	::=	“<xsd:element name=\“ ” StringLiteral “\”>” “<xsd:complexType>” RelatClause+ “</xsd:complexType>” “</xsd:element>”
[3]	RelatClause	::=	“<xsd:element name=\“ ” StringLiteral “\” minOccurs=\“0\” maxOccurs=\“unbounded\”>” “<xsd:complexType>” AttIdClause? (AttRfOpClause | AttRfRqClause | AttRfsOpClause | AttRfsRqClause | ElmOpLmClause | ElmRqLmClause | ElmOpUnlClause | ElmRqUnlClause)* “</xsd:complexType>” “</xsd:element>”
[4]	AttIdClause	::=	“<xsd:attribute name=\“ ” StringLiteral “\” type=\“xsd:ID\” use=\“required\”/>”
[5]	AttRfOpClause	::=	“<xsd:attribute name=\“ ” StringLiteral “\” type=\“xsd:IDREF\” use=\“optional\”/>”
[6]	AttRfRqClause	::=	“<xsd:attribute name=\“ ” StringLiteral “\” type=\“xsd:IDREF\” use=\“required\”/>”
[7]	AttRfsOpClause	::=	“<xsd:attribute name=\“ ” StringLiteral “\” type=\“xsd:IDREFS\” use=\“optional\”/>”
[8]	AttRfsRqClause	::=	“<xsd:attribute name=\“ ” StringLiteral “\” type=\“xsd:IDREFS\” use=\“required\”/>”
[9]	ElmOpLmClause	::=	“<xsd:element name=\“ ” StringLiteral “\” type=\“ ” TypeA “\” minOccurs=\“0\” maxOccurs=\“1\”/>”
[10]	ElmRqLmClause	::=	“<xsd:element name=\“ ” StringLiteral “\” type=\“ ” TypeA “\” />”
[11]	ElmOpUnlClause	::=	“<xsd:element name=\“ ” StringLiteral “\” type=\“ ” TypeA “\” minOccurs=\“0\” maxOccurs=\“unbounded\”/>”
[12]	ElmRqUnlClause	::=	“<xsd:element name=\“ ” StringLiteral “\” type=\“ ” TypeA “\” maxOccurs=\“unbounded\”/>”
[13]	StringLiteral	::=	[A-Za-z] ([A-Za-z0-9._] | '-')*
[14]	TypeA	::=	(“xsd:string” | “xsd:decimal” | “xsd:integer” | “xsd:boolean”)

Tabela 5.1. EBNF para Construção do Esquema XML.

Os demais atributos são representados por elementos, que possuem uma ocorrência mínima de 0 (não são obrigatórios – *null*) ou 1 (são obrigatórios – *not null*). A ocorrência máxima para esses elementos será 1, que significa que para cada atributo “isbn” diferente só pode existir um único elemento associado ou ilimitado (*unbounded*), ou seja, para cada atributo “isbn” podem existir vários elementos associados. Assim, a linha **“<xsd:element name=“preco” type=“xsd:decimal” minOccurs=“0” maxOccurs=“1”/>”** define o elemento “preço” como não obrigatório e único para cada atributo “isbn” diferente. Além disso, ele é classificado como sendo do tipo decimal. Em contrapartida, a linha **“<xsd:element name=“autor” type=“xsd:string” maxOccurs=“unbounded”/>”** especifica que o elemento “autor” é obrigatório (omissão de ocorrência mínima) e pode ocorrer inúmeras vezes para o mesmo elemento “isbn”.

5.3.1.1 Regras para Obtenção do XML *Schema*

A tabela 5.1 apresenta um modelo de especificação de um XML *Schema* utilizando a notação EBNF. Esse modelo foi utilizado para especificar o XML *Schema* apresentado na figura 5.5. De acordo com o modelo, um documento XML *Schema* conterá um elemento raiz descrito em RootClause que é composto por um ou mais sub-elementos descritos por RelatClause. Por sua vez, cada elemento do tipo RelatCaluse é composto por 1 ou mais elementos que podem ser de quaisquer dos tipos identificados na tabela 5.1 com os números de 4 a 12.

R_1: SE RootClause ENTÃO StringLiteral = nomeFonte;

R_2: SE RelatClause ENTÃO StringLiteral = nomeRelacao;

R_3: SE atributo é chave de Relação
ENTÃO tipo do elemento = AttIdClause e StringLiteral = nomeAtributo;

R_4: SE atributo é chave estrangeira de Relação e pode ser nulo e correspondência (1:1) em relação à chave
ENTÃO tipo do elemento = AttRfOpClause e StringLiteral = nomeAtributo;

R_5: SE atributo é chave estrangeira de Relação e não pode ser nulo e correspondência (1:1) em relação à chave
ENTÃO tipo do elemento = AttRfRqClause e StringLiteral = nomeAtributo;

R_6: SE atributo é chave estrangeira de Relação e pode ser nulo e correspondência (N:1) em relação à chave
ENTÃO tipo do elemento = AttRfsOpClause e StringLiteral = nomeAtributo;

R_7: SE atributo é chave estrangeira de Relação e não pode ser nulo e correspondência (N:1) em relação à chave
ENTÃO tipo do elemento = AttRfsRqClause e StringLiteral = nomeAtributo;

R_8: SE atributo pode ser nulo e correspondência (1:1) em relação à chave
ENTÃO tipo do elemento = ElmOpLmClause e StringLiteral = nomeAtributo e
TypeA = tipoAtributo;

R_9: SE atributo não pode ser nulo e correspondência (1:1) em relação à chave
ENTÃO tipo do elemento = ElmRqLmClause e StringLiteral = nomeAtributo e
TypeA = tipoAtributo;

R_{10}: SE atributo pode ser nulo e correspondência (N:1) em relação à chave
ENTÃO tipo do elemento = ElmOpUnlClause e StringLiteral = nomeAtributo e
TypeA = tipoAtributo;

R_{11}: SE atributo não pode ser nulo e correspondência (N:1) em relação à chave
ENTÃO tipo do elemento = ElmRqUnlClause e StringLiteral = nomeAtributo e
TypeA = tipoAtributo;

Figura 5.6. Regras para Construção do XML *Schema*.

A figura 5.6 apresenta um conjunto de regras para aplicação do modelo de construção do XML *Schema* descrito na tabela 5.1. No conjunto de regras apresentado, as regras 1 e 2 sempre devem ser executadas para identificar a fonte de dados e a

relação (tabela ou visão) consultadas. As regras de 3 a 11 serão executadas de acordo com a condição especificada na regra e as condições avaliadas referem-se às características do atributo avaliado e sua correspondência em relação à chave.

Um atributo possui correspondência (1:1) em relação à chave quando para um determinado valor do atributo chave só existe um único valor do atributo avaliado. Em contrapartida, uma correspondência (N:1) implica que para cada valor do atributo chave podem existir *n* valores dos atributos avaliados.

Toda a informação requerida para aplicação das regras pode, facilmente, ser extraída através de instruções SQL que consultem as tabelas de sistema (ou catálogo) onde são armazenadas as informações sobre os bancos de dados e as tabelas de usuário. Por exemplo, o MS SQL Server armazena tais informações nas tabelas de sistema *"sysobjects"*, *"syscolumns"*, *"sysindexes"* e *"sysforeignkeys"*.

A figura 5.7 apresenta um outro exemplo da obtenção de um documento de validação expresso através de XML *Schema* que foi obtido a partir de duas visões definidas em SQL, também apresentadas na figura 5.7. O documento XML *Schema* resultante é obtido através da aplicação das regras descritas na figura 5.6.

```
create view livro_ as (
select isbn,titulo,ano,preco,editoras.nome as editora,autor
from obras
join livros on livro=isbn
left outer join editoras on editora=codeditora
where ano>='1997')

create view autor as (select codautor as autor,nome from autores)

       ------------------------------------------------------------------

<xsd:schema xmlns:xsd:"http://www.w3.org/1999/XMLSchema">
 <xsd:element name="biblio">
  <xsd:complexType>
   <xsd:element name="livro_" minOccurs="0" maxOccurs="unbounded">
     <xsd:complexType>
       <xsd:attribute name="isbn"     type="xsd:ID" use="required"/>
       <xsd:attribute name="autor"    type="xsd:IDREFS"
            use="required"/>
                  . . .
     </xsd:complexType>
   </xsd:element>
   <xsd:element name="autor" minOccurs="0" maxOccurs="unbounded">
     <xsd:complexType>
       <xsd:attribute name="autor"    type="xsd:ID" use="required"/>
       <xsd:element name="nome"    type="xsd:string"/>
     </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figura 5.7. Possível XML *Schema* Obtido a Partir de uma Consulta SQL.

5.3.2 Transformando Consultas XQuery em SQL

Após a publicação do esquema da fonte de dados, esta estará apta a receber consultas enviadas pela arquitetura MDBS. Entretanto, as consultas recebidas pela interface XML estão expressas em uma linguagem para consulta de documentos XML, mas precisamente XQuery. Ainda utilizando como exemplo uma fonte de dados baseada em um SGBD relacional, é necessário transformar a consulta expressa em XQuery para uma consulta expressa em SQL.

```
<resultado> {
  for $d1 in document("d1.xml")/biblio
  return
    for $p in (1 to count($d1/livro))
    return
      <livro>
        $d1/livro[$p]/titulo,
        $d1/livro[$p]/autor
      </livro>
} </resultado>

  ------------------------------------------

use biblio
select isbn as idobj, titulo, autor from livro
```

Figura 5.8. Transformando uma Consulta XQuery em SQL.

A figura 5.8 apresenta uma consulta expressa em XQuery e uma versão dessa mesma consulta expressa em SQL. Vale ressaltar que a própria consulta XQuery foi obtida após o processo de produção de sub-consultas, onde uma consulta originalmente expressa em MXQuery gera *n* sub-consultas expressas em XQuery. Cada sub-consulta opera sobre uma única fonte de dados, portanto não necessita possuir recursos de manipulação de múltiplas fontes de dados. Por esse motivo, essa sub-consulta pode ser expressa através de XQuery puro.

A consulta apresentada na figura 5.8 baseou-se no XML *Schema* apresentado na figura 5.5. Essa consulta possui 3 (três) informações principais:

1. Ponto de entrada na floresta. A linha **"for $d1 in document("d1.xml")/biblio"** especifica esse ponto de entrada, onde é identificada a fonte de dados através de uma referência lógica. O processador de localização substituirá essa referência lógica pela localização física da fonte de dados. Além disso, é indicado o nó raiz do documento XML virtual. De acordo com as regras de

obtenção do XML *Schema* apresentadas na seção 5.3.1.1, o nó raiz é o nome do banco de dados consultado;

2. Construção do elemento "agrupador". Geralmente, uma consulta XQuery possui pelo menos um elemento que agrupa outros sub-elementos, que se referem a um mesmo objeto. Assim, a linha **"for $p in (1 to count($d1/livro))"** atribui a uma variável "$p" o resultado de uma iteração sobre elementos "livro" distintos, onde será originado um elemento "livro" para cada livro distinto encontrado;
3. Construção de sub-elementos. Dentro de cada elemento "agrupador", podem existir vários sub-elementos que especificam as características desse elemento "agrupador". A expressão **"$d1/livro[$p]/titulo"** representa um desses sub-elementos.

Uma importante característica das consultas XQuery utilizadas é que tais consultas sempre serão baseadas em documentos XML de três níveis de aninhamento. O primeiro determinando o banco de dados consultado, o segundo especificando a relação (tabela ou visão) e o terceiro identificando os atributos dessa relação. Conseqüentemente, pode-se determinar a consulta SQL através da identificação desses três níveis de aninhamento dentro do documento XML.

O ponto de entrada na floresta especifica o banco de dados consultado, assim é gerada a instrução SQL **"use biblio"**. Quando for especificado um elemento "agrupador", significa que 1 ou mais tuplas da visão consultada podem gerar um mesmo elemento "agrupador". A indicação dos sub-elementos determina quais são os atributos que devem constar na consulta SQL. A consulta **"select isbn as idobj, titulo, autor from livro"** é obtida utilizando os atributos determinados pelos sub-elementos da consulta XQuery e a cláusula "FROM" é obtida a partir da identificação do segundo nível de aninhamento. Além disso, foi introduzido um atributo extra (isbn as idobj) que tem a função de identificar as tuplas da visão "livro". Essa identificação é necessária para a construção do elemento "agrupador".

Finalmente, uma consulta XQuery pode possuir expressões condicionais que são representadas através da cláusula "WHERE". As expressões condicionais presentes nessas consultas XQuery possuem uma sintaxe bastante similar a sintaxe da cláusula "WHERE" presente em instruções SQL.

Apenas dois aspectos devem ser analisados no momento da transcrição da expressão condicional. O primeiro é a substituição das expressões de caminho pelo

nome do atributo. Para esse fim deve-se lembrar que o nome do atributo coincide com o nome do elemento em seu terceiro nível de aninhamento. O segundo aspecto se refere à adequação dos tipos de dados, que é um problema decorrente da utilização da coerção de tipos para comparar elementos e valores. A solução desse problema passa pela determinação do tipo de dado de cada atributo armazenado no XML *Schema* e, em seguida, devem-se transformar os valores comparados de acordo com o tipo de dado esperado. Ainda em relação a esse problema, pode ser necessária a utilização de funções de conversão quando a expressão condicional utilizar a comparação entre dois atributos de tipos de dados distintos.

Embora o mapeamento do esquema do banco de dados relacional para XML *Schema* padronize o formato das consultas XQuery e, conseqüentemente, facilite a transformação dessas consultas em consultas SQL, algumas informações adicionais são necessárias para complementar a transformação. O tradutor de consultas deve obter informações sobre o esquema relacional, onde devem ser identificados os tipos de dados dos atributos que compõem a consulta. Essa informação é utilizada para solucionar o problema da coerção de tipos e deve ser obtida através do componente de definição de mapeamento.

```
<resultado>
  <livro idelem="idobj">
    <titulo>$att2</titulo>
    <autor>$att3</autor>
  </livro>
</resultado>
```

Figura 5.9. Documento Modelo para Apresentação de Resultado.

O tradutor de consultas XQuery deve ainda criar um documento modelo para formatação do resultado. Esse documento modelo substitui a necessidade de armazenar e analisar a consulta XQuery no momento da geração do documento XML resultante. A figura 5.9 apresenta esse documento modelo, onde podem ser identificadas duas variáveis "$att2" e "$att3". Essas variáveis serão substituídas pelos valores das colunas 2 e 3 da consulta SQL respectivamente. Além disso, o documento utiliza o atributo "idelem" para especificar que o elemento livro é um elemento "agrupador" e que o valor do atributo "idobj" identifica unicamente o elemento livro.

O mapeamento formal de uma consulta expressa em XQuery para uma consulta expressa em SQL é uma tarefa muito mais complexa, porém foi apresentado um exemplo simples que explora as características das regras utilizadas para construção do

XML *Schema*. A intenção desse trabalho é apresentar uma forma viável de elaborar essas transformações, portanto, não tenciona esgotar o assunto sobre conversão de consultas expressas em XQuery para SQL ou de um conjunto de tuplas para documentos XML.

5.3.3 Gerando um Documento XML Resultado

Uma consulta SQL, ao ser submetida a um banco de dados relacional, resulta em um conjunto de tuplas; porém, esta não é a resposta aguardada pelo QPp. Conseqüentemente, é necessário transformar esse conjunto de tuplas no formato aguardado pela arquitetura de integração, em outras palavras, é preciso gerar um documento XML, que seja compatível com a consulta XQuery enviada.

A figura 5.10 mostra um possível resultado da consulta SQL apresentada na figura 5.8 onde é possível identificar 10 tuplas distintas. Entretanto, a consulta XQuery original determina que os sub-elementos (atributos titulo e autor) pertencentes ao mesmo livro sejam representados como sub-elementos de um mesmo elemento “livro”. Essa determinação exige que os elementos “livros” sejam identificados unicamente, por esse motivo o atributo (isbn as idobj) foi introduzido na consulta SQL. As tuplas que possuírem o mesmo valor para o atributo “idobj” devem ser representadas sob um único elemento “livro”. O atributo “idobj” é temporário e não deve constar no documento XML resultante.

A transformação do conjunto de tuplas apresentado na figura 5.10 deve originar o documento XML representado na mesma figura. Utilizando o documento modelo para formatação de resultado (figura 5.9) é criado um elemento raiz “resultado” que engloba 4 (quatro) elementos “livro”. É gerado um elemento livro para cada valor do atributo “idobj” diferente e essa determinação é especificada pelo atributo “idelem” do documento modelo.

Cada elemento “livro” é composto por sub-elementos “titulo” e “autor”, onde os valores desses sub-elementos são obtidos através da substituição dos valores dos atributos de acordo com sua posição na consulta SQL. Serão gerados tantos sub-elementos “titulo” e “autor” quantos forem os atributos “titulo” e “autor” que tiverem seus valores diferentes para o mesmo valor do atributo “idobj”. Segundo esse critério, o elemento “livro”, cujo valor do atributo “idobj” é igual a 013, é formado por um único sub-elemento “titulo” e por dois sub-elementos “autor”.

idobj	titulo	autor
013	A First Course in Database Systems	Ullman
013	A First Course in Database Systems	Widom
053	Fundamentals of Database Systems	Elmasri
053	Fundamentals of Database Systems	Navathe
346	Sistema de Banco de Dados	Silberschatz
346	Sistema de Banco de Dados	Korth
346	Sistema de Banco de Dados	Sudarshan
352	Implementação de Sistemas de BDs	Ullman
352	Implementação de Sistemas de BDs	Widom
352	Implementação de Sistemas de BDs	Garcia-Molina

```
<resultado>
  <livro>
    <titulo>A First Course in Database Systems</titulo>
    <autor>Ullman</autor>
    <autor>Widom</autor>
  </livro>
  <livro>
    <titulo>Fundamentals of Database Systems</titulo>
    <autor>Elmasri</autor>
    <autor>Navathe</autor>
  </livro>
  <livro>
    <titulo>Sistema de Banco de Dados</titulo>
    <autor>Silberschatz</autor>
    <autor>Korth</autor>
    <autor>Sudarshan</autor>
  </livro>
  <livro>
    <titulo>Implementação de Sistemas de BDs</titulo>
    <autor>Ullman</autor>
    <autor>Widom</autor>
    <autor>Garcia-Molina</autor>
  </livro>
</resultado>
```

Figura 5.10. Resultado de uma Consulta SQL e o Documento XML.

5.4 Documentos Originados do Processamento de Consultas

A seção 5.2.2 listou uma série de documentos (ou produtos) que devem ser gerados a partir de uma consulta global expressa em MXQuery. A idéia por trás dessa estratégia é separar as várias expressões da consulta proposta, de acordo com suas funções. Assim sendo, a cláusula “ALIAS” lista todas as fontes de dados consultadas e a cláusula “NULL” especifica quais dessas fontes são opcionais.

As assertivas de correspondência são obtidas principalmente a partir das especificações contidas na cláusula “EQUAL”. As cláusulas “WHERE” e “KEY” também contribuem para a definição das assertivas. As regras de construção e transformação de elementos são extraídas das cláusulas “EACH”, “EQUAL”, “FULL” e “HIDE”. Além disso, as instruções especificadas nas cláusulas “WHERE” e “RETURN” também devem ser consideradas para esse fim.

```
EACH $p
  ALIAS document("d1.xml")/bib $d1
  ALIAS document("d3.xml")/lib $d3 NULL
  EQUAL $d1/livro $d3/book IS $p/livro
  EQUAL $d1/livro/@isbn $d3/book/isbn IS $p/livro/@isbn KEY
  EQUAL $d3/book/title IS $p/livro/titulo
  EQUAL $d3/book/author IS $p/livro/autor
  EQUAL dolartoreal($d3/book/price) IS $p/livro/preco
  EQUAL $d3/book/year IS $p/livro/ano
WHERE $p/livro/ano > "1999"
RETURN
  <biblioteca>
    $p
  </biblioteca>
```

Figura 5.11. Exemplo de uma Consulta MXQuery *C*.

Extrair essas informações da consulta MXQuery possibilita a geração de sub-consultas XQuery menos complexas e que se concentram em obter a informação necessária para, posteriormente, ser manipulada e agrupada pelos componentes da arquitetura MDBS. Nesse sentido, uma sub-consulta XQuery não deve possuir nenhuma informação de manipulação de elementos XML, portanto, a aparência de qualquer sub-consulta XQuery é bastante semelhante, mesmo que o resultado final da consulta MXQuery global seja completamente diferente.

A figura 5.11 apresenta uma consulta MXQuery *C,* da qual devem ser extraídos os documentos referidos na seção 5.2.2. As seções seguintes baseiam-se nessa consulta para determinar os documentos gerados pelo processamento da consulta *C*.

5.4.1 Fontes de Dados Consultadas

A figura 5.12 apresenta o documento XML que identifica as fontes de dados envolvidas na consulta *C*. Cada consulta submetida à arquitetura de integração recebe um identificador único. Esse identificador é representado no elemento “consulta” através do atributo “id”. Além disso, o elemento “consulta” possui um atributo “var”, que

especifica a variável (ou variáveis) que identifica o documento resultado obtido após o processo de integração.

```
<consulta id="000001" var="$p">
   <fonte>
      <logico>document("d1.xml")/bib</logico>
      <alias>$d1</alias>
      <uri>http://www.nomesite.com.br/dados.xml</uri>
   </fonte>
   <fonte>
      <logico>document("d3.xml")/lib</logico>
      <alias>$d3</alias>
      <uri>http://www.outrosite.com/outrosdados.html</uri>
   </fonte>
</consulta>
```

Figura 5.12. Fontes de Dados Consultadas.

O elemento "raiz" é composto por um ou mais sub-elementos "fonte". Deve existir um sub-elemento fonte para cada declaração "ALIAS" especificada na consulta global *C*. Por sua vez, cada elemento "fonte" é composto por 3 (três) tipos distintos de sub-elementos:

1. sub-elemento "logico". Um sub-elemento "logico" que especifica o ponto de entrada na floresta do documento XML. Essa é uma referência ao nome lógico armazenado no repositório MDBS;
2. sub-elemento "alias". Um ou mais sub-elementos "alias" que especifica o nome da variável (ou variáveis) associada ao documento XML. Essa variável é utilizada nas demais cláusulas da consulta *C*;
3. sub-elemento "uri". Um sub-elemento "uri" que identifica a localização física da fonte de dados consultada. Esse endereço pode apontar para um documento XML ou o IP de um DBMS relacional ou qualquer outro endereço que seja tratado pela interface XML associada à fonte de dados. Essa informação é registrada pelo Processador de Localização.

5.4.2 Fontes de Dados Opcionais

A figura 5.13 apresenta o documento XML que identifica as fontes de dados opcionais envolvidas na consulta *C*. Essa informação poderia ter sido representada através de um atributo no documento de fontes de dados consultadas, porém isso não foi feito por questão de clareza. Quando uma fonte de dados consultada estiver indisponível o

documento de fontes de dados opcionais é consultado. Caso essa fonte de dados, representada por sua variável, esteja relacionada no referido documento, as referências a essa fonte de dados são excluídas de todos os documentos gerados no processamento de consultas.

```
<consulta id="000001">
   <fonteopicional>$d3</fonteopicional>
</consulta>
```

Figura 5.13. Fontes de Dados Opcionais.

O elemento raiz "consulta" possui um atributo "id", que identifica a consulta global *C*. Além disso, serão representados tantos sub-elementos "fonteopcional" quantas forem as fontes de dados declaradas na cláusula "ALIAS" e que utilizaram a cláusula "NULL" nessa mesma declaração.

5.4.3 Assertivas de Correspondência

As assertivas de correspondência são tipos especiais de restrições de integridade que estabelecem correspondência semântica entre componentes de esquemas distintos [Spa92, Los98]. Portanto, pode-se utilizar as assertivas de correspondência para estabelecer regras de mapeamento entre elementos de esquemas distintos e os elementos resultantes de uma consulta de integração (ver seção 5.2.2). As assertivas de correspondência estão classificadas em 4 (quatro) grupos [Los98]:

- Assertivas de Correspondência de Tipos;
- Assertivas de Correspondência de Atributos;
- Assertivas de Correspondência de Caminhos;
- Assertivas de Dependência Existencial.

A tabela 5.2 apresenta as assertivas de correspondência extraídas da consulta *C*, onde pode ser identificada uma assertiva de correspondência de tipos AC_1 que especifica a classe (documento) "$p" como o resultado de uma fusão entre as classes "$d1" e "$d3". Inicialmente, a assertiva de correspondência que representa uma classe de fusão foi definida em [Peq00], porém, naquele trabalho, a identificação de objetos equivalentes era feita automaticamente através dos identificadores de objetos. Neste trabalho, optou-se por explicitar a propriedade (ou elemento) que deve servir como identificador único do objeto.

AC_1:	$p	≡	F ($d1, $d3) <$d1/livro/@isbn, $d3/book/isbn>
AC_2:	$p/livro	≡	$d1/livro
AC_3:	$p/livro	≡	$d3/book
AC_4:	$p/livro/@isbn	≡	$d1/livro/@isbn
AC_5:	$p/livro/@isbn	≡	$d3/book/isbn
AC_6:	$p/livro/titulo	≡	$d1/livro/titulo
AC_7:	$p/livro/titulo	≡	$d3/book/title
AC_8:	$p/livro/autor	≡	$d1/livro/autor
AC_9:	$p/livro/autor	≡	$d3/book/author
AC_{10}:	$p/livro/preço	≡	$d1/livro/preco
AC_{11}:	$p/livro/preço	≡	ϒ ◦ $d3/book/price, onde ϒ: dolartoreal($d3/book/price) → $p/livro/preco
AC_{12}:	$p/livro/ano	≡	$d1/livro/ano
AC_{13}:	$p/livro/ano	≡	$d3/book/year

Tabela 5.2. Assertivas de Correspondência.

As demais assertivas são definidas como assertivas de correspondência de caminho. Embora em [Los98] as assertivas de caminho tenham sido definidas como caminhos de derivação de atributos, o conceito de expressões de caminho é similar ao conceito de caminho de derivação de atributos. Finalmente, a assertiva de correspondência AC_{11} define o mapeamento entre atributos com domínios diferentes, conseqüentemente, foi necessária a utilização da função de mapeamento de domínio ϒ.

5.4.4 Modelo para Construção de Documentos

A figura 5.14 apresenta o documento XML que especifica o modelo para construção do resultado da consulta *C*. O elemento raiz “consulta” possui um atributo “id” que identifica a consulta global *C*. Cada documento modelo deve possuir um elemento “elementoraiz”, um ou mais elementos “elemento” e um ou mais elementos “atributo”. Além disso, o elemento “elemento” pode possuir sub-elementos do tipo “elemento” ou “atributo”.

O conteúdo texto dos elementos “elementoraiz”, “elemento” e “atributo” especifica o nome do elemento ou atributo que será criado no documento resultado. Assim, o conteúdo texto de “elementoraiz” especifica que o documento resultado terá um elemento raiz “biblioteca” e o conteúdo texto do elemento “elemento” mais externo especifica que o documento resultado terá um ou mais elementos “livro”. Quando o conteúdo texto de um elemento for definido através de uma expressão de caminho, apenas o último nível dessa expressão de caminho deverá ser considerado para formação do nome do elemento no documento resultado.

```
<consulta id=“000001”>
   <elementoraiz>biblioteca
      <elemento>$p/livro
         <atributo>$p/livro/@isbn</atributo>
         <elemento>$p/livro/titulo</elemento>
         <elemento>$p/livro/autor</elemento>
         <elemento>$p/livro/preco</elemento>
         <elemento>$p/livro/ano</elemento>
      </elemento>
   </elementoraiz>
</consulta>
```

Figura 5.14. Modelo para Construção de Documento.

A hierarquia apresentada no documento modelo deve ser preservada. Portanto, o elemento “livro” deve ser um sub-elemento do elemento “biblioteca” e “isbn” deve ser um atributo do elemento “livro”.

Finalmente, o conteúdo texto dos elementos pertencentes ao documento resultado é obtido através da avaliação das expressões que representam o conteúdo texto dos elementos do documento modelo. Assim, se a expressão possuir conteúdo texto o elemento possuirá conteúdo texto e se a expressão não possuir conteúdo texto ou for uma expressão inválida, o elemento não possuirá conteúdo texto. Por exemplo, o conteúdo texto do elemento “titulo” será obtido através da avaliação da expressão “$p/livro/titulo”, o elemento raiz “biblioteca” não possuirá conteúdo texto, pois a expressão “biblioteca” não é válida.

5.4.5 Sub-consultas XQuery

Uma consulta MXQuery deve ser dividida em várias sub-consultas XQuery. Deve ser gerada pelo menos uma sub-consulta XQuery para cada fonte relacionada na consulta

global MXQuery. A figura 5.15 apresenta as sub-consultas XQuery geradas a partir da consulta MXQuery *C*.

```
<consulta id="000001" var="$d1"> {
  for $d1 in document("d1.xml")/bib
  return
    for $p in (1 to count($d1/livro))
    where $d1/livro[$p]/ano > 1999
    return
      <livro> {
        $d1/livro[$p]/@isbn,
        $d1/livro[$p]/titulo,
        $d1/livro[$p]/autor,
        $d1/livro[$p]/preco,
        $d1/livro[$p]/ano
      } </livro>
} </consulta>
```

```
<consulta id="000001" var="$d3"> {
  for $d3 in document("d3.xml")/lib
  return
    for $p in (1 to count($d3/book))
    where $d3/book[$p]/year > 1999
    return
      <book> {
        $d3/book[$p]/isbn,
        $d3/book[$p]/title,
        $d3/book[$p]/author,
        $d3/book[$p]/price,
        $d3/book[$p]/year
      } </book>
} </consulta>
```

Figura 5.15. Sub-consultas XQuery.

O elemento raiz "consulta" possui um atributo "id" que identifica a consulta global com a qual essa sub-consulta se relaciona e um atributo "var" que identifica a fonte de dados acessada. A cláusula "FOR" mais externa especifica o documento consultado especificado na cláusula "ALIAS" da consulta *C* e a cláusula "FOR" mais interna especifica o elemento agrupador.

A cláusula "WHERE" especifica uma condição baseada em variável de fonte de dados. Embora, originalmente, a consulta MXQuery utilizasse uma condição baseada em variável de resultado, essa condição foi simplificada e repassada às fontes de dados correspondentes.

Finalmente, a cláusula "RETURN" especifica as informações requeridas pela consulta global *C* e repassada às fontes de dados. Nenhuma transformação de elementos é realizada pela sub-consulta XQuery. As transformações/construções de novos elementos serão feitas pelo pós-processador de consultas onde deve ser utilizado o modelo de construção de documentos.

Normalmente, a aparência das sub-consultas XQuery é bastante similar, visto que elas não contém informações sobre transformações de elementos ou especificações de mapeamento. A sub-consulta XQuery gerada tem como única finalidade recuperar a parte da informação armazenada em sua fonte de dados. Todo o processamento de resolução de conflitos e construção do documento resultado é realizado pela arquitetura de integração.

CAPÍTULO 6

CONCLUSÃO

Esta dissertação abordou alguns aspectos sobre estratégias de integração entre fontes de dados distintas e, em especial, linguagens de consultas de bancos de dados como mecanismo de integração. As várias características analisadas apontaram para a necessidade de utilizar sistemas de integração de fontes de dados totalmente autônomas, considerando que tais fontes de dados devam ser integradas em ambientes dinâmicos e flexíveis (ver capítulo 2). Por esse motivo, destacou-se a importância de utilizar uma estratégia de integração que não necessitasse de uma etapa de pré-integração, onde os esquemas das fontes de dados locais fossem integrados em um esquema global ou esquema federado.

A abordagem de bancos de dados múltiplos possibilita a integração de fontes de dados totalmente autônomas (isolamento total). Essa característica propicia a formulação de consultas de forma mais dinâmica do que em qualquer outra abordagem. Conseqüentemente, a linguagem de consultas utilizada deve ser capaz de prover toda a informação necessária para definição das regras de integração das fontes de dados locais.

Portanto, a linguagem de consulta deve fornecer diretivas para resolução de conflitos de integração como os abordados no capítulo 2 deste trabalho. Além disso, deve possuir mecanismos para contornar problemas de disponibilidade de fontes de dados. Essa característica possui grande importância em ambientes dinâmicos, que são caracterizados por conexões e desconexões aleatórias das fontes de dados e constantes alterações nos esquemas publicados por essas fontes de dados.

Observou-se que a linguagem de marcação XML é flexível, independente de plataforma e com suporte para representar dados semi-estruturados. Essas características apontam a linguagem de marcação XML como ideal para troca e integração de dados baseados em ambientes como a Web ou redes *ad hoc*.

Foram analisadas diversas linguagens de consulta para dados XML, porém tais linguagens não apresentaram suporte necessário para prover a integração de fontes de dados heterogêneas e baseadas em um modelo de dados comum representado em XML.

Esta dissertação apresentou, portanto, uma extensão da linguagem XQuery, denominada MXQuery, com o objetivo de utilizá-la como uma linguagem de consulta (MDL) em sistemas de bancos de dados múltiplos (MDBS). A linguagem proposta permite que consultas sejam realizadas sobre fontes de dados heterogêneas. A MXQuery resolve problemas de integração de dados como heterogeneidade semântica e o tratamento de informações incompletas (indisponibilidade de fontes de dados). A estrutura da extensão é bastante flexível, pois garante a integração de um número variável de fontes de dados com diferentes graus de autonomia.

Todas as regras para a resolução de conflitos abordados na seção 2.2.3.3 podem ser especificadas na própria consulta MXQuery, não sendo necessário nenhum processo de preparação (ou pré-integração) antes de submeter as consultas globais. Essa forma de especificar consultas garante total autonomia para as fontes de dados locais, uma vez que, alterações nas fontes de dados locais não requerem alterações de esquemas integrados ou reformulação de regras utilizadas em uma etapa de pré-integração.

Além da apresentação da linguagem, foram discutidos alguns aspectos sobre a arquitetura de integração que deve dar suporte ao processamento de consultas MXQuery. Foram analisadas a produção de sub-consultas XQuery, assertivas de correspondência e modelos para construção de documentos resultado. As sub-consultas XQuery referenciam uma única fonte de dados e são encaminhadas para tais fontes de dados para obter parte da informação necessária para resolução das consultas. As assertivas de correspondência servem para definir as regras de integração (ou mapeamento de elementos) que devem ser garantidas para resolver conflitos de integração. Finalmente, o modelo de construção de documentos é utilizado para formatar o resultado de uma consulta, ou seja, definir a hierarquia dos elementos XML no documento resultado.

A estratégia de extrair de uma consulta MXQuery global vários sub-produtos (ver seção 5.2.2) possibilita a elaboração de sub-consultas XQuery mais simples e facilita a combinação/integração dos resultados realizada pelo pós-processador de consulta.

As características da arquitetura de integração e da linguagem propostas nesta dissertação propiciam uma alta independência dos participantes da conexão e a

capacidade de contornar problemas de falhas de comunicação, portanto, são ideais para ambientes como a Web ou redes *ad hoc*.

A tabela 6.1 apresenta uma comparação entre algumas linguagens de consulta, que já foram analisadas no capítulo 2, e a linguagem MXQuery proposta. A coluna modelo de dados apresenta o modelo de dados utilizado por cada linguagem e uma especificação do tipo de dados para o qual essa linguagem está orientada.

Linguagem Proposta	Modelo de Dados	Esquema Global	Pré-Integração	Resolução de Conflitos	Funções de Usuário / Reconciliação
MSQL	Relacional (estruturado)	Não	Não	N – D – A	Sim / Não
IDL	Relacional (estruturado)	Não	Não	N – E – A	Não / Não
MSQL Extensão	Relacional (estruturado)	Sim	Não	N – D – S – A	Sim / Não
YATL*	XML (semi-estruturado)	Sim	Sim	N – D – A	Sim / Não
SOQL	Orientado a Objeto (semi-estruturado)	Sim	Sim	N – D – E – A	Sim / Não
FRAQL	Objeto-Relacional (estruturado)	Sim	Não	N – D – S – E – A	Sim / Sim
XQuery	XML (semi-estruturado)	N/A	N/A	N/A	Sim / Não
MXQuery	XML (semi-estruturado)	Não	Não	N – D – S – E – A	Sim / Não

Legenda: **N** – Nome **D** – Domínio **S** – Semântico **E** – Esquema **A** – Dados
N/A – Não se aplica
(*) – A classificação apresentada refere-se a 2ª versão da linguagem YATL.

Tabela 6.1. Linguagens de Consulta como Mecanismo de Integração.

A coluna esquema global refere-se à necessidade de algumas linguagens em definir um esquema para o resultado das consultas, em seguida, esse esquema tem seus atributos mapeados para os atributos das fontes de dados locais. Embora todas as linguagens listadas, com exceção da linguagem XQuery, utilizem algum processo de mapeamento, apenas as que definem um esquema global realizam o mapeamento dissociado da consulta.

A coluna pré-integração refere-se à necessidade de executar algum processo de submissão de regras de integração antes de submeter consultas globais. Isso significa que algumas linguagens, após definirem o mapeamento entre o esquema global, devem validar o mapeamento e gerar um esquema integrado sobre o qual as consultas globais serão avaliadas.

A coluna funções de usuário / reconciliação refere-se à possibilidade do usuário definir funções que possam ser utilizadas em consultas, possivelmente, para resolver conflitos e utilizar essas funções para executar a reconciliação de dados, ou seja, utilizar funções para resolver um conflito de dados.

Como pode ser observado na tabela, a linguagem MXQuery é tão dinâmica quanto as linguagem MSQL e IDL, pois não necessita da definição de esquemas globais nem da realização de uma etapa de pré-integração. Além disso, é capaz de identificar e resolver os 5 (cinco) grupos de conflitos analisados no capítulo 2. Finalmente, é uma linguagem totalmente direcionada à consulta de dados semi-estruturados, onde o XML é utilizado como CDM.

Por outro lado, a linguagem MXQuery ainda não prevê a utilização de funções para reconciliação de dados, portanto, nesse aspecto a linguagem FRAQL é mais flexível. Os conflitos de dados são resolvidos pela linguagem MXQuery através da indicação de prioridades das fontes de dados. Quando um conflito de dados ocorre, prevalece a informação contida na fonte de dados considerada mais prioritária em relação à outra.

Atualmente, tem-se trabalhado em um protótipo da arquitetura de integração proposta nessa dissertação. Devido à natureza dos documentos produzidos pela arquitetura e da necessidade de manter uma independência de plataforma, tem-se utilizado a linguagem Java para implementar o processador de consultas MXQuery. Além disso, tem-se estudado a viabilidade de utilizar um banco de dados XML nativo para implementar a função do repositório MDBS.

A linguagem MXQuery também está sendo utilizada como linguagem de consultas para integração de bancos de dados múltiplos e móveis [Bra03]. Neste projeto, são utilizados agentes móveis para visitar as diversas fontes de dados participantes e carregar consigo as sub-consultas que devem ser executadas por essas fontes de dados.

A linguagem MXQuery ainda não possui declarações capazes de expressar instruções de atualização sobre fontes de dados, portanto, esse aspecto deve ser alvo de estudos futuros. O grande problema da especificação de atualizações sobre fontes de dados heterogêneas é a manutenção da consistência global sem violar a autonomia das fontes de dados (ver seção 2.2.2).

Como trabalho futuro, pretende-se ainda investigar a utilização de ontologias para facilitar a proposição de consultas sobre fontes de dados heterogêneas. A proposta apresentada nessa dissertação necessita que o usuário especifique todas as regras para resolução de conflitos. Conseqüentemente, a interpretação semântica das fontes de

dados é dada pelo próprio usuário. A utilização de um dicionário semântico pode contribuir para a especificação de consultas semanticamente mais uniformes.

REFERÊNCIAS BIBLIOGRÁFICAS

[Abi97a] Abiteboul, S.. Querying Semi-Structured Data. *In Proc. of International Conference on Database Theory*. 1997.

[Abi97b] Abiteboul, S., Quass, D., McHugh, J., Widom J., Wiener, J.. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries, 1(1): 68-88*. 1997.

[Abi01] Abiteboul, S., Segoufin, L., Vianu, V.. Representing and Querying XML with Incomplete Information. *In Proc. of ACM PODS'01*. 2001.

[And00] Anderson, R., et al.. Professional XML. *Ciência Moderna*. 2000.

[Ber03] Berglund, A.. Boag, S., Chamberlin, D., Fernadez, M., Kay, M., Robie, J., Siméon, J.. XML Path Language (XPath) 2.0. *W3C - Work in progress*. http://www.w3.org/TR/xpath20/. 2003. Consultado em 01/09/2003.

[Boa03] Boag, S., Chamberlin, D., Fernadez, M., Florescu, D., Robie, J., Siméon, J., Stefanescu, M.. XQuery 1.0: An XML Query Language. *W3C - Work in progress*. http://www.w3.org/TR/xquery/. 2003. Consultado em 22/08/2003.

[Bou95] Bouguettaya, A., King, R., Papazoglou, M.. On Building a Hyperdistributed Database. *Information System, 20(7):1-27*. 1995.

[Bra99] Brayner, A.. Transaction Management in Multidatabase Systems. *Shaker-Verlag*. 1999.

[Bra01] Brayner, A., Härder, T.. Global Semantic Serializability: An Approach to Increase Concurrency in Multidatabase Systems. *In Proc. of the 9th International Conference on Cooperative Information Systems*. 2001.

[Bra03] Brayner, A., Aguiar, J.. Sharing Mobile Databases in Dynamically Configurable Environments. *In Proc. of the 15th International Conference on Advanced Information Systems Engineering*. 2003.

[BrP00] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E.. Extensible Markup Language (XML) 1.0 (Second Edition). *W3C Recommendation*. http://www.w3.org/TR/REC-xml. 2000. Consultado em 19/09/2001.

[Bun96] Buneman, P., Davidson, S., Hillebrand, G., Suciu, S.. A Query Language and Optimization Techniques for Unstructured Data. *In Proc. of the International Conference on Management of Data*. 1996.

[Cha94] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J.. The TSIMMIS Project: Integration of Heterogeneous Information Sources. *In Proc. of the 10th Meeting of the Information Processing Society of Japan*. 1994.

[ChF00] Chamberlin, D., Robie, J., Florescu, D.. Quilt: An XML Query Language for Heterogeneous Data Sources. *International Workshop on the Web and Databases (WebDB'2000)*. 2000.

[Chr00] Christophides, V., Cluet, S., Siméon, J.. On Wrapping Query Languages and Efficient XML Integration. *In Proc. of the ACM SIGMOD International Conference in Management of Data*. 2000.

[Cla99] Clark, J., DeRose, S.. XML Path Language (XPath) Version 1.0. *W3C Recommendation*. http://www.w3.org/TR/xpath. 1999. Consultado em 25/10/2001.

[Clu98] Cluet, S., Delobel, C., Siméon, J., Smaga, K.. Your Mediators Need Data Convertion. In Proc. of ACM SIGMOD Conference on Management of Data. 1998.

[Clu00] Cluet, S., Siméon, J.. YATL: A Functional and Declarative Language for XML. Draft Manuscript. http://citeseer.nj.nec.com/cluet00yatl.html. 2000. Consultado em 12/01/2002.

[Coo01] Cooper, B., Sample, N., Franklin, M., Hjaltason, G., Shadmon, M.. A Fast Index for Semistructured Data. *In Proc. of 27th VLDB Conference*. 2001.

[Deu98] Deutsch, A., Fernadez, M., Florescu, D., Levy, A., Suciu, D.. XML-QL: A Query Language for XML. http://www.w3.org/TR/NOTE-xml-ql/. 1998. *W3C Review*. Consultado em 30/10/2001.

[Dom00] Domenig, R., Dittrich, K.. A Query Based Approach for Integrating Heterogeneous Data Sources. *In Proc of. 9th International Conference on Information Knowledge Management*. 2000.

[Dog95] Dogac, A., Dengi, C., Kilic, E., Ozhan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., Mancuhan, S.. METU Interoperable Database System. *ACM SIGMOD Record, Vol. 24, nº 3*. 1995.

[Dog96] Dogac, A., Dengi, C., Kilic, E., Ozhan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., Mancuhan, S.. A Multidatabase System Implementation on CORBA. *In Proc. of the 6th International Workshop on Research Issues in Data Engineering (RIDE '96)*. 1996.

[Dra03] Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/. 2003. *W3C Working Draft*. Consultado em 23/09/2003.

[Elm98] Elmagarmid, A., Bouguettaya, A., Benatallah, B.. Interconnecting Heterogeneous Information Systems. *Kluwer Academic Publishers*. 1998.

[Erw03] Erwig, M.. Xing: A Visual XML Query Language. *Journal of Visual Language and Computing, 14(1): 5-45*. 2003.

[Fal01] Fallside, D.. XML Schema Part 0: Primer. *W3C Recommendation*. http://www.w3.org/TR/xmlschema-0/. 2001. Consultado em 21/09/2001.

[Fer03] Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.. XQuery 1.0 and XPath 2.0 Data Model. *W3C - Work in progress*. http://www.w3.org/TR/xpath-datamodel/. 2003. Consultado em 05/10/2003.

[Ger98] Gertz, M., Schmitt, I.. Data Integration Techniques Based on Data Quality Aspects. *In Proc. of 3rd National Workshop on Federated Databases*. 1998.

[Gol99] Goldman, R., McHugh, J., Widom, J.. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. *In Proc. of 2nd International Workshop on the Web and Databases*. 1999.

[Gra93] Grant, J., Litwin, W., Roussopoulos, N., Sellis, T.. Query Languages for Relational Multidatabases. *In Proc. of 19th VLDB Conference*. 1993.

[Hol01] Holzner, S.. Desvendando o XML. *Campus*. 2001.

[Kri91] Krishnamurthy, R., Litwin, W., Kent, W.. Languages Features for Interoperability of Databases with Schematic Discrepancies. *ACM SIGMOD Record, 20 (2): 40-49*. 1991.

[Law01] Lawrence, R.. Automatic Conflict Resolution to Integrate Relational Schema. *Tese de Doutorado, University of Manitoba, Canada*. 2001.

[Li01] Li, Q., Moon, B.. Indexing and Querying XML Data for Regular Path Expressions. *In Proc. of 27th VLDB Conference*. 2001.

[Lit89] Litwin, W., Abdellatif, A., Zeroual, A., Nicolas, B., Vigier, Ph.. MSQL: A Multidatabase Language. *Information Sciences, (49)*. 1989.

[Lit90] Litwin, W., Mark, L., Roussopoulos, N.. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys, 22(3):267-293*. 1990.

[Los98] Lóscio, B.. Atualização de Múltiplas Bases de Dados através de Mediadores. *Tese de Mestrado, Universidade Federal do Ceará, Brasil.* 1998.

[Man01] Manolescu, I., Florescu, D., Kossmann, D.. Answering XML Queries over Heterogeneous Data Sources. *In Proc. of 27th VLDB Conference*. 2001.

[McH99a] McHugh, J., Widom, J.. Query Optimization for Semistructured Data. *Techinical Report.* 1999.

[McH99b] McHugh, J., Widom, J.. Query Optimization for XML. *In Proc. of 25th VLDB Conference*. 1999.

[Men02] Mendonça, N., Brayner, A., Monteiro, J.. Mobile Database Communities: An Approach for Sharing Autonomous Mobile Databases in Ad Hoc Networks. *Submitted for Publication*. 2002.

[Mil98] Milo, T., Zohar, S.. Using Schema Matching to Simplify Heterogeneous Data Translation. *In Proc. of 24th VLDB Conference*. 1998.

[Mis95] Missier, P., Rusimkiewicz, M.. Extending a Multidatabase Manipulation Language to Resolve Schema and Data Conflicts. *In Proc. of the 6th IFIP TC-2 Working Conference on Data Semantics.* 1995.

[Nur96] Nural, S., Koksal, P., Ozcan, F., Dogac, A.. Query Decomposition and Processing in Multidatabase Systems. *In Proc. of International Conference on Engineering Systems Design & Analysis*. 1996.

[Ozc97] Ozcan, F., Nural, S., Koksal, P., Evrendilek, C.. Dynamic Query Optimization in Multidatabases. *Bulletin of the IEEE Computer Society Techinical Committee on Data Engineering*. 1997.

[Özs91] Özsu, M., Valduriez, P.. Distributed Database Systems: Where Are We Now *IEEE, Computer, 24(8): 68-78*. 1991.

[Özs99] Özsu, M., Valduriez, P.. Principles of Distributed Database Systems. *Prentice–Hall, 2nd edition*. 1999.

[Pan02] Pankowski, T.. XML-SQL: An XML Query Language Based on SQL and Path Tables. *In Proc. of International Conference on Extending Database Technology*. 2002.

[Pap95] Papakonstantinou, Y., Garcia-Molina, H., Widom, J.. Object Exchange Across Heterogeneous Information Sources. *In Proc. of Data Engineering Conference*. 1995.

[Pap96] Papakonstantinou, Y., Abiteboul, S., Garcia-Molina, H.. Object Fusion in Mediator Systems. *In Proc. of 22th VLDB Conference*. 1996.

[Par98] Parent, C., Spaccapietra, S.. Issues on Approaches of Database Integration. *Communications of the ACM, 41(5es):166-178*. 1998.

[Peq00] Pequeno, V.. Auto-Manutenção de Classes de Fusão em Visões de Integração de Dados. *Tese de Mestrado, Universidade Federal do Ceará, Brasil*. 2000.

[Rob98] Robie, J., Lapp, J., Schach, D.. XML Query Language (XQL). *W3C*. http://www.w3.org/TandS/QL/QL98/pp/xql.html. 1998. Consultado em 21/11/2001.

[Rob00] Robie, J., Chamberlin, D., Florescu, D.. Quilt: An XML Query Language. *Presented at XML Europe*. 2000.

[Sat00] Sattler, K., Conrad, S., Saake, G.. Adding Conflict Resolution Features to a Query Language for Database Federations. *In Proc. of 3rd International Workshop on Engineering Federated Information Systems*. 2000.

[Sel01] Seligman, L., Rosenthal, A.. XML's Impact on Databases and Data Sharing. *IEEE, Computer*. 2001.

[Sha99] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J.. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In Proc. of 25th VLDB Conference*. 1999.

[She90] Sheth, A., Larson, J.. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys, 22(3):183-236*. 1990.

[Sil99] Silberschatz, A., Korth, H., Sudarshan, S.. Sistema de Banco de Dados. *Makron Books, 3a ed.*. 1999.

[Spa92] Spaccapietra, S., Parent, C., Dupont, Y.. Model Independent Assertions for Integration of Heterogeneous Schemas. *VLDB Journal, 1(1): 81-126*. 1992.

[Tom96] Tomasic, A., Raschid, L., Valduriez, P.. Scaling Heterogeneous Databases and the Design of Disco. *In Proc. of the International Conference on Distributed Computing Systems*. 1996.

[XDK03] Oracle XML Developer's Kit for PL/SQL. *Oracle Technology Network*. http://otn.oracle.com/tech/xml/xdk_plsql/indesx.html. 2003. Consultado em 22/10/2003.

APÊNDICE A

DOCUMENTOS EXEMPLO

Documento: d1.xml

```
<bib>
   <livro isbn="393">
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <autor>Özsu</autor>
      <autor>Valduriez</autor>
      <ano>1999</ano>
      <preco>89.00</preco>
   </livro>
   <livro isbn= "352">
      <titulo>Implementação de Sistemas de BDs </titulo>
      <autor>Garcia-Mollina</autor>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <ano>2001</ano>
      <editora>Campus</ editora >
      <preco>110.00</preco>
   </livro>
</bib>
```

Documento: d2.xml

```
<biblioteca>
   <livro isbn="346">
      <titulo>Sistema de Banco de Dados</titulo>
      <autor>Silberschatz</autor>
      <autor>Korth</autor>
      <autor>Sudarshan</autor>
      <publicadoem>1999</publicadoem>
      <editora>Makron Books</editora>
   </livro>
</biblioteca>
```

Documento: d3.xml

```
<lib>
   <book>
      <isbn>053</isbn>
      <title>Fundamentals of Database Systems</title>
      <author>Elmasri</author>
      <author>Navathe</author>
      <year>1994</year>
      <press>Addison-Wesley</press>
      <price>54.00</price>
   </book>
   <book>
      <isbn>013</isbn>
      <title>A First Course in Database Systems</title>
      <author>Ullman</author>
      <author>Widom</author>
      <year>1997</year>
      <press>Prentice Hall</press>
      <price>45.00</price>
   </book>
   <book>
      <isbn>352</isbn>
      <title>Database System Implementation</title>
      <author>Garcia-Mollina</author>
      <author>Ullman</author>
      <author>Widom</author>
      <year>1999</year>
      <press>Prentice Hall</press>
      <price>50.00</price>
   </book>
</lib>
```

Documento: d4.xml

```
<library>
   <book autref=“A001 A002”>
      <isbn>053</isbn>
      <title>Fundamentals of Database Systems</title>
      <year>1994</year>
      <press>Addison-Wesley</press>
   </book>
   <book autref=“A003 A004 A005”>
      <isbn>352</isbn>
      <title>Database System Implementation</title>
      <year>1999</year>
      <press>Prentice Hall</press>
   </book>
```

```
   <author id="A001">
      <name>Elmasri</name>
   </author>
   <author id="A002">
      <name>Navathe</name>
   </author>
   <author id="A003">
      <name>Garcia-Molina</name>
   </author>
   <author id="A004">
      <name>Ullman</name>
   </author>
   <author id="A005">
      <name>Widom</name>
   </author>
</library>
```

Documento: d5.xml

```
<lib>
   <Addison-Wesley>
      <book>
         <isbn>053</isbn>
         <title>Fundamentals of Database Systems</title>
         <author>Elmasri</author>
         <author>Navathe</author>
         <year>1994</year>
         <price>53.50</price>
      </book>
   </Addison-Wesley>
   <PrenticeHall>
      <book>
         <isbn>013</isbn>
         <title>A First Course in Database Systems</title>
         <author>Ullman</author>
         <author>Widom</author>
         <year>1997</year>
         <price>44.30</price>
      </book>
      <book>
         <isbn>352</isbn>
         <title>Database System Implementation</title>
         <author>Garcia-Mollina</author>
         <author>Ullman</author><author>Widom</author>
         <year>1999</year>
         <price>50.60</price>
      </book>
   </PrenticeHall>
</lib>
```

APÊNDICE B

RESULTADOS

Resultado: consulta 1

```
<biblioteca>
   <livro>
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <autor>Özsu</autor>
      <autor>Valduriez</autor>
   </livro>
   <livro>
      <titulo>Implementação de Sistemas de BDs</titulo>
      <autor>Garcia-Mollina</author>
      <autor>Ullman</autor>
      <autor>Widom</autor>
   </livro>
   <livro>
      <titulo>Fundamentals of Database Systems</titulo>
      <autor>Elmasri</autor>
      <autor>Navathe</autor>
   </livro>
   <livro>
      <titulo>A First Course in Database Systems</titulo>
      <autor>Ullman</autor>
      <autor>Widom</autor>
   </livro>
   <livro>
      <titulo>Database System Implementation</titulo>
      <autor>Garcia-Mollina</autor>
      <autor>Ullman</autor>
      <autor>Widom</autor>
   </livro>
</biblioteca>
```

Resultado: consulta 2

```
<biblioteca>
   <livro>
      <isbn>393</isbn>
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <autor>Özsu</autor>
      <autor>Valduriez</autor>
      <preco>89.00</preco>
   </livro>
   <livro>
      <isbn>352</isbn>
      <titulo>Implementação de Sistemas de BDs</titulo>
      <autor>Garcia-Mollina</autor>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <preco>110.00</preco>
   </livro>
   <livro>
      <isbn>053</isbn>
      <titulo>Fundamentals of Database Systems</titulo>
      <autor>Elmasri</autor>
      <autor>Navathe</autor>
      <preco>160.92</preco>
   </livro>
   <livro>
      <isbn>013</isbn>
      <titulo>A First Course in Database Systems</titulo>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <preco>134.1</preco>
   </livro>
</biblioteca>
```

Resultado: consulta 3

```
<biblioteca>
   <livro isbn="393">
      <titulo>Princípios de SBDs Distribuídos</titulo>
      <ano>1999</ano>
      <preco>89.00</preco>
   </livro>
   <livro isbn= "352">
      <titulo>Implementação de Sistemas de BDs </titulo>
      <ano>2001</ano>
      <editora>Campus</ editora >
      <preco>110.00</preco>
   </livro>
```

```
   <livro isbn="346">
      <titulo>Sistema de Banco de Dados</titulo>
      <publicadoem>1999</publicadoem>
      <editora>Makron Books</editora>
   </livro>
</biblioteca>
```

Resultado: consulta 4

```
<biblioteca>
  <livro isbn="352">
     <titulo>Implementação de Sistemas de BDs</titulo>
  </livro>
</biblioteca>
```

Resultado: consulta 5

```
<biblioteca>
   <livro>
      <titulo_o>Database System Implementation</titulo_o>
      <editora_o>Prentice Hall</editora_o>
      <titulo_t>Implementação de Sistemas de BDs</titulo_t>
      <editora_t>Campus</editora_t>
   </livro>
</biblioteca>
```

Resultado: consulta 6

```
<biblioteca>
   <livro>
      <titulo>Implementação de Sistemas de BDs </titulo>
      <autor>Garcia-Mollina</autor>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <editora>Campus</editora>
      <origem>document("d1.xml")/bib</origem>
   </livro>
   <livro>
      <titulo>A First Course in Database Systems</titulo>
      <autor>Ullman</autor>
      <autor>Widom</autor>
      <editora>PrenticeHall</editora>
      <origem>document("d5.xml")/library</origem>
   </livro>
</biblioteca>
```

APÊNDICE C

XML *SCHEMA*

Esquema: sc1.xsd

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
<xsd:element name=“bib” type=“bibType”/>
<xsd:complexType name=“bibType”>
  <xsd:element name=“livro” maxOccurs=“unbounded”>
    <xsd:complexType>
      <xsd:attribute name=“isbn” type=“xsd:ID” use=“required”/>
      <xsd:element name=“titulo” type=“xsd:string”/>
      <xsd:element name=“autor” type=“xsd:string” maxOccurs=“unbounded”/>
      <xsd:element name=“ano” type=“xsd:string”/>
      <xsd:element name=“editora” type=“xsd:string” minOccurs=“0” maxOccurs=“1”/>
      <xsd:element name=“preco” type=“xsd:decimal”/>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>
```

Esquema: sc2.xsd

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
<xsd:element name=“biblioteca” type=“bibType”/>
<xsd:complexType name=“bibType”>
  <xsd:element name=“livro” maxOccurs=“unbounded”>
    <xsd:complexType>
      <xsd:attribute name=“isbn” type=“xsd:ID” use=“required”/>
      <xsd:element name=“titulo” type=“xsd:string”/>
      <xsd:element name=“autor” type=“xsd:string” maxOccurs=“unbounded”/>
      <xsd:element name=“ano” type=“xsd:string”/>
      <xsd:element name=“editora” type=“xsd:string”/>
      <xsd:element name=“preco” type=“xsd:decimal” minOccurs=“0” maxOccurs=“1”/>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>
```

Esquema: sc3.xsd

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
<xsd:element name=“lib” type=“libType”/>
<xsd:complexType name=“libType” content=“elementOnly”>
  <xsd:element name=“book” maxOccurs=“unbounded”>
    <xsd:complexType>
      <xsd:element name=“isbn” type=“xsd:string”/>
      <xsd:element name=“title” type=“xsd:string”/>
      <xsd:element name=“author” type=“xsd:string” maxOccurs=“unbounded”/>
      <xsd:element name=“year” type=“xsd:string”/>
      <xsd:element name=“press” type=“xsd:string”/>
      <xsd:element name=“price” type=“xsd:decimal”/>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>
```

Esquema: sc4.xsd

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
<xsd:element name=“library” type=“libraryType”/>
<xsd:complexType name=“libraryType”>
  <xsd:element name=“book” type=“bookType” maxOccurs=“unbounded”/>
  <xsd:element name=“author” type=“authorType” maxOccurs=“unbounded”/>
</xsd:complexType>
<xsd:complexType name=“bookType”>
  <xsd:attribute name=“autref” type=“xsd:IDREFS” use=“required”/>
  <xsd:element name=“isbn” type=“xsd:string”/>
  <xsd:element name=“title” type=“xsd:string”/>
  <xsd:element name=“year” type=“xsd:string”/>
  <xsd:element name=“press” type=“xsd:string”/>
</xsd:complexType>
<xsd:complexType name=“authorType”>
  <xsd:attribute name=“id” type=“xsd:ID” use=“required”/>
  <xsd:element name=“name” type=“xsd:string”/>
</xsd:complexType>
```

Esquema: sc5.xsd

```
<xsd:schema xmlns:xsd:“http://www.w3.org/1999/XMLSchema”>
<xsd:element name=“lib” type=“libraryType”/>
<xsd:complexType name=“libraryType”>
  <xsd:element name=“Addison-Wesley” type=“pressType”/>
  <xsd:element name=“PrenticeHall” type=“pressType”/>
</xsd:complexType>
```

```
<xsd:complexType name="pressType">
  <xsd:element name="book" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:element name="isbn" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string" maxOccurs="unbounded"/>
      <xsd:element name="year" type="xsd:short"/>
      <xsd:element name="price" type="xsd:decimal"/>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo