UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DEISE DE BRUM SACCOL

# Detecção, Gerenciamento e Consulta a Réplicas e a Versões de Documentos XML

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Profa. Dra. Nina Edelweiss
Orientadora

Profa. Dra. Renata de Matos Galante
Co-orientadora

Porto Alegre, dezembro de 2008.

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Saccol, Deise de Brum

Detecção, Gerenciamento e Consulta a Réplicas e a Versões de Documentos XML / Deise de Brum Saccol – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

148 f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Orientadora: Nina Edelweiss; Co-orientadora: Renata de Matos Galante.

1.Versões 2.XML 3.*Peer-to-Peer*. I. Edelweiss, Nina. II. Galante, Renata de Matos. III. Título.

*“O futuro não pode ser previsto, mas pode ser inventado. É a nossa habilidade de inventar o futuro que nos dá esperança para fazer de nós o que somos”.*

– Dennis Gabor

# AGRADECIMENTOS

Meu agradecimento a todas as pessoas que incentivaram e contribuíram para o desenvolvimento deste trabalho. Em especial agradeço:

À minha orientadora, professora Nina Edelweiss, que soube me conduzir serenamente ao longo desta jornada. Que soube dar as “cacetadas” quando necessário, mas que também soube dar a palavra de incentivo no momento certo.

À minha co-orientadora Renata Galante, que sofreu um *upgrade* nestes quatro anos e se tornou também uma grande amiga. Obrigada pela ajuda, técnica e psicológica ☺, que tornou possível chegarmos à finalização deste trabalho.

Ao professor Carlo Zaniolo, que me recebeu de braços abertos em seu grupo de pesquisa, e que sempre achou tempo para discussões técnicas com a “*Miss Deise*”, com sua humildade e inteligência sublimes.

Agradeço à minha família e aos meus amigos. Em especial aos queridos Adrovane Kade e Viviane Orengo, que acompanharam (e compartilharam) todas as crises, depressões e alegrias que o doutorado nos proporciona. Desejo tudo de bom para vocês!

Também agradeço aos alunos Márcio Mello, Rodrigo Noll, Rodrigo Santos e Felipe Giacomel, que contribuíram com a implementação dos protótipos e tornaram possível a realização de alguns dos experimentos.

Agradeço à UFRGS, ao CNPq e à CAPES, pela oportunidade e pelo financiamento.

Por último, mas mais importante, agradeço ao meu amado Eduardo, que me “agüentou” por estes quatro anos. Obrigada pelo incentivo constante. E por fim, um apertão no Kiko e no Fofão, que proporcionaram momentos de alegria e descontração neste último ano.

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AHP | Processo Analítico Hierárquico |
| API | *Application Programming Interface* |
| BD | Banco de Dados |
| CBR | Raciocínio Baseado em Casos |
| CVS | *Concurrent Version System* |
| DAML | *DARPA Agent Markup Language* |
| DARPA | *Defense Advanced Research Projects Agency* |
| DetVX | Detector de Réplicas e de Versões de Documentos XML |
| DetVX_P2P | Detector de Réplicas e de Versões de Documentos XML em um Ambiente *Peer-to-Peer* |
| DIFF | Algoritmo de Detecção de Diferenças |
| DOEM | *Delta Object Exchange Model* |
| DOM | *Document Object Model* |
| DTD | *Document Type Definition* |
| GAV | *Global-as-View* |
| H-Doc | *Historical Document* |
| HTML | *HyperText Markup Language* |
| IP | *Internet Protocol* |
| LAV | *Local-as-View* |
| MCC | Modelo Conceitual Canônico |
| MD5 | *Message Digest Algorithm* |
| OEM | *Object Exchange Model* |
| OIL | *Ontology Interchange Language* |
| OWL | *Ontology Web Language* |

| | |
|---|---|
| P2P | *Peer-to-Peer* |
| PTS | *Proportional Transportation Similarity* |
| RCS | *Revision Control System* |
| SCCS | *Source Code Control System* |
| SPaR | *Sparse Preorder and Range* |
| SQL | *Structured Query Language* |
| TE | *Time-End* |
| TF-IDF | *Term Frequency-Inverse Document Frequency* |
| TS | *Time-Start* |
| TVSE | *Temporal and Versioning Model for Schema Evolution* |
| XML | *Extensible Markup Language* |
| XSD | *XML Schema* |
| XSL | *Extensible Stylesheet Language* |
| XSLT | *XSL Transformations* |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

O objetivo geral desta tese é a detecção, o gerenciamento e a consulta às réplicas e às versões de documentos XML. Denota-se por *réplica* uma cópia idêntica de um objeto do mundo real, enquanto *versão* é uma representação diferente, mas muito similar, deste objeto. Trabalhos prévios focam em gerenciamento e consulta a versões conhecidas, e não no problema da detecção de que dois ou mais objetos, aparentemente distintos, são variações (versões) do mesmo objeto. No entanto, o problema da detecção é crítico e pode ser observado em diversos cenários, tais como detecção de plágio, *ranking* de páginas Web, identificação de clones de *software* e busca em sistemas *peer-to-peer* (P2P).

Nesta tese assume-se que podem existir diversas réplicas de um documento XML. Documentos XML também podem ser modificados ao longo do tempo, ocasionando o surgimento de versões. A detecção de réplicas é relativamente simples e pode ser feita através do uso de funções *hash*. Já a detecção de versões engloba conceitos de similaridade, a qual pode ser medida por várias métricas, tais como similaridade de conteúdo, de estrutura, de assunto, etc. Além da análise da similaridade entre os arquivos também se faz necessária a definição de um mecanismo de detecção de versões. O mecanismo deve possibilitar o gerenciamento e a posterior consulta às réplicas e às versões detectadas.

Para que o objetivo da tese fosse alcançado foram definidos um conjunto de funções de similaridade para arquivos XML e o mecanismo de detecção de réplicas e de versões. Também foi especificado um *framework* onde tal mecanismo pode ser inserido e os seus respectivos componentes, que possibilitam o gerenciamento e a consulta às réplicas e às versões detectadas. Foi realizado um conjunto de experimentos que validam o mecanismo proposto juntamente com a implementação de protótipos que demonstram a eficácia dos componentes do *framework*. Como diferencial desta tese, o problema de detecção de versões é tratado como um problema de classificação, para o qual o uso de limiares não é necessário. Esta abordagem é alcançada pelo uso da técnica baseada em classificadores *Naïve Bayesianos*. Resultados demonstram a boa qualidade obtida com o mecanismo proposto na tese.

**Palavras-Chave:** XML, versionamento, similaridade.

**Detection, Management and Querying of Replicas and Versions of XML Documents**

# ABSTRACT

The overall goals of this thesis are the detection, management and querying of replicas and versions of XML documents. We denote by *replica* an identical copy of a real-world object, and by *version* a different but very similar representation of this object. Previous works focus on version management and querying rather than version detection. However, the version detection problem is critical in many scenarios, such as plagiarism detection, Web page ranking, software clone identification, and peer-to-peer (P2P) searching.

In this thesis, we assume the existence of several replicas of a XML document. XML documents can be modified over time, causing the creation of versions. Replica detection is relatively simple and can be achieved by using hash functions. The version detection uses similarity concepts, which can be assessed by some metrics such as content similariy, structure similarity, subject similarity, and so on. Besides the similarity analysis among files, it is also necessary to define the version detection mechanism. The mechanism should allow the management and the querying of the detected replicas and versions.

In order to achieve the goals of the thesis, we defined a set of similarity functions for XML files, the replica and version detection mechanism, the framework where such mechanism can be included and its components that allow managing and querying the detected replicas and versions. We performed a set of experiments for evaluating the proposed mechanism and we implemented tool prototypes that demonstrate the accuracy of some framework components. As the main distinguishing point, this thesis considers the version detection problem as a classification problem, for which the use of thresholds is not necessary. This approach is achieved by using *Naïve Bayesian* classifiers.

**Keywords:** XML, versioning, similarity.

# 1 INTRODUÇÃO

Documentos XML podem estar replicados em diversos arquivos[1]. Além disso, um documento pode sofrer modificações ao longo do tempo, ocasionando o surgimento de diversas versões[2] deste objeto. Trabalhos prévios focam no gerenciamento e na consulta a versões conhecidas (KATZ et al., 1987; CHIEN et al., 2001e; RONNAU et al., 2005; WESTFECHTEL et al., 2001), e não na detecção propriamente dita destas versões. No entanto, o problema da detecção de versões de documentos XML é crítico e pode ser observado em diversos cenários. Por exemplo, sistemas P2P tradicionais não costumam tratar a existência de réplicas e de versões de recursos, e por isso acabam gerando um aumento na complexidade em nível lógico (o usuário precisa analisar os arquivos retornados e distingüir réplicas e versões) e na ineficiência em nível físico (o sistema consulta réplicas e versões de documentos eventualmente já retornados ao usuário).

O problema da existência de réplicas e de versões pode ser exemplificado na Figura 1.1. Para a consulta "*XML Databases*" pode-se observar que todos os resultados retornados parecem ser os mesmos, o que indicaria que o sistema retornou réplicas de um mesmo documento. Também poderia-se supor que alguns dos resultados retornados são, na verdade, diferentes versões do mesmo objeto. Em ambos os casos, cabe ao usuário a responsabilidade de navegar no conjunto de resultados retornados e verificar quais são réplicas e/ou versões do objeto pesquisado. Além disso, o sistema gasta tempo na busca de recursos que eventualmente já estão contidos no conjunto de resultados (no caso de réplicas) ou na busca de recursos desnecessários (no caso de versões). Estas desvantagens podem ser reduzidas com o uso de um mecanismo automático de detecção de versões.

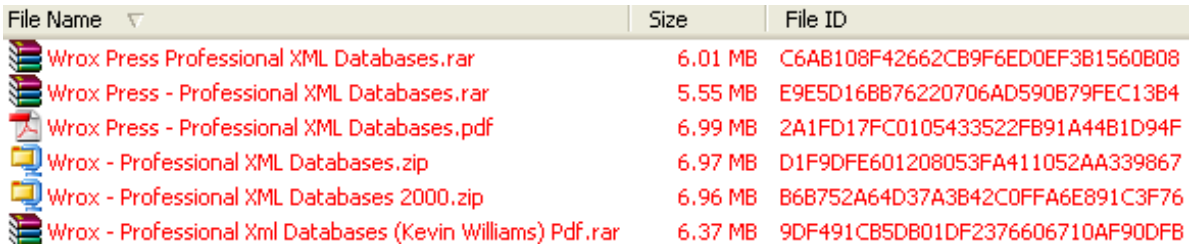
Porém, um mecanismo de detecção de versões baseado no uso de funções de similaridade apresenta dois desafios importantes: (i) como medir a similaridade entre arquivos; e (ii) como definir um grau mínimo de similaridade exigido para ser considerada uma versão. Para solucionar o **primeiro** problema, estão disponíveis algumas funções de similaridade para

---

[1] O termo *arquivo* refere-se à representação física de um objeto do mundo real (exemplo: *c:\curriculum.doc*); *documento* refere-se à representação lógica de um objeto do mundo real (exemplo: *curriculum vitae* da pesquisadora *Deise Saccol*). Em outras palavras, um documento pode ser armazenado em vários arquivos, ou porque o documento está replicado, ou porque possui várias versões.

[2] *Versão* é a descrição de um objeto em um período de tempo ou sob certo ponto de vista, cujo registro é importante para uma determinada aplicação.

valores atômicos e para valores complexos. No entanto, encontrar funções de similaridade para documentos XML que sejam efetivas para a detecção de versões ainda representa um desafio na área de pesquisa. Funções de similaridade para este propósito devem considerar, além da similaridade de conteúdo e de estrutura, outras características relevantes para a aplicação em questão, tais como a importância das modificações na evolução dos documentos. Por exemplo, uma modificação na data de nascimento no currículo de uma pessoa é uma característica com grande importância ao se analisar versões de um documento. Por outro lado, uma alteração no endereço residencial desta pessoa já tem uma relevância menor. Em outras palavras, endereços diferentes podem facilmente referir-se à mesma pessoa (isto é, versões de um mesmo documento). No entanto, o registro de duas datas de nascimento distintas indica uma maior probabilidade de se tratar de pessoas diferentes (isto é, documentos diferentes).



| File Name | Size | File ID |
|---|---|---|
| Wrox Press Professional XML Databases.rar | 6.01 MB | C6AB108F42662CB9F6ED0EF3B1560B08 |
| Wrox Press - Professional XML Databases.rar | 5.55 MB | E9E5D16BB76220706AD590B79FEC13B4 |
| Wrox Press - Professional XML Databases.pdf | 6.99 MB | 2A1FD17FC0105433522FB91A44B1D94F |
| Wrox - Professional XML Databases.zip | 6.97 MB | D1F9DFE601208053FA411052AA339867 |
| Wrox - Professional XML Databases 2000.zip | 6.96 MB | B6B752A64D37A3B42C0FFA6E891C3F76 |
| Wrox - Professional Xml Databases (Kevin Williams) Pdf.rar | 6.37 MB | 9DF491CB5DB01DF2376606710AF90DFB |

Figura 1.1: Consulta "*XML Databases*" em um sistema P2P[3]

O uso de funções de similaridade coloca um **segundo** desafio, que é o de determinar o valor de limiar que deva ser utilizado. Esta dificuldade ocorre porque a distribuição dos valores de escore gerados por funções de similaridade distintas pode ser completamente diferente – pode mesmo variar quando a mesma função de similaridade é aplicada a diferentes conjuntos de dados (STASIU et al., 2005; STASIU 2007a). Além disso, a definição do limiar é geralmente uma tarefa realizada pelo usuário, e tende a ser propensa a erros. O usuário necessita testar vários valores diferentes de limiar para obter um resultado satisfatório. É fundamental compreender o comportamento das funções de similaridade a fim de selecionar o melhor limiar. Conforme identificado nos experimentos apresentados em (STASIU et al., 2005) e também em (BILENKO, 2003), algumas funções de similaridade são mais adequadas do que outras, uma vez que os valores de revocação e de precisão tendem a ser mais altos e menos dependentes dos valores de limiares. A definição do limiar também depende do conjunto de dados e da qualidade esperada dos resultados. Um valor alto de limiar considera como versões somente pares de arquivos que apresentam alta similaridade (isto é, que são quase cópias idênticas). Por outro lado, um valor baixo de limiar considera como versões um grande número de pares de arquivos com baixa similaridade (isto é, documentos diferentes).

Para transpor os dois desafios discutidos nesta seção, esta tese propõe um mecanismo inovador para detecção de versões de documentos XML, baseada em classificadores *Naïve Bayesianos*. Desta forma, o problema de detecção é tratado como um problema de

[3] Alguns resultados da consulta obtidos com o sistema *eMule* (*www.emule-project.net/*)

classificação. Este mecanismo de detecção é parte do *DetVX*, um *framework* proposto nesta tese para detecção, gerenciamento e consulta de réplicas e de versões de documentos XML.

## 1.1 Objetivo e Contribuições

O objetivo da tese é a definição de um mecanismo automático para a detecção, o gerenciamento e a consulta a réplicas e a versões de documentos XML. Neste contexto, as principais contribuições do trabalho são:

- a definição de um mecanismo de detecção de réplicas baseado no uso de funções *hash*. O mecanismo de detecção trata réplicas como cópias idênticas, o que elimina a necessidade da análise de similaridade e a dependência do uso de limiares;
- a definição de um conjunto de funções de similaridade para arquivos XML, usado como base para a detecção de versões. As funções não são restritas a uma aplicação específica e podem ser adaptadas para considerar outras características relevantes de similaridade para outros cenários;
- a definição de um mecanismo de detecção de versões baseado em técnicas de classificação. O mecanismo de detecção baseia-se no uso de classificadores *Naïve Bayesianos*, o que elimina a dependência do uso de limiares;
- a definição de um *framework* onde o mecanismo de detecção de réplicas e de versões pode ser inserido. O *framework* também especifica os componentes usados no gerenciamento e na consulta de réplicas e de versões de documentos XML e pode ser estendido para cenários específicos. Nesta tese propõe-se a extensão do *framework* para ambientes P2P.

## 1.2 Validação

As três primeiras contribuições foram validadas através de um conjunto de experimentos:

**Definição de um mecanismo de detecção de réplicas baseado no uso de funções *hash*.** Com base em arquivos de entrada, foram calculados os resultados *hash*, visando a posterior detecção de réplicas.

**Definição de um conjunto de funções de similaridade para arquivos XML, usadas como base para a detecção de versões.** Com base em valores sintéticos de características[4] (gerados randomicamente), foram calculados valores de similaridade para diversos pares de arquivos, visando a posterior detecção de versões.

**Definição de um mecanismo de detecção de versões baseado em técnicas de classificação.** Com base nos valores de similaridade gerados no passo anterior, foi realizado

[4] Características, neste contexto, correspondem aos atributos (*features*) que são enviados ao classificador. Exemplos de características incluem *percentual de elementos que tiveram mudança de conteúdo*, *percentual de elementos que tiveram mudança de estrutura*, etc.

um conjunto de experimentos que analisaram a revocação e a precisão do mecanismo proposto.

**Definição de um *framework* onde o mecanismo de detecção de réplicas e de versões pode ser inserido.** Esta contribuição foi validada através da implementação de ferramentas e da realização de um conjunto de experimentos para avaliar a eficácia do mecanismo proposto e da sua respectiva implementação:

- *agrupamento e consulta a versões de documentos XML* - foi desenvolvida a ferramenta *XVersion*, uma ferramenta para geração e consulta ao arquivo que contém o histórico de versões de um documento XML;
- *geração de ontologias* - foi desenvolvida a ferramenta *Ontogen*, uma ferramenta para geração de ontologias a partir da integração de esquemas de documentos XML;
- *casamento entre ontologias e arquivos XML* - foi desenvolvida a ferramenta *The Matcher*, uma ferramenta para casamento entre ontologias e arquivos XML.

## 1.3 Organização do Texto

O texto é estruturado conforme segue:

- o capítulo 2, **Fundamentação Teórica,** apresenta a base conceitual que identifica o escopo de atuação da tese. Três assuntos são descritos: evolução e versionamento de documentos, similaridade entre arquivos, e técnicas de classificação de dados;
- o capítulo 3, **Trabalhos Relacionados**, apresenta o estado da arte dos trabalhos que tratam de temas relacionados à tese. Quatro assuntos são descritos: evolução de documentos, versionamento de documentos, similaridade entre arquivos e classificação de documentos;
- o capítulo 4, **Detecção de Réplicas e de Versões,** apresenta o mecanismo de detecção de réplicas e de versões de documentos XML proposto na tese. São descritos as funções de similaridade, a técnica de classificação utilizada e os experimentos realizados que validam o mecanismo proposto;
- o capítulo 5, ***Framework* para Detecção, Gerenciamento e Consulta a Réplicas e a Versões,** apresenta o *framework DetVX*. O *framework* é uma proposta de ambiente onde o mecanismo de detecção de réplicas e de versões pode ser inserido;
- o capítulo 6, **Estudo de caso: Aplicação do *Framework* em Sistemas P2P**, enfatiza as diferenças e as principais adaptações que devem ser realizadas no *framework DetVX*, visando a detecção, o gerenciamento e a consulta a documentos XML replicados e versionados em ambientes P2P;
- o capítulo 7, **Conclusões**, reforça as contribuições da tese, apresenta as publicações originadas, descreve o comparativo com os trabalhos relacionados e aponta trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

O objetivo geral da tese é a definição de um mecanismo automático para detecção de réplicas e de versões de documentos XML. Os principais itens relacionados à tese são descritos neste capítulo. São apresentados:

- as principais características de evolução e de versionamento de documentos, necessários ao entendimento do conceito de versão adotado na tese;
- os conceitos sobre similaridade entre documentos, necessários para a definição das funções de similaridade utilizadas pelo mecanismo de detecção de versões;
- o processo de classificação de dados e a técnica baseada nos classificadores *Naïve Bayesianos*, necessária para a definição do mecanismo de detecção de versões.

## 2.1 Evolução e Versionamento de Documentos

A característica de passar por atualizações é um aspecto fundamental de qualquer sistema de informações persistentes. Dados e esquemas tendem a evoluir ao longo do tempo, de forma a refletir alterações do mundo real, mudanças nos requisitos do usuário, correções de erros de projeto, ou mesmo para permitir manutenção incremental de um repositório. Muitos trabalhos foram desenvolvidos para adicionar tempo a modelos de Bancos de Dados (BD) e fornecer capacidades de armazenamento, consulta e atualização de dados históricos (TANZEL et al., 1993).

Bancos de dados temporais permitem armazenar todos os estados de uma aplicação (presentes, passados e futuros), registrando sua evolução com o passar do tempo. Para que isso seja possível, informações temporais são associadas aos dados armazenados, identificando quando a informação foi definida ou o tempo de sua validade. Neste contexto, duas dimensões são geralmente utilizadas: tempo de transação e tempo de validade. *Tempo de validade* é o tempo do mundo real, o qual denota o tempo em que um fato é verdadeiro na realidade. *Tempo de transação* é o tempo do sistema e denota o tempo durante o qual o fato está presente no banco como um dado armazenado (JENSEN et al., 1998).

Existem alguns elementos primitivos de representação temporal, dentre os quais são citados o *instante no tempo* e o *intervalo temporal*. O elemento primitivo *instante no tempo* representa um ponto no tempo em particular. Um *intervalo temporal* é caracterizado pelo tempo decorrido entre dois instantes (ou seja, é representando pelos dois instantes que o

delimitam). Um intervalo temporal é representado por [*t1*,*t2*], onde *t1* é o primeiro ponto do intervalo (limite inferior) e *t2* é o último (limite superior).

Duas abordagens principais tratam a questão da evolução:

- a primeira consiste em permitir modificações, sem manter o estado anterior à modificação;
- a segunda consiste em manter os estados anterior e posterior à modificação. Isto implica em gerenciar um conjunto de versões.

A derivação de versões pode seguir duas formas de versionamento. No versionamento linear, a evolução do documento cria uma seqüência linear de versões $V_1$, $V_2$, . . ., $V_j$, onde $V_j$ é a versão corrente. Já no versionamento ramificado, uma nova versão pode ser derivada a partir de qualquer outra versão prévia, criando uma árvore de versões.

O gerenciamento de versões consiste em manter as versões de cada documento XML e eficientemente recuperar versões passadas e presentes. Duas abordagens são observadas na literatura para o gerenciamento de versões:

- armazenar somente a última versão do documento e os *scripts* delta reversos para reconstruir versões passadas. Nesta abordagem, o custo de reconstruir versões anteriores aumenta à medida que o documento é modificado sucessivas vezes;
- armazenar todas as versões do documento. Nesta abordagem, o custo de armazenamento pode ser bastante alto.

Sistemas de gerenciamento de documentos utilizados no suporte à configuração de *software* e outras aplicações são baseados em dois esquemas de versionamento (CHIEN et al., 2001a):

- *Source Code Control System* (SCCS) (ROCHKIND, 1975) - esta abordagem insere operações no documento original e associa um par de rótulos temporais a cada segmento do documento para especificar seu ciclo de vida. Desta forma, versões são recuperadas a partir de uma busca no sistema de arquivos, recuperando os segmentos válidos com base em seus rótulos temporais;
- *Revision Control System* (RCS) (TICHY, 1985) - este esquema é baseado em edição, ou seja, utiliza *scripts* para representar mudanças nos documentos e para reconstruir incrementalmente outras versões. RCS armazena a versão mais atual, enquanto as versões anteriores são armazenadas como *scritps* reversos, os quais representam como retroceder na história do documento. Com exceção da versão corrente, é necessário realizar um processamento extra para recuperar versões anteriores.

A maioria destes sistemas compartilha algumas propriedades (RONNAU et al., 2005): os objetos básicos de controle de versão são arquivos; o objeto é de controle genérico de versões, independente do formato dos arquivos; os sistemas armazenam diferenças entre duas versões de um arquivo, utilizando deltas (processados através de algoritmos de detecção

de diferencas - *diff*[5]); deltas são utilizados para reconstruir versões, anteriores ou posteriores, dos arquivos; implementam controle de concorrência otimista, ou seja, os arquivos não são bloqueados na etapa de *checkout*, o que mais tarde pode acarretar na resolução de conflitos no arquivo.

A popularidade do XML tem tornado bastante importante o problema de possibilitar eficiente armazenamento e recuperação de documentos multiversionados[6]. No entanto, como descrito anteriormente, os sistemas tradicionais de controle de versões modelam documentos como uma seqüência de linhas de texto e utilizam *scripts* para representar diferenças entre as versões. Estes sistemas mostram-se inadequados para representar versões de documentos XML, uma vez que não preservam a estrutura lógica do documento original (CHIEN et al., 2001c).

## 2.2 Similaridade entre Arquivos

As funções de similaridade têm por finalidade atribuir uma medida do grau de semelhança entre dois objetos. A similaridade pode ser considerada em diferentes níveis de abstração (BERTINO et al., 2004): nível de dados (similaridade entre dados), nível de tipos (similaridade entre tipos, esquemas e modelos) ou entre níveis (similaridade entre dados e tipos). Avaliar a similaridade entre dados é importante para criar agrupamentos de informações relacionadas a um mesmo assunto. Avaliar a similaridade entre tipos é interessante em integração de esquemas (BATINI et al., 1986) e em agrupamento de esquemas (LEE et al., 2002). Avaliar a similaridade entre dados e tipos é relevante para identificar geradores de dados, e desta forma aplicar aos dados as propriedades específicas do seu tipo ou modelo.

Dados dois objetos *s1* e *s2*, a medida de similaridade é a função $sim(s1, s2) \rightarrow s$ que calcula um escore *s* para um par de valores *s1* e *s2*. Quanto mais alto o valor do escore, mais similares os dois valores *s1* e *s2* são entre si. Um escore igual a *zero* significa que os dois objetos analisados são totalmente diferentes, e um escore igual a *um* indica que são iguais.

As técnicas de análise de similaridade podem ser classificadas em (EUZENAT et al, 2007):

- *baseadas em cadeias de caracteres* - são freqüentemente usadas para realizar o casamento entre nomes. Estas técnicas consideram cadeias de caracteres como seqüências de letras em um alfabeto. Baseiam-se na seguinte intuição: quanto mais similares são as cadeias de caracteres, maior é a chance de representarem o mesmo conceito. Geralmente, funções de distância mapeiam um par de cadeias de caracteres para um número real, onde um número menor (pequena distância)

[5] Algoritmos *diff* podem ser utilizados para detectar um conjunto de diferenças entre documentos. Sua entrada é um par de documentos e sua saída é geralmente outro documento (chamado delta) que representa as diferenças entre os documentos de entrada.

[6] Um documento multiversionado é aquele que sofreu modificações ao longo do tempo e que possui várias versões.

indica uma grande similaridade entre as cadeias. Alguns exemplos de técnicas baseadas em cadeias de caracteres que são bastante usadas em sistemas de casamento por nome são *edit distance* (LEVENSHTEIN, 1966; HALL et al., 1980) e *N-gram* (NAVARRO, 2001). Tais métodos tipicamente encontram as classes *Book* e *Textbook* como similares, mas não as classes *Book* e *Volume*;

- *baseadas em recursos lingüísticos* - recursos lingüísticos, tais como dicionários ou *thesauri* específicos para um domínio, são usados para casar palavras baseadas nas relações lingüísticas entre elas (como, por exemplo, sinônimos, hipônimos, etc).
- *baseadas em grafos* - considera as entradas (como, por exemplo, documentos), como grafos rotulados. A comparação de similaridade entre um par de nós é baseada na análise de suas posições dentro do grafo. A idéia é que se dois nós de dois modelos são similares, então suas vizinhanças também devem ser similares.

Nesta tese foram usadas, principalmente, as técnicas baseadas em cadeias de caracteres, razão pela qual são descritas a seguir.

### 2.2.1 Técnicas de Análise de Similaridade com base em Cadeias de Caracteres

A seguir são descritas algumas das técnicas de análise de similaridade baseadas em cadeias de caracteres apresentadas em (EUZENAT et al., 2007):

- *técnicas de normalização* - reduzem as cadeias de caracteres que estão sendo comparadas a um formato comum. Exemplos de formatos comuns incluem:
  - a transformação de todos os caracteres para maiúsculo ou para minúsculo;
  - a remoção de múltiplos espaços (tabulações, etc) para um único espaço em branco;
  - a remoção de símbolos, tais como: ‘, -, ., e _. Exemplo: *peer-reviewed* passa a ser considerado como *peer reviewed*;
  - a remoção de dígitos. Exemplo: *Book123* passa a ser considerado como *Book*;
- *técnicas de igualdade de cadeias de caracteres* - retornam *zero* se as cadeias de caracteres não são idênticas, e *um* se são idênticas;
- *técnicas de sub-cadeias de caracteres ou subseqüências* - baseiam-se na similaridade de letras comuns entre as cadeias de caracteres;
- *técnicas de distância de edição* - o termo distância de edição entre dois objetos é definido como o custo mínimo de operações que pode ser aplicado a um dos objetos a fim de obter o outro. Pode-se associar um custo a cada operação de edição e, desta forma, a distância de edição fica definida como a soma dos custos de suas operações componentes (CHAWATHE, 1999). Estas técnicas foram projetadas para analisar a similaridade entre cadeias de caracteres que podem ter erros de grafia. As operações geralmente consideradas são inserção de um caractere, substituição de um caractere por outro, e remoção de um caractere. Exemplos: *Edit distance* (LEVENSHTEIN, 1966; HALL et al., 1980); *N-gram*

(NAVARRO, 2001); *Jaccard* (JACCARD, 1912); *Jaro* (JARO, 1989); *JaroWinkler* (WINKLER, 1999);

- *técnicas de comparação de caminhos* - calculam a similaridade entre dois caminhos, com base no número mínimo de operações sobre nomes de elementos necessárias para transformar um caminho no outro. As comparações entre nomes de elementos são efetuadas por funções de similaridade e os valores são utilizados na computação do valor final de similaridade entre os caminhos.

Uma classificação similar das funções de similaridade genéricas para dados do tipo texto é discutida também em (COHEN et al., 2003), que as categoriza em dois grupos: (i) as funções de similaridade baseadas em distância de edição, caractere a caractere, e (ii) as funções de similaridade baseadas em *token* (ou freqüência de termo). Como exemplo do primeiro grupo podem ser citadas *Edit distance* (LEVENSHTEIN, 1966; HALL et al., 1980), *N-gram* (NAVARRO, 2001), *Jaccard* (JACCARD, 1912), *Jaro* (JARO, 1989), *JaroWinkler* (WINKLER, 1999), entre outros. Como exemplo do segundo grupo, tem-se *TFIDF* (*Term Frequency-Inverse Document Frequency*) (SALTON et al., 1983) como uma função bem conhecida na área de recuperação de informação.

Estas técnicas são resumidas a seguir:

- *Edit distance* (LEVENSHTEIN, 1966; HALL et al., 1980) - esta função calcula o número mínimo de mudanças (inserção, exclusão e substituição) que são necessárias para fazer com que duas palavras ou seqüência de caracteres fiquem iguais;
- *Acronyms* (DORNELES et al., 2004) - esta função é útil para analisar compatibilidade entre siglas e acrônimos com a forma não abreviada, como, por exemplo, igualar “SBBD” com “Simpósio Brasileiro de Banco de Dados”;
- *Guth* (GUTH, 1976) - esta função é designada para detectar variações em nomes próprios;
- *N-gram* (NAVARRO, 2001) - o escore é calculado com base no número de caracteres que estão na mesma posição em cada *gram* (unidade mínima em que a seqüência de caracteres foi dividida);
- *Jaccard* (JACCARD, 1912) - esta função simples determina a similaridade entre duas cadeias de caracteres *s1* e *s2*. A similaridade entre duas cadeias de caracteres é definida como a razão das cardinalidades resultantes das operações de interseção e união;
- *Jaro* (JARO, 1989) - esta função baseia-se no número e na ordem de caracteres comuns entre dois termos ou seqüências de caracteres;
- *JaroWinkler* (WINKLER, 1999) - é uma variação da função *Jaro* que enfatiza a coincidência dos primeiros caracteres;
- *TF-IDF* (SALTON et al., 1983) - a idéia do *TF-IDF* é o uso de pesos para dar maior importância às palavras menos freqüentes. Para o casamento de uma seqüência de

caracteres, TF é a freqüência do termo desta seqüência e IDF é a freqüência inversa do termo em relação à coleção completa.

No contexto de XML, a necessidade de avaliar similaridade tem aumentado, principalmente por causa do número crescente de aplicações que necessitam recuperar, acessar e tratar documentos (BERTINO et al., 2004). No nível de dados, muitas abordagens têm sido propostas para medir a similaridade entre documentos XML com o objetivo de agrupar documentos relacionados a um mesmo assunto.

Observa-se que há uma variedade de funções de similaridade, sejam para valores atômicos (como por exemplo, *Levenshtein* ou *Edit Distance* (*Edit*) (NAVARRO, 2001), *Guth* (GUTH, 1976) e *N-grams* (NAVARRO, 2001)) ou para documentos (BAEZA-YATES et al., 1999; FLESCA et al., 2005; NIERMAN et al., 2002). No entanto, estas funções são comumente mais adequadas para interesses específicos. Por exemplo, algumas funções de similaridade para XML focam em estrutura, enquanto outras focam em conteúdo. No entanto, para o problema de detecção de versões, muitas características difererentes devem ser consideradas ao mesmo tempo, tais como a similaridade de estrutura, de conteúdo e a importância das modificações realizadas. Nesta tese, as funções propostas consideram diferentes características, sendo que tais características podem ter pesos diferentes, o que produz uma abordagem mais flexível para a análise de similaridade.

## 2.3 Classificação de Dados

Classificação é o processo de encontrar um modelo (ou uma função) que descreva e diferencie classes de dados ou conceitos, com o propósito de usar tal modelo para prever a classe de objetos para os quais o rótulo da classe é desconhecido. O modelo derivado é baseado na análise de um conjunto de dados de treinamento, isto é, objetos de dados para os quais o rótulo da classe é conhecido (HAN et al., 2006).

### 2.3.1 Processo de Classificação de Dados

Classificação é uma forma de análise de dados que pode ser usada para extrair modelos que descrevem classes de dados. Muitos métodos de classificação têm sido propostos nas áreas de aprendizado de máquina, reconhecimento de padrões e estatísticas. A maioria dos algoritmos reside em memória e assume pequenos conjuntos de dados. Pesquisas recentes em mineração de dados têm trabalhado no desenvolvimento de técnicas escaláveis de classificação, capazes de manipular grandes quantidades de dados residentes em disco (HAN et al., 2006).

A classificação de dados é um processo dividido em dois passos:

**Aprendizado**. É construído um classificador que descreve um conjunto pré-determinado de classes de dados ou de conceitos. Esta é a fase de aprendizado ou treinamento, onde um algoritmo de classificação constrói o classificador, analisando ou aprendendo a partir de um conjunto de treinamento de tuplas e suas respectivas classes. Uma tupla $X$ é representada por um vetor de atributos, $X = (x_1, x_2, ..., x_n)$, o qual descreve valores para $n$ medidas (isto é, atributos) da tupla. Cada tupla $X$ pertence a uma classe pré-definida, descrita no atributo *rótulo de classe*. O atributo *rótulo de classe* é um valor discreto e desordenado. Também é

categórico, no sentido em que cada valor serve como uma categoria ou classe. As tuplas que fazem parte do conjunto de treinamento são chamadas de tuplas de treinamento (HAN et al., 2006).

Uma vez que o rótulo de classe de cada tupla de treinamento é fornecido, este passo é também chamado de aprendizado supervisionado (isto é, o aprendizado do classificador é supervisionado no sentido em que é informada a classe a que cada tupla pertence). No aprendizado não supervisionado (por exemplo, agrupamento – *clustering*), o rótulo de classe de cada tupla de treinamento não é conhecido e o número de classes a serem aprendidas pode não ser conhecido no início do processo. Para os experimentos realizados na tese, considerou-se um conjunto de dados de treinamento rotulados, ou seja, para os quais já se conhecia a classe atribuída. Por estas razões, deste ponto em diante, são consideradas apenas técnicas de classificação de dados.

A etapa de aprendizagem do processo de classificação também pode ser vista como o aprendizado de um mapeamento ou função, $y=f(X)$, que prevê o rótulo de classe associado $y$ de uma dada tupla $X$. Nesta visão, deseja-se aprender uma função ou mapeamento que separa (isto é, classifica) os dados. Este mapeamento pode ser representado na forma de regras de classificação, árvores de decisão ou fórmulas matemáticas.

**Classificação.** O modelo é usado para classificação. Primeiramente, a precisão do classificador é estimada. A precisão de um classificador é dada pelo percentual de tuplas que foram corretamente classificadas. O rótulo de classe associada a cada tupla é comparado com a classe informada pelo classificador para aquela tupla. Se a precisão do classificador for considerada aceitável, o classificador pode ser usado para classificar futuras tuplas para as quais o rótulo da classe é desconhecido. Para a classificação de novas tuplas, é usado um conjunto de dados de teste, composto por tuplas de teste e seus respectivos rótulos de classes. Estas tuplas são escolhidas randomicamente a partir de um conjunto de dados e são independentes do conjunto de treinamento, ou seja, não devem ter sido usadas para construir o classificador.

Os seguintes passos de pré-processamento devem ser aplicados aos dados para melhorar a precisão, a eficiência e a escalabilidade do processo de classificação (HAN et al., 2006):

- *limpeza de dados* - refere-se ao pré-processamento de dados a fim de remover ou reduzir “ruído” dos dados ou tratar dados ausentes (por exemplo, preenchendo um valor ausente com o valor que ocorre mais vezes para aquele atributo);
- *análise de relevância* – muitos dos atributos podem ser redundantes ou irrelevantes. Para a eliminação de atributos redundantes, uma análise de correlação pode ser usada para identificar se dois atributos são relacionados estatisticamente. Para eliminar atributos irrelevantes, uma seleção de atributos pode ser usada para encontrar um conjunto reduzido de atributos, tal que a distribuição das probabilidades resultantes seja semelhante à distribuição se todos os atributos fossem usados;
- *redução e transformação de dados* – os dados podem ser transformados através da normalização. A normalização envolve escalar todos os valores de um dado

atributo em uma certa faixa de valores, por exemplo, de zero a um. Esta etapa é particularmente útil para atributos de valores contínuos.

Métodos de classificação podem ser comparados e avaliados de acordo com alguns critérios, tais como precisão (habilidade do classificador de corretamente informar o rótulo da classe das tuplas novas), velocidade (custo computacional envolvido na geração e no uso do classificador), robustez (habilidade do classificador em fazer classificações corretas, mesmo na presença de ruídos ou dados ausentes) e escalabilidade (habilidade de construir o classificador eficientemente para grandes conjuntos de dados).

Para esta tese, foram estudadas e avaliadas diversas técnicas de classificação (HAN et al., 2006), as quais incluem árvores de decisão, classificadores *Bayesianos*, classificação baseadas em regras e classificação por retropropagação. Classificadores *Naïve Bayesianos* foram escolhidos como parte da solução do mecanismo de detecção de versões. Apesar da premissa simplista de independência de atributos, o classificador reporta o melhor desempenho em várias tarefas de classificação, fenômeno este discutido em (McCALLUM et al., 1998) e em (CHAKRABARTI, 2002). Estudos comparativos entre algoritmos de classificação têm mostrado que o classificador *Naïve Bayesiano* é comparável em desempenho com os classificadores baseados em árvore de decisão e redes neurais. Estes classificadores têm mostrado alta precisão quando aplicados a grandes bases de dados (DOMINGOS et al., 1997; RISH, 2001; ZHANG, 2004). Além disso, têm uma taxa de erro mínima em comparação a outros classificadores (HAN et al., 2006).

### 2.3.2 Classificadores *Bayesianos*

Classificadores *Bayesianos* são classificadores estatísticos, os quais podem prever a probabilidade de uma dada tupla pertencer a uma determinada classe. Estes classificadores assumem que o efeito de um valor de atributo em uma classe é independente dos valores de outros atributos. Esta premissa é chamada de independência condicional de classes ou de independência entre atributos ($P(x_1,\ldots,x_k|C) = P(x_1|C)*\ldots*P(x_k|C)$) [26]. Se o *i-th* atributo é discreto, então $P(x_i|C)$ é estimado como a freqüência relativa dos exemplares terem $x_i$ como *i-th* atributo na classe *C*.

O classificador *Naïve Bayesiano* funciona da seguinte forma:

(1) Seja *D* um conjunto de tuplas de treinamento e seus respectivos rótulos de classe. Uma tupla X é representada por um vetor de atributos, $X = (x_1, x_2, ..., x_n)$, o qual descreve valores para *n* medidas (isto é, atributos) da tupla.

(2) Suponha que há *m* classes, $C_1, C_2, ..., C_m$. Dada uma tupla *X*, o classificador prevê a que classe *X* pertence, a partir da probabilidade mais alta de *X* pertencer a uma dada classe. O classificador prevê que a tupla *X* pertence a uma classe $C_i$ se e somente se:

$$P(X|C_i)P(C_i) > P(X|C_j)P(C_j) \text{ para } 1<=j<=m,\ j\neq i$$

onde o rótulo da classe prevista é $C_i$ desde que $P(X|C_i)P(C_i)$ é a máxima probabilidade.

(3) Como se assume a independência entre atributos, logo:

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) = P(x_1|C_i) * P(x_2|C_i) * \ldots * P(x_n|C_i)$$

Pode-se facilmente estimar as probabilidades $P(x_1|C)$, $P(x_2|C)$, ..., $P(x_n|C)$ a partir das tuplas de treinamento (onde $x_k$ refere-se ao valor de atributo $A_k$ para a tupla $X$). Para cada atributo $A_k$, verifica-se se o atributo é categórico ou contínuo. Se $A_k$ é categórico, então $P(x_k|C_i)$ é estimado como a freqüência relativa dos exemplares terem $x_k$ como *k-th* atributo na classe $C_i$. $P(x_k|C_i)$ é o número de tuplas na classe $C_i$ em $D$ que têm o valor $x_k$ para $A_k$, dividido por $|C_i,D|$ (número de tuplas da classe $C_i$ em $D$). Se o *k-th* atributo é contínuo, então $P(x_i|C_i)$ é estimado por uma função de densidade *Gaussian* (HAN et al., 2006).

## 2.4 Considerações Finais

Esta seção discutiu os principais conceitos necessários ao entendimento da tese. Foi apresentado o problema da evolução de dados e também foram discutidas algumas características de sistemas tradicionais de gerenciamento de versões. Como o objetivo da tese é propor um mecanismo de detecção de réplicas e de versões (independente do uso de limiares), foi discutido o processo de classificação de dados e citadas as principais técnicas encontradas na literatura.

O classificador *Naïve* Bayesiano é provavelmente o classificador mais utilizado em aprendizado de máquina (CHAKRABARTI, 2002). O classificador é denominado ingênuo (*naïve*) por assumir que os atributos são condicionalmente independentes, ou seja, a informação de um evento não é informativa sobre nenhum outro. Apesar desta premissa simplista, o classificador reporta o melhor desempenho em várias tarefas de classificação. Este fenômeno é discutido em (McCALLUM et al., 1998) e em (CHAKRABARTI, 2002). Estudos comparativos entre algoritmos de classificação têm mostrado que o classificador *Naïve Bayesiano* é comparável em desempenho com os classificadores baseados em árvore de decisão e redes neurais. Classificadores *Naïve Bayesianos* também têm mostrado alta precisão quando aplicados a grandes bases de dados (DOMINGOS et al., 1997; RISH, 2001; ZHANG, 2004). Na teoria, classificadores *Bayesianos* têm uma taxa de erro mínima em comparação a outros classificadores (HAN et al., 2006).

Outro benefício significativo dos classificadores *Bayesianos* é que eles podem classificar instâncias que apresentem valores com atributo desconhecido e nulo, os quais são simplesmente omitidos do cálculo de probabilidades (SILBERSCHATZ et al., 2006). Ao contrário, os classificadores de árvore de decisão não podem tratar de forma significativa situações em que uma instância a ser classificada possui um valor nulo para um atributo de particionamento usado para percorrer a árvore de decisão.

No contexto desta tese, o problema da classificação de dados é utilizado para a classificação de documentos. A classificação ou categorização de documentos corresponde à tarefa de atribuir um documento eletrônico a uma ou mais categorias (neste caso, *versão* e *não-versão*).

# 3 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados à tese, os quais incluem evolução e versionamento de documentos, similaridade entre arquivos, e classificação de documentos (realizado nesta tese com o propósito de detecção de versões).

## 3.1 Evolução de Documentos

O gerenciamento de tempo tem sido extensivamente estudado nas últimas décadas no contexto da pesquisa em BD (KATZ et al., 1987; OZSOYOGLU et al., 1995; CHAWATHE et al., 1998; TANZEL et al., 1993). Com o avanço dos dados semi-estruturados, incluindo XML, alguns modelos foram propostos para representar o carácter evolutivo das informações e permitir consultas (SU et al., 2001). Estas abordagens geralmente usam modelos de dados temporais e algumas extensões às linguagens de consulta padrão (GAO et al., 2004) ou funções definidas pelo usuário (WANG et al., 2003b) para recuperação histórica.

Chawathe et al. (1998) apresentam um modelo para representação de modificações em dados semi-estruturados e uma linguagem de consulta que permite acesso ao histórico das modificações. O modelo utilizado como base é o OEM (*Object Exchange Model*) (Goldman et al., 1996). O histórico de um banco OEM é representado como um banco DOEM (*Delta Object Exchange Model*). Uma linguagem de consulta, chamada *Chorel*, é apresentada, que permite consultar o histórico das informações armazenadas (CHAWATHE et al., 2000).

Em (LAMMEL et al., 2001) é proposto um mecanismo para adaptação de documentos XML às suas DTDs. A abordagem apresentada trata o problema de evolução de formatos baseados em XML. Para isso, duas idéias essenciais são discutidas. Primeiro, as mudanças nos formatos dos documentos são representadas como transformações básicas nas respectivas DTDs. Segundo, a migracão dos dados XML é induzida pelas transformações ocorridas na DTD. O artigo foca em conceitos de evolução de formatos, como transformações correspondentes, propriedades de transformações e expressividade para implementar estas transformações.

Su et al. (2001) propõem um *framework* para gerenciar a evolução de dados e esquemas XML. O *framework* fornece uma taxonomia de primitivas básicas de modificações. Estas primitivas são classificadas em dois grupos, mudanças de dados e mudanças de esquemas. Em mudanças de dados, a abordagem garante que o documento modificado é válido pela sua estrutura. Em mudanças de esquema, a abordagem garante que a nova estrutura é uma

estrutura válida e que todos os documentos XML são transformados para serem válidos de acordo com a nova estrutura.

O trabalho de (LÓSCIO, 2003) propõe uma abordagem incremental para o desenvolvimento da camada de mediação, com base na evolução dos esquemas das fontes locais e na evolução dos requisitos dos usuários. Desta forma, a adição de uma nova fonte de dados ao sistema não implica que as consultas da camada de mediação sejam completamente refeitas. Através da adoção de um formalismo para representar as consultas mediadas, é possível descrever as fontes de dados relevantes para processar uma entidade da camada de mediação. Esta representação em alto nível facilita a identificação das entidades que são afetadas por uma mudança de esquema e que devem, portanto, ser reescritas.

Wang et al. (2003b) propõem um mecanismo para representar o histórico de bancos de dados relacionais como documentos XML. Com base nos documentos publicados, pode-se especificar consultas temporais, expressas em *XQuery*, para recuperar dados históricos. Assim como os dados, o histórico dos esquemas também pode ser armazenado e consultado. Bry et al. (2004) introduz as linguagens *Xcerpt* e *Xchange*, e mostra como suas características as tornam bastante adequadas para consulta e atualização de dados convencionais e dados da Web Semântica. O artigo enfatiza que para um bom uso de dados representados na Web Semântica, é necessário fornecer linguagens para recuperação e evolução de dados.

Grandi et al. (2000) propõem o uso de uma *tag* de marcação *valid* para documentos XML/HTML, a fim de possibilitar visualização temporal em *browsers* utilizando-se XSLT. Em Gergatsoulis et al. (2003), é proposto um método baseado em dimensões para gerenciar mudanças em documentos XML, mas não é mostrado como as consultas temporais poderiam ser tratadas. Em Amagasa et al. (2000) e Amagasa et al. (2001) também é proposto um modelo de dados temporal para representar documentos XML. O suporte a consultas é feito via extensões a API DOM e à linguagem *XPath*. Dyreson (2001) propõe outra extensão da linguagem *XPath* para possibilitar consultas temporais. Em Gao et al. (2004) é proposta a linguagem τXQuery, uma extensão à linguagem *XQuery* com suporte à temporalidade. Em Wang et al. (2005b) utiliza-se *XQuery* para consultas temporais, mas sem extensões ao modelo de dados XML nem à linguagem de consulta.

Nesta tese, documentos XML podem evoluir ao longo do tempo. Tal evolução é caracterizada pela existência de diversos arquivos físicos correspondentes às diferentes versões do documento. Assume-se que cada arquivo possui um tempo de registro e um tempo de modificação. Rótulos temporais, representados por *time-start* (TS) e *time-end* (TE) são derivados a partir do tempo de modificação dos arquivos e utilizados para representar a evolução temporal dos documentos XML. *TS* e *TE* são representados como *instante no tempo*, conforme discutido na seção 2.1, mas juntos definem um *intervalo temporal*, caracterizando o tempo decorrido entre os dois instantes.

## 3.2 Versionamento de Documentos

Várias técnicas de versionamento foram propostas na área de BD, focadas geralmente em problemas como gerenciamento de tempo de transação em bancos temporais (OZSOYOGLU

et al., 1995), suporte de versões de artefatos em bancos orientados a objetos (KATZ et al., 1987) e gerenciamento de mudanças em dados semi-estruturados (CHAWATHE et al., 2000). No entanto, abordagens de versionamento para BD são diferentes das abordagens de versionamento para documentos. Sistemas de BD são projetados para suportar consultas mais complexas e assumem que a ordem dos objetos não é importante (CHIEN et al., 2001d). Em sistemas de gerenciamento de documentos, estas duas assertivas normalmente não são verdadeiras.

Alguns trabalhos foram localmente desenvolvidos pelo grupo de pesquisa em BD da UFRGS. O trabalho pioneiro do grupo de pesquisa foi o "Modelo de Versões", proposto por Golendizner em (GOLENDZINER, 1995), que estende um modelo de dados orientados a objetos, acrescentando conceitos e mecanismos que suportam a definição e a manipulação de objetos, versões e configurações. A preocupação com questões relativas à evolução de esquemas foi primeiramente abordada por (GALANTE, 1998) e (GALANTE et al., 1998), através da especificação de um modelo de evolução de esquemas conceituais para BD orientados a objetos com a utilização do conceito de versões. (GALANTE et al., 2005) definiu um modelo que utiliza os conceitos de tempo e de versão para permitir o gerenciamento da evolução dinâmica de esquemas em bancos de dados orientados a objetos. O resultado, o Modelo Temporal de Versionamento com suporte à Evolução de Esquemas (TVSE - *Temporal and Versioning Model for Schema Evolution*), é capaz de gerenciar o processo de evolução de esquemas em todos os seus aspectos: versionamento e modificação de esquemas, propagação de mudanças e manipulação de dados. O trabalho também contempla a parte de fundamentos da computação, apresentando uma especificação formal para a linguagem definida.

O conceito de versão é bem conhecido no gerenciamento de configuração de *software* (CONRADI et al., 1998; WESTFECHTEL et al., 2001). Há várias ferramentas existentes para controle de versões, como *CVS*[7], *CVSNT*[8] e *SubVersion*[9]. Estes sistemas geralmente modelam os documentos como seqüências de linhas de texto, armazenando a última versão e usando *scripts* reversos para recuperar as versões prévias. No entanto, estes sistemas não preservam a estrutura lógica do documento original (CHIEN et al., 2001d), o que é essencial no formato XML, e também não suportam consultas complexas. Algumas propostas específicas para controle de versões XML tratam desta questão (CHIEN et al., 2001a; GRANDI et al., 2005; WANG et al., 2005).

Há alguns trabalhos recentes em modelos XML temporais (SU et al., 2001; GRANDI et al., 2000), extensões a linguagens de consultas (GAO et al., 2004), bibliotecas temporais (WANG et al., 2003b) e controle de versões (CHIEN et al., 2001e). Alguns modelos foram propostos para representar o caráter evolutivo de dados XML (SU et al., 2001). A proposta de (WANG et al., 2003c) apresenta algumas técnicas para gerenciamento e consultas temporais de documentos multiversionados. A proposta consiste em representar as

[7] Disponível em: *http://www.nongnu.org/cvs*

[8] Disponível em: *http://www.march-hare.com/cvspro*

[9] Disponível em: *http://subversion.tigris.org*

sucessivas versões de um documento XML como outro documento XML que implementa um modelo de dados temporal. A abordagem utiliza algoritmos *diff* para processar os tempos de validade dos elementos no documento (WANG et al., 2005). Por fim, a proposta utiliza *XQuery* para formular consultas temporais nos documentos, a partir do uso de funções temporais que podem ser escritas pelo usuário.

Chien et al. (2001a) apresentam uma proposta para armazenamento e consulta a documentos XML versionados. É proposto o esquema de versionamento *SPaR* (*Sparse Preorder and Range*). A técnica de versionamento de documentos proposta utiliza rótulos temporais e números de nós duráveis para preservar a estrutura e o histórico dos documentos durante a evolução. Chien et al. (2001b) propõem um mecanismo de versionamento baseado em referências entre objetos. Este mecanismo preserva a estrutura lógica do documento entre versões, tornando possível a recuperação de versões e de consultas baseadas em conteúdo. O mecanismo de referências também permite que toda a história de um documento versionado possa ser representada como outro documento XML. Em Chien et al. (2001c) é proposta uma abordagem onde a estrutura do documento é preservada e seus sub-objetos são rotulados hierarquicamente com rótulos temporais para possibilitar reconstrução das versões passadas e da versão atual.

*Xyleme* (2001) apresenta uma proposta de um *warehouse* dinâmico para dados XML da Web, com suporte à avaliação de consultas, controle de mudanças e integração de dados. Para cada página da Web, o sistema proposto obtém *snapshots*. É possível versionar algumas páginas através de representações delta. Identificadores persistentes são atribuídos aos elementos dos documentos. Estes identificadores são utilizados para representar e controlar as modificações. Vagena et al. (2003) e Vagena et al. (2004) propõem um mecanismo para possibilitar consultas baseadas em expressões de caminho para documentos XML versionados. É proposta uma representação de documentos que mantém relacionamentos entre nós de documentos. Desta forma, consultas podem ser respondidas em diferentes versões dos documentos.

O trabalho de Grandi et al. (2005) apresenta um mecanismo para gerenciamento temporal de textos representados em XML. A abordagem apresenta um modelo temporal XML que permite representar a evolução dos documentos e suas respectivas versões através do uso de quatro dimensões temporais: tempo de publicação, tempo de validade, tempo de eficácia e tempo de transação. Este trabalho apresenta um modelo de dados multiversionados baseado em XML e define mecanismos para gerenciar estes documentos. Também é apresentado um protótipo da implementação realizada. O modelo proposto é aplicado em um domínio de normas e leis, e introduz metadados temporais para possibilitar consultas com base em restrições temporais.

Wang et al. (2003c) apresentam algumas técnicas para gerenciamento de documentos multiversionados e consultas temporais a estes documentos. A proposta consiste basicamente em: representar as sucessivas versões de um documento XML como outro documento XML que implementa um modelo de dados temporal e utilizar *XQuery* para expressar consultas sobre o conteúdo de uma determinada versão. O mecanismo proposto elimina a redundância entre versões, uma vez que nós idênticos são agrupados e representados por intervalos de versões. Simanovsky (2004) propõe um *framework* para evolução de esquemas de sistemas

relacionais com interface XML. O *framework* auxilia na transformação semi-automática de esquemas e dados, a fim de gerar uma nova versão deste conjunto. As operações básicas apresentadas permitem adição/remoção de atributo, adição/remoção de arcos, adição/remoção de vértices, e *merge* e *split* de vértices. Aplicando estas operações, pode-se gerar uma versão modificada de uma DTD.

No artigo de Wang et al. (2004) é mostrada uma abordagem para representar bancos de dados de tempo de validade, de tempo de transação e bitemporais baseada em um modelo de dados XML. Representando a história do banco em XML, consultas complexas em SQL podem ser mais facilmente expressas utilizando-se a linguagem *XQuery*. A proposta pode ser aplicada para a representação histórica de dados relacionais, documentos XML armazenados em bancos de dados nativos, e no gerenciamento de versões em arquivos. Vagena et al. (2004) propõem um mecanismo eficiente de armazenamento de documentos com versionamento ramificado, com um fator pequeno de replicação. Através de vários experimentos realizados, o artigo trata o problema de avaliação de consultas sobre os documentos.

A proposta de Santos et al. (2007), também desenvolvida localmente pelo grupo de pesquisa em BD da UFRGS, apresenta uma abordagem de gerenciamento da evolução de conteúdo de documentos XML, através da união dos conceitos de bitemporalidade e de versionamento. Através dos metadados adicionados aos documentos XML, é possível recuperar visões presentes e passadas sobre o documento em qualquer ponto no tempo. Também é proposta uma estensão a *XQuery*, permitindo o acesso a informações temporais e versionadas dentro do documento XML.

A abordagem de Wang et al. (2005) utiliza algoritmos *diff* para processar os tempos de validade dos elementos no documento. A saída gerada pelo algoritmo *diff* é utilizada para representar a história dos documentos em um modelo de dados temporal. Por fim, a proposta utiliza *XQuery* para formular consultas temporais nos documentos, a partir do uso de funções temporais que podem ser escritas pelo usuário. Ronnau et al. (2005) apresentam uma avaliação do estado da arte em algoritmos *diff* e identifica características destes algoritmos para o tratamento de documentos do pacote *office*. Uma API é apresentada para ser usada em sistemas de controle de versões destes documentos. A abordagem trata também da detecção de conflitos e *merging* semi-automático de documentos.

O trabalho de Chien et al. (2001a) propõe uma técnica de versionamento baseada em rótulos temporais e números de nós duráveis para preservar a estrutura e o histórico dos documentos durante a evolução. Vagena et al. (2004, 2003) propõem um mecanismo para possibilitar consultas baseadas em expressões de caminho para documentos XML versionados. Uma representacao de documentos também é proposta para manter os relacionamentos entre nós de documentos. Por fim, Grandi et al. (2005) apresenta um mecanismo para gerenciamento temporal de textos em XML através do uso de dimensões temporais e de metadados para possibilitar consultas com base em restrições temporais.

Nesta tese, versões de um mesmo documento são representadas em arquivos físicos separados. Com base na data de modificação destas versões, consegue-se representar a evolução histórica do documento. O acesso às versões pode ser feito de várias formas: consultando cada uma das versões, consultando várias versões (através de restrições

temporais nas datas de modificação destes arquivos) e consultando a versão consolidada (um novo arquivo físico que contém todo o histórico das versões). O gerenciamento destas versões é realizado pelo uso de metadados.

## 3.3 Similaridade entre Arquivos

Pesquisas existentes em detecção de mudanças podem ser usadas como base para a análise de similaridade. Algumas abordagens utilizam algoritmos *diff* para detectar diferenças entre arquivos (COBENA et al., 2002a; WANG et al., 2003). No entanto, resultados *diff* são um delta com pouca informação semântica relacionada à similaridade entre os arquivos. Outra possibilidade é analisar as representações em árvore, calculando a distância de edição, isto é, o custo mínimo para transformar uma árvore em outra, usando as operações básicas (como, por exemplo, *inserção* e *remoção*) (CHAWATHE, 1999). Da mesma forma, resultados da distância de edição em árvores não contêm informação semântica útil quanto ao nível de similaridade que possa ser usado para a detecção de versões. Além disso, algumas abordagens para a análise de similaridade consideram somente o conteúdo textual dos documentos (BAEZA-YATES et al., 1999), enquanto outras consideram a estrutura (FLESCA et al., 2005; NIERMAN et al., 2002).

O trabalho de Nierman et al. (2002) propõe um método para agrupar documentos XML de acordo com sua similaridade estrutural. Documentos XML são modelados como árvores rotuladas ordenadas. Desta forma, para medir a distância entre as duas árvores, utiliza-se a noção de *tree edit distance* (SELKOW, 1977; ZHANG et al., 1992). O algoritmo mede a similaridade estrutural entre documentos utilizando a *edit distance* entre as árvores de estruturas. No entanto, os autores argumentam que o custo desta abordagem é muito alto para aplicações práticas. Em Flesca et al. (2002), é proposta uma técnica baseada na idéia de representar a estrutura de um documento como uma *time series*, na qual cada ocorrência de uma *tag* corresponde a um dado impulso. Analisando as frequências da transformação de *Fourier* correspondente, pode-se calcular o grau de similaridade entre os documentos.

A proposta de Lian et al. (2004) agrupa sumários de grafos, os quais são bem menores que os documentos originais, e define uma métrica de similaridade com menor custo para ser processada. A abordagem propõe a idéia de um grafo de estruturas, o qual suporta uma métrica de distância entre documentos e conjuntos de documentos. Esta métrica produz o algoritmo de agrupamento. O grafo de estruturas de dois documentos é o conjunto de nós e arcos que aparecem no primeiro ou no segundo documento.

No trabalho de Dalamagas et al. (2004b), documentos XML são modelados como árvores rotuladas ordenadas. A abordagem propõe aplicar algoritmos de agrupamento utilizando distâncias que estimam a similaridade entre as árvores em termos de relacionamentos hierárquicos de seus nós. A proposta apresenta um *framework* para agrupar documentos XML por estrutura, explorando distâncias que estimam a similaridade entre árvores em termos dos relacionamentos hierárquicos de seus nós. Propõe-se o uso de sumários estruturais, representados como árvores. Estes sumários mantêm os relacionamentos estruturais entre os elementos de um documento XML, reduzindo repetição e aninhamento de elementos. É apresentado um algoritmo para calcular a *tree edit distance* e uma métrica de distância para estimar a similaridade estrutural entre os sumários de duas árvores.

Na proposta de Yang et al. (2005), utiliza-se uma representação para documentos XML chamada de *structured link vector model*. A proposta permite que elementos XML individuais tenham seus próprios pesos na similaridade total do documento e permite que a similaridade entre elementos também seja capturada. Wan et al. (2006) apresentam uma abordagem para medir a similaridade entre documentos XML, levando em consideração a semântica entre elementos XML. Nesta proposta, os documentos são processados como documentos não estruturados, através da remoção de *tags*. Para medir a similaridade entre documentos, o artigo propõe uma técnica chamada PTS, *Proportional Transportation Similarity*. PTS admite que o texto de um elemento de um documento corresponda ao texto de qualquer elemento de outro documento.

O trabalho proposto por Castano et al. (2000), aplicado em um contexto de integração de fontes Web, define um esquema global como uma visão integrada dos dados armazenados nas fontes locais. Para identificar classes semanticamente relacionadas, utiliza-se coeficientes de afinidade (valores entre 0 e 1) para cada par de classes, baseado nos relacionamentos terminológicos de um *Thesaurus*. Os coeficientes de afinidade determinam o grau de relacionamento semântico de duas classes baseado em seus nomes e em seus atributos. Um valor (ou peso) de afinidade é então atribuído, chamado de coeficiente global de afinidade. Estes coeficientes são usados por um algoritmo de agrupamento, que classifica as classes de acordo com o grau de afinidade, e produz uma árvore de afinidades. A partir deste ponto, os grupos para integração são selecionados da árvore de afinidades, usando um mecanismo de limiar. Para cada agrupamento selecionado na árvore, uma classe global (representativa das classes contidas no agrupamento) é definida.

Outro trabalho é apresentado em Jeong et al. (2001) para derivação de uma visão integrada de DTDs em um *framework* de integração de informações. Um dos passos da abordagem consiste no agrupamento de DTDs de domínios similares. Nesta etapa, tipos de elementos similares são unificados para melhorar o desempenho do processo de agrupamento. Este módulo usa um método de agrupamento aglomerativo para realizar os agrupamentos. Jeong et al. (2001) definem um método genérico de casamento de árvores para calcular a similaridade de DTDs para fins de agrupamento.

Outra abordagem é apresentada em Lee et al. (2002), a qual propõe definir agrupamentos de DTDs que sejam sintática e semanticamente similares. Para isso, propõe um algoritmo de casamento baseado em semântica, descendentes imediatos e similaridade no contexto. O processo de agrupamento é aplicado recursivamente às DTDs dos agrupamentos até que se tenha um número gerenciável de DTDs obtidas. Para gerar agrupamentos de DTDs, inicialmente gera-se o grau de similaridade entre DTDs. Para calcular a similaridade de DTDs, é necessário avaliar a similaridade entre elementos das DTDs. A partir disso, a similaridade entre duas DTDs pode ser obtida pelo somatório de similaridade dos elementos do melhor par de elementos correspondentes, e então se normalizando este resultado. Para agrupar DTDs em agrupamentos, a abordagem propõe o uso de uma técnica hierárquica, a qual começa com agrupamentos com uma única DTD e vai gradualmente adicionando DTDs similares a estes agrupamentos.

O trabalho de Francesca et al. (2003) aborda o problema de agrupamento de documentos XML com estruturas similares. A abordagem baseia-se em eleger um documento XML

representativo que contenha as características mais relevantes do conjunto de documentos XML do agrupamento. O casamento estrutural permite identificar similaridades estruturais entre dois documentos e construir um representante. O representante de um agrupamento de documentos XML é um documento XML que reflete todo o conteúdo estrutural dentro do agrupamento.

O trabalho de Dalamagas et al. (2004a) apresenta uma metodologia de agrupamento de documentos XML com estruturas similares, utilizando algoritmos de agrupamento. Documentos XML são modelados como árvores e desta forma a abordagem explora o problema de agrupamento de árvores, utilizando técnicas de distância que estimam a similaridade entre as árvores em termos de relacionamentos hierárquicos dos nós. A fim de melhorar o desempenho do cálculo da distância, esta abordagem propõe o uso de sumários estruturais.

Nesta tese, a similaridade entre arquivos é o ponto principal a ser analisado na detecção de versões. Para tal, são apresentados os requisitos de similaridade, descritos em funções de similaridade, que são analisados entre versões de um mesmo documento. Exemplos destes requisitos incluem similaridade de estrutura, similaridade de conteúdo e relevância das modificações entre versões. Para produzir os valores destes requisitos, são usadas algumas técnicas já existentes, como, por exemplo, similaridade de texto no conteúdo de um elemento XML, ou similaridade semântica, através do uso de um dicionário de dados.

## 3.4 Classificação de Documentos

Alguns classificadores foram usados em trabalhos anteriores para a classificação de documentos, tais como redes neurais (FARKAS, 1994), máquinas vetoriais (JOACHIMS, 1998), programação genética (SVINGEN, 1998) e classificadores *Naïve Bayesianos* (NIGAM et al., 1999). Conforme já discutido na seção 2.3, esta tese baseia-se no uso de classificadores *Naïve Bayesianos* para a detecção de versões. Há alguns trabalhos em classificação de documentos baseados em classificadores *Naïve Bayesianos* (WANG et al., 2003a; PON et al., 2007; DAI et al., 2007). Nestes trabalhos, o objetivo é atribuir um único documento eletrônico a uma categoria que seja a mais relevante.

Em (WANG et al., 2003a), é apresentado um sistema automático de classificação de documentos, chamado *WebDoc*, que classifica documentos Web de acordo com um esquema de classificação pré-definido. O sistema constrói uma base de conhecimento a partir dos dados de treinamento e então classifica novos documentos com base nas informações disponíveis na base. Um dos algoritmos utilizados é o classificador *Bayesiano*. O artigo apresenta um estudo de desempenho e mostra que a abordagem proposta produz resultados eficientes e eficazes na classificação de documentos Web.

Pon et al. (2007) baseia-se no pressuposto de que nem todos artigos considerados relevantes para uma dada consulta podem ser interessantes para o usuário. Por exemplo, se o artigo é antigo ou se traz pouca informação nova ao usuário, o artigo não seria interessante. Isso ocorre porque os escores de relevância não levam em consideração o que torna um artigo interessante. Para sanar estas deficiências, Pon et al. (2007) apresentam um *framework* que possibilita medir e definir o grau de interesse de artigos, através da participação do

usuário. No mecansimo proposto, várias características são extraídas, tais como relevância do tópico e reputação da fonte. Os experimentos avaliaram vários classificadores, inclusive os *Naïve Bayesianos*¸ com o objetivo de combinar estas características para encontrar um escore global de interesse do artigo. Através da participação do usuário, o classificador encontra características que são úteis na previsão do grau de interesse.

Vários artigos tratam sobre o bom desempenho dos classificadores *Naïve Bayesianos*. Em (ZHANG, 2004), é mostrado que o classificador é eficaz mesmo em situações com dependências entre os atributos. (KOTSIANTIS et al., 2004) reforçam as vantagens do uso dos classificadores *Naïve Bayesianos* em termos de precisão, simplicidade, velocidade de aprendizado e classificação, e espaço de armazenamento, mas também propõe um mecanismo de melhoria de desempenho deste classificador, baseado em discretização de valores e seleção de características.

Nesta tese, o objetivo é categorizar pares de arquivos em duas classes, isto é, *versões* e *não versões*. Há um conjunto de características que deve ser considerado ao se definir os requisitos necessários para que um arquivo seja considerado uma versão de outro arquivo. Esta tese define estes requisitos em funções de similaridade e aplica a técnica de classificação para a detecção das versões. O uso de técnicas de classificação elimina a dependência e as desvantagens do uso de limiares.

## 3.5 Considerações Finais

As seções 3.1 e 3.2 discutiram alguns trabalhos relacionados à evolução e versionamento de documentos XML. O problema da evolução de dados foi apresentado e algumas soluções existentes foram discutidas. Também as características de sistemas tradicionais de gerenciamento de versões foram apresentadas. Observa-se que as propostas encontradas na literatura focam na representação e acesso a versões já conhecidas pelo sistema (CHIEN et al., 2001c; XYLEME, 2001; GRANDI et al., 2003; WANG et al., 2003c; VAGENA et al., 2004; WANG et al., 2005), e não na detecão destas versões.

Além disso, as propostas para detecção de réplicas focam no problema de identificar múltiplas representações do mesmo objeto do mundo real (Weis et al., 2004; Weis et al., 2006). Estas representações podem ter diferença de conteúdo e de estrutura. No entanto, no contexto deste trabalho o termo *duplicata* ou *réplica* é utilizado para definir uma cópia idêntica de um documento XML. Logo, as abordagens encontradas na literatura não são adequadas.

A seção 3.3 discutiu alguns trabalhos relacionados à similaridade entre arquivos. A tese propõe um mecanismo para a detecção de réplicas e de versões, baseado em critérios de similaridade entre os documentos. Algoritmos *diff* podem ser utilizados para detectar um conjunto de diferenças entre documentos. No entanto, sua saída é geralmente um documento delta que representa as diferenças entre os dois documentos de entrada. Semanticamente, este conjunto de diferenças não traz muita informação em relação ao grau de similaridade entre os documentos analisados.

Os trabalhos discutidos baseiam-se em uma variedade de funções de similaridade, seja para valores ou para documentos. No entanto, estas funções são freqüentemente mais

adequadas para requisitos específicos. Por exemplo, algumas funções de similaridade para arquivos XML focam mais em estrutura do que em conteúdo. No entanto, para o problema de detecção de versões, muitas características diferentes devem ser consideradas ao mesmo tempo, como, por exemplo, estrutura, conteúdo, relevância das modificações, entre outras.

Nesta tese, duas versões de um mesmo documento podem ter diferenças estruturais e de conteúdo bastante significativas, o que inviabiliza o uso destas abordagens. Neste sentido, o trabalho aqui apresentado visa focar em duas deficiências observadas: detecção de réplicas e de versões de documentos XML, e gerenciamento das réplicas e das versões detectadas, de forma a permitir eficiente recuperação.

# 4 DETECÇÃO DE RÉPLICAS E DE VERSÕES

Neste capítulo é apresentado o mecanismo de detecção de réplicas e de versões proposto nesta tese. Para tal, são descritos: o mecanismo para a detecção de réplicas, baseado no uso de funções *hash*[10]; o mecanismo para a detecção de versões, baseado na similaridade entre os arquivos e no uso de classificadores; os experimentos realizados, que comprovam a eficácia do mecanismo proposto; e as ferramentas implementadas que avaliam as etapas da detecção.

## 4.1 Mecanismo para a Detecção de Réplicas

Assume-se que réplicas e versões de documentos XML são armazenadas em um repositório qualquer (como, por exemplo, em um diretório de arquivos). O termo "duplicata" ou "réplica" é utilizado para definir uma cópia idêntica de um arquivo XML. A detecção de réplicas verifica se há dois ou mais arquivos idênticos armazenados. Cópias idênticas podem ser facilmente identificadas com o uso de funções *hash* (RIVEST, 1992; RIVEST, 1992a; NSA, 2004; BARRETO et al., 2000). Para esta verificação, comparam-se os resultados *hash* dos arquivos existentes; se dois ou mais arquivos apresentarem o mesmo resultado *hash*, então se assume que estes arquivos são cópias idênticas (isto é, réplicas) de um mesmo documento.

A detecção de réplicas compara os resultados *hash* dos arquivos aos pares, seguindo uma combinação sem repetição. Uma combinação sem repetição, em análise combinatória, indica quantas variedades de subconjuntos diferentes com *s* elementos existem em um conjunto *U* com *n* elementos. De modo simples, indica quantos subconjuntos de *s* elementos diferentes em um conjunto de *n* elementos diferentes podem ser formados. A fórmula de cálculo de uma combinação é a seguinte:

$$C_s^n = \binom{n}{s} = \frac{n!}{s! \cdot (n-s)!}$$

onde *s* é dois (uma vez que os arquivos são comparados aos pares) e *n* é o número de arquivos de entrada. Por exemplo, em um conjunto com três arquivos *f1, f2 e f3,* são necessárias três comparações (*f1-f2*, *f1-f3*, *f2-f3*).

A função de comparação é definida como:

[10] Um *hash* é uma seqüencia de *bits* gerada por um algoritmo de dispersão. Essa seqüencia busca identificar um arquivo ou informação univocamente.

*v = compareFunction (hash(fm), hash(fn))*

onde *fm* e *fn* são arquivos do conjunto de arquivos a ser analisado e *v* é o valor retornado pela função de comparação (retorna *um* quando os valores *hash* forem idênticos e *zero*, caso contrário).

Alguns testes foram realizados utilizando-se o *Message-Digest Algorithm 5* (MD5) (RIVEST, 1992a), uma função criptográfica *hash* de 128 *bits*. O MD5 foi escolhido por ser um algoritmo rápido e largamente utilizado para verificação de integridade de conteúdo, suficiente para a maior parte das aplicações. Genericamente o MD5 mapeia uma cadeia de caracteres de comprimento variável numa pequena cadeia de caracteres encriptada e de tamanho fixo.

Como exemplo, considere um conjunto de arquivos XML contendo informações relacionadas a currículos, conforme ilustrados na Figura 4.1.

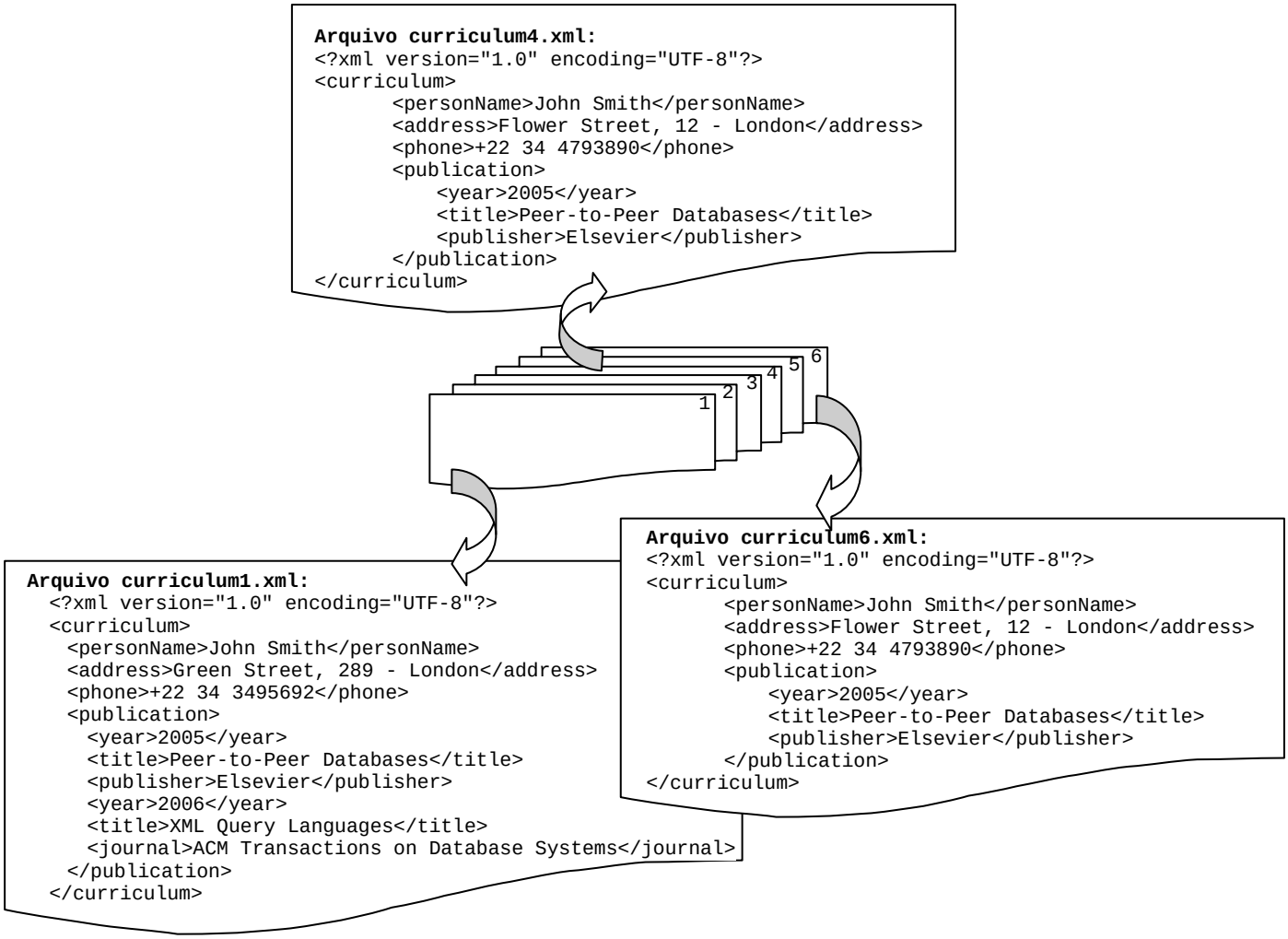



Figura 4.1: Arquivos XML "*curriculum1.xml*", "*curriculum4.xml*" e "*curriculum6.xml*"

Inicialmente, a função *hash* é aplicada a todos os arquivos. Apenas como ilustração, considerando os arquivos "*curriculum1.xml*", "*curriculum4.xml*" e "*curriculum6.xml*", a função *hash* retorna os seguintes valores (representados aqui em hexadecimal):

*hash* (*curriculum1.xml*) = 8165a11ae99f88dcc9a18a5fc7415929

*hash* (*curriculum4.xml*) = 0a39f4e268f66ba3a6976903851a90f2

*hash* (*curriculum6.xml*) = 0a39f4e268f66ba3a6976903851a90f2

A seguir, os resultados *hash* são comparados aos pares, a fim de identificar possíveis réplicas. O *hash* idêntico retornado pelo algoritmo MD5 demonstra que os arquivos "*curriculum4.xml*" e "*curriculum6.xml*" são réplicas. Os valores diferentes de *hash* do par de arquivos "*curriculum1.xml*" e "*curriculum6.xml*" demonstram que são documentos diferentes.

A implementacao em Java utilizada para a geração dos resultados *hash* mostrados nesta seção pode ser encontrada no Anexo A.

## 4.2 Mecanismo para a Detecção de Versões

O mecanismo de detecção de versões verifica se dois arquivos são versões do mesmo documento e baseia-se no uso de funções de similaridade. Uma função de similaridade $f(fm,fn) \rightarrow sim$ gera um escore *sim* para um par de arquivos *fm* e *fn*. Quanto maior o valor de *sim*, mais similares são os arquivos. Com base nos valores de similaridade, as versões poderiam ser detectadas com o uso de limiares: pares de arquivos para os quais o valor de *sim* é maior ou igual a certo limiar seriam considerados versões de um mesmo documento; pares de arquivos para os quais o valor de *sim* é menor que certo limiar seriam considerados documentos diferentes. Conforme já discutido no capítulo 1, este trabalho objetiva eliminar a dependência de limiares. Para isso, o mecanismo de detecção de versões não se baseia apenas no valor final de similaridade calculado para cada par de arquivos, mas sim em um conjunto de atributos (isto é, características) que deve ser avaliado pelo classificador.

O mecanismo proposto para a detecção de versões é composto por três atividades. A primeira atividade é chamada *análise de similaridade* e é responsável por aplicar uma função de similaridade em cada par de arquivos presente no repositório. A segunda atividade é a *classificação*, responsável por detectar versões, com base nos dados gerados na tarefa anterior. Finalmente, alguns metadados coletados durante o processo de detecção de versões são mantidos pela atividade de *gerenciamento de metadados*. Estes metadados, descritos na seção 4.2.3, descrevem informações relacionadas às versões detectadas, como por exemplo, o grau de similaridade entre cada par de versões detectado, e podem ser bastante úteis no processamento de consultas.

A função de similaridade produz um valor de similaridade a partir da combinação de um conjunto de valores de atributos calculados para o par de arquivos que está sendo analisado. Assume-se que o escore gerado é um valor discreto, variando dentro do intervalo limitado por [0,1], com variações de 0.01. Desta forma, há 100 possíveis valores diferentes para o resultado da função de similaridade. Estes valores de atributos são repassados para o classificador, o qual classifica os pares de arquivos como versões ou não versões, conforme

descrito na seção 4.2.2. Dado que um repositório pode possuir vários arquivos de um mesmo domínio de aplicação, há a necessidade de se decidir quais pares de arquivos precisam ser comparados para o processo de detecção de versões. A estrutura da árvore a ser varrida depende do tipo de versionamento usado, conforme discutido a seguir.

**Versionamento Linear.** No versionamento linear, o mecanismo de detecção compara pares de arquivos (como por exemplo, *f1* and *f2*), onde *f1* é a versão candidata e *f2* é a versão corrente de um documento existente. A versão corrente pode ser assumida por *default*, considerando-se a data de modificação do arquivo, ou seja, o arquivo que foi modificado mais recentemente é considerado a versão corrente. A versão candidata é o arquivo que se deseja analisar para verificar se é uma versão de algum outro arquivo já existente (como por exemplo, um arquivo que foi recentemente inserido ou atualizado).

Como exemplo, considere dois conjuntos de versões de arquivos, como mostrado na Figura 4.2: a primeira seqüência (isto é, *f1, f2* e *f3*) corresponde a três versões do documento *D1*; a segunda seqüência (isto é, *f4* e *f5*) corresponde a duas versões do documento *D2*. As versões correntes destes conjuntos são, respectivamente, *f3* e *f5*. O valor de similaridade entre arquivos aparece acima das setas[11].

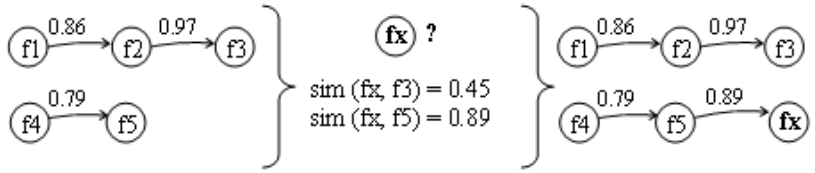



Figura 4.2: Versionamento linear

Desta forma, o arquivo analisado *fx* é comparado com o *f3* e *f5*, produzindo os valores de similaridade *sim (fx, f3) = 0.45* e *sim (fx, f5) = 0.89*. Se o mecanismo da tese tivesse como base apenas o uso de limiares, então o par de arquivos com o valor mais alto de similaridade seria considerado uma seqüência de versões lineares. No entanto, a proposta deste trabalho utiliza um classicador para a detecção de versões. Assim, os valores dos atributos considerados na função de similaridade é que são repassados ao classificador, de tal forma que o mecanismo de detecção analisa não somente o valor *sim*, mas também os valores gerados para os atributos que compõem a função de similaridade (conforme detalhado nas seções a seguir). No exemplo da Figura 4.2, o mecanismo de detecção de versões classifica o arquivo *fx* como uma versão do arquivo *f5*.

Se o arquivo *fx* não for versão de nenhum outro arquivo já existente, então este passa a ser o nó inicial de uma seqüência de derivações de versões para um novo documento, a partir do qual outras versões podem ser derivadas. Um arquivo não é considerado versão de outro arquivo se o mecanismo de classificação atribuir a classe “não-versão”.

[11] Os valores dos atributos (isto é, características) calculados entre cada par de arquivos comparado não são apresentados na Figura 4.2, por motivos de clareza. Estes atributos são discutidos na seção 4.2.1.

**Versionamento Ramificado.** Quando é usado o versionamento ramificado, o mecanismo de detecção compara pares de arquivos (por exemplo, *f1* e *f2*), onde *f1* é a versão candidata e *f2* é qualquer versão existente na árvore.

Considere a árvore de versões mostrada na Figura 4.3. Assuma que esta árvore corresponde a cinco versões de um documento *D1*. De forma similar ao exemplo ilustrado na Figura 4.2, o arquivo candidato *fx* é comparado com um conjunto das versões existentes. Seguindo a definição do versionamento ramificado, o arquivo candidato *fx* pode ser derivado de qualquer versão prévia (isto é, *f1*, *f2*, *f3*, *f4*, ou *f5*). Desta forma, em uma solução ótima, *fx* deve ser comparado com todos os nós da árvore. Há diferentes abordagens para percorrer a árvore e escolher o conjunto e a ordem dos arquivos a comparar, como descrito a seguir:

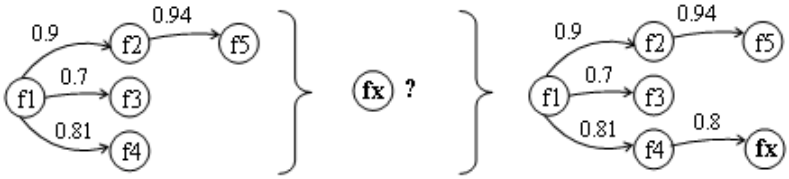



Figura 4.3: Árvore de versões ramificadas

1. comparar *fx* com todos os nós, seguindo uma busca em profundidade ou largura. Esta abordagem é simples e ótima, uma vez que *fx* é comparado com todas as versões prévias. No entanto, pode ter um alto custo dependendo do número de nós da árvore;
2. comparar *fx* com todos os nós folha da árvore. Esta abordagem não é ótima, uma vez que o arquivo candidato não é comparado com os nós intermediários. No entanto, esta abordagem apresenta um custo mais baixo, uma vez que somente um subconjunto da árvore é considerado. A adoção desta abordagem tende a linearizar a árvore de ramificações, uma vez que novas versões serão sempre derivadas das folhas;
3. comparar *fx* com todos os nós, seguindo uma busca em profundidade ou largura em ordem reversa (isto é, partindo-se das folhas), e finalizando quando atingir uma condição de parada. Uma condição de parada pode ser: tempo de processamento, grau mínimo de similaridade exigido pelo usuário, número de nós visitados, ou número de *K*-nós visitados (onde *K* é o número de arquivos que foram modificados mais recentemente). Esta abordagem não é ótima, mas tem um menor custo associado.

Uma destas três abordagens deve ser escolhida com base no comportamento de evolução dos documentos para uma determinada aplicação. Por exemplo: se as versões tendem a evoluir a partir das últimas versões geradas, a segunda abordagem é recomendada, uma vez que os nós folha representam as últimas versões criadas em um ramo. Se as versões tendem a evoluir a partir da última versão modificada, então a terceira abordagem é recomendada (para $K$=1). No entanto, se o documento não segue um comportamento padrão de evolução e o usuário deseja uma solução ótima, então a primeira abordagem é recomendada. Além disso, uma combinação de abordagens pode ser utilizada para aplicações específicas.

### 4.2.1 Análise de Similaridade

A etapa de análise de similaridade é responsável por gerar um valor de similaridade *sim* para um par de arquivos. Considere o conjunto $Rs = \{x \in R: x >= 0 \ \wedge x <=1\}$ como o conjunto de todos os valores reais em $R$ no intervalo [0,1]. A função de similaridade *s* recebe um conjunto de pares de arquivos *P* e gera um valor de similaridade no intervalo *Rs*: $s: P \rightarrow Rs$. Seja $G = \{(f1, f2), \ldots, (fm, fn)\}$ um conjunto de pares de arquivos. O valor de similaridade para todos os pares pode ser computado por $\forall (fm, fn) \in G : s(fm,fn)$.

Os tipos básicos de evolução que são considerados para a análise de similaridade são:

- *evolução de conteúdo* - o conteúdo do elemento modifica-se entre versões, mas a estrutura do elemento permanece a mesma. O exemplo da Figura 4.4 mostra que o elemento *x* muda seu conteúdo da versão (a) para a versão (b). Em termos de implementação, considera-se que o elemento *x* foi atualizado;
- *evolução de estrutura* - a estrutura do elemento modifica-se entre versões, mas o conteúdo permanece o mesmo. O exemplo da Figura 4.4 mostra que o elemento *x* muda sua estrutura da versão (a) para a versão (c). Em termos de implementação, considera-se que o elemento *x* foi removido e o elemento *z* foi adicionado;
- *evolução de estrutura e conteúdo* - o conteúdo e a estrutura modificam-se entre versões. O exemplo da Figura 4.4 mostra que o elemento *x* muda sua estrutura e conteúdo da versão (a) para a versão (d). Em termos de implementação, considera-se que o elemento *x* foi removido e os elementos *y* e *z* foram adicionados.

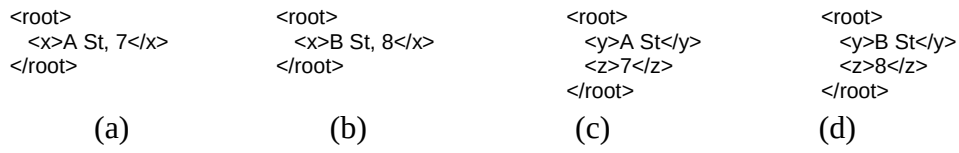



Figura 4.4: Diferentes representações em versões de documentos XML

Evolução de *estrutura* e de *conteúdo e de estrutura* são tratadas num mesmo grupo. Desta forma, são definidas funções de similaridade para o grupo de evolução de conteúdo e para o grupo de evolução de conteúdo e de estrutura.

#### *4.2.1.1 Características Analisadas na Função de Similaridade para Evolução de Conteúdo*

Considera-se o termo “correspondente” (*matched*) para se referir a um elemento que tem conteúdo idêntico em ambos os arquivos (por exemplo, *name*); “não correspondente” (*unmatched*), caso contrário (por exemplo, *salary*). A fim de avaliar a similaridade de conteúdo entre arquivos que seguem um padrão de evolução de conteúdo, as características (isto é, atributos) a seguir são observadas: elementos correspondentes, elementos não correspondentes e importância das modificações.

**Primeira característica: elementos correspondentes.** Para avaliar o grau de similaridade de conteúdo entre dois arquivos, considera-se primeiramente o uso de um algoritmo *diff*[12]. O algoritmo *diff* retorna as diferenças entre os dois arquivos em uma representação delta, representada como um conjunto de operações básicas de edição. Utilizando um algoritmo *diff*, as diferenças entre os arquivos são detectadas e analisadas, possibilitando desta forma determinar o percentual de elementos que não sofreram alterações na segunda versão.

Suponha os arquivos *f1* e *f2* mostrados na Figura 4.5. Uma vez que apenas o conteúdo modificou-se nestes arquivos, o número de elementos permanece o mesmo. No exemplo, cada arquivo tem seis elementos (isto é, o elemento raiz *employee* tem 6 descendentes diretos e indiretos).

**Arquivo f1:**

```
<employee>
        <name>Marcos</name>
        <hiringDt>10/10/03</hiringDt
        <job>engineer</job>
        <salary>3700</salary>
        <address>7 St</address>
        <phone>65982541</phone>
</employee>
```

**Arquivo f2:**

```
<employee>
        <name>Marcos</name>
        <hiringDt>10/10/03</hiringDt>
        <job>manager</job>
        <salary>4900</salary>
        <address>7 St</address>
        <phone>65982541</phone>
</employee>
```

Figura 4.5: Arquivos XML com mudanças no conteúdo dos elementos

Como mostrado na Figura 4.6, o conteúdo dos elementos *salary* e *job* não correspondem no segundo arquivo (ou seja, sofreram alteração na segunda versão). Em outras palavras, 67% dos elementos originais não se modificaram no segundo arquivo (4 elementos dentre os 6 filhos existentes). A idéia é que quanto maior a percentagem de elementos que não se modificam, maior é a chance destes arquivos serem versões do mesmo documento.

```
<delta><Deleted update="yes"  pos="0:0:3:0">3700</Deleted>
    <Deleted update="yes" pos="0:0:2:0">engineer</Deleted>
    <Inserted update="yes"  pos="0:0:2:0">manager</Inserted>
    <Inserted update="yes" pos="0:0:3:0">4900</Inserted>
</delta>
```

Figura 4.6: Resultado *diff* para os arquivos *f1* e *f2*

Para este trabalho foi utilizado o algoritmo *XyDiff* (COBENA et al., 2002a), embora a arquitetura permita a escolha de outras implementações de algoritmos *diff*. *XyDiff* é muito eficiente em termos de velocidade e espaço de memória; além disso, o algoritmo também considera a operação *move*, operação essencial para o tratamento de sub-árvores, além das tradicionais operações de inserção, remoção e atualização. De acordo com (DULUCQ et al., 2003), a complexidade deste algoritmo não é mais que *O(n*log(n))*.

**Segunda característica: elementos não correspondentes.** Outra característica interessante é analisar o quanto mudaram os elementos que sofreram algum tipo de modificação. A idéia é que quanto mais similares são os respectivos elementos não correspondentes, maior é a chance de serem versões do mesmo documento.

[12] Outras possíveis abordagens são discutidas nao seção 2.2.

Analisando os elementos *salary* and *job* e utilizando uma (combinação de) função (ões) de similaridade de cadeias de caracteres, pode-se calcular um valor que demonstre o quão similares são estes elementos. Cabe ressaltar que o objetivo deste trabalho não é descrever funções de similaridade de cadeias de caracteres, sendo que uma boa análise de funções existentes pode ser encontrada em (SILVA et al., 2007).

**Terceira característica: importância das modificações.** Alguns conceitos do domínio podem sofrer modificações mais freqüentemente que outros. Considere, por exemplo, um elemento *address*. Dois endereços diferentes podem facilmente se referir a uma mesma pessoa; no entanto, duas datas de nascimento distintas sugerem que são dois objetos diferentes no mundo real. Em outras palavras, a importância da mudança recebe pesos diferentes, dependendo do conceito que se está analisando. Nesta característica, usa-se a média das importâncias para calcular a similaridade. A idéia é que quanto menor a importância da modificação, maior é a chance dos arquivos serem versões do mesmo documento. A definição dos valores dos pesos é baseada no método de *Processo Analítico Hierárquico*, descrito na seção 4.2.1.5.

#### *4.2.1.2 Função de Similaridade Definida para Evolução de Conteúdo*

Com base nas três características apresentadas, a função de similaridade *simC* entre dois arquivos *f1* e *f2* é definida como:

$$simC(f1,f2) = (w_1*F_1 + w_2*F_2 + w_3*F_3 + ... + w_n*F_n)$$

onde $w_n$ é um fator que pesa a importância de uma característica $F_n$. Um fator pode ser positivo ou negativo (se influencia o aumento ou a redução da similaridade, respectivamente). Considerando $w_x, w_{x+1},...w_y$ como fatores positivos e $w_z, w_{z+1},...w_q$ como fatores negativos, assume-se que $w_x + w_{x+1} +...+w_y = 1$ e $-1<=w_z + w_{z+1} +...+ w_q<=0$, onde o somatório dos pesos dos fatores positivos deve ser igual a 1 e o somatório dos pesos dos fatores negativos deve ser entre 0 e -1.

As três características são combinadas para produzir a seguinte função de similaridade para evolução de conteúdo:

$$simC(f1,f2) = w_1*P + w_2*S + w_3*R$$

onde: *P* é a percentagem de elementos correspondentes, *S* é a média de similaridade dos elementos não correspondentes, e *R* é a média das importâncias das modificações dos elementos não correspondentes. Os fatores *P* e *S* ($w_1$ e $w_2$, respectivamente) são valores positivos (quanto maior estes valores, mais similares são os arquivos) e o fator *R* ($w_3$) é um valor negativo (quanto menor este valor, menor é a importância da modificação e mais similares são os arquivos). Os fatores ($w_1, w_2,..., w_n$) são definidos com base na importância das três características para cada aplicação e também dependendo das medidas esperadas para revocação e precisão.

O intervalo das variáveis *P*, *S* e *R* é definido dentro do intervalo [0,1] (ou seja, $\{P|P \in [0,1]\}$, $\{S|S \in [0,1]\}$, $\{R|R \in [0,1]\}$). Analisando os valores mínimo e máximo de *P*, *S* e *R*, e as restrições de soma para os fatores positivos e negativos, conclui-se que a função de similaridade produz um valor *simC* que varia de -1 a 1, isto é, $\{simC|simC \in [-1,1]\}$.

Para calcular *P*, usa-se a função *calcP* que retorna o percentual de elementos correspondentes, com base no resultado *diff*. *S* é calculado pela (combinação de) função (ões) de similaridade de *strings (StrSim())* e é definido como a média de valores de similaridade dos elementos não correspondentes (*ue*). Considerando-se que o número de elementos não correspondentes é denotado por *t*, a função é detalhada como segue:

$$simC(f1,f2) = w_1 * calcP(diff(f1,f2)) + w_2 * \frac{\sum_{x=1}^{t} StrSim(ue1_x, ue2_x)}{(t)} + w_3 * \frac{\sum_{x=1}^{t} R(ue_x)}{(t)}$$

Considere que *P*, *S* e *R* têm a mesma importância para uma data aplicação (isto é, $w_1=0.5$, $w_2=0.5$ e $w_3=-0.5$). A Figura 4.7 mostra a distribuição dos valores da função de similaridade. Os valores da função não estão uniformemente distribuídos. Para uniformizar tais valores, os *m* resultados da função são ordenados e mapeados para *n* classes. O mapeamento, representado em uma tabela de transformação, classifica *m/n* membros em cada classe. Uma vez que são possíveis 100 diferentes valores de similaridade, esta transformação gera *0.01*m* membros em cada classe.

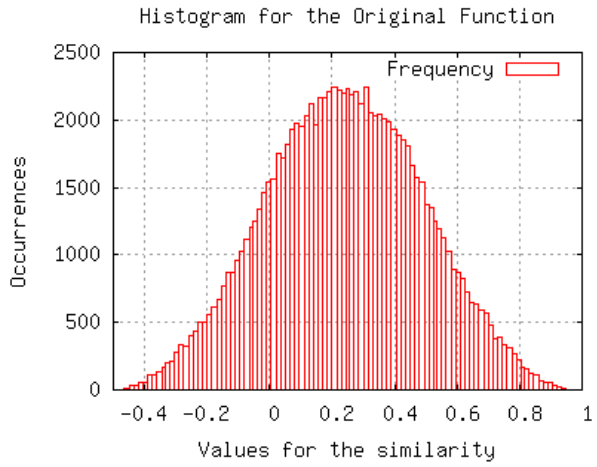



Figura 4.7: Histograma da função original

A Figura 4.8 mostra a distribuição dos valores uniformizados. Foram gerados 1.000.000 de valores de acordo com a função original, usando 0.5, 0.5 e -0.5 como pesos, e agrupados em 100 classes. Estas classes foram mapeadas para valores $\in [0,1]$, a fim de uniformemente distribuir os valores da função. Para garantir que o mapeamento foi feito corretamente, outros 100.000 valores foram gerados e mapeados para a tabela.

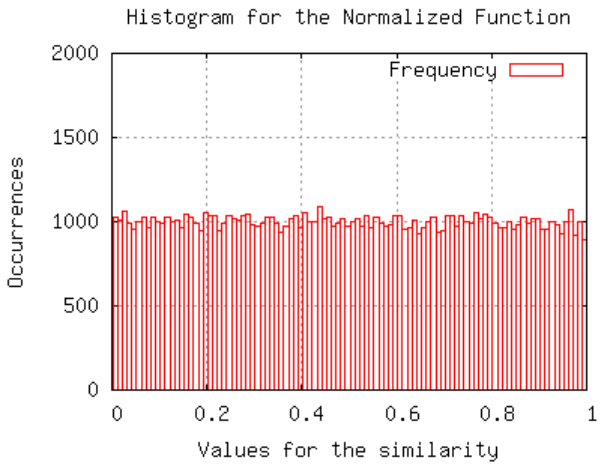



Figura 4.8: Histograma para a função normalizada

#### *4.2.1.3 Características Analisadas na Função de Similaridade para Evolução de Conteúdo e de Estrutura*

Além da primeira (elementos correspondentes) e terceira (importância das modificações) características discutidas na seção anterior, outra característica deve ser considerada, conforme apresentado a seguir.

**Elementos adicionados e removidos.** As diferenças entre os arquivos são detectadas utilizando-se um algoritmo *diff*. Analisando-se os arquivos e o resultado *diff*, pode-se observar o número de elementos adicionados e removidos do primeiro para o segundo arquivo. Seja "adicionados" (*added*) o termo usado para denotar os elementos novos (por exemplo, *address*) e seja "removidos" (*deleted*) o termo usado para denotar os elementos removidos (por exemplo, *job*). Os conceitos adicionados e removidos são similares às idéias apresentadas em (BERTINO et al., 2004), o qual considera elementos *plus*, *minus* e *common* para medir a similaridade entre um documento e uma DTD.

Como exemplo, considere dois arquivos *f3* e *f4*, mostrados na Figura 4.9.

**Arquivo f3:**

```
<employee>
        <name>Marcos</name>
        <hiringDt>10/10/03</hiringDt>
        <job>engineer</job>
        <salary>3700</salary>
        <phone>65982541</phone>
</employee>
```

**Arquivo f4:**

```
<employee>
        <name>Marcos</name>
        <salary>4500</salary>
        <address>7 St</address>
        <phone>65982541</phone>
</employee>
```

Figura 4.9: Arquivos XML com mudanças de conteúdo e estrutura

Analisando-se os arquivos e o resultado *diff* (apresentado na Figura 4.10), pode-se observar que *f4* adicionou um elemento (*address*) e removeu dois elementos (*job* e *hiringDt*).

```
<delta>
    <Deleted update="yes"  pos="0:0:3:0">3700</Deleted>
    <Deleted update="yes" pos="0:0:2:0">engineer</Deleted>
    <Inserted update="yes"  pos="0:0:2:0">manager</Inserted>
    <Inserted update="yes" pos="0:0:3:0">4900</Inserted>
</delta>
```

Figura 4.10: Resultado *diff* para os arquivos *f3* e *f4*

Como o algoritmo *XyDiff* considera a operação *move*, os conceitos adicionados referem-se somente a novos elementos; ações de movimento não são identificadas como adição e remoção.

Para evolução de conteúdo e de estrutura, o conceito "correspondente" é redefinido em relação à seção anterior. Aqui, o termo correspondente refere-se a um elemento que tem a mesma estrutura e o mesmo conteúdo em ambos os arquivos (por exemplo, *name* e *phone*). O termo não correspondente ainda é usado para denotar mudanças no conteúdo do elemento (por exemplo, *salary*), uma vez que mudanças na estrutura são classificadas como elementos adicionados e removidos.

#### *4.2.1.4 Função de Similaridade Definida para Evolução de Conteúdo e de Estrutura*

A função usada para processar o valor de similaridade para evolução de conteúdo e de estrutura é descrita como segue:

$$simE(f3,f4) = simC(f3,f4) + w_4*A + w_5*D$$

onde *simC* é o valor de similaridade de conteúdo entre os elementos comuns (isto é, que não tiveram alteração de estrutura), *A* é a percentagem de elementos adicionados e *D* é a percentagem de elementos removidos. Os pesos para *A* e *D* ($w_4$ e $w_5$, respectivamente) são valores negativos (quanto menores estes valores, mais similares são os arquivos).

O domínio das variáveis varia no seguinte intervalo: $\{A|A \in [0,1]\}, \{D|D \in [0,1]\}$. Analisando-se os valores mínimos e máximos de *simC*, *A*, *D*, e as restrições de soma para fatores positivos e negativos, conclui-se que a função de similaridade produz um valor *simE* $\in [-3,2]$. Para calcular *A*, usa-se uma função *calcA* que retorna a percentagem de elementos adicionados, com base no resultado *diff*. Para calcular *D*, usa-se uma função *calcD* que retorna a percentagem de elementos removidos, também com base no resultado *diff*. Desta forma, a função de similaridade é detalhada como segue:

$$simE(f3,f4) = simC(f3,f4) + w_4*calcA(diff(f3,f4)) + w_5*calcD(diff(f3,f4))$$

Semelhante à função de evolução de conteúdo, os valores desta função também não são uniformemente distribuídos. O processo detalhado na seção anterior é aplicado novamente.

#### *4.2.1.5 Definição dos Valores dos Pesos*

O Processo Analítico Hierárquico (AHP) (SAATY, 1990) é uma técnica de tomada de decisão para avaliar um conjunto de diferentes alternativas de solução para um dado problema. Este processo tem como objetivo encontrar uma solução ótima usando análise de decisão qualitativa e quantitativa. O AHP é composto por cinco passos: (a) definição do problema e seus objetivos, (b) representação hierárquica, (c) estimativa das prioridades, (d) síntese, e (e) análise de consistência dos resultados.

A *definição do problema e seus objetivos* estabelecem o contexto no qual o processo de tomada de decisão irá ocorrer. Após este passo, o problema é *hierarquicamente representado*, com os objetivos tendo diversos critérios associados ($C_1$, ..., $C_n$) e cada critério tendo diversas alternativas ($A_1$, ..., $A_n$).

No passo de *estimativa de prioridades*, o desenvolvedor define as prioridades para os critérios e alternativas. A importância relativa de cada critério sobre os demais e cada alternativa sobre as demais é representada usando comparações par-a-par. A escala na Tabela 4.1 é usada para expressar numericamente a importância relativa entre os critérios e as alternativas.

Tabela 4.1: Valores numéricos de cada importância relativa

| **Valor** | **Importância Relativa** |
|---|---|
| 1 | Mesma Importância |
| 2 | Levemente mais importante |
| 3 | Fracamente mais importante |
| 4 | Fracamente a moderadamente mais importante |
| 5 | Moderadamente mais importante |
| 6 | Moderadamente a fortemente mais importante |
| 7 | Fortemente mais importante |
| 8 | Muito mais importante |
| 9 | Absolutamente mais importante |

Neste processo, uma matriz par-a-par pode ser criada, contendo a importância relativa de cada um dos critérios sobre os demais. Considere, por exemplo, um conjunto de critérios $C=\{c_1, \ldots, c_n\}$ e uma matriz $M$ representando a importância relativa de cada critério sobre os demais $W=\{w_{11}, w_{12}, \ldots, w_{nn}\}$. A matriz de comparação par-a-par neste caso, é construída como segue:

$$\mathcal{M} = \begin{bmatrix} 1 & w_{12} & \cdots & w_{1n} \\ 1/w_{21} & 1 & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1/w_{n1} & 1/w_{n2} & \cdots & 1 \end{bmatrix}$$

Por exemplo, considere três critérios $c_1$, $c_2$, $c_3$, onde $c_1$ é *fracamente mais importante* que $c_3$, $c_2$ é *levemente mais importante* que $c_1$ e $c_2$ é *fracamente a moderadamente mais importante* que $c_3$. Uma matriz par-a-par $M$ contendo a importância de cada critério sobre os demais pode ser criada como segue:

$$\mathcal{M} = \overbrace{\begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 4 \\ 1/3 & 1/4 & 1 \end{bmatrix}}^{c_1 \quad c_2 \quad c_3} \begin{matrix} c_1 \\ c_2 \\ c_3 \end{matrix}$$

O mesmo processo é feito para as alternativas. Para cada critério, um conjunto de comparações par-a-par é criado entre as alternativas e uma matriz par-a-par é criada para representar estas comparações.

O próximo passo do AHP, chamado de *síntese*, computa um vetor contendo os pesos relativos de todos os critérios da matriz e um vetor para cada matriz de alternativas. Estes vetores representam o *ranking* de prioridades dado por cada matriz par-a-par. Para calcular cada vetor, a respectiva matriz é elevada ao quadrado sucessivamente. Em cada iteração, as somas das colunas são calculadas e normalizadas. A computação pára quando as diferenças entre as somas de dois cálculos consecutivos são menores que um valor pré-definido. Normalmente, de duas a quatro iterações são suficientes.

Tomando a matriz *M* como exemplo, o *ranking* de prioridades pode ser derivado a partir do vetor de pesos da matriz. Para calcular este vetor, a matriz é elevada ao quadrado, é computada a soma dos valores das colunas e são normalizados os valores:

| Matriz ao quadrado | | | | Soma de linhas | | normalizado | |
|---|---|---|---|---|---|---|---|
| 3.00 | 1.75 | 8.00 | = | 12.7500 | Autovetor: | 0.3194 |
| 5.33 | 3.00 | 14.00 | | 22.3332 | | 0.5595 |
| 1.17 | 0.67 | 3.00 | | 4.8333 | | 0.1211 |
| | | | Total | 39.9165 | Total normalizado | 1.0000 |

Iterando o processo novamente, o vetor calculado é: $V' = \langle\ 0.3196\ \ 0.5584\ \ 0.1220\ \rangle$

Iterações adicionais não modificam estes valores (ao menos usando uma precisão de quatro dígitos). Este vetor normalizado é chamado de *eigenvector* principal da matriz (SAATY, 2003). Para calcular o *ranking* geral das alternativas em termos dos critérios selecionados, uma matriz cujas colunas são os *eigenvectors* das matrizes de alternativas é multiplicada pelo *eigenvector* da matriz de critérios. O vetor resultante é o *ranking* geral de alternativas *vs*. critérios.

O ultimo passo do AHP, *análise de consistência dos resultados*, avalia o nível de consistência das matrizes de comparação par-a-par. Isto é necessário porque inconsistências podem ser introduzidas no processo, dado que a importância relativa de um critério em relação aos demais é resultado de diversos julgamentos sobre o relacionamento do critério. Estes julgamentos podem ser conflitantes uns com os outros. O processo é considerado aceitável quando o índice de consistência é menor que 10%. Mais detalhes sobre como calcular o índice de consistência são descritos em (SAATY, 1990).

### 4.2.2 Classificação

A segunda tarefa na detecção de versões é responsável por verificar se dois arquivos *f1* e *f2* são versões de um mesmo documento. Uma típica tarefa de classificação recebe como entrada um conjunto de tuplas de treinamento, cada uma rotulada com um rótulo de classe. O modelo de saída (classificador) atribui rótulos de classe para cada tupla com base em seus atributos. O modelo pode ser usado para prever a classe de novas tuplas, para as quais o rótulo é desconhecido. A classificação é definida como uma técnica de aprendizado

supervisionada, uma vez que o mecanismo utiliza exemplares de treinamento com classes conhecidas para classificar novos dados. As tuplas (exemplares) são particionadas em dois conjuntos: conjunto de treinamento e conjunto de teste.

A classificação também é realizada em dois passos: fase de treinamento e fase de testes. A fase de treinamento é responsável por construir o modelo a partir do conjunto de treinamento. A fase de testes é responsável por verificar a eficácia do modelo, usando o conjunto de testes. A eficácia do modelo gerado pode ser medida através da verificação da classe atribuída (corretamente classificado ou não). A taxa de eficácia é fornecida pela percentagem de exemplares de teste que foram corretamente classificados pelo modelo.

O conjunto de treinamento consiste em um conjunto de pares de arquivos com valores de características (isto é, *P, S, R* para evolução de conteúdo ou *P, S, R, A, D* para evolução de estrutura e conteúdo), o valor de similaridade e uma classe *k* (k | k ∈ {“versão”, “não versão”}). O conjunto de teste consiste na mesma estrutura, mas o valor de *K* é desconhecido. Desta forma, a função *c* computa a classe *k* para cada par. Seja *G = {(f1, f2), … , (fm, fn)}* um conjunto de pares de arquivos, onde cada par é associado com uma *6-tuple* F={*P, S, R, A, D, sim*}. O valor *k* para todos os pares pode ser computado por $\forall (fm, fn) \in G : c(fm, fn)$, onde cada par de arquivos tem uma classe associada computada por uma função *c*.

Para a etapa de classificação, são utilizados os classificadores *Naïve Bayesianos*. Conforme discutido na seção 2.3.2, decidiu-se optar pelos classificadores *Naïve Bayesianos* como parte da solução do mecanismo de detecção de versões a partir da análise feita em relação às técnicas de classificação apresentadas. Resultados experimentais da classificação são apresentados na seção 4.3.

### 4.2.3 Gerenciamento de Metadados

A última atividade realizada pelo mecanismo de detecção de versões é o gerenciamento de metadados. Tais metadados são usados para descrever informações relacionadas às versões detectadas, como, por exemplo, o valor de similaridade *sim* entre cada par de versões, o valor das características (isto é, atributos) entre cada par de versões, e a representação lógica da estrutura (isto é, a descendência) da árvore de versionamento. Os metadados podem ser úteis no processamento de consultas sobre as diversas versões de um documento XML, uma vez que tornam possível recuperar toda a descendência de versões dos documentos. Também possibilitam consultar algumas metainformações sobre as versões detectadas, como, por exemplo, o grau de similaridade e o valor das características extraídas pelas funções de similaridade.

O modelo de metadados também suporta a representação de arquivos não-versionados (isto é, arquivos que não encontraram versões nos arquivos existentes). Arquivos não versionados podem ser representados no modelo de metadados como raízes de árvores, para futuras descendências. Informações dos metadados são estruturadas de acordo com a DTD mostrada na Figura 4.11. A idéia geral é representar, para cada par de versões (onde elemento *node* representa cada arquivo do par), quais os valores das caracterísicas (atributos *P, S, R, D* e *A*) e o valor final de similaridade (atributo *sim*) calculado.

```
<!ELEMENT metadata (node*)>
<!ELEMENT node (features*, node*)>
<!ATTLIST node file CDATA #REQUIRED>
<!ELEMENT features EMPTY>
<!ATTLIST features  P CDATA #REQUIRED    S CDATA #REQUIRED
      R CDATA #REQUIRED    sim CDATA #REQUIRED
      D CDATA #IMPLIED    A CDATA #IMPLIED>
```

Figura 4.11: Estrutura dos metadados

A estrutura lógica das versões lineares e ramificadas pode ser representada utilizando o modelo de metadados. A Figura 4.12 descreve os metadados para a Figura 4.2.

```
<metadata><node file="f1">
  <node file="f2">
      <features P="0.7" S="0.9" R="0.1" sim="0.86"/>
      <node file="f3">
          <features P="0.8" S="0.9" R="0.1" sim="0.97"/>
      </node>
  </node>
</node>
<node file="f4">
    <node file="f5">
        <features P="0.6" S="0.7" R="0.1" sim="0.79"/>
        <node file="fx">
            <features P="0.7" S="0.8" R="0.1" sim="0.89"/>
        </node>
    </node>
</node></metadata>
```

Figura 4.12: Exemplo de metadados para o versionamento linear

A Figura 4.13 descreve os metadados para a Figura 4.3.

```
<metadata><node file="f1">
    <node file="f2">
        <features P="0.7" S="0.9" R="0.1" sim="0.9"/>
        <node file="f5">
            <features P="0.8" S="0.9" R="0.1" sim="0.94"/>
        </node>
    </node>
    <node file="f3">
        <features P="0.6" S="0.7" R="0.1" sim="0.7"/>
    </node>
    <node file="f4">
        <features P="0.7" S="0.8" R="0.1" sim="0.81"/>
        <node file="fx">
            <features P="0.7" S="0.8" R="0.2" sim="0.8"/>
        </node>
    </node>
</node></metadata>
```

Figura 4.13: Exemplo de metadados para o versionamento ramificado

A descendência de versões é representada como um relacionamento *pai-filho*. Percorrendo-se a árvore, a descendência inteira de uma versão pode ser recuperada. Por exemplo, há dois grupos de versões lineares na Figura 4.2. O primeiro grupo é formado pelos arquivos *f1*, *f2* e *f3*. Na Figura 4.12, este grupo é representado pelo seguinte relacionamento *parent-children*:

*/metadata/node[@file="f1"]/node[@file="f2"]/node*

Outro exemplo é mostrado na Figura 4.3, a qual contém três ramos de versões. O terceiro ramo é formado pelos arquivos *f1*, *f4* e *fx*. Na Figura 4.13, este ramo é representado pelo seguinte relacionamento *parent-children*:

*/metadata/node[@file="f1"]/node[@file="f4"]/node*

Os identificadores de arquivos (denotados pelo atributo *file* nos metadados) são gerados utilizando funções *hash*, uma abordagem bastante comum na identificação de arquivos. Por questões de simplicidade, os identificadores baseados em *hash* não são apresentados nos exemplos acima. O resultado *hash* é substituído por uma notação mais simples (por exemplo, o resultado *hash ece50ed4d6d48dac839bfe8fa719fcff* foi denotado por *f5*).

## 4.3 Experimentos

Para testar o mecanismo proposto na detecção de réplicas e de versões, foram realizados alguns experimentos. Estes experimentos verificam a revocação e a precisão obtidas com o método de detecção de versões, baseado nas funções de similaridades apresentadas na seção 4.2.1, e no uso de classificadores *Bäyesianos*, apresentados na seção 4.2.2. Os valores de revocação e de precisão obtidos demonstram que a aplicação do mecanismo proposto produz resultados com boa qualidade para a detecção de versões.

Para avaliar a qualidade do classificador *Naïve Bayesiano*, os experimentos analisam a corretude através de métricas de revocação e precisão, medidas bastante utilizadas em recuperação de informação (BAEZA-YATES et al., 1999). A definição clássica de revocação é a razão entre documentos relevantes que são recuperados e os documentos relevantes disponíveis. Precisão é definida como a razão entre os documentos recuperados relevantes e os documentos recuperados. Seja *A* o conjunto de pares de arquivos que são versões e *B* o conjunto de pares de arquivos que são detectados como versões pelo classificador. Seja "|" a cardinalidade de um conjunto (isto é, o número de elementos do conjunto). Desta forma, a revocação é definida como *Recall:* $|A \cap B|/|A|$, e a precisão é definida como *Precision:* $|A \cap B|/|B|$. A expressão $|A \cap B|$ corresponde ao número de versões que foram corretamente detectadas como versões pelo classificador. Em outras palavras, revocação e precisão são definidas como:

*revocação: (nº. de versões corretamente detectadas)/(nº. de versões existentes)*

*precisão: (nº. de versões corretamente detectadas)/(nº. de versões detectadas)*

A fim de avaliar a eficácia/corretude da função de similaridade e do mecanismo de detecção de versões baseado no classificador *Bayesiano*, dividiu-se os experimentos em dois grupos. O primeiro grupo considerou apenas evolução de conteúdo. O segundo grupo considerou evolução de conteúdo e de estrutura, conforme apresentado na seção 4.2.1. Estes experimentos foram realizados com dados sintéticos.

Para cada tipo de evolução, os experimentos foram divididos em quatro fases: aquisição dos dados, treinamento dos dados, teste dos dados, e análise de resultados. Na primeira fase, os dados foram adquiridos. Na segunda fase, o classificador usou os dados adquiridos para o treinamento, a fim de obter um modelo de classificação. Na terceira fase, o classificador usa

o modelo de classificação nos dados de teste. Finalmente, a fase de análise avaliou a corretude dos resultados, usando métricas de revocação e precisão.

### 4.3.1 Evolução de Conteúdo

Para estes experimentos considerou-se a função de similaridade apresentada na seção 4.2.1.1. Os experimentos foram intencionalmente baseados em valores simulados (isto é, sintéticos), a fim de avaliar a escalabilidade do classificador conjuntamente com a análise da qualidade dos resultados. Os experimentos foram realizados como segue.

a) Aquisição dos dados: esta fase foi responsável pela aquisição do conjunto de dados necessário a ser usado como dados de treinamento para o classificador. Algumas atividades foram realizadas, conforme descrito.

1. Foram gerados 9000 valores randômicos para os atributos (isto é, características) considerados na função de similaridade: *P*, *S* e *R*, onde *P*, *S* e $R \in [0,1]$. Para estes experimentos, foram usados os pesos 0.5, 0.5 e -0.5, respectivamente.
2. Foi aplicada a função de similaridade *simC* a estes valores, obtendo-se valores de similaridade para 9000 pares de arquivos. Em outras palavras, foram calculados 9000 valores de similaridade para um conjunto de pares de arquivos. As amostras geradas foram uniformemente distribuídas entre versões (50%) e não versões (50%).
3. Finalmente, os valores de similaridade foram uniformente distribuídos, usando uma transformação uniforme de mapeamento, descrita na seção 4.2.1.1.

b) Treinamento dos dados: esta fase foi responsável por fornecer os dados adquiridos ao classificador. O classificador usou este conjunto de dados para gerar um modelo que mais tarde é usado na fase de teste. Realizando-se as atividades 1, 2 e 3 acima descritas, o conjunto de treinamento foi gerado e usado pelo classificador. O conjunto de treinamento foi armazenado em um banco de dados seguindo a seguinte estrutura:

*trainingTable (pairID, P, S, R, simCNormalized, class)*

onde: *pairID* é o identificador de um par de arquivos; *P*, *S* e *R* são as características consideradas na função de similaridade; *simCNormalized* é o valor de similaridade após a função de transformação; e *class* é a categoria de cada par de arquivos que foi comparado (isto é, versão ou não versão).

A partir do conjunto de treinamento calculou-se a probabilidade de cada valor de característica para as classes. Estas probabilidades são usadas pelo classificador e são armazenadas na seguinte estrutura:

*featureProbability (class, feature, value, probability)*

onde: *class* é a categoria (isto é, versão ou não versão); *feature* é uma das características consideradas na função de similaridade (isto é, *P*, *S* ou *R*); *value* é o valor possível para uma característica (entre 0 e 1, variando na escala 0.01); e *probability* é a probabilidade de que um certo valor apareça na característica para uma certa classe (por exemplo, 0.000648). A probabilidade é definida como descrita na seção 4.2.2 e é computada por meio de consultas SQL sobre a relação *trainingTable*.

c) Teste dos dados: esta fase foi responsável por testar vários conjuntos de dados no modelo gerado pelo classificador na fase de treinamento. A fim de avaliar a qualidade dos resultados em diferentes tamanhos de conjuntos de dados de testes, foram escolhidos três grupos de conjuntos de dados: o primeiro grupo continha três conjuntos de dados com 1000 amostras cada (isto é, 1000 pares de arquivos), o segundo grupo continha três conjuntos de dados com 500 amostras cada, e o terceiro grupo continha três conjuntos de dados com 100 amostras cada. A geração destes conjuntos de dados seguiu os mesmos passos descritos na fase de aquisição de dados. Os dados de teste foram armazenados em um banco de dados seguindo a seguinte estrutura:

*testingTable (pairID, P, S, R, simCNormalized)*

onde: *pairID* é o identificador de um par de arquivos; *P*, *S* e *R* são as características consideradas na função de similaridade; e *simCNormalized* é o valor de similaridade após a função de transformação.

Para cada par de arquivos (*pairID*), computou-se a probabilidade do valor *xi* aparecer no *i-ésimo* atributo da classe *C*. Por exemplo, considerando-se a seguinte tupla na relação *testingTable*:

*testingTable (1, 0.15, 0.13, 0.55, 0.07)*

As probabilidades calculadas são:

$$P(0.15,0.13,0.55|non\text{-}version) = 0.0000018102$$

$$P(0.15,0.13,0.55|version) = 0.0000000709$$

Uma vez que a primeira probabilidade é maior do que a segunda probabilidade, o par de arquivos foi classificado como não versão. O classificador foi aplicado para cada conjunto e retornou a categoria de cada par de arquivos.

d) Análise dos dados: esta fase foi responsável por avaliar a corretude dos resultados em termos de revocação e de precisão. Os seguintes experimentos foram realizados: *e1*, *e2* e *e3* (1000 pares de arquivos); *e4*, *e5* e *e6* (100 pares de arquivos); *e7*, *e8* e *e9* (500 pares de arquivos). Os resultados são apresentados na Figura 4.14. Conforme observado, as taxas médias de revocação e de precisão foram, respectivamente, 92.13% e 92.49%. Mesmo os piores casos para revocação (88.89% no experimento 4) e precisão (87.27% no experimento 4) ainda foram bons. Em outras palavras, o classificador classificou corretamente mais do que 92% das versões existentes, e mais do que 92% das versões detectadas foram corretamente classificadas.

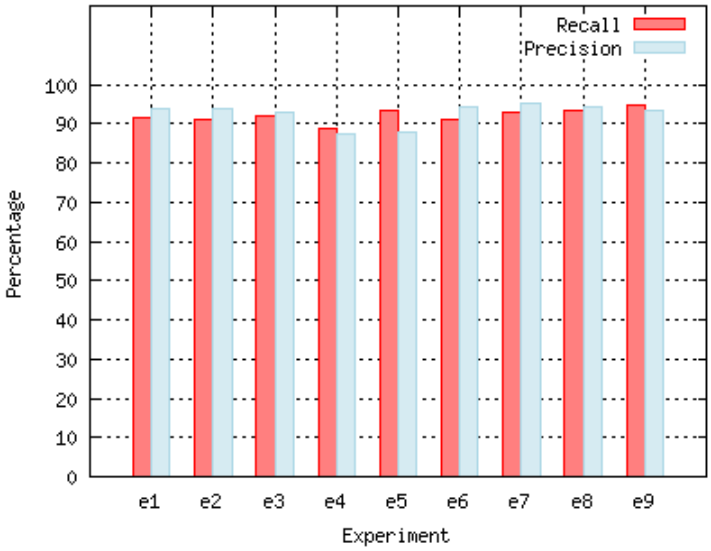



Figura 4.14: Resultados de revocação e de precisão para o grupo 1

Para os resultados apresentados na Figura 4.14 foi considerada a mesma proporção (50%) de versões e não versões para os conjuntos de treinamento e de teste. Foram realizados outros experimentos a fim de avaliar como o classificador se comporta com diferentes proporções de versões e não versões no conjunto de treinamento e de teste. Foram usadas as mesmas configurações de dados descritas anteriormente. No entanto, nestes experimentos considerou-se que 80% dos conjuntos de treinamento e de teste eram representados por versões. Os resultados destes experimentos são mostrados na Figura 4.15.

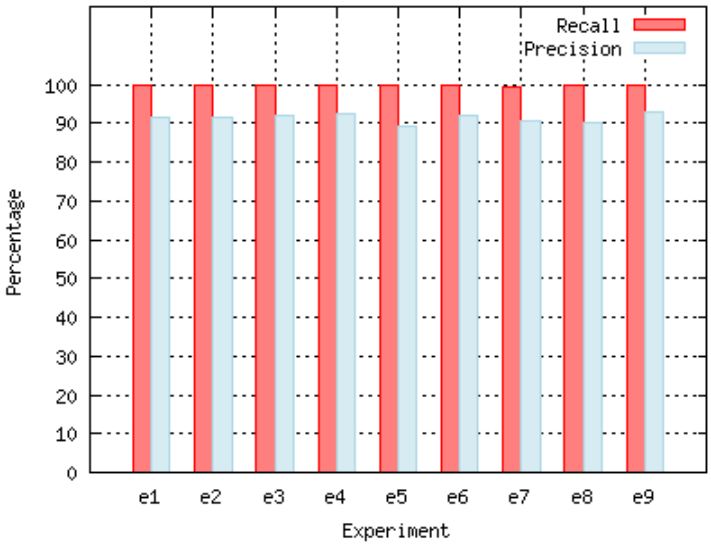



Figura 4.15: Resultados de revocação e de precisão para o grupo 2

Como mostrado na Figura 4.15, as taxas de revocação foram melhores neste caso. A taxa média de revocação e de precisão foram, respectivamente, 99.83% e 91.35%. Mesmo os piores casos para revocação (99.28% no experimento 7) e para precisão (89.13% no experimento 5) ainda mostraram boas taxas. Em outras palavras, o classificador detectou

corretamente mais que 99% das versões existentes, e mais de 91% das versões detectadas foram corretamente classificadas.

### 4.3.2 Evolução de Estrutura e de Conteúdo

Para estes experimentos foi considerada a função de similaridade apresentada na seção 4.2.1.3. Similarmente à seção prévia, os experimentos foram realizados da seguinte forma:

a) Aquisição dos dados: as seguintes atividades foram realizadas:

1. Foram gerados 9000 valores randômicos para os atributos (isto é, características) considerados na função de similaridade: *P*, *S*, *R*, *A* e *D*, onde *P*, *S*, *R*, *A* e $D \in [0,1]$. Para estes experimentos, foram usados os pesos 0.5, 0.5, -0.33, -0.33 e -0.33, respectivamente.
2. Foi aplicada a função de similaridade *simE*, obtendo-se valores de similaridade para 9000 pares de arquivos. As amostras geradas foram uniformemente distribuídas entre versões (50%) e não versões (50%).
3. Finalmente, os valores de similaridade foram uniformemente distribuídos usando uma transformação uniforme de mapeamento, descrita na seção 4.2.1.1.

b) Treinamento dos dados: os dados de treinamento foram armazenados em um banco de dados seguindo a seguinte estrutura:

*trainingTable (pairID, P, S, R, A, D, simENormalized, class)*

Foi gerada a tabela para as probabilidades das características (tabela *featureProbability*, conforme apresentada anteriormente).

c) Teste dos dados: esta fase foi similar à fase de treinamento de dados apresentada na seção anterior. Novamente, a fim de avaliar a qualidade dos resultados em diferentes tamanhos de conjuntos de dados de testes, foram escolhidos três grupos de conjuntos de dados: o primeiro grupo continha três conjuntos de dados com 1000 amostras cada (isto é, 1000 pares de arquivos), o segundo grupo continha três conjuntos de dados com 500 amostras cada, e o terceiro grupo continha três conjuntos de dados com 100 amostras cada. A geração destes conjuntos de dados seguiu os mesmos passos descritos na fase de aquisição de dados. Os dados de teste foram armazenados em um banco de dados seguindo a seguinte estrutura:

*testingTable (pairID, P, S, R, A, D, simENormalized)*

O classificador foi aplicado para cada conjunto, retornando a categoria para cada par de arquivos (isto é, versão ou não versão).

d) Análise dos dados: vários experimentos foram realizados, conforme descritos: *e1*, *e2* e *e3* (1000 pares de arquivos); *e4*, *e5* e *e6* (100 pares de arquivos); *e7*, *e8* e *e9* (500 pares de arquivos). Os resultados são apresentados na Figura 4.16. Conforme observado, as taxas médias de revocação e de precisão foram, respectivamente, 91.11% e 91.05%. Mesmo os piores casos para revocação (87.23% no experimento 5) e precisão (90.04% no experimento 1 e no experimento 7) ainda foram bons. Em outras palavras, o classificador classificou corretamente mais do que 91% das versões existentes, e mais do que 91% das versões detectadas foram corretamente classificadas.

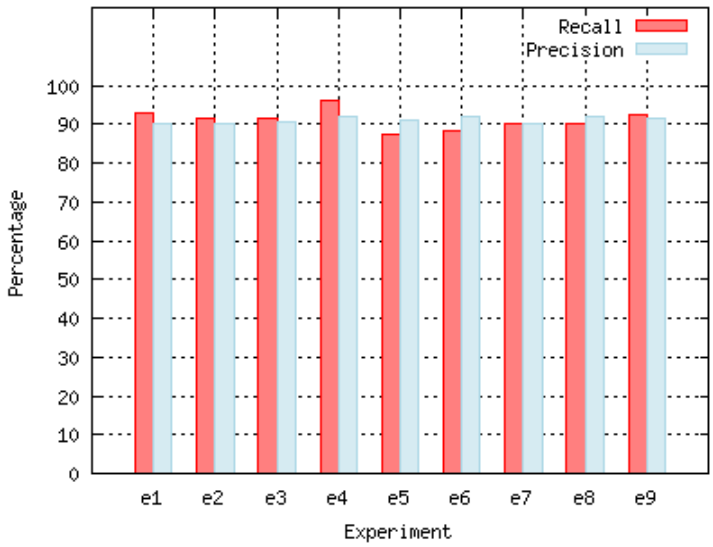



Figura 4.16: Resultados de revocação e de precisão para o grupo 3

Todos os resultados apresentados (grupos 1 ao 3) foram executados sobre um conjunto de treinamento com 9000 amostras. No entanto, era desejável também avaliar como os valores de revocação e de precisão comportavam-se usando um tamanho diferente para o conjunto de treinamento. Desta forma, foram executados novamente os experimentos acima (*e1* ao *e9*), usando agora um conjunto de treinamento com somente 3000 amostras (grupo 4). A mesma configuração foi usada, isto é, mesma distribuição entre versões e não versões.

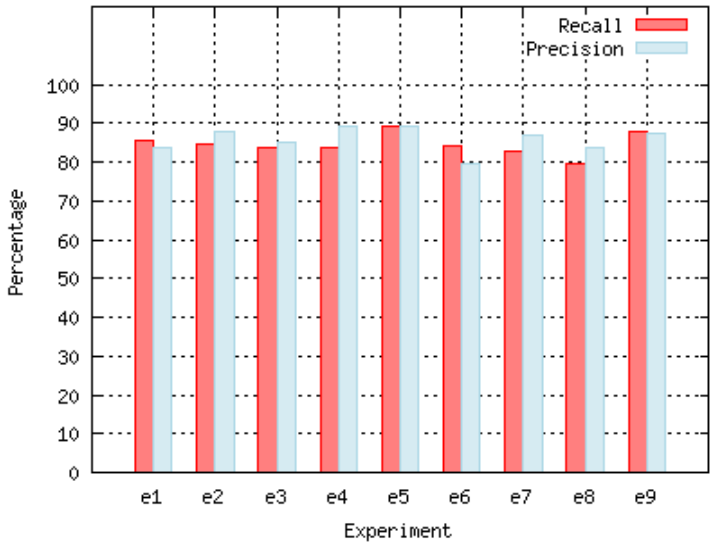



Figura 4.17: Resultados de revocação e de precisão para o grupo 4

As taxas médias de revocação e de precisão foram, respectivamente, 84.66% e 85.87%. Mesmo os piores casos para revocação (79.66% no experimento 8) e para precisão (79.63% no experimento 6) ainda mostraram taxas razoáveis. Em outras palavras, o classificador detectou corretamente mais que 84% das versões existentes, e mais de 85% das versões detectadas foram corretamente classificadas. Os resultados das taxas de revocação e precisão

(mais baixos que nos experimentos anteriores) levam à conclusão de que o classificador produz melhores resultados quando o conjunto de treinamento tem mais amostras. Considerando os resultados apresentados na Figura 4.16 e na Figura 4.17, as taxas de revocação e de precisão foram, respectivamente, 7.07% e 5.68% mais altas com um conjunto de treinamento maior.

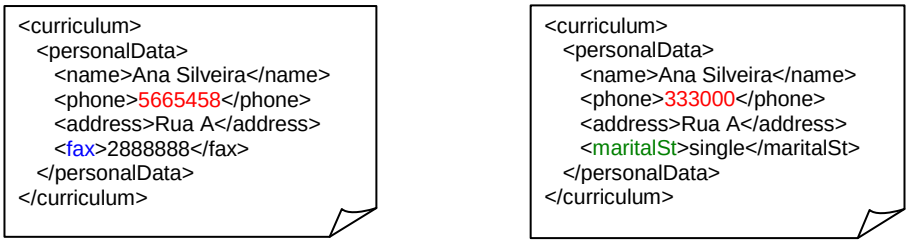
Finalmente, todos os experimentos (considerando os grupos 1 ao 4) mostraram a corretude e eficácia do mecanismo proposto. Foram usados dois conjuntos de treinamento diferentes com a mesma configuração (isto é, a mesma proporção de versão e não versões, e o mesmo tamanho), resultando em quatro conjuntos de treinamento. Os conjuntos de teste também foram diversos: 27 conjuntos diferentes de teste, cujos tamanhos variaram de 100 a 1000 amostras (isto é, pares de arquivos). A taxa média de revocação, considerando os três primeiros grupos de experimentos (grupos 1 ao 3), foi de 94.35% e a taxa média de precisão foi de 91.63%. Incluindo o quarto grupo nesta análise (com taxas piores por causa do tamanho menor do conjunto de treinamento), as taxas médias ainda foram boas: 91.93% para revocação e 90.19% para precisão.

## 4.4 Agrupamento de Versões de Documentos

A partir de *n* versões de um documento, deve ser gerada uma representação consolidada contendo todas as suas modificações. Esta nova representação deve ser armazenada como um novo arquivo físico. Desta forma, para cada documento com *n* versões, onde *n>1*, uma nova representação é criada e armazenada. Por questão de desempenho, esta parece ser uma boa abordagem, uma vez que somente um arquivo precisa ser consultado para cada instância de documento. Obviamente, nem todos os documentos teriam uma versão consolidada armazenada, mas somente aqueles documentos que são mais freqüentemente acessados pelo sistema. Para isso, é necessária a análise dos arquivos que são mais frequentemente acessados. Representações consolidadas podem ser utilizadas para um processamento mais rápido de certas consultas que solicitam histórico de documentos. Dessa maneira, não há a necessidade de acessar todas as versões de um documento espalhadas pelo repositório. No entanto, os metadados devem prever a representação destas versões consolidadas.

Por exemplo, considerando o documento XML mostrado na Figura 4.18(a), registrado em 10/10/2008. A Figura 4.18 (b) contém uma nova versão deste documento, registrada em 20/11/2008. Nota-se que existem algumas diferenças entre as duas versões: o elemento *phone* foi modificado; o elemento *fax* foi removido; e o elemento *maritalSt* foi adicionado.

Versões prévias, como mudanças delta sobre o estado corrente de certo documento, não são armazenadas. Ao invés disso, propõe-se gerar uma representação única, contendo todas as modificações do documento. Rótulos temporais são responsáveis por validar e invalidar dados em versões específicas, baseados nos tempos iniciais (*TS*) e tempos finais (*TE*). *TS* e *TE* representam o período de tempo no qual uma informação é válida no mundo real. A Figura 4.19 mostra uma representação consolidada representada na forma de árvore.




(a) (b)

Figura 4.18: Versão 1 e versão 2 de um documento XML

A tarefa de gerar uma versão consolidada contendo todo o histórico do documento é similar ao processo de *merging* de arquivos, bastante utilizado em sistemas de versões concorrentes (CVS). Para o mecanismo proposto, não existem conflitos a serem resolvidos, uma vez que todo o histórico do documento é armazenado, com os respectivos rótulos temporais. A representação consolidada (*H-Doc* file: ***H**istorical **Doc**ument*) do documento mostrado na Figura 4.19 pode ser armazenada fisicamente em um novo arquivo XML, conforme detalhado na seção 4.4.2.

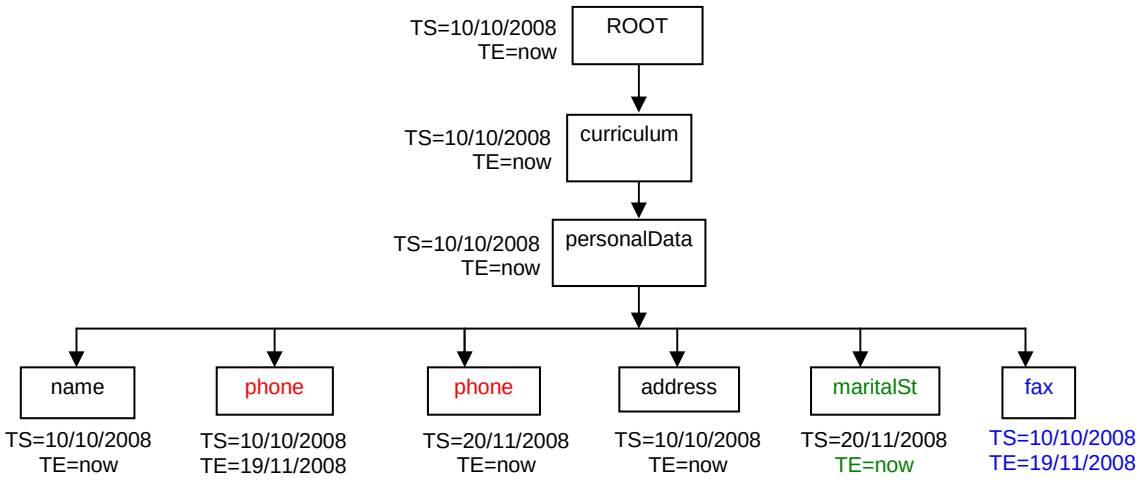



Figura 4.19: Representação em árvore

O agrupamento de versões assume que a evolução dos documentos cria uma seqüência linear de versões temporalmente ordenadas: $V_1$, $V_2$, ..., $V_n$, onde a última versão $V_n$ é a atual. Logo, não são geradas representações consolidadas para documentos que seguem o versionamento ramificado. Uma versão pode possuir apenas uma versão antecessora e outra sucessora. A fim de ordenar as versões de arquivos em uma linha de tempo, levam-se em consideração os tempos de modificação destes arquivos. Assume-se que o arquivo com tempo de modificação mais recente é uma derivação de uma versão anterior. Supondo duas versões de arquivos, *F1* e *F2*, com as respectivas datas de modificação *10/10/2007* e *03/04/2008*, conclui-se que o arquivo *F2* é uma versão modificada do arquivo *F1*.

### 4.4.1 Detecção de Diferenças entre Versões de Documentos

Considere dois arquivos XML contendo informações relacionadas a currículos, mostrados na Figura 4.20(a) e Figura 4.20(b). Nota-se que algumas modificações foram realizadas no segundo arquivo: o elemento *phone* foi atualizado, o elemento *fax* foi removido e o elemento *maritalSt* foi inserido.

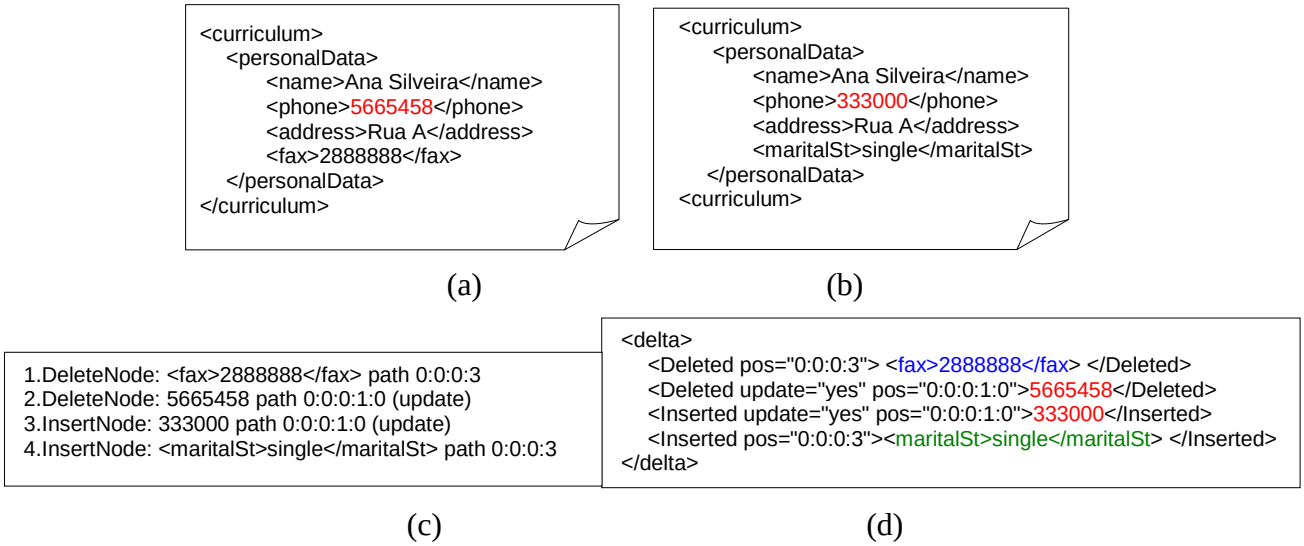



Figura 4.20: Documento XML original e modificado, com alterações detectadas

Utilizando-se o algoritmo *XyDiff*, obtém-se as seguintes operações, mostradas na Figura 4.20(c). As operações 2 e 3 correspondem à atualização do elemento *phone*. A operação 1 corresponde à remoção do elemento *fax*. A operação 4 corresponde à inserção do elemento *maritalSt*. Estas operações são estruturadas em um documento XML, mostrado na Figura 4.20(d).

### 4.4.2 Geração da Representação com o Histórico de Versões

Assume-se que cada elemento possui dois atributos temporais, *TS* e *TE*. *TS* é inicializado com o tempo de modificação do arquivo no qual o elemento foi adicionado/modificado. *TE* é inicializado com *now* e permanece com este valor enquanto for válido; um elemento é considerado válido até sua próxima modificação/remoção. *TE* é atualizado quando um elemento é modificado ou removido; neste caso, *TE* é encerrado quando uma nova instância de elemento é criada.

A representação consolidada do documento mostrado na Figura 4.19 pode ser armazenada fisicamente em um novo arquivo XML, mostrado na Figura 4.21.

Os metadados do *framework* devem prever a representação das versões consolidadas, de forma que o processador de consultas possa tirar vantagem dos arquivos *H-Doc*, quando estes existirem. O processamento de consultas sobre o arquivo *H-Doc* é descrito na implementação da ferramenta, apresentado no capítulo 5.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT TS="10/10/2008" TE="now">
<curriculum TS="10/10/2008" TE="now">
    <personalData TS="10/10/2008" TE="now">
        <name TS="10/10/2008" TE="now">…</name>
        <phone TS="10/10/2008" TE="19/11/2008">…</phone>
        <phone TS="20/11/2008" TE="now">…</phone>
        <address TS="10/10/2008" TE="now">…</address>
        <marital TS="20/11/2008" TE="now">…</marital>
        <fax TS="10/10/2008" TE="19/11/2008">…</fax>
    </personalData>
</curriculum>
</ROOT>
```

Figura 4.21: Versão consolidada – arquivo *H-Doc*

### 4.4.3 Implementação: *XVersion*

Esta seção detalha a ferramenta construída para o agrupamento de versões. A implementação da ferramenta *XVersion* (GIACOMEL, 2006) disponibiliza um ambiente que agrupa em um único arquivo de saída os *n* arquivos XML de entrada, através do processamento de diferenças. O agrupamento destes arquivos baseia-se no uso de algoritmos *diff* e rótulos temporais que especificam o histórico de cada elemento dentro do documento de saída gerado. Com a geração da representação única do histórico dos documentos, o usuário pode submeter consultas temporais, utilizando *XQuery* (XQUERY, 2008).

A Figura 4.22 apresenta a tela inicial da ferramenta *XVersion*, que tem como principais funcionalidades abrir documentos XML existentes ou criar novos arquivos, salvar alterações nos arquivos XML, gerar o arquivo *H-Doc* a partir de *n* arquivos XML, detectar diferenças entre versões de arquivos XML, e executar consultas sobre o documento agrupado. Esta última funcionalidade possibilita a utilização de filtros temporais (restrições nos valores dos atributos temporais *TS* e *TE*).

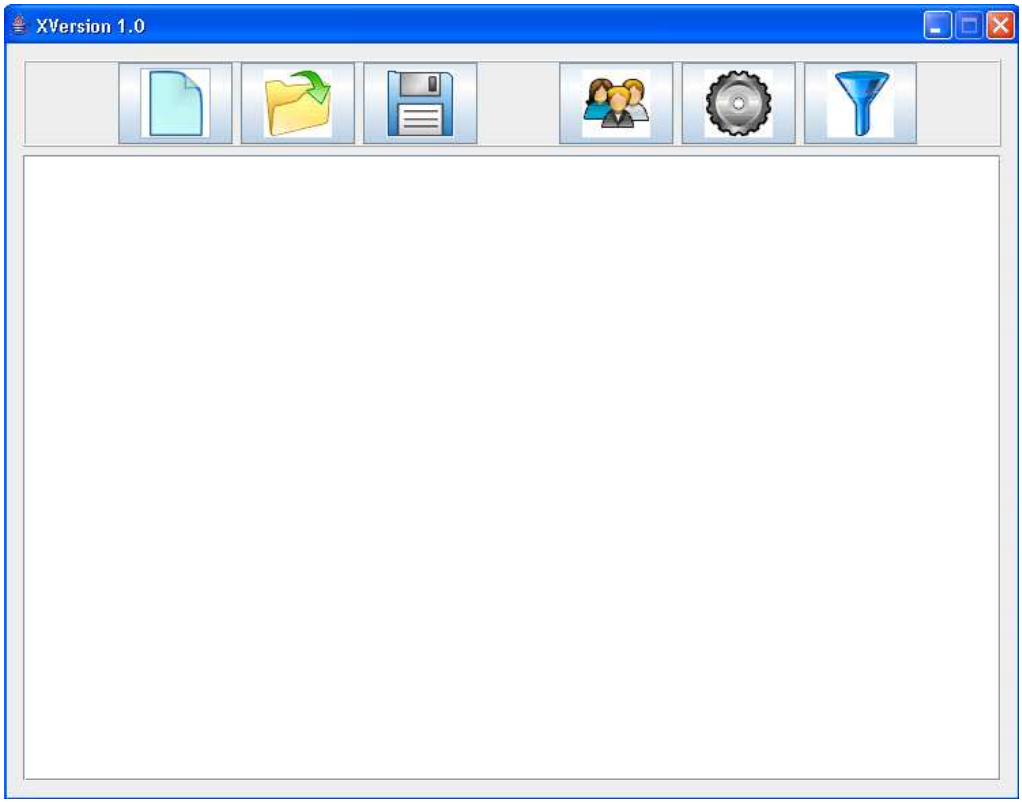



Figura 4.22: *XVersion* - Tela inicial da ferramenta

A partir de *n* documentos XML fornecidos como entrada, executa-se o *XyDiff* para cada par de arquivos $a_i$ e $a_{i+1}$ (onde *i* é o índice do documento XML dentro da lista de documentos fornecidos), obtendo assim *n-1* documentos representando as diferenças entre cada par de arquivos fornecido. A Figura 4.23 mostra os arquivos delta gerado para três versões de documentos XML fornecidas, utilizando *XyDiff*. Considere que as três versões possuem as seguintes datas de modificação, respectivamente: *09/09/2006*, *05/11/2006* e *11/11/2006*.

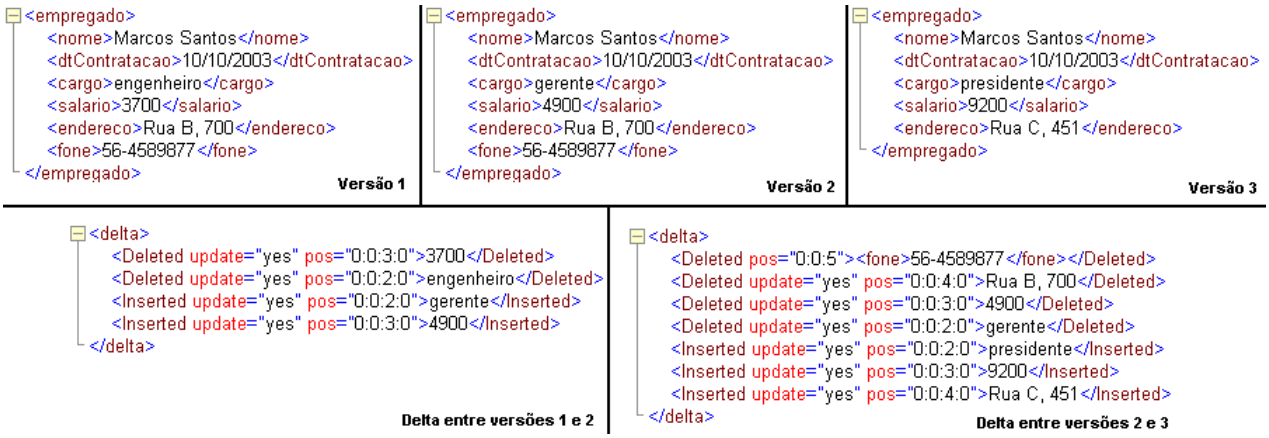



Figura 4.23: Arquivos delta gerados para três versões de um documento XML

A seguir, é criada a representação do arquivo consolidado para a representação do histórico das versões. Para isto, é criado um arquivo XML representando a união de cada par de versões fornecidas. Esta união é feita a partir dos dois arquivos fornecidos e do arquivo delta contendo as diferenças entre estes. O último passo é juntar todas as uniões geradas em um único arquivo. A Figura 4.24 mostra o documento *H-Doc* resultante, contendo o histórico das versões dos arquivos mostrados na Figura 4.23.

```
1.  <empregado posicao="0:0" TS="09/09/2006" TE="~NOW">
2.      <nome posicao="0:0:0" TS="09/09/2006" TE="~NOW">Marcos Santos</nome>
3.      <dtContratacao posicao="0:0:1" TS="09/09/2006" TE="~NOW">10/10/2003</dtContratacao>
4.      <cargo posicao="0:0:2" TS="09/09/2006" TE="04/11/2006">engenheiro</cargo>
5.      <salario posicao="0:0:3" TS="09/09/2006" TE="04/11/2006">3700</salario>
6.      <endereco posicao="0:0:4" TS="09/09/2006" TE="10/11/2006">Rua B, 700</endereco>
7.      <fone posicao="0:0:5" TS="09/09/2006" TE="10/11/2006">56-4589877</fone>
8.      <cargo posicao="0:0:2" TS="05/11/2006" TE="10/11/2006">gerente</cargo>
9.      <salario posicao="0:0:3" TS="05/11/2006" TE="10/11/2006">4900</salario>
10.     <cargo posicao="0:0:2" TS="11/11/2006" TE="~NOW">presidente</cargo>
11.     <salario posicao="0:0:3" TS="11/11/2006" TE="~NOW">9200</salario>
12.     <endereco posicao="0:0:4" TS="11/11/2006" TE="~NOW">Rua C, 451</endereco>
13. </empregado>
```

Figura 4.24: Históricos das versões - arquivo *H-Doc* resultante

Conforme observado na Figura 4.23, alguns elementos mudaram seu conteúdo ao longo das versões. Como exemplo, o elemento *cargo* teve seu conteúdo alterado de “engenheiro” para “gerente” e posteriormente para “presidente”. O histórico deste elemento é representado no arquivo *H-Doc*, através da inclusão de três instâncias para o elemento *cargo*, conforme mostrado nas linhas 4, 8 e 10 da Figura 4.24.

## 4.5 Considerações Finais

O problema da detecção de versões é importante em muitas aplicações, tais como identificação de clones de *software*, ranqueamento de páginas Web, detecção de plágio e busca em sistemas P2P. Uma abordagem comum e bastante intuitiva baseia-se no uso de análise de similaridade entre os arquivos. A maioria das técnicas propostas baseia-se no uso de limiares, mas a definição do valor do limiar é problemática por várias razões. Em particular: (i) o valor do limiar não é o mesmo quando se considera diferentes funções de similaridade; e (ii) o limiar não tem significado semântico para o usuário. Para solucionar este problema, este trabalho propõe um mecanismo de detecção de versões de documentos XML baseado em classificadores *Näive Bayesianos*. Desta forma, a abordagem proposta trata o problema de detecção como um problema de classificação.

Neste capítulo foram apresentadas as funções de similaridade que contemplam diversas características que podem ser levadas em consideração ao se detectar versões. As funções não são restritas a uma aplicação específica e podem ser adaptadas para considerar outras características relevantes em diferentes cenários. Além disso, cada característica tem um peso associado, o que torna a abordagem proposta mais flexível do que as encontradas na literatura. O mecanismo de detecção de versões apresentado baseia-se em classificadores, o que elimina a necessidade da definição de limiares.

O problema de detecção de versões não é um assunto novo. Também não é inédito o uso de classificadores para categorizar documentos. No entanto, o uso desta técnica para solucionar o problema de detecção de versões requer a definição de variáveis (ou seja, atributos ou características) que descrevam cada categoria, o que geralmente é uma tarefa árdua, dependente de domínio, para a detecção de versões. Neste contexto, o capítulo apresentou uma solução eficaz para um antigo problema, conforme validado na seção de experimentos apresentada ao longo deste trabalho.

Este capítulo resultou nas seguintes publicações: (SACCOL et al., 2007a; SACCOL et al., 2007b; SACCOL et al., 2007). A implementação da ferramenta *XVersion* foi realizada durante a co-orientação de um Trabalho de Conclusão de Curso (GIACOMEL, 2006). Deste capítulo, também resultaram duas co-orientações de trabalhos de iniciação científica e a produção de um relatório técnico (SACCOL et al., 2007c). Estas produções estão detalhadas no capítulo 7.

# 5 UM FRAMEWORK PARA DETECÇÃO, GERENCIAMENTO E CONSULTA A RÉPLICAS E A VERSÕES

Este capítulo apresenta a arquitetura básica do *framework* onde o mecanismo de detecção de réplicas e de versões pode ser inserido. O *framework* também especifica os componentes usados no gerenciamento e na consulta de réplicas e de versões de documentos XML. São apresentados os componentes principais da arquitetura e as atividades em cada componente, incluindo os metadados necessários para manter e gerenciar o mecanismo proposto.

## 5.1 Visão Geral

*DetVX* (**Det**ector de Réplicas e de **V**ersões de Documentos **X**ML) é um *framework* para detecção, gerenciamento e consulta de réplicas e versões de documentos XML. Assume-se que cada arquivo é armazenado em um *host*[13], o qual pode armazenar diversas réplicas e versões de um mesmo documento, representadas em diferentes arquivos físicos. Por exemplo, o *host* pode armazenar três réplicas da primeira versão de um documento e duas réplicas da segunda versão do documento, totalizando cinco arquivos.

Os arquivos podem pertencer a domínios de aplicação diferentes, como, por exemplo, arquivos referentes a currículos, projetos de pesquisa, cadastro de alunos, etc. Assume-se que cada arquivo pertence a um único domínio de aplicação descrito por uma ontologia[14]. Conseqüentemente, a busca por réplicas e versões de um documento dá-se somente entre os arquivos que pertencem a um mesmo domínio.

O uso de domínios de aplicação traz duas vantagens para o *framework* proposto:

---

[13] Computador conectado à internet, identificado pelo seu endereço IP.

[14] Uma ontologia fornece um vocabulário que descreve um domínio de interesse e uma especificação do significado dos termos deste vocabulário. Segundo Gruber (1993), uma ontologia pode ser entendida como "uma especificação explícita de uma conceitualização consensual de um domínio". Por conceitualização, entende-se um vocabulário de termos, um conjunto de fatos e regras, ou um conjunto de entidades e relacionamentos considerados em um universo de discurso que se deseja representar. Por consensual, entende-se que esta especificação é senso comum para um conjunto de especialistas do universo de discurso.

1) restringe o espaço de busca na detecção de réplicas e versões - não é necessário buscar por uma versão ou réplica entre arquivos de domínios de aplicação diferentes;

2) aumenta a eficiência no processamento de consultas - consultas relativas a um determinado domínio de aplicação não são processadas sobre arquivos que não possuem conceitos pertencentes a este domínio.

Para gerenciar a existência de réplicas e de versões, assim como seus respectivos domínios de aplicação, o *host* possui um conjunto de metadados com informações sobre os arquivos. Os metadados são atualizados sempre que um novo arquivo é registrado, sendo intensamente acessados durante o processamento de consultas. Basicamente, os metadados descrevem informações sobre os seus arquivos, como: identificadores de arquivos, identificadores de documentos, resultados *hash* de arquivos, tempos de registro e última data de atualização de cada arquivo. O detalhamento dos metadados é apresentado na seção 5.6.

Conforme mostrado na Figura 5.1, os arquivos são descritos por ontologias. As ontologias são gerenciadas por um componente chamado gerenciador de ontologias, o qual possui um conjunto de metadados necessário ao gerenciamento dos diversos domínios de aplicação. O usuário acessa os arquivos através da submissão de consultas ao processador de consultas, formuladas com base em uma determinada ontologia e relacionadas a um domínio de aplicação. O processador de consultas processa a consulta e retorna os resultados ao usuário.

Conforme mencionado, um determinado documento pode estar replicado ou multiversionado no *host*. Se o usuário submete uma consulta solicitando o endereço atual de uma pessoa, o sistema deve retornar somente a última versão desta informação. Para descobrir qual(is) arquivo(s) é(são) necessário(s) acessar para responder uma determinada solicitação, o sistema consulta os metadados disponíveis.

Versões e réplicas de documentos são detectadas à medida que um determinado arquivo é criado. Modificações posteriores à criação de um arquivo também são observadas, a fim de verificar o surgimento de uma nova versão ou réplica, de forma transparente ao usuário. Embora o módulo de consultas não seja o foco deste trabalho, o sistema permite que a partir dos metadados definidos seja possível submeter requisições ao sistema. Para isso, nenhuma modificação na edição dos arquivos é exigida.

## 5.2 Arquitetura do *Framework DetVX*

A fim de fornecer as funcionalidades para o *framework* de detecção de réplicas e de versões, é proposta uma arquitetura, conforme ilustrado na Figura 5.2. De maneira geral, o usuário interage com o sistema via uma interface, a qual possibilita registrar[15] arquivos do *host* (através do gerenciador de arquivos, apresentado na Seção 5.3) e posteriormente submeter consultas a estes arquivos (através do processador de consultas, apresentado na

[15] O registro de arquivos no *host* corresponde à tarefa de informar ao *framework* que estes arquivos devem ser considerados no processo de detecção, gerenciamento e consulta a réplicas e versões. O registro se faz necessário porque podem existir arquivos no *host* para os quais o usuário não deseja realizar tal gereciamento.

seção 5.7). Quando um arquivo é registrado, é necessário verificar o seu domínio de aplicação, tarefa esta realizada pelo gerenciador de ontologias (apresentado na seção 5.4). Depois de registrados, o ambiente verifica se estes arquivos referem-se a versões ou réplicas de documentos previamente registrados no sistema. Esta última tarefa é realizada pelo gerenciador de réplicas e de versões (apresentado na Seção 5.5) .

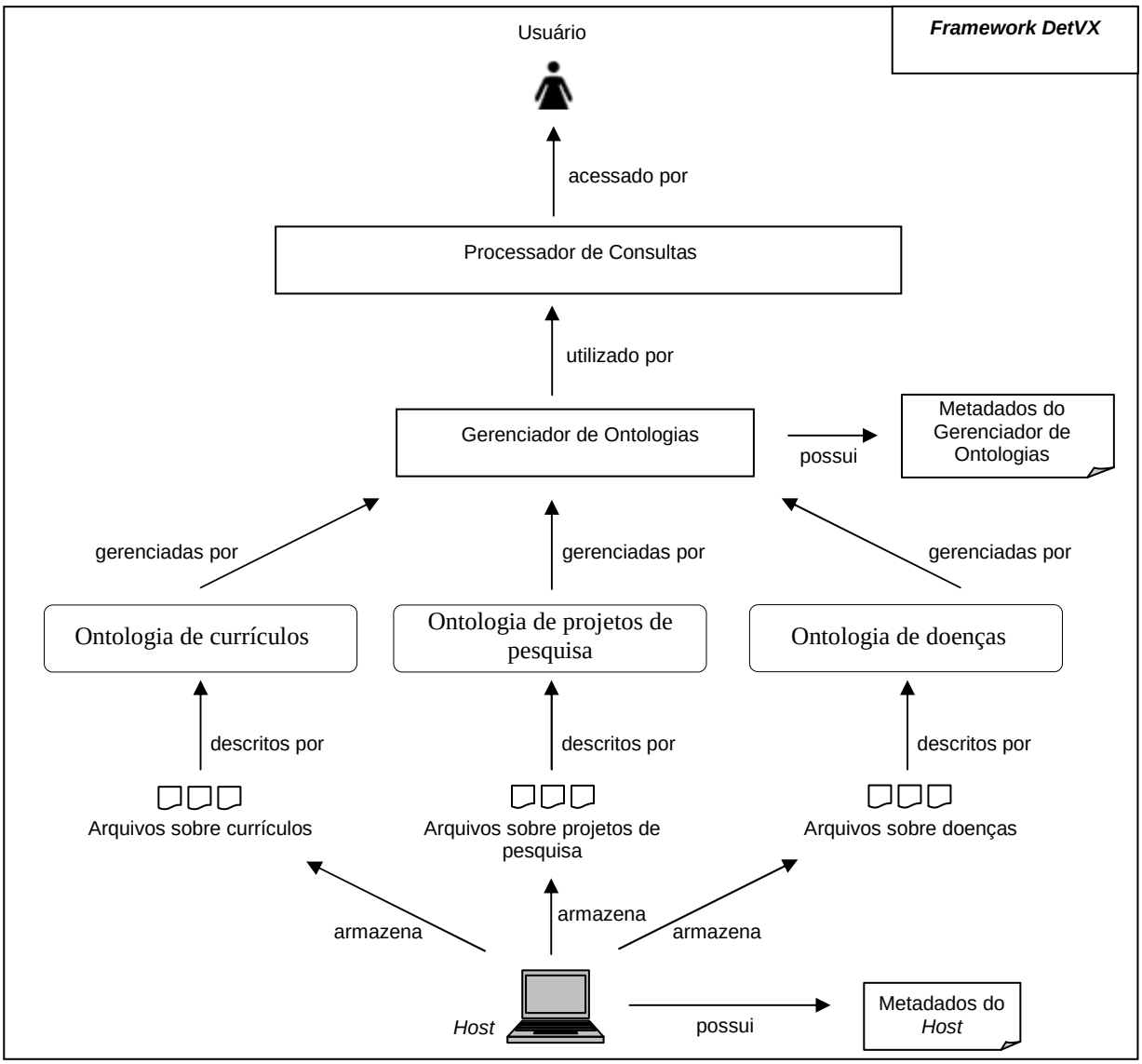



Figura 5.1: Visão geral do *framework DetVX*

### 5.2.1 Arquivos e Documentos

Um arquivo possui um identificador local e um identificador global. Identificadores locais (denotados por *fileID*) são usados para identificar um arquivo no *host*. Neste trabalho é usada uma abordagem de geração de identificadores baseada em funções *hash*, bastante comum para identificação de arquivos. Um arquivo e sua réplica possuem o mesmo

identificador local. Identificadores globais (denotados por *GFID*) são usados para identificar uma versão específica de um arquivo e são definidos como:

*GFID: <fileID>.<versionNo>*

onde:

- *fileID* é um valor que identifica um arquivo no *host*, mapeado a partir do resultado de uma função *hash (fileID=map(hashFunction(file)))*. Exemplo: *fileID=f7*;
- *versionNo* identifica a versão do documento no *host.* Exemplo: *versionNo=1*.

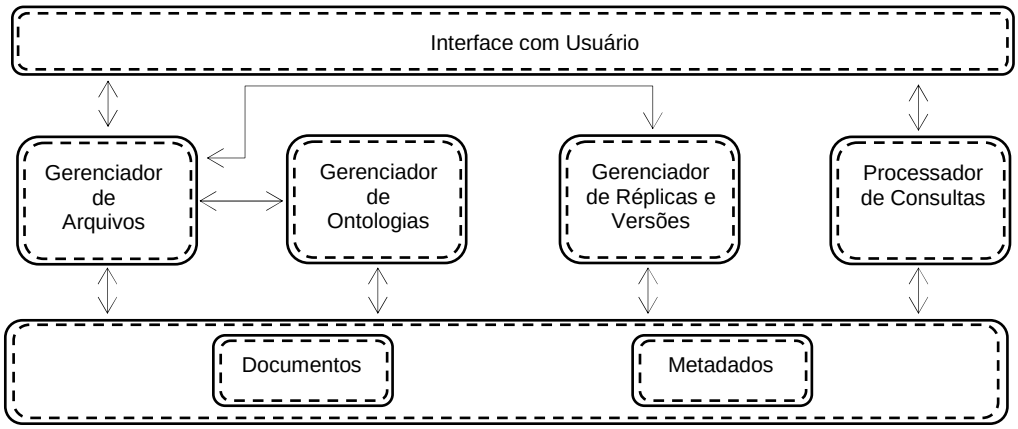



Figura 5.2: Arquitetura do *framework DetVX*

Documentos também possuem identificadores (*docID*) e são usados para identificar objetos do mundo real (por exemplo, o documento *D7*). Todos os identificadores de arquivos e documentos ficam disponíveis em um serviço de nomes localizado no *host*.

Com base nestes identificadores locais e globais, tem-se:

1. se o arquivo registrado no sistema é uma nova instância - o arquivo recebe um novo identificador local e um novo identificador global. Um novo identificador de documento também é atribuído. Exemplo:

   *fileID=f5, GIF=f5.1, docID=D9*

2. se o arquivo registrado é uma réplica de algum arquivo já existente no *host*, terá o mesmo identificador local e global deste arquivo (mesmo resultado *hash*). O identificador de documento também se mantém o mesmo para todas as réplicas. Exemplo:

   réplica 1: *fileID=f5, GIF=f5.3, docID=D9*

   réplica 2: *fileID=f5, GIF=f5.3, docID=D9*

3. se o arquivo registrado é uma versão de algum arquivo já existente, seus identificadores locais serão diferentes (resultados *hash* diferentes), e seus identificadores globais, por conseqüência, também o serão. No entanto, todas as

versões de um mesmo recurso recebem o mesmo identificador de documento. Exemplo:

versão 1: *fileID=f5, GIF=f5.3, docID=D9*

versão 2: *fileID=f6, GIF=f6.3, docID=D9*

O processo de detecção de réplicas e de versões é executado entre os arquivos pertencentes a um mesmo domínio de aplicação, conforme detalhado no Capítulo 4.

## 5.3 Gerenciador de Arquivos

Quando um usuário deseja gerenciar réplicas e versões de seus arquivos, deve registrar estes arquivos no *framework*. O gerenciador de arquivos é responsável por gerenciar os registros de arquivos e por verificar periodicamente se o *host* teve alguma modificação em seus arquivos registrados. O processo de registro é feito em dois passos:

- o usuário envia uma solicitação de registro de *host* ao gerenciador de arquivos, informando o seu endereço IP;
- o gerenciador de arquivos solicita ao usuário a lista de arquivos e/ou diretórios que participarão do processo de gerenciamento de réplicas e de versões.

O sistema deve verificar se esta é a primeira solicitação de registro do *host*. Se este for o caso, então é realizada a escolha do domínio de aplicação da lista de arquivos a serem registrados, e a posterior atualização dos metadados. Caso contrário, verifica-se se ocorreu qualquer modificação nos arquivos previamente registrados, desde a última conexão do *host*. Se houve alguma modificação, então o *framework* verifica se a mudança causou uma alteração no domínio de aplicação dos arquivos previamente registrados, o que eventualmente pode levar à troca de ontologias e conseqüente atualização dos metadados.

As atividades do gerenciador de arquivos são mostradas na Figura 5.3 e detalhadas a seguir.

**Solicitar registro do *host*.** O *host* conecta-se ao *framework*, inicializando o serviço de registro de *host*.

**Verificar registro prévio do *host*.** Verifica se o *host* já esteve conectado anteriormente no ambiente. Para isso, consultam-se os metadados e verifica-se se existem informações do *host* cadastradas. Em outras palavras, verifica-se se o identificador do *host*, dado pelo IP da máquina, aparece nos metadados. Se o identificador deste *host* não é encontrado, então se assume que este *host* nunca foi registrado ao sistema. Caso contrário, o próximo passo é verificar se houve modificações neste *host* desde o último registro.

**Verificar modificações no *host*.** O *framework* dispara um serviço de verificação de modificações no *host*. Este serviço verifica se ocorreu alguma mudança nos arquivos desde a última vez em que o *host* esteve conectado. Uma mudança pode ser a inserção e a remoção de um arquivo, ou a alteração no conteúdo de um arquivo já existente. O serviço de verificação de modificações executa uma função *hash* em cada arquivo. Os resultados da

função *hash* são estruturados em uma mensagem XML, contendo o identificador local do arquivo, tempo de registro, última data em que o arquivo foi modificado e o resultado *hash*. Como exemplo, alguns testes apresentados neste trabalho foram realizados utilizando-se o *Message-Digest Algorithm 5*.

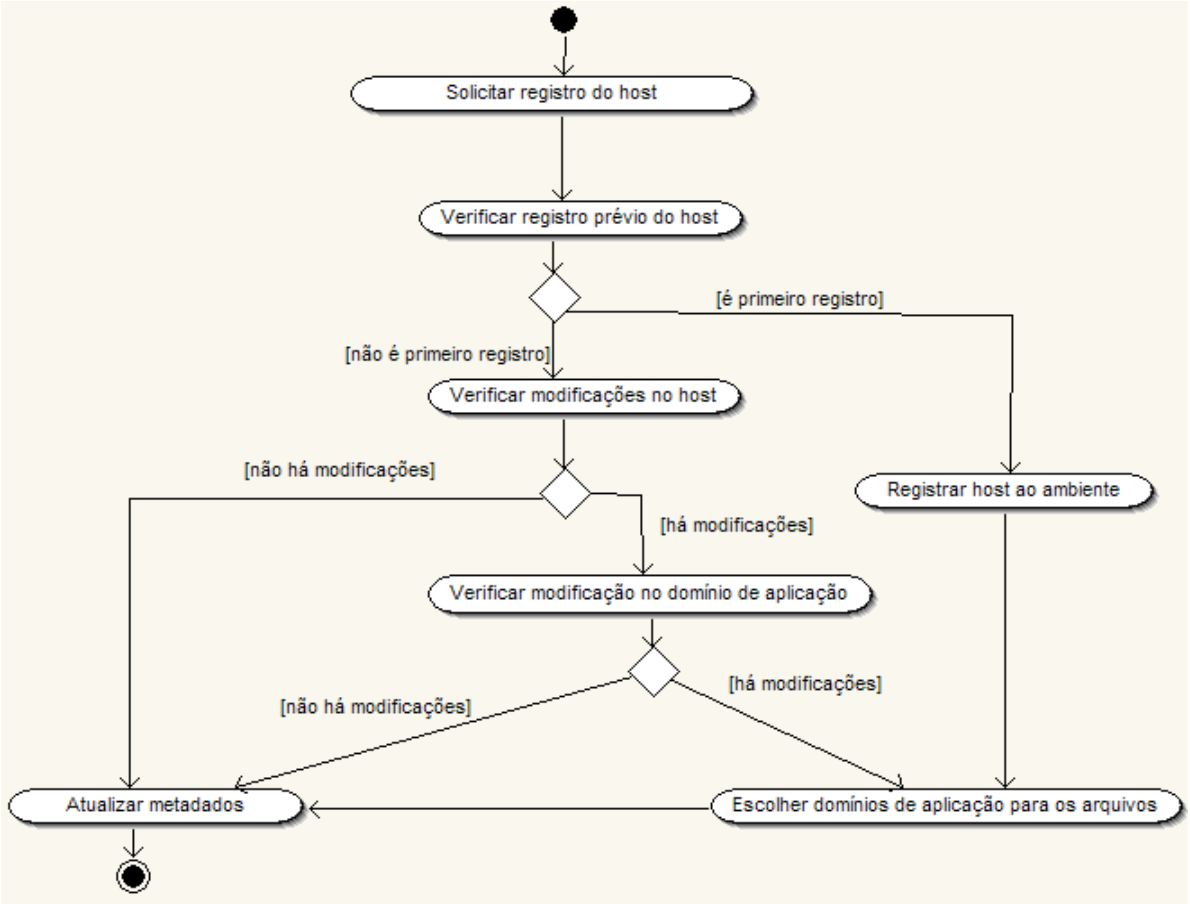



Figura 5.3: Diagrama de atividades do gerenciador de arquivos

Estes resultados são comparados com as informações relacionadas ao *host*, as quais estão localizadas nos metadados, descritos na Seção 5.6. Por exemplo, comparando os dados da mensagem XML da Figura 5.4 com os dados encontrados nos metadados, pode-se concluir se o *host* teve alguma alteração em seus arquivos desde a última conexão.

```
<?xml version="1.0" encoding="UTF-8"?>
<message hostID="Host01">
        <file fileIDID="F1" hashResult="ece50ed4d6d48dac839bfe8fa719fcff" registeringTime="10/10/2005"
        modificationTime="08/08/2004"/>
        <file fileIDID="F2" hashResult="e3732b09b5b2a9aa452b8ef7802db638" registeringTime="10/15/2005"
        modificationTime="08/08/2004"/>
        <file fileIDID="F3" hashResult="73cbe8e94c7fa839ba1246b34b2a49cd" registeringTime="10/20/2005"
        modificationTime="08/08/2004"/>
</message>
```

Figura 5.4: Mensagem XML

**Verificar modificações no domínio de aplicação.** Para os arquivos que sofreram alguma modificação, o sistema dispara um serviço de verificação de casamento entre o arquivo e as ontologias existentes. Esta atividade verifica qual a ontologia que apresenta maior similaridade com o arquivo em questão. Se houver modificações no domínio de aplicação do arquivo, então há a necessidade de atribuir outra ontologia para este arquivo.

**Escolher domínios de aplicação para os arquivos.** O ambiente solicita ao gerenciador de ontologias uma ontologia que descreva o domínio de aplicação do arquivo. A ontologia escolhida é aquela que apresentar maior similaridade com o arquivo, desde que este limiar esteja acima de um valor especificado. A escolha de ontologias é detalhada no módulo gerenciador de ontologias, discutido na seção 5.4.

**Registrar *host* ao ambiente.** Esta atividade será realizada quando for a primeira ocorrência de registro deste *host*.

**Atualizar metadados.** Os metadados do *host* são atualizados.

Depois do *host* ter-se conectado ao sistema, todos os arquivos registrados estarão sendo tratados pelo mecanismo de gerenciamento de réplicas e de versões. Registros de arquivos pertencentes ao mesmo domínio de aplicação de arquivos anteriormente registrados causarão apenas atualização dos metadados. Registros relacionados a um novo domínio implicarão em escolher outra ontologia, além de atualizar os metadados necessários.

## 5.4 Gerenciador de Ontologias

Arquivos do mesmo domínio de aplicação estão espalhados pelo *host*, o que pode gerar ineficiência na busca de recursos quando o usuário submete consultas. Para solucionar este problema, este trabalho propõe agrupar documentos do mesmo domínio de aplicação. Desta forma, o processamento de consultas é realizado somente nos arquivos pertencentes a este domínio.

A tarefa de agrupamento é realizada por um módulo gerenciador de ontologias, composto por três etapas: geração da ontologia, casamento entre a ontologia e os arquivos, e gerenciamento de metadados. Este módulo é responsável pela manutenção do repositório de ontologias e pela associação de ontologias a arquivos. As ontologias são armazenadas no repositório de ontologias, um conjunto de arquivos armazenados em um sistema de arquivos.

As atividades do gerenciador de ontologias são mostradas na Figura 5.5 e detalhadas a seguir.

**Mostrar ontologias disponíveis.** Esta atividade retorna uma breve descrição de cada ontologia armazenada no repositório de ontologias, contendo os principais termos e relações que representam o domínio de conhecimento. Desta forma, o usuário pode escolher uma ontologia para ser utilizada para cada conjunto de documentos pertencentes a um mesmo domínio de aplicação.

**Realizar o casamento entre arquivo e ontologia.** Caso o usuário não escolha uma das ontologias disponíveis, então esta atividade é responsável por sugerir uma ontologia que descreva os conceitos existentes no(s) arquivo (s). O casamento de ontologias e documentos objetiva encontrar correspondências entre entidades semanticamente relacionadas nas duas

representações. Para descobrir qual ontologia melhor descreve um arquivo, este trabalho baseia-se em medidas de similaridade. A ontologia que apresenta maior similaridade (acima de um limiar definido pelo usuário) é escolhida para representar o domínio de aplicação do arquivo. Um *thesaurus* é usado para auxiliar na descoberta dos relacionamentos terminológicos, conforme descrito em (NOLL et al., 2007).

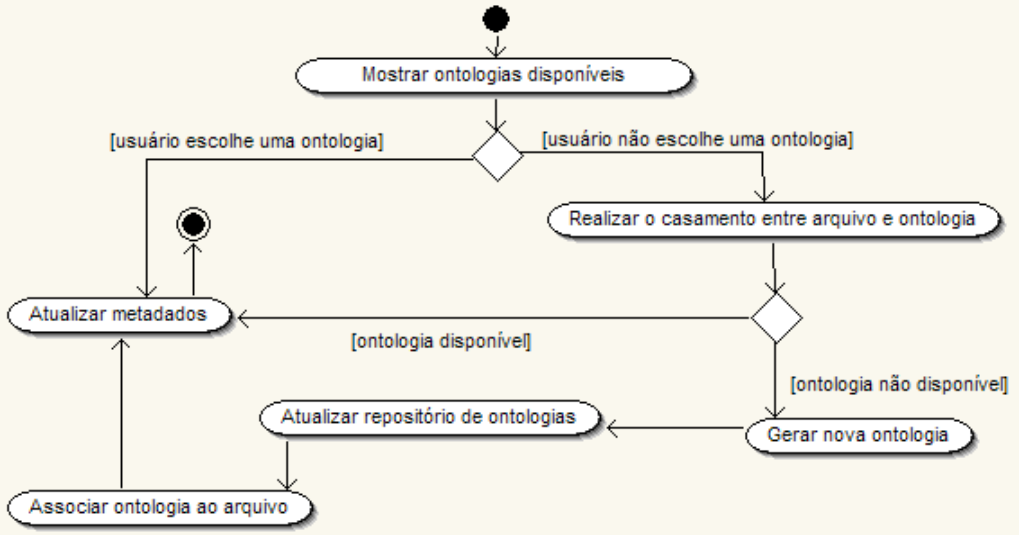



Figura 5.5: Diagrama de atividades do gerenciador de ontologias

**Gerar nova ontologia.** Esta tarefa é realizada sempre que o sistema não encontrar uma ontologia adequada para um (conjunto de) arquivo (s) localizado no *host*. A geração de ontologias é feita a partir da integração dos esquemas dos arquivos compartilhados e é detalhada na seção 5.4.1.

**Atualizar o repositório de ontologias.** Esta atividade é responsável por inserir a ontologia criada no repositório de ontologias.

**Associar uma ontologia ao arquivo.** Supondo que uma nova ontologia foi criada a partir de arquivos XML, esta ontologia precisa ser associada a estes arquivos. Alguns metadados devem ser atualizados.

**Atualizar metadados.** Esta atividade atualiza os metadados do gerenciador de ontologias. Os metadados descritos permitem especificar qual a ontologia (`ontology Oid`) que descreve um determinado arquivo (`fileID`), um rótulo descritivo do domínio (`domain`) e o arquivo em que a ontologia encontra-se armazenada (`file`), conforme mostrado na Figura 5.6.

Estes metadados são mantidos sempre que a atividade de casamento de ontologias e arquivos é realizada. Para um arquivo (`fileID`) armazenado no *host* (`host id`), associado a uma ontologia (`ontology`), os metadados descrevem o respectivo domínio (`domain`) a que este arquivo pertence. Dada uma consulta pertencente a um determinado domínio, o sistema acessa os metadados e verifica quais arquivos podem responder à consulta.

```
<AssociatedOntologies>
  <ontology Oid="Ont1"><domain>Research  Projects</domain> <file>ResearchProjects.owl</file>
    <Files>
      <host id="Host01"><fileID>F2</fileID> <fileID>F7</fileID> <fileID>F8</fileID></host>
    </Files></ontology>
  <ontology Oid ="Ont2"><domain>Diseases</domain> <file>Diseases..owl</file>
    <Files>
      <host id="Host01"><fileID>F66</fileID><fileID>F17</fileID> <fileID>F28</fileID></host>
    </Files></ontology>…
</AssociatedOntologies >
```

Figura 5.6: Metadados do gerenciador de ontologias

As atividades detalhadas na Figura 5.5 são estruturadas na arquitetura ilustrada na Figura 5.7.

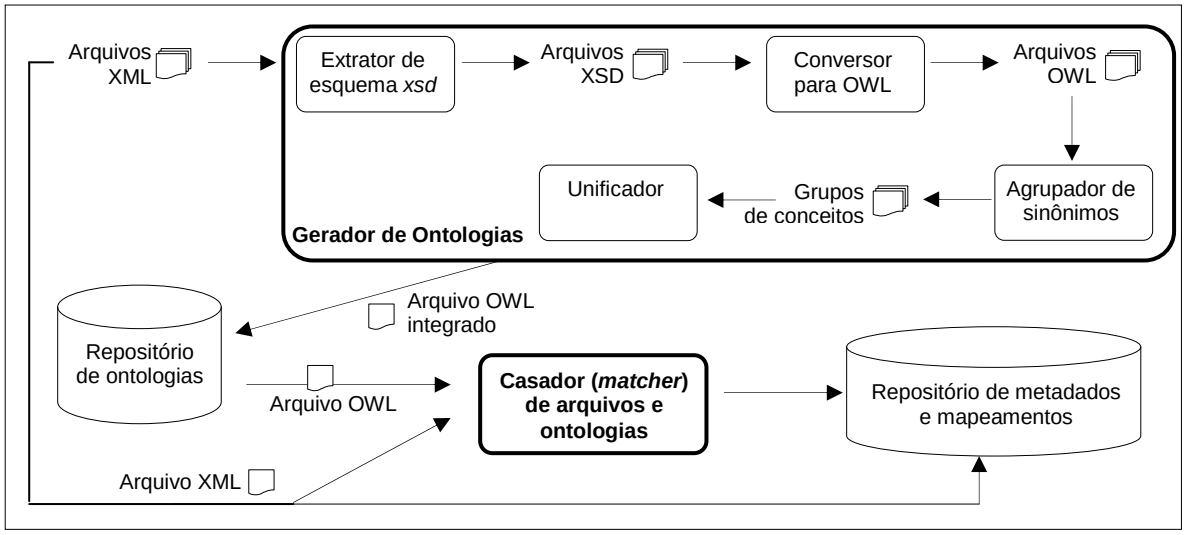



Figura 5.7: Arquitetura do gerenciador de ontologias

O gerenciador de ontologias encapsula a heterogeneidade dos recursos, fornecendo informações sobre as ontologias existentes e os arquivos. Também é responsável por gerar novas ontologias quando nenhuma das existentes é apropriada para um determinado (conjunto de) arquivo (s) XML. A geração da ontologia é feita a partir da integração dos esquemas dos arquivos envolvidos, ou seja, arquivos relacionados a um mesmo domínio de aplicação, para os quais ainda não existe uma ontologia associada. O casamento entre arquivos e ontologias é responsável pela identificação da ontologia que melhor descreve um arquivo.

### 5.4.1 Gerador de Ontologias

Este módulo é responsável por gerar uma ontologia que descreva os conceitos e os relacionamentos existentes em um domínio de aplicação. A ontologia é criada automaticamente a partir da integração de esquemas dos arquivos pertencentes a um determinado domínio de aplicação, para o qual ainda não existe uma ontologia associada. Como a ontologia é criada a partir de um conjunto de arquivos, assume-se que o usuário informa quais os arquivos relacionados a um mesmo domínio de aplicação, para que se possa

gerar a ontologia correspondente a este domínio. Primeiramente, gera-se o esquema XML (isto é, o esquema *xsd*[16]) correspondente ao arquivo compartilhado. O esquema *xsd* é então traduzido para um formato OWL, o qual basicamente descreve os elementos complexos (i.e., não léxicos) e atômicos (i.e., léxicos). Através da aplicação de várias regras de integração, um esquema integrado é gerado, também representado em OWL. Este esquema integrado, representado em OWL, corresponde à ontologia do domínio da aplicação.

O mecanismo considera três tipos de elementos no processo de integração: atômico *x* atômico, complexo *x* não complexo, e complexo *x* atômico. O processo de integração segue uma abordagem binária e utiliza um *thesaurus* para auxiliar na inferência de relacionamentos terminológicos, a fim de realizar a equivalência de conceitos.

#### *5.4.1.1 Abordagem para a Geração de Ontologias*

A geração de ontologias está implementada na ferramenta *Ontogen* (MELLO, 2007). O objetivo geral da ferramenta *OntoGen* é a integração de esquemas XML, gerando ao final do processo uma ontologia que represente os esquemas integrados. No primeiro nível estão os documentos de entrada, que podem ser documentos XML ou esquemas XML representados na linguagem *XML Schema*. Os esquemas XML são repassados diretamente para a próxima etapa, enquanto os documentos XML precisam passar por uma etapa a mais, em que a ferramenta deve extrair o esquema do documento XML de entrada, baseando-se nos dados e na estrutura do documento. Esta etapa é feita automaticamente, utilizando a biblioteca *Castor* (CASTOR, 2008). No último nível está o artefato de saída da ferramenta: o esquema global (isto é, a ontologia). Para obter o esquema global, os esquemas locais devem passar por duas etapas que compõem os dois módulos principais da implementação: a conversão do XSD e a integração semântica.

O módulo de conversão do XSD é responsável pela conversão de cada esquema XML em um modelo conceitual canônico. O uso de um modelo conceitual facilita a etapa posterior, a integração semântica, pois permite uma maior abstração do esquema XML, deixando de lado os detalhes técnicos de estruturação e colocando o foco na semântica dos seus dados (MELLO, 2002). Em (GARCIA, 2005) é apresentada uma ferramenta que tem o objetivo de extrair um modelo conceitual canônico de um documento *XML Schema*, utilizada como base para a implementação da ferramenta *OntoGen*.

O módulo de integração semântica é responsável pela integração das ontologias geradas na etapa de conversão do XSD. Na integração semântica são aplicadas as regras de unificação, para que seja gerada ao final do processo uma única ontologia, no mesmo formato das ontologias geradas anteriormente. Essa ontologia deve representar todas as ontologias de entrada e será considerada a ontologia global, objeto de saída da ferramenta.

Os dois módulos principais da integração semântica são detalhados a seguir.

**Agrupamento de Sinônimos.** É responsável por verificar equivalência semântica entre conceitos de diferentes ontologias, agrupando os conceitos equivalentes em grupos (*clusters*)

---

[16] Optou-se por trabalhar com os esquemas *xsd* ao invés de com os arquivos XML, uma vez que vários arquivos XML podem ser descritos pelo esquema *xsd*.

de afinidade. Este módulo recebe duas ontologias em OWL. Para cada conceito da primeira ontologia, o agrupador de sinônimos procura um conceito semanticamente equivalente na segunda ontologia. Uma função de afinidade é responsável por comparar dois conceitos, devolvendo um coeficiente de afinidade que indica se os dois conceitos são semanticamente equivalentes ou não. Os conceitos semanticamente equivalentes são separados em grupos de afinidade, através do uso das funções de afinidade.

A função de afinidade é responsável por verificar se dois conceitos, de ontologias diferentes, são semanticamente equivalentes. Por questões de simplificação, a função está baseada unicamente no nome dos conceitos. Se os nomes dos conceitos forem idênticos ou sinônimos, estes dois conceitos serão tomados como semanticamente iguais e a função de afinidade retornará 1. Caso contrário, a função de afinidade retornará zero. O dicionário de sinônimos utilizado é o *WordNet*[17]. Exemplos:

Afinidade("*Author*","*Paper*") = 0  Afinidade("*Paper*","*Paper*") = 1  Afinidade("*Author*","*Writer*") = 1

Um grupo de afinidade pode ser composto por no mínimo um e no máximo dois conceitos, pois a estratégia de unificação é binária. Ele será composto por dois conceitos quando a função de afinidade para estes conceitos retornar o valor 1, e por um conceito apenas, caso este conceito não tenha conceito semanticamente igual na outra ontologia. Exemplo:

- Conceitos da ontologia 1: *author*, *paper*, *university*;

- Conceitos da ontologia 2: writer, paper, *title*;

Grupos gerados: c1 = {author, writer}, c2 = {paper, paper}, c3 = {university}, c4 = {title}

O conjunto de grupos de afinidade obtido através do agrupamento de sinônimos de duas ontologias é repassado para a etapa de unificação.

**Unificação.** É responsável por integrar grupos de conceitos, a fim de gerar o esquema global. Essa unificação é baseada no conjunto de grupos de afinidade, gerado no módulo de agrupamento de sinônimos e nas regras de unificação definidas em (MELLO, 2002). As regras de unificação propostas em (MELLO, 2002) foram simplificadas e adaptadas para a proposta deste trabalho, detalhado em (MELLO, 2007). Para cada grupo de afinidade é gerado um conceito na ontologia global, resultado da unificação dos dois conceitos pertencentes ao grupo. Caso o grupo seja composto por um conceito apenas, este conceito é mapeado diretamente para a ontologia global, não sendo necessário aplicar as regras de unificação.

Existem três casos possíveis de unificação para os tipos de grupos: unificação léxica (conceitos léxicos), unificação não léxica (conceitos não léxicos) e unificação mista (conceito léxico e conceito não léxico). As regras implementadas baseiam-se em:

- **unificação de nomes -** caso o grupo seja composto por dois conceitos, *Ci* e *Cj*, e estes dois conceitos tenham nomes diferentes, o nome do conceito gerado a partir do grupo é igual ao nome do conceito *Ci*;

[17] *WordNet*. Disponível em *http://wordnet.princeton.edu.*

- **unificação de tipos -** caso o grupo léxico seja composto por dois conceitos, *Ci* e *Cj*, e estes dois conceitos tenham tipos de dados diferentes, o tipo de dados do conceito gerado é igual ao tipo mais genérico dentre os tipos de dados dos conceitos *Ci* e *Cj*. Exemplos:

{date, date} => date; {integer, float} => float; {string, integer} => string; {integer, date} => string;

- **unificação de relacionamentos -** para cada relacionamento *Ri* da primeira ontologia é feita uma busca por um relacionamento *Rj* na segunda ontologia que possua *afinidade* com *Ri*. Caso o relacionamento *Ri* possua afinidade com o relacionamento *Rj*, estes dois relacionamentos devem ser unificados, observando as regras descritas em (MELLO, 2007);
- **unificação de cardinalidades -** dadas duas cardinalidades, *Cardi* e *Cardj*, a cardinalidade unificada será a mais abrangente. Exemplos:

{“(0,1)”, “(1,N)”} => “(0,N)”; {“(1,1)”, “(0,1)”} => “(0,1)”; {“(1,1)”, “(1,N)”} => “(1,N)”;

A descrição completa da geração de ontologias pode ser encontrada em [Mello 2007].

### *5.4.1.2 Exemplo da Geração*

Considere o documento XML mostrado na Figura 5.8(a). Este documento contém alguns conceitos e relacionamentos descritos por um domínio de publicações, como código e nome de certo autor de artigos. A Figura 5.8(b) mostra outro documento XML, pertencente ao mesmo domínio de publicações. Este documento descreve o nome, o código e o *e-mail* de alguns autores de artigos, mas com uma estrutura diferente do primeiro documento apresentado. Além da estrutura, a presença de alguns sinônimos (como por exemplo, *author* e *writer*) torna o processo de integração mais difícil.

```
<paper>
  <author>
    <code>RC</code>
    <name>Robson Cordeiro</name>
  </author>
  <author>
    <code>RG</code>
    <name>Renata Galante</name>
  </author>
</paper>
```

(a)

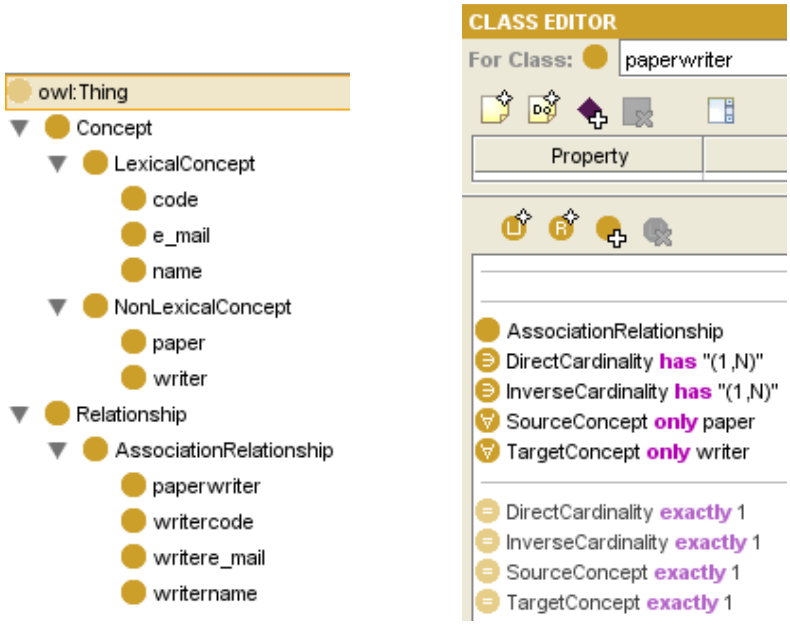
```
<paper>
  <writer code="1">
    <name>Robson Leonardo Ferreira Cordeiro</name>
    <e_mail>rlfcordeiro@inf.ufrgs.br</e_mail>
  </writer>
</paper>
```

(b)

Figura 5.8: Documento XML 1 e Documento XML 2

A ontologia global gerada para os documentos da Figura 5.8 é mostrada na Figura 5.9(a). Conforme descrito na Figura 5.9(a), o esquema integrado que representa a ontologia global contém três conceitos léxicos (*code*, *e-mail* e *name*). Os conceitos *paper* e *writer* estão representados como conceitos não-léxicos. Nota-se que o elemento *author* não aparece na ontologia gerada, uma vez que foi substituído pelo seu sinônimo (*writer*). As relações entre conceitos são expressas como relações de associação. Por exemplo, *paperwriter*

descreve a relação entre *paper* e *writer*. As propriedades das relações também são mantidas, como exemplificadas na Figura 5.9(b).




(a) (b)

Figura 5.9: Ontologia global e descrição de propriedades - visualização na ferramenta *Protegé*

### 5.4.2 Casamento entre Ontologias e Arquivos

O casamento entre ontologias e arquivos XML está implementado na ferramenta *The Matcher* (NOLL, 2007). A tarefa de casamento tem como objetivo escolher a ontologia que melhor descreve os conceitos e os relacionamentos de um arquivo XML. Esta atividade baseia-se na similaridade entre ambas as representações. Dado um arquivo XML *d* e um conjunto de ontologias $O=\{o_1,o_2,o_3,\ldots,o_n\}$, o mecanismo processa um escore de similaridade *sim(d,O)*. A ontologia com o escore mais alto (desde que superior a um limiar *t*) é escolhida para representar o domínio de aplicação do arquivo em questão. Se nenhuma ontologia é escolhida, por não apresentar um escore de similaridade igual ou superior a *t*, então o arquivo é rotulado como “sem ontologia associada”. Arquivos sem ontologias associadas são analisados pelo usuário, a fim de que se possam agrupar possíveis candidatos pertencentes a um mesmo domínio de aplicação a serem enviados para a etapa de geração de nova ontologia.

A estratégia encontra semelhanças entre os elementos do arquivo XML e os elementos da ontologia. Duas questões devem ser tratadas: 1) quais pares de elementos são comparados; e 2) quais são os critérios para determinar o quão similares são tais elementos. Várias abordagens tratam da análise de similaridade (MADHAVAN et al., 2001; MAEDCHE et al., 2002) e são geralmente baseadas em três etapas principais:

- normalização - determina quais elementos são semanticamente equivalentes. Por exemplo, através do uso de um *thesaurus* para normalizar os elementos equivalentes para uma linguagem de domínio comum;
- categorização - separa os elementos em classes, a fim de reduzir o número de comparações;
- Comparação - define o escore de similaridade processado entre os elementos nas suas categorias.

Para avaliar a similaridade entre os arquivos, dois tipos de perspectivas são considerados: a perspectiva léxica que avalia as relações entre termos, comparando as cadeias de caracteres; e a perspectiva semântica, que foca no significado e na relação entre os termos. Para a análise de similaridade, este trabalho considera ambas as perspectivas, conforme descrito a seguir.

**Análise de similaridade léxica**. Duas abordagens são utilizadas:

- funções de edição de distância (LEVENSHTEIN, 1966) - analisa o número mínimo de operações para transformar uma seqüência de caracteres em outra;
- algoritmos de extração de radicais[18] - reduz a seqüência de caracteres ao radical.

A comparação entre os principais algoritmos de análise léxica é apresentada em (KANTROWITZ et al., 2000). O algoritmo *Edit Distance* tem melhor desempenho em casos onde não há análise de grafia (isto é, em textos onde pode ocorrer algum erro de grafia). Uma vez que a taxonomia dos elementos é definida *a priori* em nossa abordagem, este trabalho utiliza algoritmos de extração de radicais para análise de similaridade léxica.

**Análise de similaridade semântica**. Dois tipos de abordagens são utilizados:

- *thesaurus* - usado para avaliar os relacionamentos terminológicos. *WordNet* é um sistema de referência léxica que agrupa palavras em categorias (substantivo, verbo, adjetivo, advérbio etc). Dentro de cada categoria, o sistema organiza as palavras por conceitos (significado) e relacionamentos semânticos. Alguns exemplos de relacionamentos são: sinonímia, antonímia, hiponímia etc;
- procedimento de sobreposição de taxonomias (MAEDCHE et al., 2002) - corresponde à comparação taxonômica entre elementos. Esta comparação não analisa individualmente os elementos, mas o contexto do elemento. Para calcular o nível de similaridade entre dois conjuntos de elementos, pode-se utilizar o coeficiente *Jaccard* (MANNING et al., 1999). A similaridade entre dois conjuntos é definida como a razão das cardinalidades resultantes das operações de interseção e união, como exemplificado na Figura 5.10.

[18] *The Lancaster Stemming Algorithm*. Disponível em *http://www.comp.lancs.ac.uk/computing/research/stemming*.

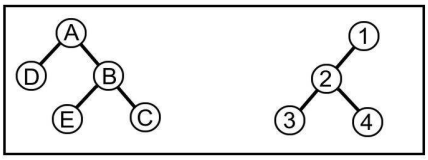



Figura 5.10: Exemplo de duas árvores

Suponha que o nó A é lexicamente equivalente ao nó 1 e que o nó B é lexicamente equivalente ao nó 3. Para analisar a sobreposição taxonômica na hierarquia formada por A, B e C, assuma que a união tem cardinalidade 3 (A, B e C) e que a interseção tem cardinalidade 2 (A é equivalente a 1 e B é equivalente a 3). Desta forma, o valor de similaridade é definido como 2 dividido por 3, ou seja, 0.667.

O mecanismo proposto neste trabalho agrega e estende as vantagens de algumas abordagens existentes, como descrito na próxima seção.

### *5.4.2.1 Abordagem para o Casamento*

Para o cálculo do escore de similaridade $sim(d, o_n)$ entre um arquivo XML *d* e uma ontologia $o_{n,}$ utiliza-se o mecanismo a seguir. O primeiro passo corresponde às etapas de normalização e categorização.

**Fase de normalização e categorização.** Para cada arquivo (XML e OWL), mapeiam-se todos os elementos componentes e armazena-se o radical (chave) e uma lista com os nomes completos de cada elemento. Se o nome do elemento é composto por *n*-palavras, então a lista também consiste de cada componente. Este mapeamento inicial corresponde à perspectiva léxica e é exemplificado na Figura 5.11.

| **Key** | **List** | | |
|---|---|---|---|
| skill | skillArea | skill | area |

Figura 5.11: Normalização léxica de elementos

O próximo passo considera os sinônimos dos elementos. A lista de sinônimos é recuperada do *WordNet*. A lista resultante é apresentada na Figura 5.12.

| **Key** | **List** | | | | | |
|---|---|---|---|---|---|---|
| skill | skillArea | skill | area | accomplishment | acquisition | domain |

Figura 5.12: Normalização semântica de elementos

A normalização ocorre em dois passos. Primeiro, normaliza-se os elementos da ontologia, como descrito acima. Percorrem-se os elementos XML e verifica-se em suas listas a existência de qualquer correspondência léxica com outros elementos da lista OWL. Estas correspondências são analisadas observando-se apenas os radicais, utilizando um extrator de radicais. Finalmente, têm-se disponíveis duas listas categorizadas e normalizadas (XML e OWL). Nesta fase, o sistema está apto a identificar as correspondências existentes entre os elementos XML e OWL, fornecendo o passo inicial para a comparação de elementos.

**Fase de comparação.** Esta fase analisa a sobreposição de taxonomias. Conforme mencionado, não são considerados os elementos individualmente, mas seu contexto. Para

cada elemento que consiste de um conjunto *root-leaf* (isto é, o caminho da raiz aos nós folha), o sistema identifica a existência de correspondências na ontologia (obtido durante a fase de normalização). Para obter o valor de similaridade, define-se *união* como o número total de elementos XML do conjunto *root-leaf*. A *intersecção* é definida como o número de relações com correspondência entre os elementos XML e da ontologia. Por relação, entende-se: uma classe OWL que se relaciona a uma subclasse OWL; uma classe OWL que se relaciona a propriedades *Object* ou *Datatype*; e relações entre classes OWL.

Por exemplo, suponha que o conjunto A, B e C, mostrado na Figura 5.10, define um conjunto *root-leaf* em uma estrutura XML. Este conjunto tem correspondências com a árvore da direita. Assuma que:

- existe uma correspondência léxica entre A e 1 e entre B e 3, definida pelos radicais de qualquer elemento presente nas listas normalizadas;

- existe uma propriedade transitiva entre 1 e 2 e entre 2 e 3. Desta forma, há uma propriedade implícita entre os elementos 1 e 3, representada pelas propriedades transitivas.

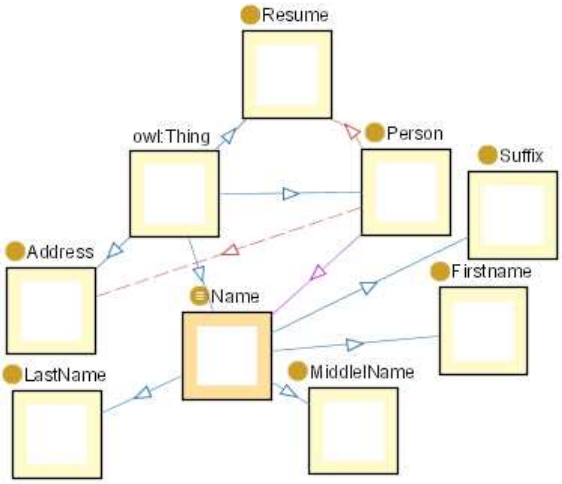
O método de sobreposição de taxonomias percorre a árvore da esquerda e analisa somente os nós que têm correspondência com a árvore da direita (isto é, o sistema verifica se os elementos lexicamente similares a A e B têm equivalência semântica). Em outras palavras, uma vez que A e B são diretamente relacionados, o sistema verifica se seus respectivos equivalentes (isto é, 1 e 3) também possuem este relacionamento. A propriedade transitiva permite identificar que 1 se relaciona com 3, de forma que a interseção vale 2. Como o número total de elementos considerados é 3 (A, B e C), o valor de similaridade é 0.667. Este procedimento é repetido para todos os conjuntos *root-leaf*. Desta forma, obtém-se uma lista com todos os valores de similaridade. O valor final é calculado como a média aritmética desta lista.

#### *5.4.2.2 Exemplo do Casamento*

Considere o arquivo XML mostrado na Figura 5.13(a). A Figura 5.13(b) apresenta parte da ontologia de currículos (algumas propriedades de *address* não são mostradas, tais como *city*, *state*, *street* e *zip*). Suponha que exista outra ontologia representativa de outro domínio, como, por exemplo, o domínio de projetos de pesquisa. O objetivo é avaliar qual a ontologia que melhor descreve o documento XML, através da análise de similaridade entre o arquivo XML e as ontologias.

Utilizando o mecanismo descrito na seção 5.4.2, a abordagem de casamento gera os valores individuais (*ISim*), parciais e finais de similaridade para os arquivos apresentados na Figura 5.13, descritos na Tabela 5.1. Observando-se a última linha desta tabela, observa-se que a primeira ontologia (*resume.owl*) apresenta a similaridade mais alta (0.899). Desta forma, esta ontologia é escolhida para descrever o domínio do arquivo XML em questão.

```
<resume>
<header>
  <name><firstname>Jo</firstname>
  <surname>Doe</surname></name>
        <address><street>123 Elm #456</street>
                    <city>Garbonzoville</city><state>NX</state>
                    <zip>99999-9999</zip></address>
       <contact><phone>555.555.5555</phone>
       <email>doe@doe.doe</email>
         <url>http://doe.com/~doe/</url></contact>
</header>
</resume>
```




(a) (b)

Figura 5.13: Documento XML e ontologia do domínio de currículos

Tabela 5.1: Valores individuais (*ISim*), parciais e finais de similaridade

| **XML (Resume)** | **Ontology (Resumé)** | **ISim** | **Ontologia (Academic Research)** | **ISim** |
|---|---|---|---|---|
| **[header, name, firstname]** | [Name, Firstname] | 0.667 | [firstName, firstName] | 0.333 |
| **[header, name, surname]** | [Name, LastName] | 0.667 | [firstName, lastName] | 0.333 |
| **[header, name]** | [name] | 0.500 | [name] | 0.500 |
| **[header, address, street]** | [Address, street] | 0.667 | [address] | 0.333 |
| **[header, address, city]** | [Address, city] | 0.667 | [address] | 0.333 |
| **[header, address, state]** | [Address, state] | 0.667 | [address, Department] | 0.667 |
| **[header, address, zip]** | [Address, zip] | 0.667 | [address, nil] | 0.333 |
| **[header, address]** | [address] | 0.500 | [address] | 0.500 |
| **[header, contact, phone]** | [phone] | 0.333 | [phone] | 0.333 |
| **[header, contact, email]** | [email] | 0.333 | [email] | 0.333 |
| **[header, contact, url]** | [url] | 0.333 | [url] | 0.333 |
| **[header, contact]** | [] | 0.000 | [] | 0.000 |
| **[header]** | [] | 0.000 | [] | 0.000 |
| **…** | … | … | … | … |
| **Valor parcial de similaridade** | | **0.462** | | **0.333** |
| **Valor final de similaridade** | | **0.899** | | **0.433** |

Enquanto analisa as similaridades parciais, o mecanismo é capaz de armazenar informações sobre os mapeamentos entre os conceitos dos documentos XML e da ontologia. Basicamente, para cada termo na ontologia a informação de mapeamento é representada como uma função de transformação. As funções de transformação são representadas em *XPath*[19]. Por exemplo, o conceito *fullName* pode ser mapeado para diferentes documentos, como, por exemplo como *name* ou *firstName+lastName*. Tais mapeamentos são usados na transformação de consultas, conforme descrito no capítulo 5.7.

### 5.4.3 Implementação: *Ontogen* e *The Matcher*

Esta seção detalha as ferramentas implementadas para a geração de ontologias e casamento entre arquivos e ontologias.

#### *5.4.3.1 Geração de Ontologias*

A abordagem de geração de ontologias foi implementada na ferramenta *OntoGen* (MELLO, 2007). Duas etapas foram realizadas: conversão do XSD e integração semântica. Na conversão do XSD, tomou-se como base o trabalho de (GARCIA, 2005), que desenvolveu uma ferramenta para extrair um modelo conceitual canônico de um documento *.xsd*. Esta ferramenta foi gentilmente cedida pelo grupo de banco de dados da UFSC, por meio do coordenador Ronaldo dos Santos Mello, para ser incorporada na ferramenta *OntoGen*. Foram realizadas algumas modificações no modelo conceitual canônico (MCC), proposto por (MELLO, 2002) e gerado por (GARCIA, 2005), tais como: representação na linguagem OWL (originalmente o modelo conceitual canônico estava sendo gerado na linguagem DAML+OIL), e simplificação do modelo conceitual (algumas construções foram retiradas do modelo para fins de simplificação ou porque estavam fora do objetivo do trabalho proposto).

A Figura 5.14 apresenta a tela inicial da ferramenta *OntoGen*, que tem como principais funcionalidades: inserir um documento XML/XSD, excluir um documento, alterar um documento e processar a integração dos documentos incluídos.

A integração das ontologias intermediárias resulta na ontologia global.

**Estudo de Caso.** Como demonstração das funcionalidades implementadas na ferramenta *OntoGen*, é apresentado um breve estudo de caso que detalha o processo de integração de três esquemas gerados a partir de três documentos XML. Para melhor entendimento e visualização dos resultados, os arquivos de entrada e os arquivos gerados pela ferramenta são apresentados utilizando representações gráficas. Os arquivos de entrada são descritos nos Anexos C, D e E.

Para prosseguir à integração, os três documentos XML devem ser carregados na lista de esquemas locais. Após os documentos serem carregados, inicia-se o processo de integração. O primeiro passo é a extração do esquema de cada documento XML, utilizando uma funcionalidade da biblioteca Castor (CASTOR, 2007). Após a extração dos esquemas dos documentos XML de entrada, a ferramenta converte cada esquema em um modelo conceitual

[19] *XML Path Language*. Disponível em *http://www.w3.org/TR/xpath20/*

canônico (ontologia). Esta conversão é realizada através da ferramenta implementada no trabalho de Garcia (2005) que foi adaptada para ser utilizada na ferramenta *OntoGen*.


Figura 5.14: *OntoGen* - Tela inicial da ferramenta

O resultado da conversão do XSD de cada esquema gera uma ontologia. Ao comparar as três ontologias geradas, diversos conflitos são identificados, como, por exemplo: nos esquemas “A” e “C” o conceito autor é denominado “author” e no esquema “B” este mesmo conceito é denominado “writer”. A etapa de integração semântica é responsável por resolver estes conflitos, através das regras de integração.

Depois de terminado o processo de integração, a ferramenta *OntoGen* salva a ontologia global em um documento OWL. A Figura 5.15 apresenta a ontologia global gerada pela ferramenta *OntoGen*, resultado da integração dos esquemas “A”, “B” e “C”.

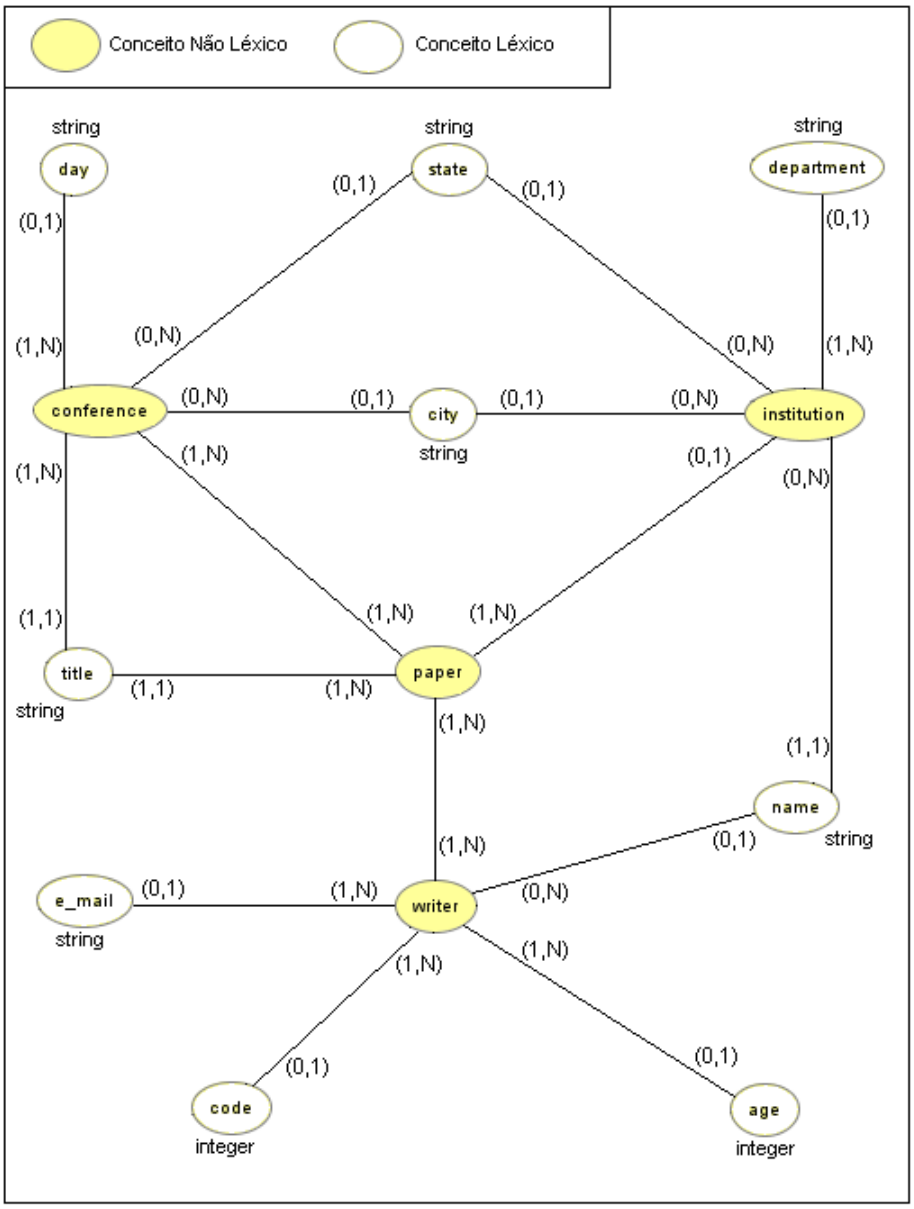



Figura 5.15: Ontologia global gerada a partir da integração dos esquemas A, B e C

### *5.4.3.2 Casamento entre Ontologias e Arquivos XML*

A abordagem de casamento entre ontologias e arquivos XML foi implementada em uma ferramenta chamada *The Matcher* (NOLL, 2007). A ferramenta mede a similaridade léxica e semântica entre arquivos XML e ontologias OWL. A Figura 5.16 apresenta a tela inicial da ferramenta *The Matcher*, que tem como principais funcionalidades escolher um documento XML, escolher uma ontologia OWL e calcular o valor de similaridade.

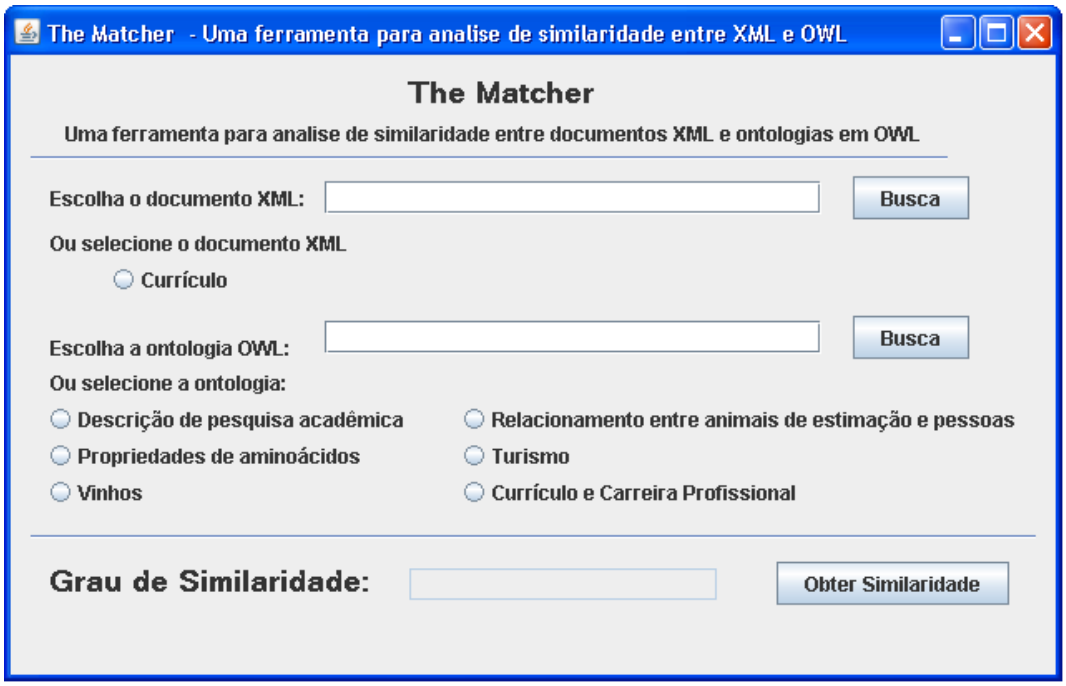



Figura 5.16: *The Matcher* - Tela inicial da ferramenta

A seção a seguir detalha um estudo de caso realizado na ferramenta.

**Estudo de Caso.** Foram realizados vários experimentos para calcular a similaridade entre um arquivo contendo dados de um currículo e algumas ontologias. O objetivo dos experimentos é provar que um arquivo pertencente a um domínio D tem similaridade mais alta do que ontologias de outros domínios. A entrada consiste de um arquivo XML (apresentado no Anexo B) e seis arquivos OWL. As ontologias são relacionadas aos seguintes domínios: *academic research* (O1), *amino acids* (O2), *wine* (O3), *pets and owners* (O4), *travel* (O5), and *curriculum* (O6). Os resultados da similaridade são apresentados na Figura 5.17.

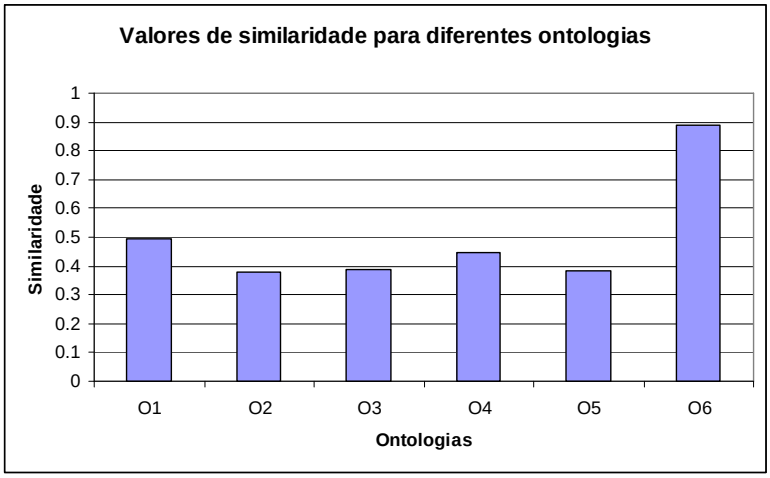



Figura 5.17: Valores de similaridade entre o arquivo XML e as ontologias

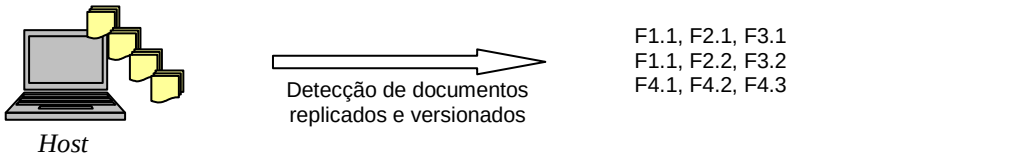
Os valores de similaridade mostram que a ontologia *resume.owl* tem a similaridade mais alta com o documento XML usado no experimento (89.9%). Outras ontologias de domínios diferentes apresentaram similares bem mais baixas (entre 38% e 49%). Desta forma, a ontologia *Resume* é escolhida para descrever o documento.

## 5.5 Gerenciador de Réplicas e de Versões

O módulo de gerenciamento de réplicas e de versões, foco deste trabalho, é responsável por detectar e gerenciar réplicas e versões de documentos. Para isso, devem-se observar algumas situações que podem ocorrer quando um arquivo é registrado:

- o arquivo é uma nova instância - isso significa que este arquivo é um novo documento no sistema e o *host* ainda não possui este documento armazenado;
- o arquivo é uma nova versão de um documento já existente. Neste caso, outra versão deste documento já existe no *host*;
- o arquivo é uma réplica de um outro documento armazenado no *host*.

Para a detecção de réplicas e de versões de documentos XML, este trabalho propõe um método específico, detalhado no capítulo 3. A Figura 5.18(a) mostra um ambiente tradicional, com vários arquivos XML. Na Figura 5.18(b), diferentes versões e réplicas do mesmo documento são detectadas. A partir da Figura 5.18, pode-se concluir que o *host* possui réplicas da versão 1 do documento 1 (F1.1). Além disso, possui três versões sucessivas do arquivo 4 (*F4.1, F4.2, F4.3*).




(a) Documentos replicados e versionados (b) Documentos replicados e versionados detectados

Figura 5.18: Mecanismo automático de detecção e de representação de versões

O trabalho proposto assume que a evolução do documento cria seqüências lineares ou ramificadas de versões temporalmente ordenadas, conforme detalhado no Capítulo 3.

### 5.5.1 Detecção de Réplicas e de Versões

O mecanismo de detecção de réplicas e de versões é executado sempre que um arquivo é registrado (inserido ou atualizado) no *host*. Quando um arquivo é removido do *host*, somente os metadados precisam ser atualizados. Para detectar quando um *host* foi modificado, propõe-se um serviço de verificação de modificações, o qual é responsável por periodicamente observar o *host* e notificar o ambiente quando uma alteração for detectada.

A Figura 5.19 sumariza as atividades relacionadas à detecção de réplicas e de versões, mostradas em um diagrama de atividades. Estas atividades são detalhadas a seguir.

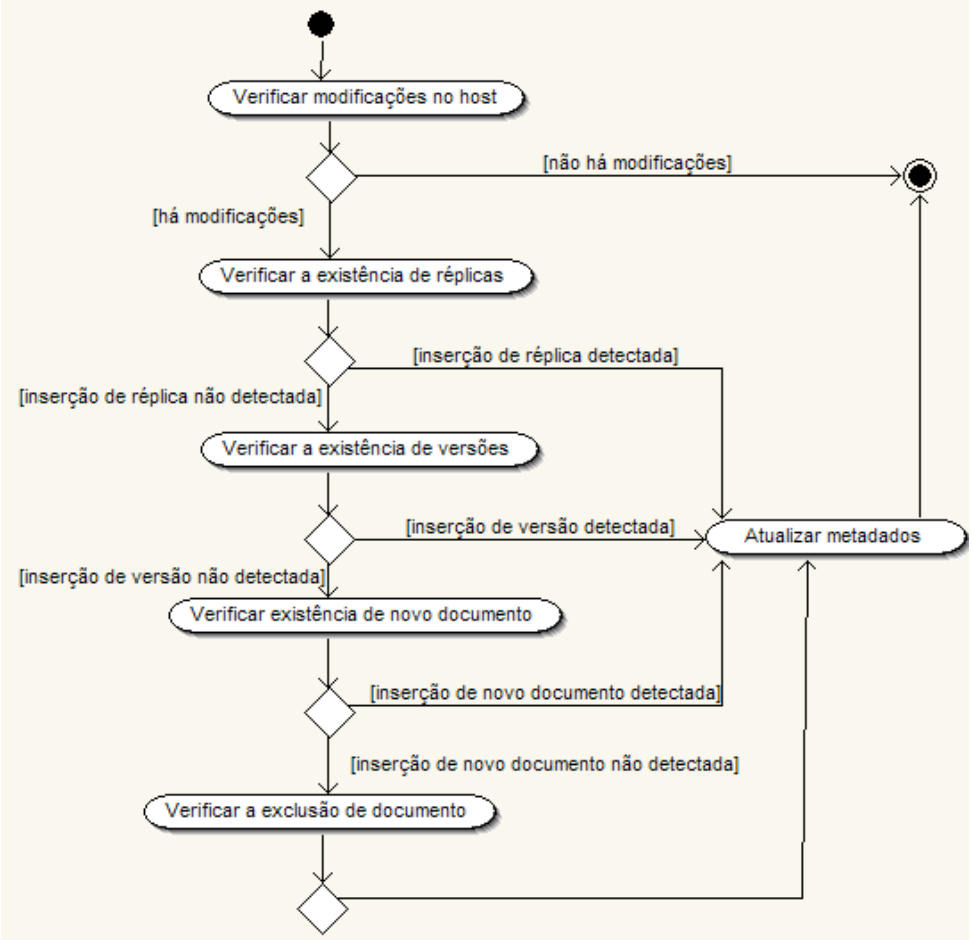



Figura 5.19: Diagrama de atividades do gerenciador de réplicas e de versões

**Verificar modificações no *host*.** O ambiente dispara um serviço de verificação de mudanças no *host*. Esta atividade é similar à atividade de verificar modificações no *host* apresentada no gerenciador de arquivos.

Comparando-se a mensagem XML e os metadados dos documentos no *host*, pode-se concluir quais arquivos não foram modificados desde a última verificação (tempos de modificação e resultado *hash* não sofreram alteração). Uma modificação num *host* pode ser a inserção (de réplicas, versões ou documentos novos), exclusão (de réplicas ou de versões) ou atualização (de versões) de arquivos. Esta atividade retorna uma lista de itens (arquivos) modificados. Para cada arquivo modificado, a lista apresenta o resultado *hash*, o tempo de registro, o tempo de modificação e o identificador local do documento.

**Verificar a existência de réplicas.** Esta atividade verifica se um arquivo modificado corresponde a uma réplica de algum outro arquivo armazenado no *host*. Compara-se o resultado *hash* do arquivo com todos os resultados *hash* armazenados nos metadados. Se um

casamento for detectado, isso significa que o arquivo é uma réplica de outro arquivo existente. A detecção de réplicas é detalhada no Capítulo 3.

**Verificar a existência de versões.** Se o arquivo modificado não é uma réplica, ele pode ser uma versão de algum outro documento armazenado no *host*. A detecção de versões é detalhada no Capítulo 3.

**Verificar a existência de um novo documento.** Se a modificação detectada não é a inserção de uma réplica nem de uma versão de um documento existente, então esta atividade verifica se o arquivo é um novo documento. Se o resultado *hash* deste arquivo não for encontrado nos metadados do *host* nem possuir similaridade alta com outros arquivos (não é versão de um arquivo existente), então se conclui que é um novo documento.

**Verificar a exclusão de um documento.** Se a lista de itens modificados finalizar e nenhuma versão, réplica ou documento novo for detectado, conclui-se que a modificação ocorrida no *host* trata-se de uma exclusão de arquivo. Assim, apenas é necessário atualizar os metadados (próxima atividade), para remover as informações referentes ao(s) arquivo(s) excluído(s). Sabe-se que um arquivo foi excluído, comparando-se os metadados do *host* com a mensagem XML retornada. Metadados sobre os arquivos para os quais existem informações no *host*, mas não existem informações na mensagem XML, devem ser removidos.

**Atualizar metadados.** Esta atividade é responsável por atualizar metadados. Dependendo das modificações que foram detectadas, diferentes atualizações são necessárias.

## 5.6 Metadados

Os módulos propostos na arquitetura fazem uso de metadados para gerenciamento das réplicas e das versões armazenadas no *host.* Considere que o *host* armazena dois arquivos XML, *f1* e *f2*, conforme mostrado na Figura 5.20.

```
<resume>
    <name>
        <firstname>Hu</firstname>
        <surname>Doe</surname>
    </name>
    <address>
            <street>123 Elm #456</street>
            <city>Garbonzoville</city>
            <state>NX</state>
            <zip>99999-9999</zip>
            <contact>555.555.5555</contact>
    </address>
</resume>
```

(a)

```
<resume>
    <fullName>Jo Doe</fullName>
    <address>
            <adr>123 Elm #456, Garbonzoville </adr>
            <state>NX</state>
            <zipCode>99999-9999</zipCode>
    </address>
    <contact>
        <phone>555.555.5555</phone>
        <email>doe@doe.doe</email>
        <url>http://doe.com/~doe/</url>
         </contact>
</resume>
```

(b)

Figura 5.20: Dois arquivos XML armazenados no *host*

Considere a consulta *Q1*: “obter o nome das pessoas que vivem em *Garbonzoville*”. O objetivo é garantir que o usuário não se preocupe com as incompatibilidades estruturais e de conteúdo entre os arquivos, nem com a localização do arquivo no *host*. Dois problemas devem ser solucionados (MENA et al., 2001):

- descoberta de recursos - quais os arquivos que satisfazem a consulta? Onde estes arquivos estão localizados?
- problema de vocabulário - como lidar com incompatibilidades estruturais e de conteúdo? Como acessar a informação desejada?

Dois aspectos devem ser considerados para solucionar estas questões. Primeiro, documentos com estruturas diferentes podem ser descritos pela mesma ontologia, como exemplificado na Figura 5.21. Desta forma, o usuário consulta o sistema, expressando descrições estruturais e de conteúdo com base na ontologia. O sistema encontra e recupera a informação, com base nos metadados disponíveis.

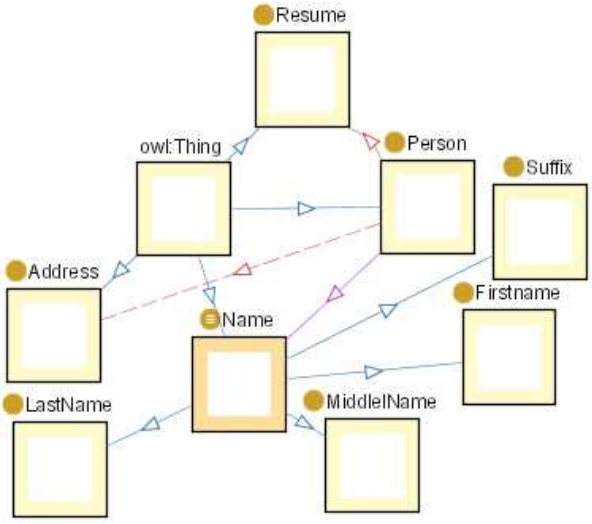



Figura 5.21: Conceitos do domínio de aplicação descritos por uma ontologia

O foco desta seção é a manutenção dos metadados e dos mapeamentos, gerados pelo gerenciador de arquivos, pelo gerenciador de ontologias e pelo gerenciador de réplicas e de versões. A sua manutenção permite aos usuários submeter consultas com base na ontologia. Por exemplo, a consulta *Q1* formulada sobre a ontologia é expressa como "obter o primeiro nome da pessoa cujo endereço contém *Garbonzoville*". Com base nos metadados e mapeamentos, o sistema verifica:

- onde estão localizados os arquivos associados à ontologia utilizada - esta análise é feita, consultando-se os metadados do gerenciador da ontologia, descritos na seção 5.6.2;

- quais os arquivos associados à ontologia utilizada para a formulação da consulta – esta análise é feita, consultando-se os do *host*, descritos na seção 5.6.1;

- como extrair os conceitos solicitados na consulta dos arquivos XML - esta análise é feita, consultando-se os mapeamentos entre os termos da ontologia e dos arquivos XML, descritos na seção 5.6.3.

Neste trabalho, metadados são descritos em XML e classificados em dois níveis: metadados do *host* e metadados do gerenciador de ontologias. Os níveis de metadados são descritos a seguir.

### 5.6.1 Metadados do *Host*

Cada *host* possui metadados com informações sobre seus arquivos. Estes metadados podem ser estruturados em um documento XML, como mostrado na Figura 5.22. Os metadados especificam as versões e as réplicas disponíveis no *host*, os identificadores locais (`fileID`), os resultados hash (`hashResult`), os tempos de registro (`registeringTime`) e a última data de atualização (`modificationTime`) de cada arquivo no *host*. O atributo *fileID* é um valor único mapeado a partir do resultado de uma função *hash.* Os metadados também descrevem os rótulos temporais correspondentes para cada elemento (`TS, TE`) encontrado em um dado arquivo (`fileID`). Estes rótulos temporais são derivados a partir das datas de modificação dos arquivos (`modificationTime`).

```
1.  <Metadata>
2.  …
3.  <document docID=”D1” fileID="F7" HDoc=’’YES”>
4.  <version   versionID="1" registeringTime="10/10/2005" modificationTime="08/08/2004" duplicate="no"
5.                  hashResult="d49622ddab3733549e54749755fd52b5">
6.    <element name="author" TS="08/08/2004" TE="10/15/2004"/>
7.    <element name="address" TS="08/08/2004" TE="10/15/2004"/></version>
8.  <version versionID="2" …</version>
9.  </document>
10. …
11. <document docID=”D1” fileID="F8" HDoc=’’YES”>
12. <version   versionID="2" registeringTime="10/16/2004" modificationTime="08/08/2005" duplicate="no"
13.                 hashResult="749752ddab3733549e54749755f78g65">
14.   <element name="author" TS="08/08/2005" TE="now"/>
15.   <element name="address" TS="08/08/2005" TE="now"/></version>
16. <version versionID="2" …</version>
17. </document>
18. …
19. </Metadata>
```

Figura 5.22: Metadados do *host*

Assume-se que:

- cada elemento possui dois atributos temporais, *TS* e *TE*;
- *TS* é inicializado com o tempo de modificação do arquivo (`modificationTime`) no qual o elemento foi adicionado/modificado;
- *TE* é inicializado com o valor *now* (que representa o momento atual) e permanece com este valor enquanto for válido. Um elemento é considerado válido até sua próxima modificação;
- *TE* é atualizado quando um elemento é modificado. Neste caso, *TE* é encerrado quando uma nova instância de elemento é criada (com *TS* e *TE* iguais a *now*).

Os metadados são atualizados sempre que um novo arquivo é registrado ou quando os arquivos do *host* são modificados. Estes metadados são acessados nas seguintes situações:

- verificar a ocorrência de alguma modificação em um host - modificações no *host* são detectadas quando o resultado da função *hash* aplicada sobre um arquivo retorna um valor diferente dos resultados armazenados nos metadados. Também pode ser levado

em consideração o atributo *modificationTime*, o qual representa a última data de atualização deste arquivo no *host;*

- inserir metadados na primeira conexão de um *host* - neste caso, os metadados são atualizados com informações como: qual é o resultado *hash* dos arquivos, qual é o tempo de registro destes arquivos (tempo em que o arquivo foi registrado pelo *host* para gerenciamento pelo *framework*) e qual é a última data de atualização deste arquivo (`modificationTime`);
- atualizar os metadados após a reconexão de um *host* (quando os arquivos do *host* foram modificados).

Os rótulos temporais precisam ser atualizados sempre que uma nova versão de documento é detectada em um *host.* Três situações precisam ser observadas ao representar um documento:

1. o documento é replicado, mas não versionado - neste caso, o *TS* do elemento é definido como o tempo de modificação da réplica mais antiga; *TE* é definido como *now*;
2. o documento é versionado, mas não replicado - em uma versão *n*, o *TS* de um elemento é definido como o tempo de modificação da *n-ésima* versão; *TE* é definido como o (tempo de modificação - 1) da versão *n-ésima+1*, onde *n-ésima+1* é a versão em que o elemento foi modificado ou removido do documento;
3. o documento é replicado e versionado - em uma versão *n*, o *TS* é definido como o tempo de modificação da réplica mais antiga, e *TE* é definido como o (tempo de registro -1) da primeira representação da *n-ésima+1* versão, onde *n-ésima+1* é a versão em que o elemento foi modificado ou removido do documento.

### 5.6.2 Metadados do Gerenciador de Ontologias

Os metadados do gerenciador de ontologias descrevem informações que relacionam os arquivos XML e a ontologia que descreve tais arquivos. Informações sobre a associação de ontologias e arquivos são descritas em um arquivo XML, como mostrado na Figura 5.23:. Os metadados permitem especificar quais ontologias (atributo ontology Oid) descrevem o *host*, um rótulo descritivo do domínio (domain) e o arquivo onde a ontologia está descrita (file). Metadados também representam quais arquivos são descritos por uma dada ontologia (fileID).

```
1.  <AssociatedOntologies>
2.    <ontology Oid="Ont1"><domain>Research  Projects</domain> <file>ResearchProjects.owl</file>
3.      <Files>
4.        <fileID>F2</fileID> <fileID>F7</fileID> <fileID>F8</fileID>
5.      </Files></ontology>
6.    <ontology Oid ="Ont2"><domain>Diseases</domain> <file>Diseases.owl</file>
7.      <Files>
8.        <fileID>F66</fileID><fileID>F17</fileID> <fileID>F28</fileID>
9.      </Files></ontology>…
10. </AssociatedOntologies >
```

Figura 5.23: Metadados do gerenciador de ontologias

No exemplo citado, a ontologia *Research Projects* (linha 2) está descrita no arquivo *ResearchProjects.owl* (linha 2). O *host* possui alguns arquivos deste domínio, mais

especificamente os arquivos *F2*, *F7* e *F8* (linha 4). Outro fato que pode-se observar é que a ontologia *Diseases* (linha 6) está descrita no arquivo *Diseases.owl* (linha 6). O *host* também possui alguns arquivos deste domínio, mais especificamente os arquivos *F66*, *F17* e *F28* (linha 8).

Estes metadados são mantidos sempre que for realizado o casamento de arquivos XML e ontologias. Dada uma consulta para certo domínio de aplicação, o sistema acessa os metadados e avalia quais arquivos devem ser acessados para responder tal consulta. Assume-se que uma consulta é relacionada a um domínio de aplicação; desta forma, não há a necessidade de correlações entre domínios. Por exemplo, uma consulta submetida sobre o domínio *research projects* é formulada na ontologia *Ont1*, como descrito na Figura 5.23: (linha 2). Analisando-se os metadados, o sistema sabe quais os arquivos são associados ao domínio desejado (*F2*, *F7* e *F8*, linha 4). Desta forma, apenas os arquivos necessários são acessados no processamento da consulta, ao invés de enviar a consulta a todos os arquivos registrados no *host*.

### 5.6.3 Mapeamentos

Instâncias correspondentes aos termos na ontologia são armazenadas nos arquivos XML do *host*. Informações de mapeamento relacionam os termos na ontologia aos elementos de dados nos arquivos XML. Mapeamentos são informações necessárias para recuperar os dados correspondentes a cada termo na ontologia. O gerenciamento envolve a geração e o armazenamento destes mapeamentos.

Duas abordagens gerais podem ser usadas para especificar mapeamentos entre ontologias e repositórios de dados, e desta forma usar os mapeamentos para reformulação de consultas (FLORESCU et al., 1998): *global-as-view* (GAV) e *local-as-view* (LAV). Na abordagem LAV, esquemas locais são definidos em termos do esquema global, e o mapeamento é especificado definindo-se cada construção do esquema local como uma visão da construção do esquema global. Na abordagem GAV, o esquema global é definido em termos dos esquemas locais, e o mapeamento é especificado definindo-se cada construção do esquema global como uma visão sobre os esquemas locais.

Este trabalho assume a abordagem GAV, uma vez que uma consulta sobre os arquivos é formulada para cada termo na ontologia. Esta consulta especifica como obter as instâncias dos repositórios. Desta forma, cada termo na ontologia tem associado uma informação de mapeamento que relaciona o termo nos arquivos XML. Os mapeamentos são definidos/atualizados sempre que o casamento entre as ontologias e os arquivos é realizado, e são bastante importantes para a transformação de consultas, como mostrado na Seção 5.7.

A informação de mapeamento é representada como uma função de transformação. A definição de um mapeamento na ontologia é uma lista de mapeamentos (um mapeamento para cada documento XML que tenha estrutura diversa). As funções de transformação são representadas em *XPath* (XPATH, 2008). Além de relacionar conceitos, as funções de transformação também podem alterar valores de um elemento ou atributo, de forma que o significado da informação não seja alterado. Por exemplo, se os dados no documento XML estão armazenados em um formato (por exemplo, peso em gramas), mas o conceito foi

definido na ontologia em um formato diferente (por exemplo, peso em quilos), os mapeamentos devem especificar uma função que transforme tais representações.

Alguns mapeamentos são descritos na Tabela 5.2 para a ontologia apresentada na Figura 5.21 e os arquivos XML mostrados na Figura 5.20. Assuma que o elemento *address* na ontologia corresponde à concatenação de rua, número, cidade e estado.

Tabela 5.2: Funções de transformação em *XPath* para a ontologia *resumé*

| **Conceito da ontologia** | **Mapeamento *XPath*** | |
|---|---|---|
| | **Arquivo 1 (a)** | **Arquivo 1 (b)** |
| Person | | |
| Person/resume | //resume | //resume |
| Person/Name | /resume/name | /resume/fullName |
| Person/Name/Firstname | /resume/name/firstname | substring-before(/resume/fullName, “ ”) |
| Person/Name/MiddleName | - | - |
| Person/Name/LastName | /resume/name/surname | substring-after /resume/fullName, “ ”) |
| Person/Name/Suffix | - | - |
| Person/Address | concat(/resume/address/street,../city,../state) | concat(/resume/address/adr, ../state) |

Nesta tabela, a função *concat* retorna a concatenação de seus argumentos; *substring-after* retorna a subcadeia de caracteres do primeiro argumento que segue a primeira ocorrência do segundo argumento no primeiro argumento. O poder da expressividade do mapeamento é limitado pela expressividade do *XPath*. No entanto, *XPath* fornece vários construtores interessantes, tais como: caminhos absolutos e relativos; predicados (isto é, filtros); funções para manipulação de *strings*, números e booleanos; funções para conjuntos de nós (por exemplo, primeira ou última posição) etc.

Os mapeamentos são gerados pelo gerenciador de ontologias durante a fase de casamento entre arquivos e ontologias. A tarefa de casamento estabelece correspondências entre a ontologia e os documentos XML, determinando os conceitos equivalentes.

## 5.7 Consulta a Réplicas e a Versões

O módulo de processamento de consultas é responsável por verificar quais os arquivos necessários para processar uma determinada consulta, processá-la e retornar os resultados ao usuário final. Antes de processar uma consulta no *host*, o sistema deve conhecer quais versões de quais documentos o *host* possui. Se a consulta solicita pelo estado corrente de um elemento ou atributo, somente os arquivos que possuem a última versão dos documentos devem ser acessados no processamento da requisição. A mesma situação aplica-se a outros tipos de consultas, como, por exemplo, as que solicitam a primeira versão de certa informação. Deste modo, o sistema não é sobrecarregado com requisições que apenas determinados arquivos estão aptos a responder. Para avaliar quais arquivos devem ser consultados para responder uma dada consulta, o trabalho propõe fazer uso de metadados, como descrito na Seção 5.6.

Não é o foco do trabalho definir extensões a uma linguagem de consulta para permitir consultas temporais. Alguns trabalhos na literatura já tratam desta questão, como os apresentados em (WANG et al., 2005b) e (GAO et al., 2004). O objetivo deste trabalho, no entanto, é especificar um ambiente que possam ser utilizados para a submissão de consultas. Os metadados definidos no nível administrativo, no nível do *host* e no nível do gerenciador de ontologias possibilitam que consultas temporais possam ser respondidas. Como exemplo, considere as consultas descritas a seguir: 1) obter a primeira versão do endereço dos autores; 2) obter a última versão do endereço dos autores. A submissão de uma consulta segue o fluxo descrito a seguir. O usuário submete uma consulta a um *host* específico. Esta consulta é relacionada a um domínio de aplicação (isto é, formulada com base nos conceitos e nas relações de uma ontologia), tal como *obter a primeira versão do salário dos empregados.* Duas opções devem ser analisadas:

- se o *host* puder respondê-la, a consulta é processada e os resultados são retornados ao usuário. Por exemplo, este é o caso quando a consulta está solicitando a primeira versão do salário dos empregados e o *host* possui esta informação. Para verificar se o *host* tem condições de responder a consulta, o sistema precisa verificar os metadados do *host*, conforme detalhados na seção 5.6.1. Consultando estes metadados, têm-se condições de descobrir quais são os arquivos aptos a responder a consulta. Além disso, uma vez que se conheçam os arquivos necessários para uma dada consulta, consultam-se os metadados do gerenciador de ontologias, a fim de se obter o mapeamento correspondente para acessar a informação desejada no arquivo correspondente. Finalmente, a consulta é processada e os resultados são retornados ao usuário;
- caso o usuário solicite uma informação que não exista nos arquivos do *host*, uma mensagem de “informação não localizada” é retornada ao usuário.

Com os dois exemplos mostrados nesta seção, verifica-se que a infra-estrutura disponibilizada pelos metadados fornece um ambiente adequado para submissão de consultas temporais. Outros exemplos de consultas poderiam incluir integração de resultados de dois arquivos diferentes. No entanto, distribuição de consultas e integração de resultados não são detalhadas neste trabalho.

O processador de consultas permite ao usuário formular consultas aos arquivos. Assume-se que uma determinada consulta se refere a um único domínio de aplicação, ou seja, envolve conceitos e relacionamentos descritos em uma única ontologia.

O processador de consultas solicita a ontologia representativa de um domínio de aplicação ao gerenciador de ontologias. O usuário, então, navega na ontologia disponível e edita a consulta desejada. Antes da consulta ser processada, o sistema deve conhecer quais arquivos estão armazenados no *host*, de forma a descobrir quais arquivos são relevantes para a consulta formulada. Por exemplo, se o usuário solicita o valor corrente de um elemento ou atributo, somente os arquivos que possuem a última versão deste recurso devem receber a requisição. A mesma situação aplica-se para outros tipos de consultas, como, por exemplo, as que solicitam apenas a primeira versão de certa informação. Desta forma, são envolvidos no processamento da consulta apenas os arquivos necessários para a requisição.

Para analisar quais arquivos devem ser acessados para responder a certa requisição, o mecanismo proposto faz uso de metadados, apresentados na seção 5.6. Tais metadados são fundamentais para otimizar o processo de busca de arquivos.

### 5.7.1 Consulta aos Metadados

A infra-estrutura disponível no *framework* é suficiente para permitir análise de metadados e roteamento de consultas. Os metadados definidos na Seção 5.6 são usados para este propósito. Por exemplo, considerando a consulta 'obter a primeira versão do endereço de uma pessoa', a submissão funciona da seguinte maneira: o usuário submete a consulta ao processador de consultas. Esta consulta pertence a um determinado domínio, como, por exemplo, o domínio de *curriculum*. Duas situações devem ser consideradas:

1. o domínio da consulta e de pelo menos um dos arquivos do *host* é o mesmo, representado por *D*. Neste caso, a consulta é processada, acessando-se os arquivos necessários à consulta, e os resultados são retornados ao usuário;
2. o domínio da consulta não corresponde ao domínio de nenhum arquivo armazenado no *host*. Neste caso, a consulta não pode ser processada e uma mensagem de aviso deve ser retornada ao usuário.

No exemplo a seguir, considere o primeiro caso (domínio da consulta é o mesmo domínio de pelo menos um dos arquivos do *host*). Com base nos metadados do *host*, mostrados na Figura 5.24, é possível acessar a versão $v_i$ de um elemento $e_j$, o histórico completo de um elemento $e_j$, a versão $v_i$ de um documento D, o histórico de um elemento $e_j$ ou um documento D entre um período de tempo $t_x$ e $t_y$, e assim por diante.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.    <Metadata">
3.
4.    <document docID=”D1” fileID="F7" HDoc=’’YES”>
5.    <version   versionID="1" registeringTime="10/10/2005" modificationTime="08/08/2004" duplicate="no"
6.                    hashResult="d49622ddab3733549e54749755fd52b5">
7.      <element name="author" TS="08/08/2004" TE="10/15/2004"/>
8.      <element name="address" TS="08/08/2004" TE="10/15/2004"/></version>
9.    <version versionID="2" …</version>
10.   </document>
11.
12.   <document docID=”D1” fileID="F8" HDoc=’’YES”>
13.   <version   versionID="2" registeringTime="10/16/2004" modificationTime="08/08/2005" duplicate="no"
14.                   hashResult="749752ddab3733549e54749755f78g65">
15.     <element name="author" TS="08/08/2005" TE="now"/>
16.     <element name="address" TS="08/08/2005" TE="now"/></version>
17.   <version versionID="2" …</version>
18.   </document>
19.   …
20.  </Metadata>
```

Figura 5.24: Metadados do *host*

Algumas exemplos de recuperação de versões específicas são descritos a seguir:

- recuperar a versão $v_i$ de um elemento $e_j$ - por exemplo, deseja-se recuperar a primeira versão (`versionID="1",` linha 5) do elemento *author* (`element name="author",`

linha 7). Pelo número da versão representado nos metadados, verifica-se que a primeira versão do elemento solicitado encontra-se no arquivo *F7* (`fileID="F7"`, linha 5). Logo, basta acessar este arquivo e retornar a informação solicitada;

- para recuperar o histórico completo do elemento endereço, deve-se consultar os metadados, buscando por todas as versões (`versionID`) do elemento *address* (`element name="address"`). A última versão deste elemento é representada pelo TE=*now* (linha 16);
- recuperar a versão $v_i$ de um documento D - para recuperar uma determinada versão de um documento, deve-se procurar pelo número da versão desejada (por exemplo, `versionID="1"`) do documento desejado (por exemplo, `docId="D1"`). Pelos metadados, verifica-se em qual arquivo esta versão está disponível e acessa-se este arquivo em específico. Para saber qual arquivo deve ser acessado, pode-se verificar no *host* qual arquivo possui o mesmo valor para o atributo `modificationTime`. Se mais de um arquivo possuir o mesmo *modificationTime*, então calcula-se e compara-se o *hash* destes arquivos para decidir qual deles deve ser consultado e retornado;
- recuperar o histórico de um elemento $e_j$ ou um documento D entre um período de tempo $t_x$ e $t_y$ - por exemplo, para recuperar o histórico do documento *D1* entre 10/10/2005 e 01/01/2006, deve-se procurar em todas as versões (`versionID`) deste documento (`docId="D1"`) e acessar os arquivos necessários no *host*. Para melhor facilitar a visualização, pode-se retornar também os rótulos temporais de acordo com a validade dos elementos ao longo do tempo (`TS` e `TE`). Esta consulta também pode ser processada no arquivo *H-Doc*, uma vez que o documento *D1* possui uma representação com todo o histórico (`HDoc="YES"`, linha 4). O processamento de consultas no arquivo *H-Doc* é descrito na seção 5.7.2.1.

### 5.7.2 Processamento de Consultas

Uma vez consultados os metadados, o sistema já tem conhecimento de quais arquivos precisa acessar. A partir deste momento, inicia-se a etapa de processamento de consultas, propriamente dita.

O processador de consultas responde consultas formuladas sobre uma ontologia, traduzindo a consulta e acessando os arquivos necessários. Os seguintes passos são executados para se obter os dados correspondentes a uma dada consulta:

- construção da consulta - o usuário formula a consulta usando os termos da ontologia (*Qo*);
- tradução da consulta - a consulta formulada sobre a ontologia é reescrita (*Qm*) em termos dos arquivos XML;
- recuperação dos dados - a expressão de mapeamento é enviada ao *host*, os dados são acessados, os resultados individuais (*Ru*) são combinados e a resposta final (*Ro*) é retornada ao processador de consultas.

Este processo, descrito na Figura 5.25, é detalhado a seguir.

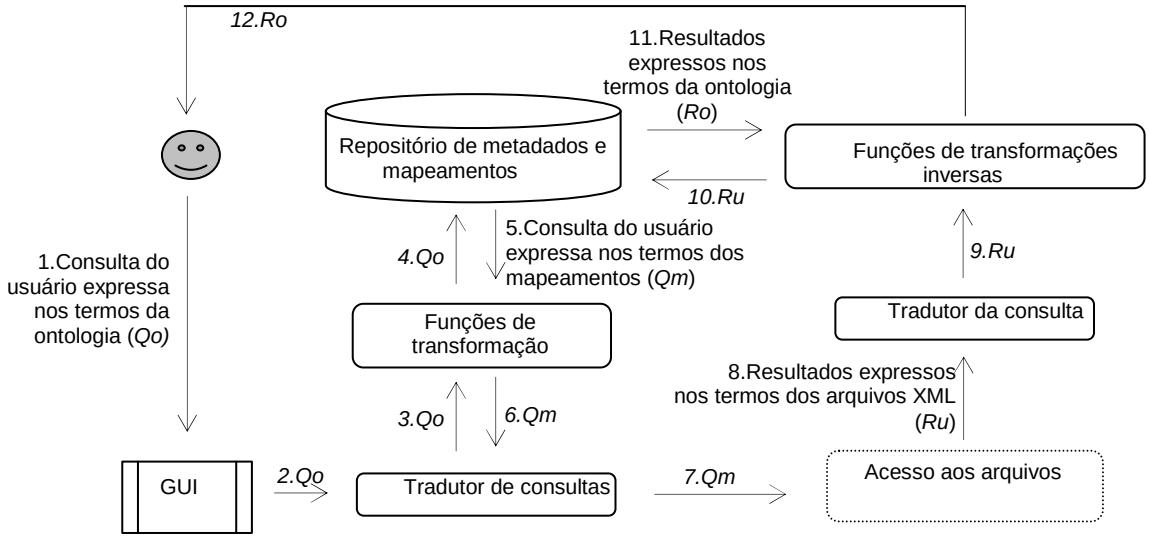



Figura 5.25: Fluxo do processamento de consultas

**Construção da consulta.** O usuário constrói a consulta através da interface. Para esta tarefa, o sistema mostra as ontologias disponíveis, a fim de restringir o domínio da consulta. Após a escolha da ontologia, o usuário pode construir a consulta. Nenhuma intervenção extra do usuário é necessária. Como exemplo, a consulta *Qo* "obter o nome das pessoas que vivem em *Garbonzoville"* é representada em *XPath* como:

```
/Person/Name/Firstname [contains(../../address," Garbonzoville"]
```

onde: a função *contains* retorna verdadeiro se o primeiro argumento contém o segundo argumento, e falso, no caso contrário.

Como já mencionado, assume-se que os usuários formulem consultas sobre uma única ontologia. Após obter a consulta do usuário, o processador invoca o tradutor de consultas. O tradutor utiliza as informações de mapeamento pré-definidas que relacionam termos na ontologia aos termos nos arquivos XML. Os dados são posteriormente recuperados com a ajuda das funções de transformação e do repositório dos metadados.

**Tradução da consulta.** O sistema recebe a consulta construída nos termos da ontologia, transforma-a em várias expressões de mapeamento, e executa as consultas nos arquivos XML. Na submissão da consulta, o usuário deve preocupar-se somente com a semântica dos dados, e não com aspectos relacionados com sintaxe, localização, estrutura etc; estas questões devem ser tratadas pelo sistema. Por exemplo, usando as definições de mapeamento apresentadas na Tabela 5.2, a consulta *Qo* é traduzida nas duas expressões:

*Qm1:*

```
/resume/name/firstname[../../address/city=Garbonzoville"]
```

*Qm2:*

```
substring-before
(/resume/address/adr[contains(.,"Garbonzoville")]/../../fullName," ")
```

**Acesso aos dados.** O sistema recupera os dados correspondentes à consulta e que são válidos pela ontologia escolhida. Para esta tarefa, o sistema usa as informações de mapeamento para traduzir a consulta do usuário em diferentes sub-consultas para os arquivos XML. Analisando os metadados descritos neste trabalho, o sistema sabe quais arquivos XML estão relacionados à ontologia escolhida e como acessar a informação desejada.

**Tradução de resultados.** Dados são recuperados dos arquivos em diferentes estruturas e formatos. Desta forma, diferentes sub-árvores devem ser combinadas (correlacionadas) e apresentadas ao usuário. A etapa de correlação tem dois objetivos (MENA et al., 2001): solucionar a heterogeneidade estrutural e juntar respostas vindas de diferentes arquivos. As respostas vindas dos arquivos estão na estrutura do arquivo XML consultado. Desta forma, cada resposta é mudada para estrutura da ontologia correspondente suportada por este arquivo. Após o processo de aplicação de funções de transformação inversas, todas as respostas estão expressas no formato da ontologia que descreve tais arquivos.

Por exemplo, a consulta *Qm* produz os seguintes resultados:

*Ru1:* `<firstname>Hu</firstname>` *Ru2:* `Jo`

Analisando as expressões de mapeamento definidas na seção 5.6.3, a seguinte transformação é realizada:

*Ru1(*`<firstname>Hu</firstname>`*)→Ro1:* `Firstname: Hu`

Uma vez que o segundo resultado é somente uma subcadeia de caracteres (*Ru2:* Jo), a resposta retornada para este arquivo é *Ro2:* `Jo.`

Além disso, a fim de correlacionar subrespostas, duas questões devem ser tratadas: (i) identificação de entidades – como identificar representações do mesmo objeto do mundo real em diferentes arquivos? e (ii), conflitos nos valores de atributos – como lidar com diferenças nos valores de dados entre atributos que representam o mesmo objeto do mundo real? O trabalho apresentado em (SACCOL et al., 2002) discute a abordagem proposta para solucionar estas questões.

Finalmente, os resultados são retornados ao usuário.

#### *5.7.2.1 Consulta ao Histórico de Documentos (H-Doc)*

Com o agrupamento de versões em um único arquivo físico (*H-Doc*), determinadas consultas podem ser diretamente respondidas, sem acessar os vários arquivos espalhados pelo *host*. Caso o *host* não tenha condições de responder a consulta, uma mensagem de aviso é enviada ao usuário. Nesta seção será considerado apenas o processo de consulta em arquivos *H-Doc*.

Considerando um elemento qualquer *e* de um documento *D*, os filtros temporais aplicados podem ser baseados em uma data *x* ou em um intervalo de tempo *x* e *y*. Algumas cláusulas utilizadas para a execução de consultas temporais são mostradas a seguir:

- *select_Before (e, x)* - esta cláusula retorna o conteúdo dos elementos *e* que são válidos no documento *H-Doc* antes da data *x*, ou seja, procura-se por elementos cujo tempo inicial de validade (*TS*) seja menor à data *x* especificada;
- *select_After (e, x)* - esta cláusula retorna o conteúdo dos elementos *e* que são válidos no documento *H-Doc* após a data *x*, ou seja, procura-se por elementos cujo tempo final de validade (*TE*) seja maior à data *x* especificada;
- *select_Between (e, x, y)* - esta cláusula retorna o conteúdo dos elementos *e* que são válidos no documento *H-Doc* entre o período *x* e *y*, onde $x<y$, ou seja, procura-se por elementos cujo tempo inicial de validade (*TS*) seja menor ou igual a *y* e cujo tempo final de validade (*TE*) seja maior ou igual a *x*;
- *select_Now (e)* - esta cláusula retorna o conteúdo dos elementos *e* que são válidos no documento *H-Doc* no momento atual, ou seja, procura-se por elementos cujo tempo final de validade (*TE*) seja igual a *now*;
- *select_Before (D, x)* - esta cláusula retorna o conteúdo do documento *D* que é válido antes da data *x*, ou seja, procura-se por documentos cujo tempo inicial de validade (*TS*) do elemento raiz seja menor à data *x* especificada;
- *select_After (D, x)* - esta cláusula retorna o conteúdo do documento *D* que é válido após a data *x*, ou seja, procura-se por documentos cujo tempo final de validade (*TE*) do elemento raiz seja maior à data *x* especificada;
- *select_Between (D, x, y)* - esta cláusula retorna o conteúdo do documento *D* que é válido entre o período *x* e *y*, ou seja, procura-se por documentos cujo tempo inicial de validade (*TS*) do elemento raiz seja menor ou igual a *y* e cujo tempo final de validade (*TE*) do elemento raiz seja maior ou igual a *x*;
- *select_Now (E)* - esta cláusula retorna o conteúdo do documento *D* que é válido no no momento atual, ou seja, procura-se por documentos cujo tempo final de validade (*TE*) do elemento raiz seja igual a *now*.

Alguns exemplos de consultas temporais são mostrados na seção a seguir.

### 5.7.3 Implementação

A segunda funcionalidade implementada (GIACOMEL, 2006) na ferramenta *XVersion* (a primeira foi detalhada na seção 4.4) é a possibilidade de se executar consultas sobre o arquivo *H-Doc*. Desta forma, a consulta é expressa sobre um único arquivo físico. Através de filtros temporais, informações históricas podem ser retornadas sem a necessidade de processar a consulta em todos os arquivos espalhados pelo *host.* Na versão corrente da ferramenta, utiliza-se *XQuery* como linguagem de consulta

Algumas consultas executadas na ferramenta são mostradas na Figura 5.26 e Figura 5.27. A primeira consulta solicita o salário atual dos empregados. Para responder a esta consulta, a ferramenta busca todos os elementos cujo tempo de validade final seja *now*. O resultado da consulta é mostrado na parte inferior da Figura 5.26.

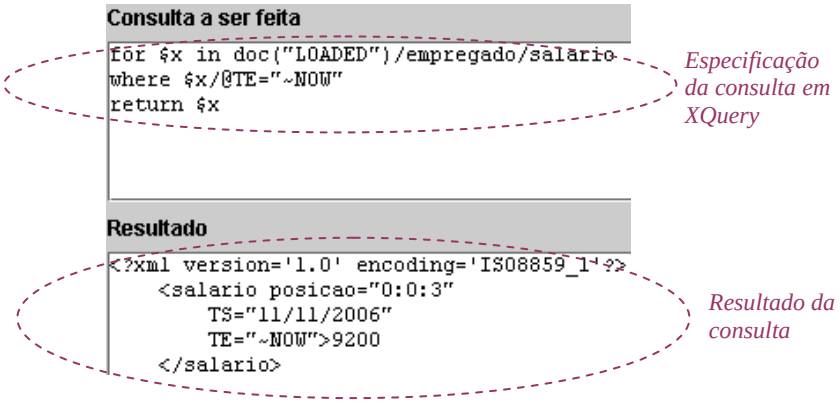



Figura 5.26: Consulta 1 sobre o arquivo *H-Doc*

A segunda consulta retorna os valores de salário válidos até a data *10/11/2006*. Para responder a esta consulta, a ferramenta busca todos os elementos *salário* cujo tempo inicial de validade seja inferior à data especificada. O resultado da consulta é mostrado na parte inferior-esquerda da Figura 5.27.

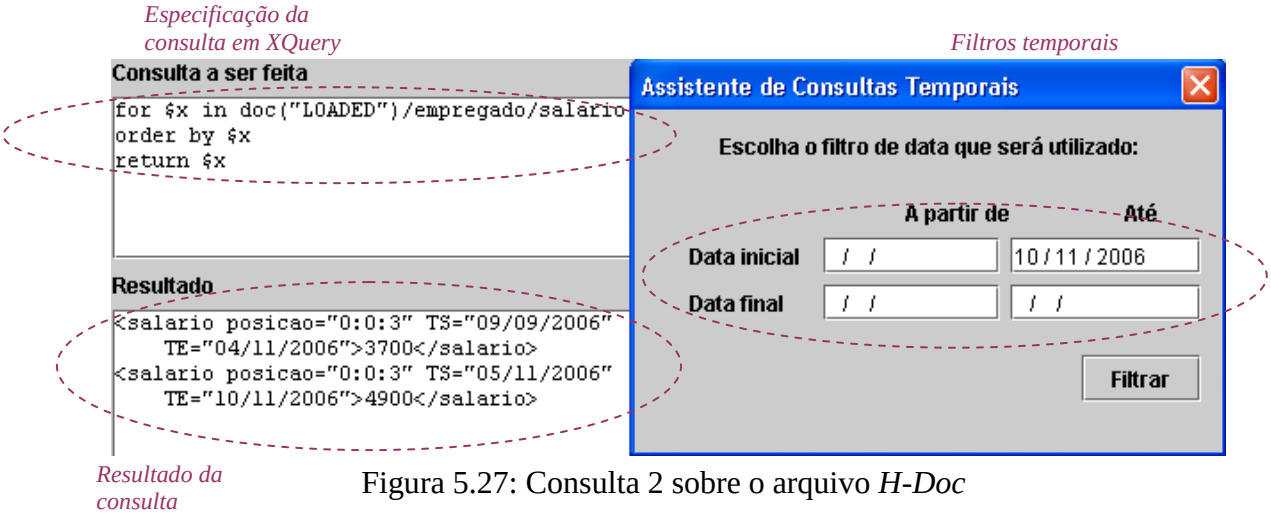



Figura 5.27: Consulta 2 sobre o arquivo *H-Doc*

Para a execução das consultas foi utilizada a biblioteca *Qizx/Open*[20], distribuída sob a licença GNU.

## 5.8 Considerações Finais

Dois problemas foram levantados no início deste trabalho: possibilidade de existência de duplicatas e de múltiplas versões de um mesmo recurso disponível em um *host*. Estes dois problemas levam à necessidade de tratar as seguintes questões:

[20] Biblioteca *Qizx/Open*. Disponível em *http://www.xfra.net/qizxopen/*

- detectar réplicas e versões de documentos XML espalhadas pelo *host*;
- representar réplicas e versões detectadas, de forma a permitir eficiente recuperação.

Para solucionar estas duas questões, este capítulo especificou uma arquitetura para um ambiente de detecção, gerenciamento e consulta a réplicas e versões. Algumas considerações finais são feitas a seguir sobre o *framework* proposto.

**Sobre o detector de réplicas e versões.** Convém destacar que os trabalhos relacionados a versões encontrados na literatura focam na representação e acesso a versões previamente conhecidas. Não há a detecção destas versões, nem em ambientes centralizados nem em ambientes distribuídos. Existem algumas pesquisas em detecção de diferenças em documentos XML que podem ser utilizadas como uma importante base para a detecção de réplicas e de versões. Entretanto, estas abordagens geralmente produzem um conjunto de diferenças pobre em semântica, que não traz muita informação em relação ao grau de similaridade entre os documentos analisados. Além disso, as propostas para detecção de réplicas focam no problema de identificar múltiplas representações do mesmo objeto do mundo real. No entanto, estas propostas não se aplicam ao problema levantado, uma vez que, neste contexto, o termo réplica é utilizado para definir uma cópia idêntica de um documento XML.

**Sobre o gerenciador de ontologias.** Motores de busca existentes fornecem suporte para recuperação automática de informações. No entanto, a tarefa de extrair a informação relevante fica para o usuário. Desta forma, há alguns desafios que devem ser superados, tais como a falta de mecanismos para representar e traduzir informações heterogêneas, e descrições de conteúdo. Conseqüentemente, há uma clara necessidade por mais semântica e interoperabilidade (FENSEL, 2001).

Em se tratando de recursos compartilhados, há um gargalo extra: aumentar a qualidade dos resultados enquanto se otimiza o espaço de busca. Neste cenário, duas questões devem ser tratadas: como encontrar arquivos relevantes para a consulta do usuário (com baixo custo) e como aumentar a semântica dos recursos disponíveis. Este trabalho propõe um mecanismo baseado em ontologias para solucionar estes problemas, que pode ser utilizada para aperfeiçoar as técnicas tradicionais de recuperação de informações. O mecanismo proposto reduz o tráfego e aumenta a semântica com relação ao armazenamento e busca das informações. A otimização do espaço de busca é feita pelo agrupamento de arquivos em domínios, descritos por ontologias. O aumento da semântica é feito pela adoção de ontologias na descrição de conceitos e relacionamentos. Neste contexto, este capítulo apresentou o gerenciador de ontologias, detalhando as etapas de geração de ontologias e casamento de ontologias e arquivos XML.

**Sobre o processador de consultas.** O processador de consultas permite ao usuário formular consultas aos arquivos armazenados no *host*. Assume-se que uma determinada consulta refere-se a um único domínio de aplicação, ou seja, envolve conceitos e relacionamentos descritos em uma única ontologia. No entanto, antes da consulta ser processada, o sistema deve conhecer quais arquivos estão armazenados neste *host*, de forma a descobrir quais arquivos são relevantes para a consulta formulada. Para avaliar quais arquivos devem ser consultados para responder a uma determinada requisição, o mecanismo proposto neste

trabalho faz uso de metadados, apresentados na seção 5.6. Tais metadados são fundamentais para otimizar o processo de busca de arquivos, descrito neste capítulo. Uma vez consultados os metadados, o sistema já tem conhecimento de quais arquivos precisa acessar. A partir deste momento, inicia-se a etapa de processamento de consultas, propriamente dita.

A seção sobre o processamento de consultas descreveu uma possível arquitetura para o ambiente proposto. Duas abordagens foram apresentadas: o acesso aos diversos arquivos físicos do *host* e o acesso às versões que armazenam o histórico de recursos, representadas nos arquivos *H-Doc*. Desta forma, consultas podem ser expressas através da restrição de valores nos rótulos temporais, eliminando a necessidade de processar consultas sobre vários arquivos distribuídos no *host*. Com a abordagem proposta, espera-se melhorar o desempenho das consultas em documentos freqüentemente acessados.

**Sobre os metadados.** Sistemas baseados em palavras-chave não recuperam o arquivo necessário quando um sinônimo é usado como parte da consulta. Esta situação acontece porque diferentes termos podem ser utilizados para descrever conceitos similares. Além disso, sistemas automáticos têm dificuldades em encontrar, extrair e integrar informação espalhada sobre a rede. Desta forma, uma questão importante deve ser respondida no contexto deste trabalho: como lidar com incompatibilidades estruturais e de conteúdo?

Suponha duas aplicações que necessitam trocar dados. Uma abordagem possível é construir um adaptador que transforma dados e estruturas entre elas. No entanto, a construção do adaptador não é uma tarefa simples, e requer conhecer a organização dos dados em ambas as aplicações. Além disso, a complexidade e o tempo de desenvolvimento tendem a ser quadráticos em relação ao número de aplicações componentes (STAAB et al., 2004). Outra possível solução baseia-se no uso de metadados para descrever a semântica dos arquivos. Mas este cenário traz duas questões críticas: como lidar com diferentes conceitos usados para descrever a mesma informação, e como adquirir e manter os metadados necessários para solucionar o problema de compartilhamento de vocabulário. Para solucionar estes problemas de heterogeneidade semântica, foram utilizados os conceitos de ontologias, de metadados e de mapeamentos para a melhoria do processo de interoperabilidade de informação.

Para tratar esta questão, este trabalho propõe o uso de anotações (representadas por metadados e funções de mapeamento) que permitem definições estruturais e semânticas dos documentos. O uso de tais anotações possibilita um processamento de consultas mais inteligente que torna possível ao usuário submeter consultas sem o conhecimento da localização e da estrutura dos arquivos XML.

Este capítulo resultou nas seguintes publicações: (SACCOL et al., 2008; SACCOL et al., 2008b; SACCOL et al., 2008c). A implementação da ferramenta *Ontogen* foi realizada durante a co-orientação de um Trabalho de Conclusão de Curso (MELLO, 2007). A implementacao da ferramenta *The Matcher* também resultou da co-orientação de uma monografia de especialização (NOLL, 2007). Estas produções estão detalhadas no capítulo 7.

# 6 ESTUDO DE CASO: APLICAÇÃO DO FRAMEWORK EM SISTEMAS PEER-TO-PEER

Este capítulo apresenta a aplicação do *framework* proposto, considerando um cenário P2P. São enfatizadas as diferenças e as principais adaptações que devem ser realizadas no *framework DetVX*, visando a detecção, o gerenciamento e a consulta a documentos XML replicados e versionados em ambientes P2P. O *framework* estendido é tratado neste capítulo como *DetVX_P2P*. Para facilitar o entendimento das diferenças e adaptações, este capítulo mantém uma estrutura similar à do capítulo 5.

## 6.1 Escolha da Aplicação

O termo P2P refere-se a sistemas de aplicações que usam recursos distribuídos para realizar tarefas em um contexto descentralizado (TATARINOV et al., 2003; ABERER et al., 2002; BERNSTEIN et al., 2002). A usabilidade destes sistemas é dependente principalmente das técnicas utilizadas para encontrar e recuperar os recursos desejados. A qualidade de tais resultados pode ser medida por algumas métricas, tais como o tamanho do conjunto de resultados, a satisfação do usuário com os resultados retornados e o tempo gasto no processamento (YANG et al., 2002).

Quatro problemas podem ser observados neste ambiente. O primeiro problema refere-se à possibilidade de existir múltiplas representações de um mesmo recurso na rede. A existência de recursos duplicados pode ser interessante por questões de desempenho, possibilitando que o usuário submeta uma consulta e que os resultados sejam retornados de um *peer* específico, atendendo critérios de tempo de resposta e de disponibilidade. No entanto, para que se tire vantagem da replicação de recursos, é necessário que se tenha um processo de identificação destas múltiplas representações do mesmo objeto compartilhado. Caso contrário, respostas redundantes podem ser retornadas para o usuário.

O segundo problema advém do comportamento evolutivo e dinâmico de alguns recursos disponíveis na rede P2P. Este problema é ainda mais evidente em se tratando de documentos XML, com freqüentes mudanças de estrutura e de conteúdo. A característica de temporalidade em documentos XML compartilhados em ambientes P2P pode ser tratada com o uso de versões. Através do uso de versões pode-se preservar o conteúdo antigo e o conteúdo atual do documento, possibilitando representação da evolução temporal e acesso aos estados passados do documento. A existência de múltiplas versões de um mesmo recurso

coloca, então, outra questão a tratar: como identificar que duas representações referem-se a versões modificadas de um mesmo documento.

O terceiro problema é como localizar arquivos relevantes para uma consulta, com baixo custo. Arquivos pertencentes a um mesmo domínio de aplicação e que são necessários para responder uma determinada consulta podem estar espalhados em vários *peers*. O uso da técnica *flooding* (inundação) é necessário para acessar todos os arquivos existentes. Esta técnica é cara, uma vez que todos os *peers* recebem a mensagem de consulta e geralmente apenas alguns deles estão aptos a respondê-la. Outra abordagem para balancear custo é usar alguma técnica variante da *flooding*, tais como busca em largura e em profundidade na árvore formada pelos *peers*. No entanto, o uso destas variantes não garante o retorno ótimo dos resultados, uma vez que nem todos os *peers* recebem a mensagem da consulta.

O quarto problema advém da falta de semântica dos recursos. Considerando duas aplicações que necessitam trocar dados, uma abordagem possível é construir um adaptador que transforme dados e estrutura entre elas. Entretanto, a construção destes adaptadores é difícil e requer conhecer a organização dos dados em ambas as aplicações. Além disso, a complexidade e o tempo de desenvolvimento tendem a ser quadráticos em relação ao número de aplicações componentes (STAAB et al., 2004). Outra solução pode utilizar metadados para descrever a semântica dos recursos. Mas este cenário traz duas questões (MENA et al., 2001): (i) como lidar com diferentes conceitos usados para descrever a mesma informação; e (ii), como adquirir e manter os metadados para tratar o problema de compartilhamento de vocabulários.

Visando tratar as deficiências acima relatadas, este capítulo estende o *framework* proposto no capítulo 5 para um ambiente P2P.

## 6.2 Visão Geral

No *framework DetVX_P2P*, cada *peer* participante atua tanto como cliente quanto como servidor, fornecendo acesso a arquivos XML compartilhados. Estes arquivos estão armazenados em *peers*, segundo uma arquitetura *super peer* (SCHOLLMEIER, 2001). A arquitetura *super peer* arega um subconjunto de *peers* em *super peers*. Neste tipo de arquitetura, a comunicação é estabelecida somente entre os *super peers*, e destes com os seus *peers* agregados, o que pode contribuir para tornar a pesquisa mais rápida.

Similarmente ao apresentado na seção 5.4, os arquivos pertencem a um domínio de aplicação, descrito por uma ontologia. As ontologias são usadas como critério de agrupamento de *peers* em *super peers*. Todos os *peers* que possuem documentos pertencentes a certo domínio de aplicação são agrupados no *super peer* descrito pela respectiva ontologia. Assume-se que cada arquivo é descrito por uma ontologia. Como um *super peer* agrupa *peers* que armazenam arquivos de diversos domínios de aplicação, um *super peer* pode possuir mais de uma ontologia associada a ele, conforme mostrado na Figura 6.1. Sabendo-se que um domínio é representado em somente um *super peer*, dois documentos *d1* e *d2* pertencentes ao mesmo domínio não serão encontrados em *super peers* diferentes. Com base nesta afirmação, o ambiente garante que uma determinada consulta relacionada a certo domínio será roteada somente dentro da sub-rede de um *super peer*.

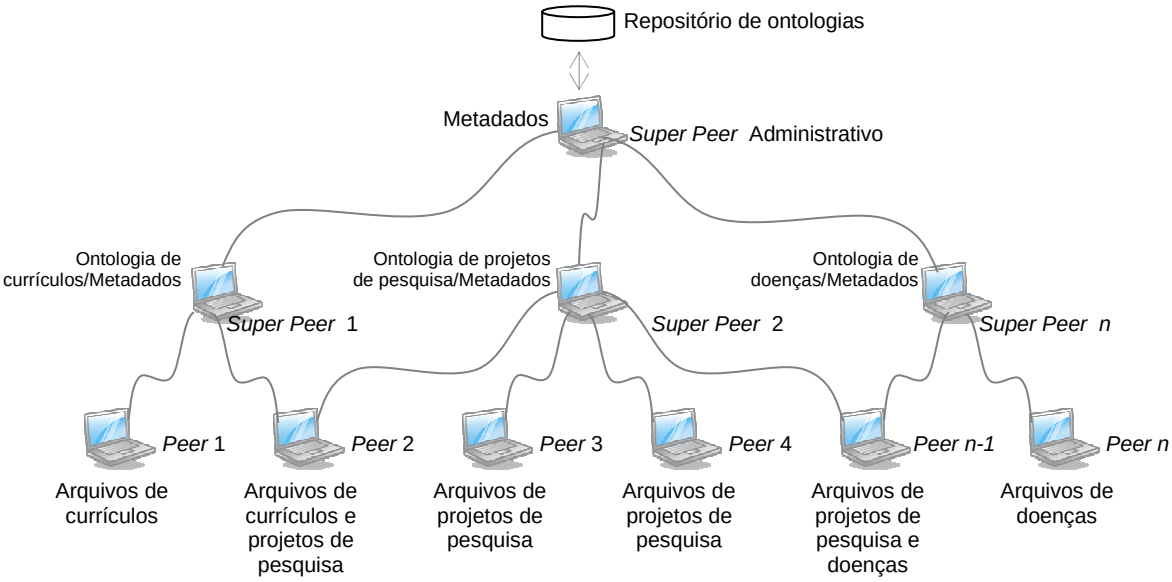



Figura 6.1: Representação de *peers* no *framework DetVX_P2P*

Com a extensão do *framework* para um ambiente distribuído, os metadados disponíveis também precisam ser adaptados. Foram descritos dois níveis de metadados na abordagem proposta no capítulo 5: metadados do *host* e metadados do gerenciador de ontologias. No *framework* estendido, as consultas são submetidas ao *super peer*, com base nos conceitos e relações existentes em um domínio de aplicação. Desta forma, cada *super peer* também possui metadados com informações sobre seus *peers* agregados e seus respectivos arquivos registrados. Os metadados especificam as versões e réplicas disponíveis em um determinado *super peer*, e os rótulos temporais correspondentes a cada elemento encontrado em um dado arquivo de um *peer*. Os metadados são atualizados sempre que um novo documento é registrado em um *peer* e são acessados durante o processo de consulta. Metadados dos *super peers* são descritos na seção 6.3.5.2.

Como existem diversos *super peers*, o *framework* prevê a existência de um *super peer* administrativo, o qual funciona como responsável pelo gerenciamento da rede. Este *peer* conhece todos os *super peers* e tem ligação com o repositório de ontologias. Os metadados do *super peer* administrativo descrevem os *super peers* existentes, seus *peers* agregados e informações sobre os arquivos de cada um destes *peers*, como identificadores locais de documento, resultados *hash*, tempos de registro e última data de atualização de cada arquivo no *peer*. Metadados do *super peer* administrativo são descritos na seção 6.3.5.1.

Neste ambiente, o usuário submete uma consulta a um *peer* específico. Esta consulta é relacionada a um domínio de aplicação, tal como *obter todas as versões do artigo 'Peer Databases'*. Se a consulta não puder ser respondida localmente, ela é roteada para o *super peer* do *peer* solicitante. O *super peer*, então, verifica seus *peers* disponíveis que estão aptos a responder à consulta e reenvia a consulta a estes *peers*. Os *peers* processam a consulta e retornam os resultados ao *super peer*. Finalmente, os resultados são retornados ao usuário. Convém ressaltar que o usuário pode submeter consultas relacionadas a qualquer domínio em qualquer *peer*. Se nem o *peer* onde a consulta foi submetida nem o seu *super peer*

puderem responder a consulta (por não possuírem os dados relativos ao domínio da consulta), então a consulta será roteada a outro *super peer* apropriado.

A informação pode estar replicada em vários *peers*. Documentos multiversionados podem estar localizados em *peers* diferentes. Se o usuário submete uma consulta a um *peer* específico solicitando o endereço atual de uma pessoa, o sistema deve retornar somente a última versão desta informação. Supondo que os dados existentes no *peer* onde a consulta foi submetida estejam desatualizados, então a consulta deve ser roteada para outros *peers*. Para descobrir qual (is) documento (s) é (são) necessário(s) acessar para responder uma determinada solicitação, o sistema consulta os metadados disponíveis no *super peer*.

A Figura 6.2 enfatiza as diferenças do *DetVX_P2P* em relação à visão geral do *DETVX* apresentada na Figura 5.1.

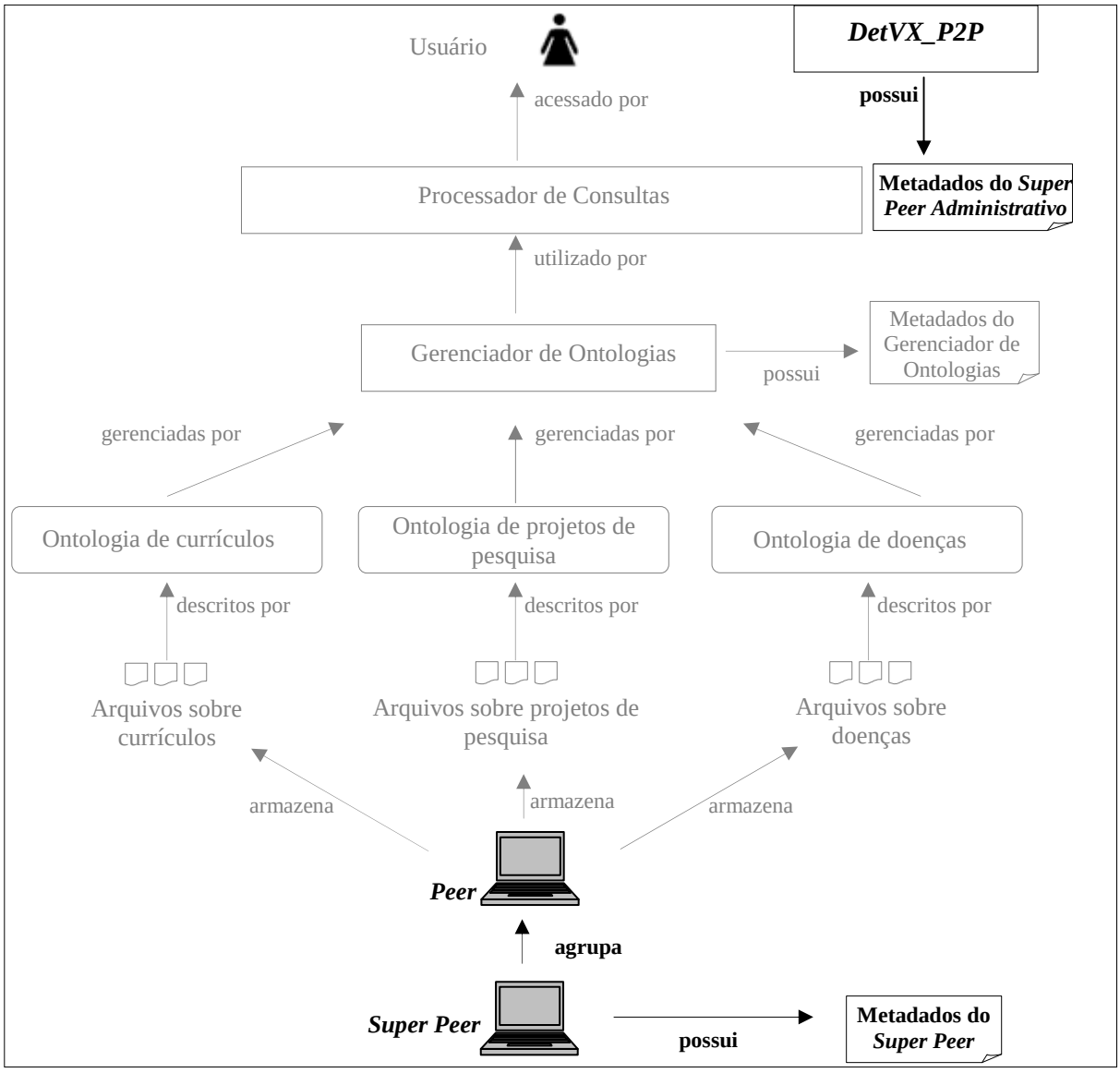



Figura 6.2: Visão geral do *framework DetVX_P2P*

Versões e réplicas de documentos podem ser detectadas à medida que os *peers* se conectam ao sistema. Modificações posteriores à conexão de um *peer* também são observadas, a fim de verificar o surgimento de uma nova versão ou réplica, de forma transparente ao usuário. O sistema permite que a partir dos metadados definidos seja possível submeter requisições ao sistema, como *obter a primeira versão de um elemento ou documento, obter a última versão de um elemento ou documento, obter o histórico do documento*, etc. Para isso, nenhuma modificação na edição dos documentos é exigida.

Eleição de *super peers* não é tratada neste trabalho. No entanto, convém ressaltar que para ser um *super peer*, um *peer* deve possuir algumas características que permitam o bom funcionamento do sistema. Desta forma, a escolha de um *super peer* é feita com base em algumas características, como (ZHU et al., 2003): estabilidade, largura de banda e acesso rápido, capacidade de processamento, capacidade de armazenamento e memória. Um algoritmo para este propósito pode ser encontrado em (TRIANTAFILLOU et al., 2003).

## 6.3 Arquitetura do *Framework DetVX_P2P*

Em relação à arquitetura mostrada na Figura 5.2, o único módulo que sofre alterações é o gerenciador de arquivos, que aqui passa a ser denotado pelo termo gerenciador de *peers*, conforme mostrado na Figura 6.3. A fim de dar suporte às características de um ambiente P2P, os outros módulos sofrem alguns ajustes na definição de suas atividades. De maneira geral, o usuário interage com o sistema via uma interface, a qual possibilita registrar *peers* e documentos no ambiente (através do gerenciador de *peers*) e submeter consultas posteriores aos recursos compartilhados (através do processador de consultas). Quando um *peer* solicita conexão ao sistema, é necessário verificar a qual *super peer* o *peer* solicitante será conectado. Para isso, precisa-se descobrir os domínios de aplicação de seus documentos, tarefa esta realizada pelo gerenciador de ontologias. Depois de conectado, o ambiente verifica se os documentos recém compartilhados referem-se a versões ou réplicas de documentos já previamente compartilhados no sistema. Esta última tarefa é realizada pelo gerenciador de réplicas e de versões. A Figura 6.3 corresponde à extensão da Figura 5.2.

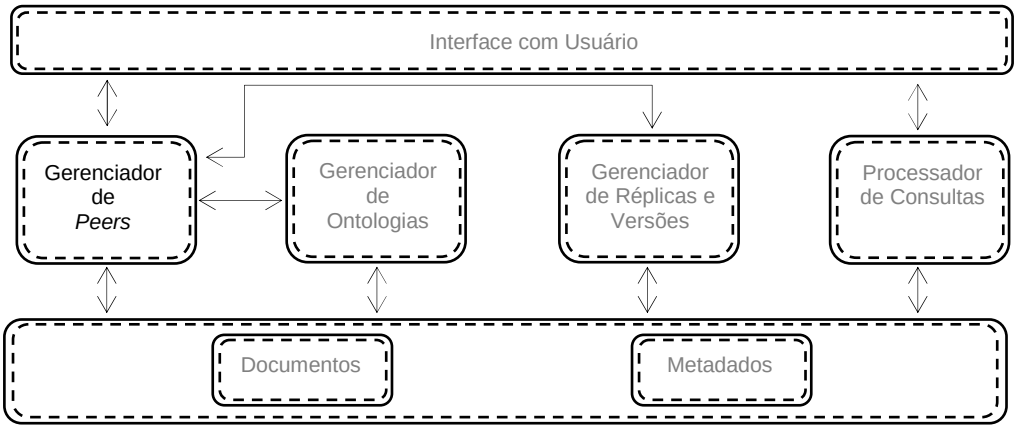



Figura 6.3: Arquitetura do *framework DetVX_P2P*

O módulo de gerenciamento de *peers* é responsável por registrar arquivos no sistema, além de verificar periodicamente se o *peer* teve alguma modificação em seus arquivos compartilhados.

### 6.3.1 Arquivos e Documentos

Um arquivo possui um identificador local e um identificador global. Identificadores locais (denotados por *fileID*) são usados para identificar um arquivo em um *peer*. Os identificadores globais estão disponíveis em um componente de serviço de nomes localizado no *super peer* administrativo. Basicamente, o identificador global descreve qual é o *super peer* e o *peer* em que este arquivo se encontra, qual é o identificador local do documento (dado pelo mapeamento a partir do resultado da função *hash*) e qual é o número da versão deste arquivo. Os identificadores globais são representados conforme mostrado a seguir:

*GlobalID: <super peer identification>.<peer identification>.<fileID>.<version number>*

onde:

- *super peer identification* - consiste de um valor único que identifica o *super peer* dentro da rede. Exemplo: SP1;
- *peer identification* - consiste de um valor único que identifica o *peer* dentro da sub-rede de um *super peer*. Exemplo: P1;
- *fileID* - consiste de um valor que identifica o documento em um *peer*. Nesta abordagem, considera-se *fileID* (identificador local do documento) como um valor mapeado a partir do resultado da função *hash* aplicada ao documento. Exemplo: Doc07;
- *version number* - consiste de um valor que identifica a versão do documento em um *peer*. Exemplo: V1.

Logo, se o arquivo inserido no sistema é uma nova instância, o documento irá receber um novo identificador global. Se o arquivo inserido é uma réplica de algum arquivo já existente, o identificador local destes arquivos será o mesmo (mesmo resultado *hash*); no entanto, seus identificadores globais podem ser diferentes (se estiverem localizados em *peers* diferentes). Se o arquivo inserido é uma versão de algum arquivo já existente, seus identificadores locais serão diferentes (resultados *hash* diferentes) e seus identificadores globais, por consequência, também serão.

O processo de detecção de réplicas e de versões é executado entre os arquivos pertencentes a um mesmo domínio de aplicação, conforme detalhado no Capítulo 3.

### 6.3.2 Gerenciador de *Peers*

Este módulo é responsável por gerenciar as conexões e reconexões de *peers* ao ambiente e por verificar periodicamente se o *peer* teve alguma modificação em seus arquivos compartilhados. Quando um *peer* deseja compartilhar arquivos ou ter acesso a arquivos compartilhados por outros *peers*, devem conectar-se a um *super peer* e registrar os arquivos a serem compartilhados. O processo de conexão é feito em dois passos:

- o *peer* envia uma solicitação a um serviço administrador, o qual informa o *super peer* adequado para a conexão. Se não houver um *super peer* adequado para o domínio de aplicação desejado, uma nova ontologia é gerada e associada a um *super peer* existente;
- o *peer* estabelece a conexão com o *super peer*.

Basicamente, o ambiente deve verificar se esta é a primeira conexão do *peer*. Se este for o caso, então se parte para a escolha do domínio de aplicação, descrito pela ontologia, e posterior conexão. Caso contrário, verifica-se se ocorreu qualquer modificação no *peer* desde sua última conexão. Se não houve modificação, o *peer* é reconectado ao mesmo *super peer* da última conexão. Se houve alguma modificação, então o sistema precisa verificar se a mudança causa uma alteração no domínio dos documentos compartilhados neste *peer*, o que eventualmente pode levar a uma conexão com um *super peer* diferente.

As atividades do gerenciador de *peers* são mostradas na Figura 6.4. Estas atividades correspondem à extensão da Figura 5.3. A seguir, são detalhadas apenas as atividades que sofreram modificações em relação à descrição apresentada na seção 5.3 ou que foram adicionadas na extensão do *framework.*

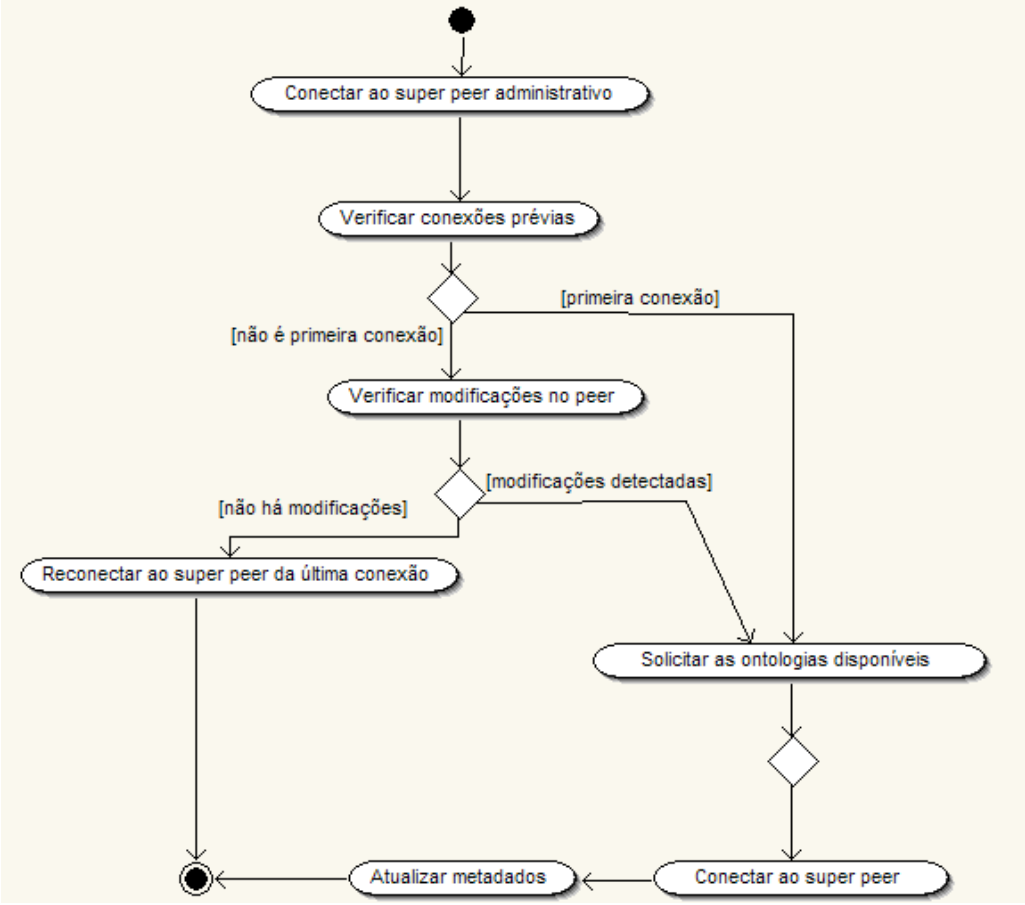



Figura 6.4: Diagrama de atividades do gerenciador de *peers*

**Conectar ao super peer administrativo.** O *peer* conecta-se ao servidor do *super peer* administrativo (equivalente à atividade '*solicitar registro do host*' no gerenciador de arquivos do *DetVX*, mostrada na seção 5.3).

**Verificar conexões prévias.** Esta atividade verifica se este *peer* já esteve conectado anteriormente no sistema (equivalente à atividade '*verificar registro prévio do host*' no gerenciador de arquivos do *DetVX*, mostrada na seção 5.3). Para isso, consultam-se os metadados do *super peer* administrativo e verifica-se se existem informações deste *peer* cadastradas. Em outras palavras, verifica-se se o identificador deste *peer*, dado pelo IP da máquina, aparece nos metadados do *super peer* administrativo. Se o identificador deste *peer* não é encontrado, então se assume que este *peer* nunca esteve conectado anteriormente ao sistema. Caso contrário, o próximo passo é verificar se houve modificações neste *peer* desde a última conexão, conforme detalhado na próxima atividade.

**Verificar modificações no *peer***. O *super peer* administrativo dispara um serviço de verificação de modificações no *peer* local (equivalente à atividade '*verificar modificações no host*' no gerenciador de arquivos do *DetVX*, mostrada na seção 5.3). Este serviço verifica se alguma mudança ocorreu nos documentos compartilhados desde a última vez em que o *peer* esteve conectado. Uma mudança pode ser a inserção e a remoção de um arquivo, ou a alteração no conteúdo de um arquivo já existente. O serviço de verificação de modificações executa uma função *hash* em cada documento. Os resultados da função *hash* são estruturados em uma mensagem XML, conforme já apresentado na seção 5.3.

**Reconectar ao *super peer* da última conexão.** Esta atividade será realizada se nenhuma mudança tiver ocorrido no *peer* desde sua última conexão. O fato de nenhuma mudança ter ocorrido implica em manter o mesmo domínio de aplicação para este *peer*. Neste caso, o *peer* é reconectado ao mesmo *super peer* da última conexão.

**Solicitar as ontologias disponíveis**. O *super peer* administrativo solicita as ontologias disponíveis ao gerenciador do repositório de ontologias;

**Conectar ao *super peer***. O *peer* conecta-se a um *super peer* específico. Esta atividade é equivalente à atividade '*registrar host ao ambiente*' no gerenciador de arquivos do *DetVX*, mostrada na seção 5.3, e é realizada em três situações:

- o usuário escolheu uma ontologia disponível;
- o usuário foi apresentado a uma ontologia já existente;
- o usuário foi apresentado a uma ontologia recentemente criada, a qual foi associada ao *peer*.

**Atualizar metadados.** Os metadados do *super peer* e do *super peer* administrativo são atualizados. Esta atividade é realizada sempre que houver qualquer mudança no *peer* desde sua última conexão, ou se é a primeira conexão deste *peer* no sistema. Esta atividade é equivalente à atividade '*atualizar metadados*' no gerenciador de arquivos do *DetVX*, mostrada na seção 5.3.

Depois do *peer* ter-se conectado a um *super peer*, todos os documentos registrados no *peer* estão disponíveis para compartilhamento. Inserções de arquivos neste *peer* pertencentes

ao mesmo domínio de aplicação dos documentos anteriormente registrados causam apenas atualização de metadados. Inserções relacionadas a um novo domínio implicam em escolher outra ontologia, além de atualizar os metadados necessários.

### 6.3.3 Gerenciador de Ontologias

Arquivos do mesmo domínio de aplicação podem estar espalhados pela rede P2P, o que pode gerar ineficiência na busca de recursos quando o usuário submete consultas. Para solucionar este problema, o módulo gerenciador de ontologias é o responsável pelo agrupamento de *peers* em *super peers*, com base no domínio de aplicação dos arquivos armazenados nos *peers*. Em outras palavras, este módulo é responsável pela manutenção do repositório de ontologias e pela associação de ontologias a *super peers*. Escolher uma ontologia implica indiretamente em escolher um *super peer* para se conectar, uma vez que cada *super peer* é descrito por uma ou mais ontologias. As principais diferenças entre o gerenciador de ontologias do *DetVX* e o gerenciador de ontologias do *DetVX_P2P* são:

- a ontologia gerada para os arquivos XML é associada ao *super peer*;
- os metadados do gerenciador de ontologias incluem informações de *super peer* e *peer*.

Informações sobre a associação de ontologias à *super peers* e seus respectivos agregados é descrita em um documento de metadados em XML, conforme mostrado na Figura 6.5. Os metadados descritos permitem especificar qual a ontologia (atributo `ontologia`) que descreve um determinado *super peer* (atributo `id`), um rótulo descritivo do domínio (`dominio`) e o arquivo em que a ontologia encontra-se armazenada (`localizacao`). Também representa os *peers* agregados (`peers`) daquele *super peer*.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <AssociacaoOntologias>
3      <superPeer id="SP2" ontologia="Ont1">
4         <dominio>Projetos de Pesquisa</dominio>
5         <localizacao>ResearchProjects.owl</localizacao>
6            <peers>
7               <peer>P3</peer>
8               <peer>P4</peer>
9               <peer>Pn-1</peer>
10           </peers>
11        </superPeer>
12     <superPeer id="SPn" ontologia="Ont2">
13        <dominio>Doenças</dominio>
14        <localizacao>Doencas.owl</localizacao>
15           <peers>
16              <peer>Pn-1</peer>
17              <peer>Pn</peer>
18           </peers>
19     </superPeer>
20  </AssociacaoOntologias>
```

Figura 6.5: Metadados do gerenciador de ontologias do *DetVX_P2P*

As atividades do gerenciador de ontologias são mostradas na Figura 6.6. A seguir, são detalhadas apenas as atividades que sofreram modificações em relação à descrição apresentada na seção 5.4 ou que foram adicionadas na extensão do *framework.* As atividades *“Mostrar ontologias disponíveis”*, *“Realizar o casamento entre arquivo e*

*ontologia”* e *“Atualizar repositório de ontologias”* seguem a mesma descrição das atividades apresentadas na seção 5.4.

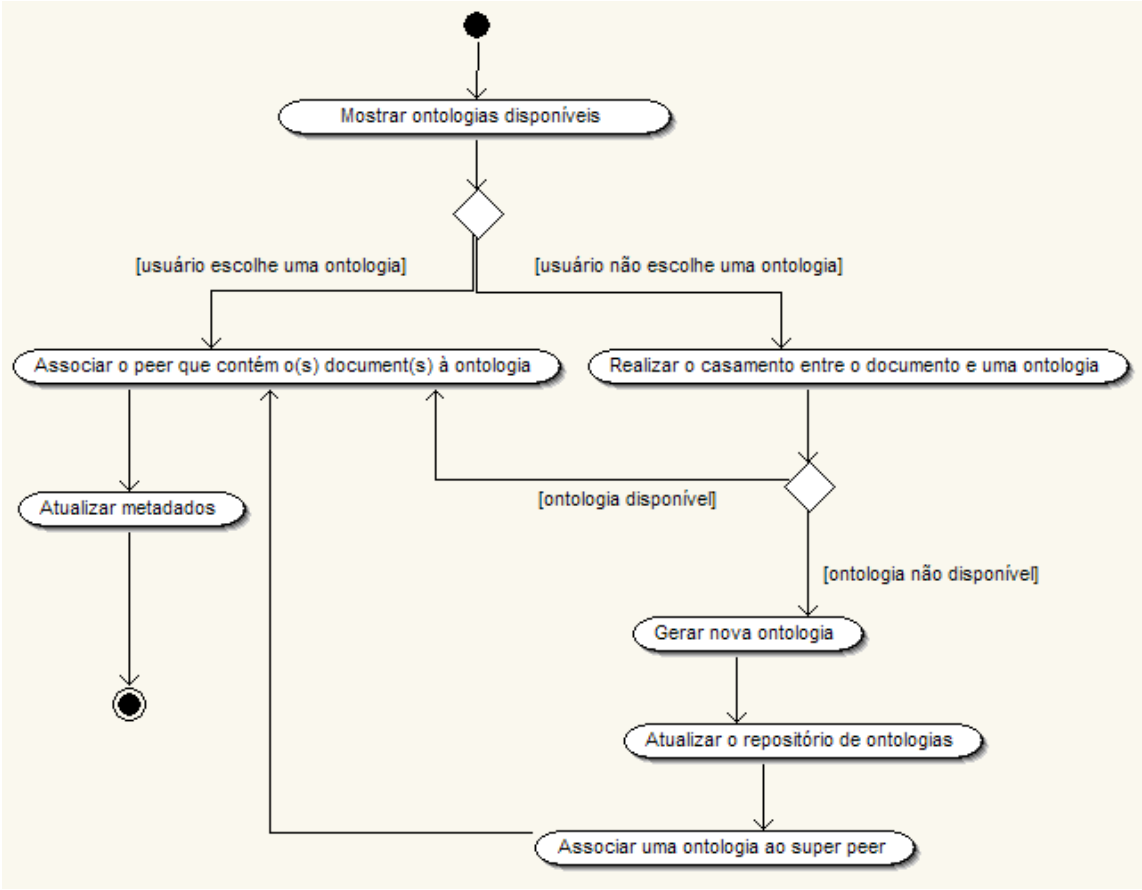



Figura 6.6: Diagrama de atividades do gerenciador de ontologias do *DetVX_P2P*

**Gerar nova ontologia.** Esta tarefa será realizada sempre que o sistema não encontrar uma ontologia adequada para um (conjunto de) arquivo (s) localizado (s) em um *peer*. Ao realizar o casamento de documentos com ontologias, este trabalho assume uma das seguintes opções:

- existe uma ontologia compatível com o arquivo - desta maneira, o *peer* que contém este documento é conectado ao *super peer* descrito por esta ontologia;
- não existe uma ontologia compatível com o arquivo - neste caso, gera-se uma ontologia que represente os conceitos deste domínio, atribui-se esta ontologia a um *super peer* e conecta-se o *peer* que contém este documento ao *super peer* descrito pela ontologia criada.

**Associar uma ontologia ao *super peer*.** Supondo que uma nova ontologia foi criada a partir de arquivos XML, esta ontologia precisa ser associada a um *super peer* existente. Alguns metadados do *super peer* administrativo precisam ser atualizados. Esta atividade substitui a atividade ‘*Associar uma ontologia ao arquivo’* descrita no *DetVX*.

Embora não seja o foco do trabalho, duas questões precisam ser comentadas. Critérios como balanceamento de carga devem ser levados em consideração no momento de escolher um determinado *super peer* para associar à ontologia. O reagrupamento de *peers* em *super peers* deve ocorrer quando um *super peer* possuir um número relativamente maior de *peers* do que outros *super peers* da mesma rede. O critério de balanceamento se dará pelo número de *peers* agregados em cada *super peer*, objetivando-se obter uma média de *peers* entre os *super peers existentes*. O reagrupamento de *peers* deve levar em consideração o critério utilizado no agrupamento (pertencer à mesma ontologia). Dessa forma, é possível que um reagrupamento de *peers* implique na associação da uma ontologia a outro *super peer* existente ou a um *super peer* recém criado.

A segunda questão é quanto ao número de *super peers* disponíveis. O número de *super peers* permitido em um sistema pode ser definido e parametrizado, de acordo com as características de cada sistema. Esta quantidade pode ser dinâmica e influenciada pela quantidade máxima de *peers* que um *super peer* pode suportar para o bom funcionamento do sistema. Para se chegar a um bom número, pode-se realizar uma avaliação do tempo de resposta a determinadas consultas submetidas em máquinas com *hardware* semelhantes, porém de diferentes localizações (BRITO et al., 2005).

**Associar o *peer* que contém o(s) documento(s) à ontologia.** O arquivo que anteriomente não possuia uma ontologia correspondente agora é associado ao *super peer* descrito pela ontologia recém criada.

### 6.3.4 Gerenciador de Réplicas e de Versões

Quando um *peer* deseja compartilhar arquivos ou ter acesso a arquivos compartilhados por outros *peers*, deve conectar-se a um *super peer* e registrar os arquivos a serem compartilhados. Algumas situações podem ocorrer quando um documento é registrado em um *peer*:

- o arquivo é uma nova instância - isso significa que este arquivo é um novo documento no sistema e nenhum *peer* possui este documento armazenado;
- o arquivo é uma nova versão de um documento já existente. Neste caso, outra versão já existe, localmente (neste *peer*) ou em outro *peer* da rede do super peer;
- o arquivo é uma réplica de um outro documento armazenado localmente (neste *peer*) ou em outro *peer* da rede do *super peer*.

Para identificar réplicas e versões de documentos no sistema P2P, considera-se que cada arquivo possui um identificador global, descrito na seção 6.3.1.

O processo de detecção de réplicas e de versões é executado entre os *peers* da rede do *super peer*.

#### *6.3.4.1 Detecção de Réplicas e de Versões*

O mecanismo de detecção de réplicas e de versões é executado sempre que um arquivo é registrado (inserido ou atualizado) em um *peer*. Quando um arquivo é removido de um *peer*, somente os metadados precisam ser atualizados. Para detectar quando um *peer* foi

modificado, propõe-se um serviço de verificação de modificações, o qual é responsável por periodicamente observar o *peer* e notificar seu *super peer* quando uma alteração for detectada. A Figura 6.7 sumariza as atividades relacionadas à detecção de réplicas e de versões, mostradas em um diagrama de atividades.

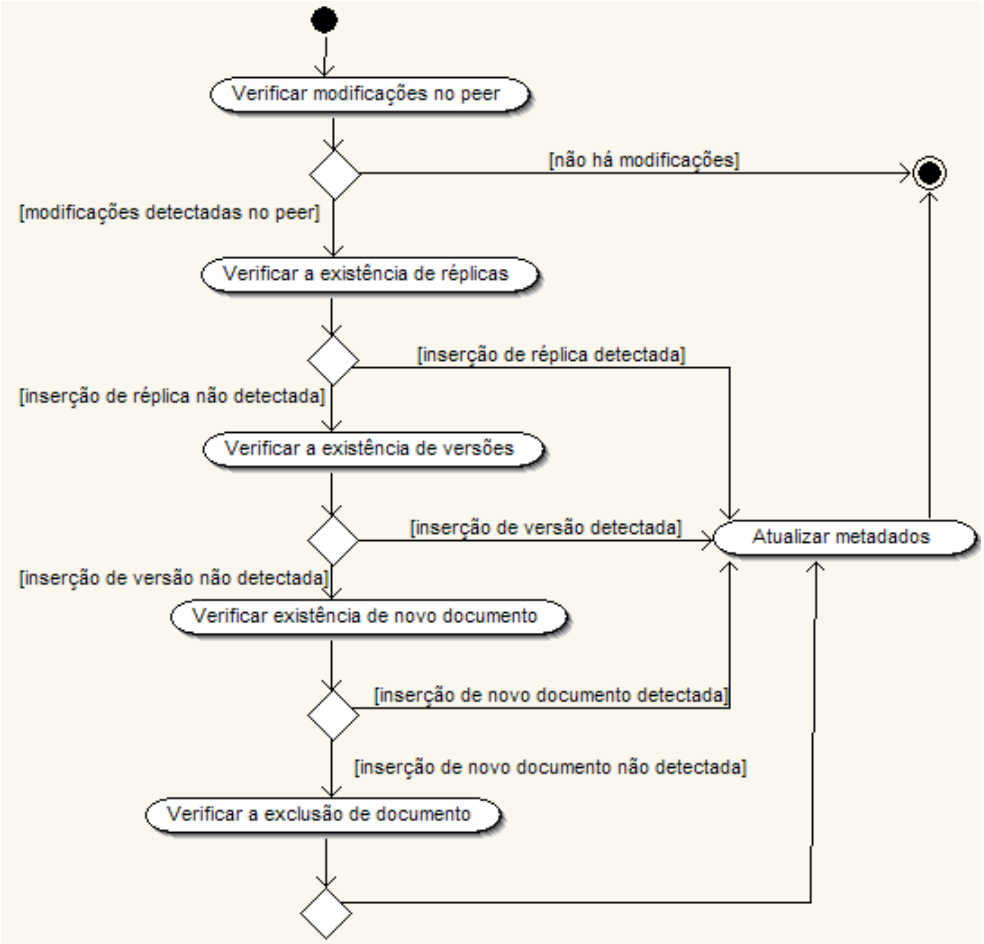



Figura 6.7: Diagrama de atividades do gerenciador de réplicas e de versões

**Verificar modificações no *peer*.** Equivalente à atividade *"Verificar modificações no host"* apresentada na seção 5.5.1. Cada *super peer* dispara um serviço de verificação de mudanças no *peer* local. A diferença entre elas reside na etapa de comparação. Esta atividade compara a mensagem XML gerada para cada *peer* com as informações relacionadas ao *peer*, localizadas nos respectivos *super peers*.

As atividades *"Verificar a existência de réplicas"*, *"Verificar a existência de versões"*, *"Verificar a existência de um novo documento"* e *"Verificar a exclusão de um documento"* seguem a mesma descrição das atividades apresentadas na seção 5.5.1. A única diferença é que como o *super peer* agrupa arquivos de certo domínio, a busca por réplicas, versões e novos documentos, dá-se entre os *peers* da sub-rede do *super peer* do domínio de aplicação considerado.

**Atualizar metadados**. Esta atividade é responsável por atualizar metadados do *super peer* e do *super peer* administrativo. Dependendo das modificações que foram detectadas, diferentes atualizações são necessárias.

A detecção de réplicas e de versões é realizada conforme o mecanismo apresentado na seção 5.5 e detalhado no capítulo 3. Como os arquivos relativos a um mesmo domínio de aplicação podem estar espalhados pela sub-rede do *super peer*, esta etapa (agrupamento de versões em um único arquivo físico - arquivo *H-Doc*, detalhado na seção 4.4) propõe armazenar a versão consolidada no *super peer*. Obviamente, nem todos os documentos teriam uma versão consolidada armazenada no *super peer*, mas principalmente aqueles documentos que são mais frequentemente acessados pelo sistema. O mecanismo para a geração do arquivo *H-Doc* não sofre alterações, e segue a abordagem apresentada na seção 4.4. Representações consolidadas armazenadas no *super peer* podem ser utilizadas para um processamento mais rápido de certas consultas que solicitam histórico de documentos. Dessa maneira, não há a necessidade de acessar todas as versões de um documento espalhadas pelos *peers.*

Em relação ao geranciamento de metadados realizado pelo detector de réplicas e versões (apresentado na seção 4.2.3), a única diferença é que agora há a necessidade de se manter a localização das versões detectadas na rede. Para isso, os metadados apresentados na Figura 4.11 são estendidos e mostrados na Figura 6.8.

```
<!ELEMENT metadata (node*)>
<!ELEMENT node (features*, node*)>
<!ATTLIST node file CDATA #REQUIRED
                idPeer CDATA #REQUIRED
                idSPeer CDATA #REQUIRED>
<!ELEMENT features EMPTY>
<!ATTLIST features  P CDATA #REQUIRED    S CDATA #REQUIRED
     R CDATA #REQUIRED    sim CDATA #REQUIRED
    D CDATA #IMPLIED    A CDATA #IMPLIED>
```

Figura 6.8: Estrutura dos metadados

Utilizando o modelo de metadados, a Figura 6.9 descreve os metadados para a Figura 4.2.

```
<metadata><node file="f1" idPeer=”P1” idSPeer=”SP1”>
  <node file="f2" idPeer=”P2” idSPeer=”SP1”>
      <features P="0.7" S="0.9" R="0.1" sim="0.86"/>
      <node file="f3" idPeer=”P3” idSPeer=”SP1”>
          <features P="0.8" S="0.9" R="0.1" sim="0.97"/>
      </node>
  </node>
</node>
<node file="f4" idPeer=”P2” idSPeer=”SP1”>
    <node file="f5" idPeer=”P2” idSPeer=”SP1”>
      <features P="0.6" S="0.7" R="0.1" sim="0.79"/>
      <node file="fx" idPeer=”P2” idSPeer=”SP1”>
          <features P="0.7" S="0.8" R="0.1" sim="0.89"/>
      </node>
    </node>
</node></metadata>
```

Figura 6.9: Exemplo de metadados para o versionamento linear

Analisando-se os metadados, identifica-se que há dois grupos de versões lineares. O primeiro grupo é formado pelos arquivos *f1*, *f2* e *f3*. O arquivo *f1* está armazenado no *peer*

*P1*, o arquivo *f2* está armazenado no *peer P2*, e o arquivo *f3* está armazenado no *peer P3*, todos eles localizados no *super peer SP1*. O segundo grupo é formado pelos arquivos *f4* e *f5*. O arquivo *f4* está armazenado no *peer P2* e o arquivo *f5* também está armazenado no *peer P2*, todos eles localizados no *super peer SP1*.

### 6.3.5 Metadados

Os metadados são classificados em três níveis:

#### *6.3.5.1 Metadados do Super Peer Administrativo*

Metadados do *super peer* administrativo não eram previstos no *framework DETVX*, uma vez que os arquivos eram armanzenados em um único *host.* Como o *framework DETVX_P2P* pressupõe a existência de diversos *peers* e *super peers*, os metadados do *super peer* administrativo descrevem os *super peers* existentes (`ID`) e seus *peers* agregados (`peerID`), os identificadores locais (`docID`), os resultados hash (`hashResult`) , os tempos de registro (`registeringTime`) e as últimas datas de atualização (`modificationTime`) de cada arquivo no *peer*. O tempo de registro de um documento é dado pelo tempo em que o arquivo foi disponibilizado no *peer* para compartilhamento no sistema. O atributo *docID* é um valor único mapeado a partir do resultado de uma função *hash.* Os metadados do *super peer* administrativo são ilustrados na Figura 6.10.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <metadata>
3      <superPeer ID="SP1">
4         <message peerID="P1">
5      <document docID="D1"
6   hashResult="ece50ed4d6d48dac839bfe8fa719fcff"  registeringTime="10/10/2005"
7   modificationTime=08/08/2004”/>
8      <document docID="D2"
9   hashResult="e3732b09b5b2a9aa452b8ef7802db638" registeringTime="10/15/2005"
10  modificationTime=12/12/2004”/>
11     <document docID="D3"
12  hashResult="73cbe8e94c7fa839ba1246b34b2a49cd" registeringTime="10/20/2005"
13  modificationTime=10/08/2005”/>
14        </message>
15        <message peerID="P2">
16             <document docID="D1"
17     hashResult="ece50ed4d6d48dac839bfe8fa719fcff" registeringTime="08/20/2005"
18  modificationTime=01/02/2004”/>
19             <document docID="D4"
20     hashResult="4324349b5b2a9aa452b8ef7fdsfdffdf" registeringTime="11/22/2005"
21  modificationTime=11/20/2004”/>
22             <document docID="D5"
23     hashResult="nnggfge94c7fa839ba1244543fdsfrar" registeringTime="10/29/2005"
24  modificationTime=07/08/2003”/>
25        </message>
25     </superPeer>
26     <superPeer ID="SP2">....</superPeer>
27  </metadata>
```

Figura 6.10: Metadados do *super peer* administrativo

Estes metadados são acessados nas seguintes situações:

- o sistema precisa verificar a ocorrência de alguma modificação no *peer* - modificações no *peer* são detectadas quando o resultado da função *hash* aplicada

sobre um arquivo retorna um valor diferente dos resultados armazenados nos metadados. Também pode ser considerado o atributo *modificationTime*, o qual representa a última data de atualização do arquivo no *peer;*

- o sistema precisa atualizar os metadados quando for a primeira conexão de um *peer* - neste caso, os metadados são atualizados com informações como: qual é o *peer*, qual(is) é(são) seu(s) *super peer*, qual é o resultado *hash* dos documentos deste *peer*, qual é o tempo de registro destes documentos no *peer* (tempo em que o arquivo foi disponibilizado no *peer* para compartilhamento no sistema) e qual é a última data de atualização deste documento no *peer* (`modificationTime`);
- o sistema precisa atualizar os metadados após a reconexão de um *peer* (quando os documentos do *peer* foram modificados).

### *6.3.5.2 Metadados do Super Peer*

Metadados do *super peer* correspondem aos metadados do *host*, descritos na seção 5.6.1. Cada *super peer* possui metadados com informações sobre seus *peers* agregados e seus respectivos documentos registrados. Estes metadados podem ser estruturados em um documento XML, como mostrado na Figura 6.11. Basicamente, os metadados especificam as versões e as réplicas disponíveis em um determinado *super peer* (`superPeerId`), e os rótulos temporais correspondentes para cada elemento (`TS, TE`) encontrado em um dado documento (`docId`) de um *peer* (`peerID`). Estes rótulos temporais são derivados a partir das datas de modificação dos arquivos (`modificationTime`). Os metadados são atualizados sempre que um novo documento é registrado em um *peer* e são acessados durante o processo de consulta.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE Metadata SYSTEM "C:\METADATA.dtd">
3  <Metadata superPeerId="SP1">
4   <document docId="D7">
5       <version versionID="1" peerID="P1" registeringTime="10/10/2005"
6       modificationTime="08/08/2004" duplicate="no"
7       hashResult="d49622ddab3733549e54749755fd52b5">
8             <element name="author" TS="08/08/2004" TE="10/15/2004"/>
9             <element name="address" TS="08/08/2004" TE="10/15/2004"/>...
10      </version>
11      <version versionID="2" peerID="P2" registeringTime="11/20/2005"
12      modificationTime="10/16/2004" duplicate="yes"
13      hashResult="7c00bb062edc60fa548729a3d55c04fd">
14            <locationDuplicate>Peer 3</locationDuplicate>
15            <element name="author" TS="10/16/2004" TE="now"/>
16            <element name="address" TS="10/16/2004" TE="now"/>
17            <element name="phone" TS="10/16/2004" TE="now"/>...
18      </version>
19 </document>
20 <document docId="D8">......
21 </document>
22 </Metadata>
```

Figura 6.11: Metadados do *super peer*

No exemplo da Figura 6.11, observa-se que a versão 2 (`versionID="2"`, linha 11) do documento D7 (`docId="D7"`, linha 4) está replicada nos *peers* 2 (`peerID="P2"`, linha 11)

e 3 (`Peer 3`, linha 14). Supõem-se os seguintes tempos de modificação para esta réplica: *peer* 2 (10/16/2004, linha 12) e *peer* 3 (12/10/2005). Nota-se que somente foi mantido o TS mais antigo para os elementos desta réplica.

### *6.3.5.3 Metadados do Gerenciador de Ontologias*

Os metadados descritos permitem especificar qual a ontologia (`ontology`) descreve um determinado *super peer* (`superPeer id`), quais *peers* (`peer id`) estão associados a este *super peer*, quais arquivos (`doc`) deste domínio estão armazenados num dado *peer* (`peer id`), um rótulo descritivo do domínio (`domain`), e o arquivo em que a ontologia encontra-se armazenada (`file`), conforme mostrado na Figura 6.12.

```
<AssociatedOntologies>
   <superPeer id="SP2" ontology="Ont1"><domain>Research Projects</domain><file>ResearchProjects.owl</file>
      <peers>  <peer id="P3"><doc>F2</doc> <doc>F7</doc> <doc>F8</doc></peer>
               <peer id="P4"><doc>F1</doc> <doc>F5</doc></peer>
      </peers>
   </superPeer>
</AssociatedOntologies >
```

Figura 6.12: Metadados do gerenciador de ontologias

Além dos metadados, algumas modificações também se fazem necessárias nos mapeamentos (apresentados na seção 5.6.3), conforme mostrado a seguir.

### *6.3.5.4 Mapeamentos*

Instâncias correspondentes aos termos na ontologia são armazenadas nos arquivos XML dos *peers*. Desta forma, informações de mapeamento relacionam termos na ontologia aos elementos de dados nos arquivos XML. Similar ao que foi apresentado na seção 5.6.3, a informação de mapeamento é representada como uma função de transformação.

Por exemplo, considere a ontologia apresentada na Figura 5.21 e os arquivos XML mostrados na Figura 5.20. Suponha que o arquivo (a) está armazenado no *peer 1*, e o arquivo (b) está armazenado no *peer 2* (ambos localizados no *super peer SP1*). Como as informações de um mesmo domínio são agrupadas em *super peers*, assume-se que os mapeamentos são mantidos para cada *super peer*. Desta forma, fica a necessidade de referenciar o *peer* para o qual o mapeamento deve ser feito.

Alguns mapeamentos são descritos na Tabela 6.1. Assume-se que o elemento *address* na ontologia corresponde à concatenação de rua, número, cidade e estado.

Tabela 6.1: Funções de transformação em *XPath* para a ontologia *resumé*

| **Conceito da ontologia** | **Mapeamento *XPath*** | |
|---|---|---|
| | ***Peer* 1** | ***Peer 2*** |
| | **Documento 1 (a)** | **Documento 1 (b)** |
| Person | | |
| Person/resume | //resume | //resume |
| Person/Name | /resume/name | /resume/fullName |

| | | |
|---|---|---|
| Person/Name/Firstname | /resume/name/firstname | substring-before(/resume/fullName, “ ”) |
| Person/Name/MiddleName | - | - |
| Person/Name/LastName | /resume/name/surname | substring-after /resume/fullName, “ ”) |
| Person/Name/Suffix | - | - |
| Person/Address | concat(/resume/address/street,../city,../state) | concat(/resume/address/adr, ../state) |

Similar ao apresentado no *DETVX*, os mapeamentos são gerados pelo gerenciador de ontologias durante a fase de casamento entre arquivos e ontologias. A tarefa de casamento estabelece correspondências entre a ontologia e os documentos XML, determinando os conceitos equivalentes.

### 6.3.6 Processador de Consultas

A principal diferença entre o processador de consultas do *DetVX_P2P* em relação ao componente proposto na seção 5.7 está no roteamento das requisições entre os *peers* da rede. No *framework DetVX_P2P*, o usuário submete uma consulta a um determinado *peer*, chamado *peer* solicitante. Antes de processar a consulta submetida, o sistema precisa verificar se esta requisição pode ser respondida localmente. Em caso afirmativo, a consulta é processada e os resultados são retornados ao usuário; caso contrário, a consulta é roteada para o *peer* adequado. Para verificar se uma consulta pode ser respondida localmente (pelo *peer* solicitante), o sistema verifica os metadados de seu *super peer*, conforme descrito a seguir.

#### *6.3.6.1 Consulta aos Metadados*

Para uma determinada consulta do usuário pertencente a certo domínio de aplicação, duas opções devem ser analisadas:

1) o usuário submete uma consulta cujo domínio é o mesmo da sub-rede do *super peer* - em outras palavras, o usuário está buscando por conceitos que existem na sub-rede deste *super peer*. Duas opções devem ser analisadas:

    - se o *peer* solicitante puder respondê-la localmente, a consulta é processada e os resultados são retornados ao usuário. Por exemplo, este é o caso quando a consulta está solicitando a primeira versão do salário dos empregados e o *peer* solicitante possui esta informação. Para verificar se o *peer* solicitante tem condições de responder a consulta, o sistema precisa verificar os metadados do seu *super peer*, exemplificados na Figura 6.11. Consultando estes metadados, têm-se condições de descobrir quais são os *peers* aptos a responder a consulta. Como a consulta solicitou a primeira versão de um dado, consulta-se os metadados e verifica-se quem é o *peer* (`peerID`) que possui a primeira versão do elemento *address*. Verificando nos metadados, conclui-se que a primeira versão deste dado (`versionID="1",` linha 5) encontra-se no peer 1 (`peerID="P1",` linha 5). Supondo que o *peer* solicitante é o *peer* 1, então a consulta pode ser respondida localmente;

- se a consulta não puder ser respondida localmente, ela é roteada para o *super peer* do *peer* solicitante. Este é o caso da consulta 2, em que é solicitada a última versão do endereço dos autores. A última versão de uma informação é aquela que possui TE=*now*, ou seja, é a versão mais atual e sua validade ainda não foi encerrada. Novamente, consultando os metados, verifica-se que o elemento *address* possui TE=*now* (linha 16) na versão 2 do documento (`versionID="2"`, linha 11) que encontra-se disponível no *peer* 2 (`peerID="P2"`, linha 11). Desta forma, a consulta precisa ser roteada para o *super peer* do *peer* solicitante, o qual se responsabilizará pelo reenvio desta consulta ao *peer* apto a respondê-la. O *peer* 2, então, processa a consulta e retorna os resultados ao *super peer*. Finalmente, os resultados são retornados ao usuário. Este processo é exemplificado na Figura 6.13;

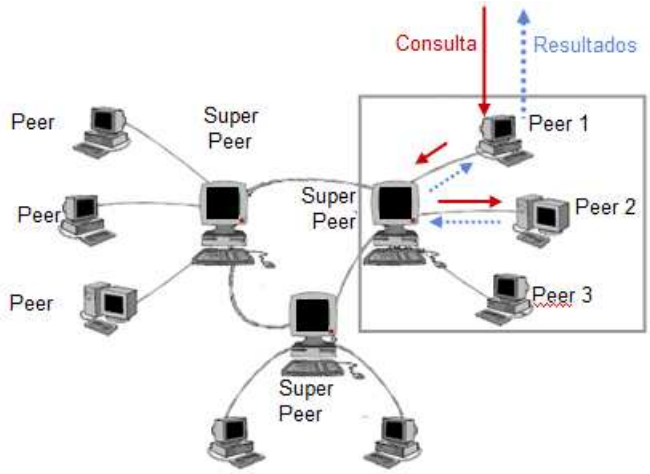



Figura 6.13: Roteamento da consulta 2 dentro da rede do *super peer*

2) o usuário submeteu uma consulta cujo domínio é diferente do domínio da sub-rede do seu *super peer* - em outras palavras, o usuário está buscando por informações que não existem na sub-rede deste *super peer*. Neste caso, a consulta precisa ser roteada para o *super peer* que possua documentos do domínio solicitado. Para verificar qual *super peer* possui documentos do domínio solicitado, consultam-se os metadados do gerenciador de ontologias, mostrados na Figura 6.12. Este trabalho não detalha o processo de casamento entre domínio da consulta e domínio descrito pela ontologia de um *super peer*. Assume-se que o usuário informa explicitamente o domínio sobre o qual a consulta é formulada.

A partir deste ponto, o processo segue de forma similar ao caso descrito no item 1. O *super peer* consulta seus metadados e verifica qual é o *peer* que possui a informação solicitada. Esta consulta é roteada para este *peer*, o qual processa a requisição e retorna os resultados para o usuário final. Este processo é exemplificado na Figura 6.14.

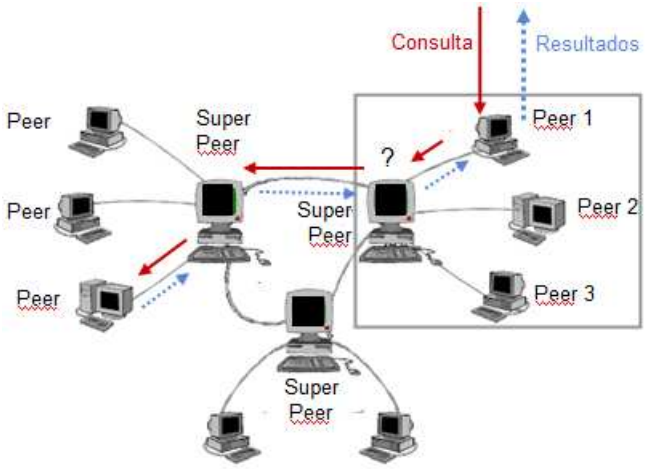



Figura 6.14: Roteamento de consultas entre redes de *super peers*

Uma vez consultados os metadados, o sistema já tem conhecimento de quais arquivos precisa acessar e onde estes arquivos encontram-se armazenados. A partir deste momento, inicia-se a etapa de processamento de consultas, propriamente dita. O processamento de consultas segue a abordagem definida na seção 5.7.2.

Com o agrupamento de versões em um único arquivo físico (*H-Doc*), determinadas consultas podem ser diretamente respondidas pelo *super peer*, sem acessar os vários arquivos espalhados pelos *peers*. Caso o *super peer* não tenha condições de responder a consulta, uma mensagem de aviso é enviada ao usuário. O processo de consulta aos arquivos *H-Doc* é similar ao apresentado na seção 5.7.2.1

Com os dois exemplos mostrados nesta seção, verifica-se que a infra-estrutura disponibilizada pelos metadados fornece um ambiente adequado para submissão de consultas. Outros exemplos de consultas poderiam incluir integração de resultados de dois documentos diferentes, vindos de *peers* diferentes. No entanto, distribuição de consultas e integração de resultados não são descritas neste trabalho.

Uma vez consultados os metadados, o sistema já tem conhecimento de quais arquivos precisa acessar e onde estes arquivos encontram-se armazenados. A partir deste momento, inicia-se a etapa de processamento de consultas, propriamente dita. Esta etapa é similar à apresentada na seção 5.7.2. A análise dos metadados do *super peer* permite descobrir se existem tais arquivos criados e onde estão armazenados. A partir daí, a consulta é processada de forma similar ao que já foi apresentado no *framework DetVX*.

## 6.4 Considerações Finais

Este capítulo detalhou a arquitetura proposta do *framework* P2P (*DETVX_P2P*) para detecção, gerenciamento e consulta a réplicas e versões. Este *framework* é uma extensão do *framework DETVX*, apresentado no capítulo 5. As principais diferenças entre os dois *frameworks* são sumarizadas na Tabela 6.2.

Este capítulo resultou nas seguintes publicações: (SACCOL et al., 2007d; SACCOL et al., 2007e). Estas produções estão detalhadas no capítulo 7.

Tabela 6.2: Tabela comparativa entre os *frameworks DETVX e DETVX_P2P*

| | **Principal diferença** | ***DETVX*** | ***DETVX_P2P*** |
|---|---|---|---|
| **Visão geral** | *Localização dos arquivos* | • No *host* | • Nos *peers*, os quais são agrupados em *super peers* |
| **Gerenciador de arquivos / *peers*** | *Nomenclatura e funcionalidade* | • Chamado de gerenciador de arquivos<br>• Gerencia o registro do *host* no ambiente, e modificações posteriores nos arquivos do *host* | • Chamado de gerenciador de *peers*<br>• Gerencia o registro dos *peers* no ambiente, e modificações posteriores nos arquivos do *peer* |
| **Gerenciador de ontologias** | *Associação de ontologias* | • Ontologia é associada aos arquivos do domínio de aplicação | • Ontologia é associada ao *super peer* que agrega *peers* com arquivos do domínio de aplicação |
| **Gerenciador de réplicas e versões** | *Detecção de réplicas e versões* | • Ocorre entre os arquivos do *host* | • Ocorre entre os arquivos do mesmo domínio de aplicação (*super peer*) |
| | *Armazenamento do arquivo H-doc* | • No *host* | • No *super peer* |
| **Metadados** | *Níveis de metadados* | • Metadados do *host*<br>• ---<br>• Metadados do gerenciador de ontologias<br>o Associa ontologia aos arquivos<br>• Mapeamentos | • Metadados do *super peer*<br>• Metadados do *super peer* administrativo<br>• Metadados do gerenciador de ontologias<br>o Associa ontologia ao *super peer*<br>• Mapeamentos |

| | | | |
|---|---|---|---|
| | | o Considera os mapeamentos da ontologia para os arquivos do *host* | o Considera os mapeamentos da ontologia para os arquivos do *super peer* |
| **Processador de consultas** | *Roteamento das consultas* | • Consulta é submetida ao *host*<br>• Domínio da consulta e do *host* são os mesmos:<br>o Processa no *host* e retorna resultados<br>• Domínio da consulta e do *host NÃO* são os mesmos:<br>o Não processa e retorna aviso ao usuário | • Consulta é submetida ao *peer*<br>• Domínio da consulta e do *peer* são os mesmos:<br>o Se o *peer* solicitante puder responder<br>❖ Processa no *peer* e retorna resultados<br>o Se o *peer* solicitante NÃO puder responder<br>❖ Roteia para o seu *super peer*<br>• Domínio da consulta e do *peer NÃO* são os mesmos:<br>o Roteia para o *super peer* adequado<br>❖ Se existe o *super peer* adequado e este pode responder: processa e retorna resultados<br>❖ Se NÃO existe o *super peer* adequado ou NÃO pode responder: retorna aviso ao usuário |

# 7 CONCLUSÕES

Esta tese apresenta um mecanismo para detecção, gerenciamento e consulta a réplicas e a versões de documentos XML. Esta tese também especifica um *framework* onde este mecanismo é usado e a extensão deste *framework* para um cenário P2P.

O problema da detecção de versões é um tópico atual de pesquisa na comunidade de Bancos de Dados. Este fato se justifica porque as aplicações do conceito de versão são diversas. Além disso, trabalhos prévios focam no gerenciamento e na consulta de versões, mas não na sua detecção. No entanto, o problema da detecção de versões é crítico em muitos cenários, tais como detecção de plágio, ranqueamento de páginas Web, identificação de clones de *software* e busca em sistemas P2P, entre outros.

Este capítulo apresenta as principais contribuições da tese, as publicações originadas, o comparativo com os principais trabalhos relacionados e as propostas de trabalhos futuros.

## 7.1 Contribuições da Tese

Esta tese especifica um mecanismo para a detecção, o gerenciamento e a consulta a réplicas e a versões de documentos XML. Neste contexto, as principais contribuições do trabalho foram a definição de um conjunto de funções de similaridade para arquivos XML, a definição de um mecanismo de detecção de versões baseado em técnicas de classificação e a definição de um *framework* onde o mecanismo de detecção de réplicas e de versões pode ser inserido.

Para que tais contribuições fossem atingidas, a solução proposta foi desmembrada nas seguintes atividades:

- *definição de um mecanismo para a detecção de réplicas* – um mecanismo simples foi proposto com base no uso de funções *hash*;
- *definição de um conjunto de funções de similaridade para a detecção de versões* – as funções definidas não são restritas a uma determinada aplicação, visto que podem ser adaptadas para que considerem também outras características em diferentes cenários, e possuem pesos associados. Isto torna esta abordagem mais flexível quando comparada a outras funções de similaridade existentes na literatura;
- *definição de um mecanismo para a detecção de versões* – o mecanismo de detecção de versões foi proposto com base em classificadores *Naïve Bayesianos*, o que eliminou as

desvantagens do uso de limiares. Embora o uso desta técnica na classificação de documentos não seja algo inédito, a definição de quais atributos devem ser considerados pelo classificador torna-se um problema à parte quando o objetivo é a detecção de versões;

- *definição de uma abordagem de "agrupamento" de versões de documentos XML em um único arquivo físico, contendo toda a evolução histórica dos documento* – estes arquivos podem ser utilizados para um processamento mais rápido para as consultas que solicitam histórico de documentos. Dessa maneira, não há a necessidade de acessar todas as versões de um documento espalhadas pelo repositório;
- *definição de um framework onde o mecanismo de detecção de réplicas e de versões está inserido.* O *framework* contempla o gerenciamento de arquivos, de ontologias, de réplicas e de versões, de metadados, além do processamento de consultas às réplicas e às versões detectadas;
- *definição de um mecanismo de agrupamento de arquivos, com base em seus domínios de aplicação* – arquivos relacionados a um domínio de aplicação são agrupados e o processamento de consultas pode ser realizado somente em um subconjunto de arquivos;
- *extensão do framework para um cenário P2P* – sistemas P2P tradicionais não fornecem suporte eficiente à recuperação de recursos replicados e versionados. Tais desvantagens podem ser reduzidas com o uso do *framework* proposto.

## 7.2 Publicações

Esta tese resultou nos seguintes artigos publicados:

1. **Deise de Brum Saccol**, Nina Edelweiss, Renata de Matos Galante, Márcio Roberto de Mello: *Managing Application Domains in P2P Systems*. In: *Proceedings of the IEEE IRI 2008 - International Conference on Information Reuse and Integration*, Las Vegas, NV, USA, 2008. Classificação Qualis: Conferência Internacional B.

2. **Deise de Brum Saccol**, Nina Edelweiss, Renata de Matos Galante: *A Metadata Model for Managing and Querying XML Resources in Peer-to-peer Systems*. In: *Proceedings of the 20th SEKE - International Conference on Software Engineering & Knowledge Engineering*, San Francisco, CA, USA, 2008. Classificação Qualis: Conferência Internacional B.

3. **Deise de Brum Saccol**, Rodrigo Perozzo Noll, Nina Edelweiss, Renata de Matos Galante: *An Ontology-Based Approach for Semantic Interoperability in P2P Systems*. In: *Proceedings of the 10th ICEIS - International Conference on Enterprise Information Systems*, Barcelona, Spain, 2008. Classificação Qualis: Conferência Internacional A.

4. Glauber Rodrigues da Silva, Renata de Matos Galante, **Deise de Brum Saccol**: *Uma Proposta para o Uso de Detecção de Versões de Páginas Web para Melhoria do Desempenho do Algoritmo de Page Rank.* In: Anais da 4th ERBD - Escola Regional de Banco de Dados, Florianópolis, 2008.

5. **Deise de Brum Saccol**, Nina Edelweiss, Renata de Matos Galante, Carlo Zaniolo: *XML Version Detection*. In: *Proceedings of the DocEng - ACM Symposium on Document Engineering*, Winnipeg, Manitoba, Canada, 2007. Classificação Qualis: Conferência Internacional A.

6. **Deise de Brum Saccol**, Nina Edelweiss, Renata de Matos Galante, Carlo Zaniolo: *Managing XML Versions and Replicas in a P2P Context*. In: *Proceedings of the 19th SEKE - International Conference on Software Engineering Knowledge Engineering*, Boston, Massachusetts, USA, 2007. Classificação Qualis: Conferência Internacional B.

7. **Deise de Brum Saccol**, Nina Edelweiss e Renata de Matos Galante: *Detecting, Managing and Querying Replicas and Versions in a Peer-to-Peer Environment*. In: *Proceedings of the The First IEEE TCSC Doctoral Symposium*, em conjunto com o *7th IEEE International Symposium on Cluster Computing and the Grid* (CCGrid), Rio de Janeiro, Brasil, 2007. Classificação Qualis: Conferência Internacional C.

8. **Deise de Brum Saccol**, Nina Edelweiss e Renata de Matos Galante: *Detecting, Managing and Querying XML Replicas and Versions in Peer-to-Peer Environments*. In: VI WTDBD - Workshop de Teses e Dissertações em Bancos de Dados, em conjunto com o Simpósio Brasileiro de Banco de Dados (SBBD), João Pessoa, Brasil, 2007.

9. **Deise de Brum Saccol**, Felipe dos Santos Giacomel, Renata de Matos Galante, Nina Edelweiss: *Agrupamento e Consulta a Versões de Documentos XML em um Ambiente Peer-To-Peer*. In: Anais do V XATA - XML: Aplicações e Tecnologias Associadas, Lisboa, Portugal, 2007.

10. Rodrigo Perozzo Noll, **Deise de Brum Saccol**, Nina Edelweiss. *Uma proposta para Análise de Similaridade entre Documentos XML e Ontologias em OWL*. In: I Sessão de Pôsteres, em conjunto com o Simpósio Brasileiro de Banco de Dados (SBBD), João Pessoa, Brasil, 2007.

Três relatórios técnicos foram produzidos:

1. **Deise de Brum Saccol**, Nina Edelweiss e Renata de Matos Galante. *XML Version Detection*. Technical Report no. 355, UFRGS, Porto Alegre, Brasil, 2007.
2. **Deise de Brum Saccol**, Nina Edelweiss e Renata de Matos Galante. *Evolução de Esquemas em Ambientes de Integração de Dados*. Trabalho Individual, UFRGS, Porto Alegre, Brasil, 2005.

3. **Deise de Brum Saccol**, Nina Edelweiss e Renata de Matos Galante. *Integração de Esquemas em Fontes Heterogêneas XML*. Trabalho Individual no. 1150, UFRGS, Porto Alegre, Brasil, 2004.

Durante o desenvolvimento da tese de doutorado, também foram realizadas as seguintes orientações e co-orientações:

1. Co-orientação do projeto de iniciação científica, UFRGS, 2007: *Implementação de Funções de Similaridade para Detecção de Versões de Documentos XML*. Aluno: Rodrigo Otávio Silva Santos.

2. Co-orientação do projeto de iniciação científica, UFRGS, 2007: *Implementação de Funções de Similaridade para Detecção de Versões de Documentos XML*. Aluna: Mell Fogliatto Santa Lucia.

3. Co-orientação do trabalho de conclusão de curso em Ciência da Computação, UFRGS, 2006: *XVersion: Uma Ferramenta Gráfica para Gerenciamento e Consulta de Versões de Documentos XML*. Aluno: Felipe dos Santos Gicomel. Este trabalho foi **vencedor do prêmio** de melhor trabalho de Graduação da Universidade Federal do Rio Grande do Sul 2006/2 - Prêmio Assespro-RS (Associação das Empresas Brasileiras de Tecnologia da Informação).

4. Co-orientação do trabalho de conclusão de curso em Ciência da Computação, UFRGS, 2006: *Ontogen: Uma Ferramenta para Integração de Esquemas XML*. Aluno: Márcio Roberto de Mello.

5. Co-orientação (não-formal) da monografia de conclusão de curso em Especialização em Web e Sistemas de Informação, UFRGS, 2006: *Uma Proposta para Análise de Similaridade entre Documentos XML e Ontologias Definidas em OWL*. Aluno: Rodrigo Perozzo Noll.

## 7.3 Comparativo com os Trabalhos Relacionados

Sistemas tradicionais de controle de versões modelam arquivos como seqüências de linhas de texto, armazenando a última versão e usando *scripts* de edição reversa para recuperar versões prévias. Estes sistemas não preservam a estrutura lógica original do documento e não suportam consultas complexas, sendo inadequados para o tratamento de versões XML. Além disso, trabalhos prévios focam no gerenciamento de versões e não na detecção destas versões. Propostas para detecção de réplicas focam na identificação de múltiplas representações do mesmo objeto do mundo real (WEIS et al., 2006), as quais podem ter diferenças de conteúdo e de estrutura. No contexto desta tese, uma réplica é considerada uma cópia idêntica de um arquivo XML, o que inviabiliza o uso destas abordagens.

Técnicas para análise de similaridade baseadas em *diff* (COBENA et al., 2002a; WANG et al., 2003) e em distância de edição (CHAWATHE, 1999) geralmente produzem um resultado pobre em semântica, com pouca informação em relação ao grau de similaridade

entre os documentos, para ser usado na detecção de versões. Além disso, funções de similaridade encontradas na literatura são geralmente adequadas para requisitos específicos (focam em similaridade de estrutura ou em similaridade de conteúdo, por exemplo). Para o problema da detecção de versões, muitas características diferentes devem ser consideradas ao mesmo tempo. Nesta tese, é definido um conjunto de funções de similaridade que consideram várias características com pesos diferentes, o que gera uma abordagem mais flexível em termos de análise de similaridade para a detecção de versões. Além disso, funções de similaridade são fortemente dependentes do uso de limiares. Esta tese utiliza um método mais robusto, baseado em classificadores *Naïve Bayesianos*, o que elimina esta dependência.

Em relação ao *framework* proposto, convém ressaltar que embora existam vários sistemas de integração de dados (BROEKSTRA et al., 2003; BOYD et al, 2004), não há um único sistema que trata de todas as questões importantes (tais como heterogeneidade, escalabilidade, mudanças nas fontes locais, complexidade do processamento de consultas, evolução do esquema global, etc) em uma única abordagem unificada. Esta tese trata o problema da heterogeneidade através do uso de funções de mapeamento e o mecanismo é projetado para preservar a escalabilidade (uma vez que o estudo de caso foi feito em sistemas P2P). Além disso, mudanças nas fontes locais são um requisito constante, uma vez que os *peers* entram e saem do sistema e alteram seus arquivos regularmente. Finalmente, a evolução do esquema global pode ser facilmente alcançada através da evolução da ontologia e dos mapeamentos necessários.

Sistemas que são baseados em palavras-chave (FREENET, 2008) não recuperam o documento necessário se um sinônimo é utilizado como parte da consulta. Além disso, sistemas automáticos falham nas tarefas de encontrar e extrair informação relevante, e integrar esta informação distribuída por várias fontes (FENSEL, 2001). Para tratar destas questões, anotações semânticas (metadados) usadas nesta tese possibilitam definições estruturais e semânticas dos documentos, fornecendo um processamento de consultas mais eficiente que possibilita aos usuários submeter consultas sem o conhecimento da localização e estrutura dos arquivos.

Com relação ao casamento entre ontologias e arquivos XML, os trabalhos encontrados na literatura são geralmente baseados em similaridade estrutural (FRANCESCA et al., 2003; DALAMAGAS et al., 2004b; LIAN et al., 2004) ou de conteúdo (BAEZA-YATES et al., 1999). No entanto, o problema desta tese inclui identificar correspondências semânticas, as quais podem existir mesmo entre representações com diferenças significativas de estrutura e de conteúdo. Embora uma ontologia e um arquivo XML possam apresentar baixa similaridade de conteúdo e de estrutura, mesmo assim eles podem se referir a um mesmo domínio de aplicação, invibilizando o uso destas técnicas para o problema tratado. Esta tese propõe um mecanismo que não se baseia apenas na similaridade estrutural e de conteúdo, mas também em um dicionário de dados (*WordNet*), para aperfeiçoar a análise de similaridade semântica entre ontologias e arquivos.

Com relação ao estudo de caso realizado, convém ressaltar que a usabilidade de sistemas P2P é dependente principalmente das técnicas usadas para localizar e recuperar os recursos (YANG et al., 2002). O uso da técnica *flooding* (inundação) garante os resultados ótimos,

mas é cara e consome tempo, uma vez que todos os *peers* recebem a mensagem de consulta e geralmente apenas alguns deles estão aptos a respondê-la. Outra alternativa para balanceamento de custo é o uso de alguma variação da técnica *flooding*, tal como varredura em largura ou varredura em profundidade na rede de *peers*. No entanto, o uso de alguma destas variações não garante a solução ótima, uma vez que nem todos os *peers* irão receber a mensagem da consulta. Nesta tese, o espaço de busca é reduzido pelo agrupamento de *peers* em *super peers* (através do uso de ontologias). O mecanismo garante que uma determinada consulta relacionada a certo domínio é roteada somente dentro da sub-rede de um *super peer*, o que diminui o custo na busca de recursos. Paralelamente, o mecanismo garante a solução ótima, uma vez que todos os arquivos do domíno de aplicação são consultados.

## 7.4 Trabalhos Futuros

Como trabalhos futuros propõe-se a extensão do *framework* para outras aplicações. Esta extensão pode requerer a adaptação do mecanismo de detecção de réplicas e de versões, como, por exemplo, a otimização das funções de similaridade, dos pesos utilizados e do método de classificação. Embora o estudo de caso tenha sido realizado para um ambiente P2P, o problema da detecção de versões é crítico em muitos cenários, tais como:

- ***ranking* de páginas Web -** métodos de *ranking* geralmente envolvem a localização e a freqüência de palavras-chave em uma página Web. Máquinas de busca verificam se as palavras-chave pesquisadas aparecem no topo da página (no cabeçalho ou nos primeiros parágrafos). A freqüência também é considerada, analisando-se a proporção de vezes em que as palavras-chave aparecem no texto, em relação a outras palavras da página Web. Outro fator que pode ser considerado no processo de *ranking* é o número de *links* de entrada de uma página (denotado por *g* - número de *links* que apontam para uma página *p*). Novas versões desta página podem ter um baixo *g* porque páginas que apontam para *p* ainda não estão cientes da existência de suas novas versões. Neste contexto, a detecção de versões e de réplicas pode ser útil para melhorar o *ranking* de novas versões de páginas, mesmo que apresentem um baixo grau de *links* de entrada;
- **detecção de plágio -** plágio é o ato de assinar ou apresentar uma obra intelectual de qualquer natureza contendo partes de uma obra que pertença a outra pessoa, sem colocar os créditos para o autor original. Arquivos digitais podem ser facilmente copiados, de uma forma parcial ou na sua totalidade. Uma forma de detectar plagiarismo é pela comparação de seus *checksums*, uma abordagem simples e suficiente para a detecção de cópias exatas. No entanto, detectar cópias parciais é mais complicado, uma vez que qualquer alteração mínima no texto já ocasiona um *checksum* diferente. Com o mecanismo de detecção de versões proposto nesta tese, arquivos similares podem ser identificados;
- **identificação de clones de *software* -** código replicado pode surgir durante o processo de desenvolvimento de *software*, o que causa um impacto negativo na manutenção destes sistemas. A detecção torna-se mais difícil por causa de pequenas diferenças entre os clones, tais como mudanças no corpo do código, mudanças de nome de variáveis, etc. Mecanismos existentes para a detecção de clones de *software* geralmente baseiam-se no uso de analisadores sintáticos, mas estas abordagens são dependentes da

sintaxe da linguagem de programação. A representação clássica de código em texto plano é conveniente para programadores, mas requer análise sintática para a verificação da estrutura do programa. Representar código em um formato estruturado, como o XML, permite uma especificação mais útil para uma série de análises da Engenharia de *Software*. Neste contexto, considerando tais clones como versões, o problema de detecção pode ser tratado com o mecanismo proposto na tese.

# REFERÊNCIAS

ABERER, K.; HAUSWIRTH, M. An Overview on Peer-to-Peer Information Systems. In: WORKSHOP ON DISTRIBUTED DATA AND STRUCTURES, WDAS, 2002. **Proceedings…** [S.l.:s.n.], 2002. p.171-188.

AMAGASA, T.; YOSHIKAWA, M.; UEMURA, S. A Data Model for Temporal XML Documents. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, DEXA, 11., 2000, London. **Database and Expert Systems Applications**: proceedings. Berlin: Springer-Verlag, 2000. p. 334-344.

AMAGASA, T.; YOSHIKAWAT, M.; UEMURA, S. Realizing Temporal XML Repositories using Temporal Relational Databases. In: INTERNATIONAL SYMPOSIUM ON COOPERATIVE DATABASE SYSTEMS FOR ADVANCED APPLICATIONS, CODAS, 2001. **Proceedings…** [S.l.:s.n.], 2001. p.60-64.

BAEZA-YATES, R.A.; RIBEIRO-NETO, B. A. **Modern Information Retrieval**. New York: ACM, 1999.

BARRETO, P.; RIJMEN, V. The Whirlpool Hashing Function. In: OPEN NESSIE WORKSHOP, 2000. **Proceedings…** [S.l.:s.n.], 2000.

BATINI, C.; LENZERINI, M.; NAVATHE, S.B. A Comparative Analysis of Methodologies for Database Schema Integration. **Computing Surveys**, New York, v.18, n.4, p. 323-364, Dec. 1986.

BERNSTEIN, P. et al. Data Management for Peer-to-Peer Computing: A Vision. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WebDB, 2002. **Proceedings…** [S.l.:s.n.], 2002. p.89-94.

BERTINO, E.; GUERRINI, G.; MESITI, M. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. **Information Systems**, Oxford, v.29, n.1, p.23-46, Mar. 2004.

BILENKO, M. et al. Adaptive Name Matching in Information Integration. **IEEE Intelligent Systems**, Piscataway, v.18, n.5, p.16-23, Sept. 2003.

BOYD, M. et al. AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In: ADVANCED INFORMATION SYSTEMS ENGINEERING CONFERENCE, CAISE, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.82-97.

BRITO, G.A.D.D.; MOURA, A.M. de C. Integração de Objetos de Aprendizado no Sistema ROSA - P2P. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 20., 2005, Uberlândia, MG. **Anais...** Porto Alegre: Universidade Federal do Rio Grande do Sul, 2005. p.235-249.

BROEKSTRA, J. et al. A Metadata Model for Semantics-Based Peer-to-Peer Systems. In: WORKSHOP ON SEMANTICS IN PEER-TO-PEER AND GRID COMPUTING, SEMPGRIG, 2003. **Proceedings…** [S.l.:s.n.], 2003.

BRY, F. et al. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In: WORKSHOP ON PRINCIPLES AND PRACTICE OF SEMANTIC WEB REASONING, PPSWR, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.34-49.

CASTANO, S. et al. An XML-Based Framework for Information Integration over the Web. In: INTERNATIONAL WORKSHOP ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS AND SERVICES, 2000. **Proceedings…** [S.l.:s.n.], 2000.

CASTOR Project. Disponível em: <http://www.castor.org>. Acesso em: maio 2008.

CHAKRABARTI, S. **Mining the Web**: Discovering Knowledge from Hypertext Data. San Francisco: Morgan Kaufmann, 2002.

CHAWATHE, S.; WIDOM, J. Representing and Querying Changes in Semistructured Data. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 1998. **Proceedings…** [S.l.:s.n.], 1998. p.4-13.

CHAWATHE, S.S.; ABITEBOUL, S.;WIDOM, J. Managing Historical Semistructured Data. **Theory and Practice of Object Systems**, New York, v. 5, n.3, p. 143-162, Aug. 1999.

CHAWATHE, S.S. Comparing Hierarchical Data in External Memory. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1999. **Proceedings…** [S.l.:s.n.], 1999. p.90-101.

CHIEN, S. et al. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In: INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS ENGINEERING, WISE, 2001. **Proceedings…** [S.l.:s.n.], 2001. p.232-241.

CHIEN, S.; TSOTRAS, V.J.; ZANIOLO, C. Efficient Management of Multiversion Documents by Object Referencing. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASE SYSTEMS, VLDB, 21., 2001. **Proceedings…** [S.l.:s.n.], 2001. p.291-300.

CHIEN, S.; TSOTRAS, V.J.; ZANIOLO, C. Version Management of XML Documents. In: INTERNATIONAL WORKSHOP WEBDB ON THE WORLD WIDE WEB AND DATABASES, 3., Dallas, USA. **Proceedings...** Berlin: Springer, 2001. p.184-200. (Lecture Notes in Computer Science, v.1997).

CHIEN, S. et al. Efficient Complex Query Support for Multiversion XML Documents. In : INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, EDBT, 2001. **Proceedings…** [S.l.:s.n.], 2001. p.161-178.

CHIEN, S.Y.; TSOTRAS, V.J.; ZANIOLO, C. XML Document Versioning. **SIGMOD Record**, New York, v.30, n.3, p.46-53, Sept. 2001.

COBENA, G.; ABITEBOUL, S.; MARIAN, A. Detecting Changes in XML Documents. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2002. **Proceedings…** [S.l.:s.n.], 2002. p.41-52.

COHEN, W. W.; RAVIKUMAR, P.; FIENBERG, S. A comparison of String Distance Metrics for Name-matching Tasks. In: WORKSHOP ON INFORMATION INTEGRATION ON THE WEB, 2003. **Proceedings…** [S.l.:s.n.], 2003. p.71-78.

CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. **Computing Surveys**, New York, v.30, n.2, p. 232-282, June 1998.

DAI, W. et al. Transferring Naive Bayes Classifiers for Text Classification. In: CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2007. **Proceedings…** [S.l.:s.n.], 2002. p.540-545.

DALAMAGAS, T. et al. Clustering XML Documents using Structural Summaries. In: EDBT WORKSHOP ON CLUSTERING INFORMATION OVER THE WEB, CLUSTWEB, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.547-556.

DALAMAGAS, T. et al. Clustering XML Documents by Structure. In: HELENIC CONFERENCE ON AI, 3., Samos, Greece. **Methods and Applications of Artificial Intelligence**: proceedings. Berlin: Springer, 2004. p.112-121. (Lecture Notes in Computer Science, v.3025).

DOMINGOS, P.; PAZZANI, M. On the optimality of the simple Bayesian classifier under zero-one loss. **Machine Learning**, Hingham, v.29, n.2-3, p. 103-137, Nov. 1997.

DORNELES, C. F. et al. Measuring similarity between collections of values. In: WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT, WIDM, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.56-63.

DULUCQ, S. ; TOUZET, H. Analysis of Tree Edit Distance Algorithms. In: ANNUAL SYMPOSIUM ON COMBINATORIAL PATTERN MATCHING, 14., Morelia, Mexico. **Combinatorial pattern matching**: proceedings. Berlin: Springer, 2003. p.83-95. (Lecture Notes in Computer Science, v.2676).

DYRESON, C. Observing Transaction-Time Semantics with /sub TT/XPath. In: INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS ENGINEERING, WISE, 2001. **Proceedings…** [S.l.:s.n.], 2001. p.193-202.

EUZENAT, J.; SHVAIKO, P. **Ontology Matching**. Berlin: Springer-Verlag, 2007.

FARKAS, J. Generating Document Clusters Using Thesauri and Neural Networks. In: CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING, CCECE, 1994. **Proceedings…** [S.l.:s.n.], 1994. p.710-713.

FENSEL, D. **Ontologies**: A Silver Bullet for Knowledge Management and Electronic Commerce. Berlin: Springer, 2001.

FLESCA, S. et al. Detecting Structural Similarities between XML Documents. In: INTERNATIONAL WORKSHOP ON WEB AND DATABASES, WebDB, 2002. **Proceedings…** [S.l.:s.n.], 2002.

FLESCA, S.; PUGLIESE, A. Fast Detection of XML Structural Similarity. **IEEE Transactions on Knowledge and Data Engineering**, Piscataway, v.17, n.2, p. 160-175, Feb. 2005.

FLORESCU, D.; LEVY, A.; MENDELSON, A. Database techniques for the world wide web: a survey. **SIGMOD Record**, New York, v. 27, n.3, p. 59-74, Sept. 1998.

FRANCESCA, F.D. et al. Distance-based Clustering of XML Documents. In: *W*ORKSHOP ON MINING GRAPHS, TREES AND SEQUENCES, MGTS, 2003. **Proceedings…** [S.l.:s.n.], 2003. p.75-78.

FREENET. Disponível em: <http://freenet.sourceforge.net>. Acesso em: jan. 2008.

GALANTE, R. M.; SANTOS, C. S.; RUIZ, D. D. A. Um Modelo de Evolução de Esquemas Conceituais para BDOO com o Emprego de Versões. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 13., 1998, Maringá, PR. **Anais...** Porto Alegre: Universidade Federal do Rio Grande do Sul, 1998. p.303-318.

GALANTE, R.M. et al. Temporal and Versioning Approach to Schema Evolution in Object-Oriented Databases. **Data & Knowledge Engineering**, [S.l.], v.53, n.2, p.99-128, May 2005.

GALANTE, R. M. **Um Modelo de Evolução de Esquemas Conceituais para Bancos de Dados Orientados a Objetos com o Emprego de Versões**. 1998. 92f. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

GAO, D.; SNODGRASS, R.T. Temporal Slicing in the Evaluation of XML Queries. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.632-643.

GARCIA, L. G. **Uma ferramenta para engenharia reversa de esquemas XML em esquemas conceituais no ambiente BInXS**. 2005. Trabalho de Conclusão (Graduação em Ciência da Computação) - Centro Tecnológico, UFSC, Florianópolis.

GERGATSOULIS, M.; STAVRAKAS, Y. Representing Changes in XML Documents using Dimensions. In: XML DATABASE SYMPOSIUM, XSYM, 2003. **Proceedings…** [S.l.:s.n.], 2003. p.208-222.

GIACOMEL, F. S. **XVersion:** uma ferramenta para a detecção de diferenças em agrupamentos de arquivos XML. 2006. Trabalho de Conclusão (Graduação em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

GOLDMAN, R. et al. **A Standard Textual Interchange Format for the Object Exchange Model (OEM)**. [S.l.]: Stanford University, 1998.

GOLENDZINER, L.G. **Um modelo de versões para banco de dados orientados objetos**. 1995. 147p. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

GRANDI, F.; MANDREOLI, F. The Valid Web: an XML/XSL Infrastructure for Temporal Management of Web Documents. In: ADVANCES IN INFORMATION SYSTEMS, ADVIS, 1., 2000, Izmir. **Proceedings...** London: [s.n.], 2000. p.294-303.

GRANDI, F. et al. A Temporal Data Model and Management System for Normative Texts in XML Format. In: INTERNATIONAL WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT, WIDM, 5., 2003, New Orleans. **Proceedings…** New York: [s.n.], 2003. p.29-36.

GRANDI, F.; MANDREOLI, F.; TIBERIO, P. Temporal Modeling and Management of Normative Documents in XML Format. **Data & Knowledge Engineering**, Amsterdam, v. 54, n. 3, p. 327-354, Sept. 2005.

GRUBER, T. R. A Translation Approach to Portable Ontologies. **Knowledge Acquisition**, [S.l.], v.5, n.2, p.199-200, 1993.

GUTH, G.J. Surname Spellings and Computerized Record Linkage. **Historical Methods Newsletter**, [S.l.], 1976.

HALL, P. A. V.; DOWLING, G. R. Approximate String Matching. **Computing Surveys**, New York, v.12, n.4, p.381-402, Dec. 1980.

HAN, J.; KAMBER, M. **Data Mining**: concepts and techniques. San Francisco: Morgan Kaufmann, 2006.

JACCARD, P. The Distribution of the Flora in the Alpine Zone. **New Phytologist**, [S.l.], v.11, n.2, p.37-50, Feb. 1912.

JARO, M. A. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa-Florida. **Journal of the American Statistical Association**, [S.l.], v.84, n.406, p.414-420, June 1989.

JENSEN, C. S. et al. The Consensus Glossary of Temporal Database Concepts. In: Etzion, O; Jajodia, S; Sripada, S. (Ed.). **Temporal Databases**: research and practice. Berlin: Springer-Verlag, 1998.

JEONG, E.; HSU, C. Induction of Integrated View for XML Data with Heterogeneous DTDs. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, CIKM, Atlanta. **Proceedings…** New York: [s.n.], 2001. p.151-158.

JOACHIMS, T. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In: EUROPEAN CONFERENCE ON MACHINE LEARNING, ECML, Chemnitz. **Proceedings…** [S.l.:s.n.], 1998. p. 137-42.

KANTROWITZ, M.; MOHIT, B.; MITTAL, V. Stemming and its effects on TFIDF ranking. In: CONFERENCE ON RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL, SIGIR, Athens. **Proceedings….** [S.l.:s.n.], 2000. p.357-359.

KATZ, R.; CHANG, E. Managing Change in a Computer-Aided Design Database. In: CONFERENCE ON VERY LARGE DATABASE SYSTEMS, VLDB, 13., 1987. **Proceedings...** [S.l.:s.n.], 1987. p.455-462.

KOTSIANTIS, S.B. ; PINTELAS, P.E. Increasing the Classification Accuracy of Simple Bayesian Classifier. In: CONFERENCE ON ARTIFICIAL INTELLIGENCE: METHODOLOGY, SYSTEMS, APPLICATIONS, 11., 2004, Varna, Bulgaria. **Proceedings...** [S.l.:s.n.], 2004. p.198-207.

LAMMEL, R.; LOHMANN, W. Format Evolution. In: INTERNATIONAL CONFERENCE ON REVERSE ENGINEERING FOR INFORMATION SYSTEMS, RETIS, 7., 2001. **Proceedings…** [S.l.:s.n.], 2001.

LEE, M.L. et al. XClust: Clustering XML Schemas for Effective Integration. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, 11., 2002, McLean, Virginia. **Proceedings…** New York: ACM Press, 2002. p.292-2002.

LEVENSHTEIN, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. **Soviet Physics Doklady**, [S.l.], v.10, p.707-727, Feb. 1966.

LIAN, W. et al. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. **IEEE Transactions on Knowledge and Data Engineering**, New York, v.16, p.82-96, 2004.

LÓSCIO, B.F. **Managing the Evolution of XML-based Mediation Queries**. 2003. Tese (Doutorado em Ciência da Computação) – UFPE, Recife.

MADHAVAN, J.; BERNSTEIN, P. A.; RAHM, E. Generic schema matching using Cupid. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 27., 2001. Rome, Italy. **Proceedings…** [S.l.:s.n.], 2001.

MAEDCHE, A.; STAAB, S. Measuring similarity between ontologies. In: INTERNATIONAL CONFERENCE ON KNOWLEDGE ENGINEERING AND KNOWLEDGE MANAGEMENT, 13., 2002, Siguenza, Spain. **Ontologies and the Semantic Web**: proceedings. Berlin: Springer, 2002. p.251-263. (Lecture Notes In Computer Science, v.2473).

MANNING, C. D.; SCHÜTZE, H. **Foundations of Statistical Natural Language Processing**. Cambridge, MA: MIT Press, 1999.

MCCALLUM, A.; NIGAM, K. A comparison of event models for Naïve bayes text classification. In: AAAI WORKSHOP ON LEARNING FOR TEXT CATEGORIZATION, 1998, Madison, Wiscosin. **Proceedings…** [S.l.]: AAAI Press, 1998. p. 41-48.

MELLO, R. S. **Uma abordagem bottom-up para integração semântica de esquemas XML**. 2002. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MELLO, M. R. **OntoGen**: uma ferramenta para integração de esquemas XML. 2007. Trabalho de Conclusão (Graduação em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MENA, E.; ILLARRAMENDI, A. **Ontology-based query processing for global information systems**. New York: Springer, 2001. 232p. (The Springer International Series in Engineering and Computer Science, v.619).

NATIONAL SECURITY AGENCY (NSA). **FIPS 180-2: Secure Hash Standard** (SHS). [S.l. : s.n.], 2004.

NAVARRO, G. A guided tour to approximate string matching. **Computing Surveys**, New York, v.33, n.1, p. 31-88, Mar. 2001.

NIERMAN, A.; JAGADISH, H.V. Evaluating Structural Similarity in XML Documents. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 5., 2002, Madison, Wiscosin. **Proceedings…** [S.l.:s.n.], 2002. p.61-66.

NIGAM, K. et al. Text Classification from Labeled and Unlabeled Documents using EM. **Machine Learning**, Hingham, v.39, n.2-3, p.103-134, May 2000.

NOLL, R.; SACCOL, D. B.; EDELWEISS, N. Uma Proposta para Análise de Similaridade entre Documentos XML e Ontologias em OWL. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 22., 2007, João Pessoa, PB. **Sessão de Pôsteres**: anais. João Pessoa: SBC, 2007. p. 47-50.

NOLL, R. P. **Uma Proposta para Análise de Similaridade entre Documentos XML e Ontologias definidas em OWL**. 2007. Trabalho de Conclusão (Especialização) - Instituto de Informática, UFRGS, Porto Alegre.

OZSOYOGLU, G.; SNODGRASS, R. Temporal and Real-Time Databases: A Survey. **IEEE Transactions on Knowledge and Data Engineering**, New York, v.7, p. 513-532, 1995.

PON, R.K. et al. iScore: Measuring the Interestingness of Articles in a Limited User Environment. In: IEEE SYMPOSIUM ON COMPUTATIONAL INTELLIGENCE AND DATA MINING, CIDM, 2007, Honolulu, HI. **Proceedings…** [S.l.: s.n.], 2007. p. 354-361.

PROTÉGÉ-OWL API. Disponível em: <http://protege.stanford.edu/plugins/owl/api/>. Acesso em: maio 2008.

RISH, I. An Empirical Study of the Naive Bayes Classifier. In: WORKSHOP ON EMPIRICAL METHODS IN ARTIFICIAL INTELLIGENCE, 2001. **Proceedings…** [S.l.:s.n.], 2001.

RIVEST, R. **The MD4 Message-Digest Algorithm**. Disponível em: <http://tools.ietf.org/html/rfc1320>. Acesso em: set. 2007.

RIVEST, R. **The MD5 Message-Digest Algorithm**. Disponível em: <http://tools.ietf.org/html/rfc1321>. Acesso em: set. 2007.

ROCHKIND, M.J. The Source Code Control System. **IEEE Transactions on Software Engineering**, New York, v.1, n.4, p.364-370. Dec. 1975.

RONNAU, S.; SCHEFFCZYK, J.; BORGHOFF, U.M. Towards XML Version Control of Office Documents. In: ACM SYMPOSIUM ON DOCUMENT ENGINEERING, DOCENG, 5., Bristol, UK. **Proceedings…** New York: ACM Press, 2005. p.10-19.

SAATY, T.L. How to make a decision: The analytic hierarchy process. **European Journal of Operational Research**, [S.l.], v.48, n.1, p.9-26, 1990.

SAATY, T.L. Decision-making with the ahp: Why is the principal eigenvector necessary? **European Journal of Operational Research**, [S.l.], v.145, n.1, p.85-91, 2003.

SACCOL, D.; HEUSER, C. Integration of XML Data. In: VLDB WORKSHOP ON EFFICIENCY AND EFFECTIVENESS OF XML TOOLS AND TECHNIQUES, 1., 2002, Hong Kong. **Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web**: revised papers. Berlin: Springer-Verlag, 2003. p.68-80.

SACCOL, D. B. et al. Agrupamento e Consulta a Versões de Documentos XML em um Ambiente Peer-to-Peer. In: CONFERÊNCIA NACIONAL EM XML: APLICAÇÕES E TECNOLOGIAS ASSOCIADAS, 5., 2007, Lisboa, Portugal. **Xata 2007**: XML: Aplicações e Tecnologias Associadas: 5ª Conferência Nacional. Lisboa: [s.n.], 2007. p.187-198.

SACCOL, D. B. et al. Managing XML Versions and Replicas in a P2P Context. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, SEKE, 19., Boston, MA. **Proceedings…** Skokie: Knowledge Systems Institute, 2007. p.680-685.

SACCOL, D. B. et al. XML Version Detection. In : ACM SYMPOSIUM ON DOCUMENT ENGINEERING, DOCENG, 7., 2007, Winnipeg, MN. **Proceedings…** New York: ACM, 2007. p.79-88.

SACCOL, D. B.; EDELWEISS, N.; GALANTE, R. M. **XML Version Detection**. Porto Alegre: UFRGS, 2007. (Technical Report, n. 355).

SACCOL, D. B.; EDELWEISS, N.; GALANTE, R. M. Detecting, Managing and Querying Replicas and Versions in a Peer-to-Peer Environment. In: INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND GRID, CCGRID, 7., 2007, Rio de Janeiro, RJ. **IEEE TCSC Doctoral Symposium**: proceedings. Washington: IEEE Computer Society, 2007. p. 881-886.

SACCOL, D. B.; EDELWEISS, N.; GALANTE, R. M. Detecting, Managing and Querying XML Replicas and Versions in Peer-to-Peer Environments. In: WORKSHOP DE TESES E DISSERTAÇÕES EM BANCOS DE DADOS, WTDBD, 6., 2007, João Pessoa, PB. **Anais...** João Pessoa: SBC, 2007. p. 73-80.

SACCOL, D. B. et al. Managing Application Domains in P2P Systems. In: IEEE INTERNATIONAL CONFERENCE ON INFORMATION REUSE AND INTEGRATION, IRI, 2008, Las Vegas, NV. **Proceedings…** Piscataway: IEEE, 2008. p. 451-456.

SACCOL, D. B. et al. An Ontology-based Approach for Semantic Interoperability in P2P Systems. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, ICEIS, 10., 2008, Barcelona, Spain. **Proceedings…** [Setubal]: ISTICC, 2008.

SACCOL, D. B.; EDELWEISS, N.; GALANTE, R. M. A Metadata Model for Managing and Querying XML Resources in Peer-to-Peer Systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING,

SEKE, 20., 2008, Redwood City, USA. **Proceedings…** Skokie: Knowledge Systems Institute Graduate School, 2008.

SALTON, G.; MCGILL, M. **Introduction to Modern Information Retrieval**. New York: McGraw-Hill, 1983.

SANTOS, R.G.; GALANTE, R. M.; EDELWEISS, N. TVX - Time and Versions in XML. **Journal of Theoretical and Applied Computing**, Porto Alegre, v.14, n.2, p.9-27, Dec.2007.

SCHOLLMEIER, R. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architetures and Applications. In: INTERNATIONAL CONFERENCE ON PEER-TO-PEER COMPUTING, P2P, 1., 2001, Linköping, Sweden. **Proceedings…** [S.l.:s.n.], 2001. p.27-29.

SELKOW, S.M. The Tree-to-Tree Editing Problem. **Information Processing Letters**, [S.l.], v.6, n.6, p.184-186, 1977.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de Banco de Dados**. Rio de Janeiro: Campus, 2006.

SILVA, R. et al. Measuring quality of similarity functions in approximate data matching. **Journal of Informetrics**, [S.l.], v.1, n.1, p.35-46, Jan. 2007.

SIMANOVSKY, A. Evolution of Schema of XML-Documents Stored in a Relational Database. In: DATABASES AND INFORMATION SYSTEMS, DB&IS, 6., 2004, Latvia. **Proceedings…** [S.l.:s.n.], 2004. p. 192-204.

STAAB. S.; STUDER, R. **Handbook on Ontologies**. Berlin: Springer, 2004.

STASIU, R. K. **Avaliação da Qualidade de Funções de Similaridade no Contexto de Consultas por Abrangência**. 2007. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

STASIU, R. K.; HEUSER, C. A.; SILVA, R. Estimating Recall and Precision for vague queries in Databases. In: INTERNATIONAL CONFERENCE ADVANCED INFORMATION SYSTEMS ENGINEERING, 17., 2005, Porto, Portugal. **Proceedings…** Berlin: Springer, 2005. p.187-200. (Lecture Notes in Computer Science, v. 3520).

SU, H. et al. XEM: Managing the Evolution of XML Documents. In: INTERNATIONAL WORKSHOP ON RESEARCH ISSUES IN DATA ENGINEERING: DOCUMENT MANAGEMENT FOR DATA INTENSIVE BUSINESS AND SCIENTIFIC APPLICATIONS, RIDE-DM, 11., 2001. **Proceedings…** Berlin: Springer, 2001.

SVINGER, B. Using genetic programming for document classification. In: INTERNATIONAL FLORIDA ARTIFICIAL INTELLIGENCE RESEARCH, FLAIRS, 11., 1998, Sanibel Island, Florida, USA. **Proceedings...** [S.l.:s.n.], 1998. p. 63-67.

TANSEL, A. et al. **Temporal Databases**: theory, design and implementation. Redwood City: Benjamin/Cummings Publishing Company, 1993.

TATARINOV I. et al. The Piazza Peer Data Management Project. **SIGMOD Record**, New York, v.32, n.3, 2003.

TICHY, W.F. RCS - A System for Version Control. **Software - Practice and Experience**, London, v.15, n.7, p.637-654, July 1985.

TRIANTAFILLOU, P. et al. Towards High Performance Peer-to-Peer Content and Resource Sharing Systems. In: CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH, CIDR, 1., 2003, Pacific Grove, CA, USA. **Proceedings…** [S.l.:s.n.], 2003.

VAGENA, Z.; TSOTRAS, V. Path-Expression Queries over Multiversion XML Documents. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2003, San Diego, CA, USA. **Proceedings…** [S.l.:s.n.], 2003. p.49-54.

VAGENA, Z.; MORO, M.; TSOTRAS, V. Supporting Branched Versions on XML Documents. In: INTERNATIONAL WORKSHOP ON RESEARCH ISSUES ON DATA ENGINEERING: WEB SERVICES FOR E-COMMERCE AND E-GOVERNMENT APPLICATIONS, RIDE-WS-ECEG, 14., 2004, Boston, MA, USA. **Proceedings…** [S.l.:s.n.], 2004. p.137-144.

WAN, X.; YANG, J. Using Proportional Transportation Similarity with Learned Element Semantics for XML Document Clustering. In: INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, WWW, 15., 2006, Edinburgh, Scotland, UK. **Proceedings…** [S.l.:s.n.], 2006. p. 961-962.

WANG, Y.; DEWITT, D. J.; CAI, J. X-Diff: An Effective Change Detection Algorithm for XML Documents. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 19., 2003, Bangalore, India. **Proceedings…** [S.l.:s.n.], 2003. p. 519-530.

WANG, Y.; HODGES, J.; TANG, B. Classification of Web Documents Using a Naive Bayes Method. In: IEEE INTERNATIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE, ICTAI, 15., 2003, Sacramento, CA, USA. **Proceedings…** [S.l.:s.n.], 2003. p.560-564.

WANG, F.; ZANIOLO, C. Representing and Querying the Evolution of Databases and their Schemas in XML. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, SEKE, 15., 2003, San Francisco Bay, CA, USA. **Proceedings….** [S.l.:s.n.], 2003. p. 33-38.

WANG, F.; ZANIOLO, C. Temporal Queries in XML Document Archives and Web Warehouses. In: INTERNATIONAL SYMPOSIUM ON TEMPORAL REPRESENTATION AND REASONING AND FOURTH INTERNATIONAL CONFERENCE ON TEMPORAL LOGIC, TIME, 10., 2003, Cairns, Queensland, Australia. **Proceedings…** [S.l.:s.n.], 2003. p. 47-55.

WANG, F.; ZANIOLO, C. XBiT: An XML-based Bitemporal Data Model. In: INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING, ER, 23., 2004, Shanghai, China. **Proceedings…** [S.l.:s.n.], 2004. p.810-824.

WANG, F. et al. Managing Multiversion Documents and Historical Databases: a Unified Solution Based on XML. In: INTERNATIONAL WORKSHOP ON THE WEB, WEBDB, 8., 2005, Baltimore, Maryland. **Proceedings…** [S.l.:s.n.], 2005. p. 151-153.

WANG, F.; ZANIOLO, C. An XML-Based Approach to Publishing and Querying the History of Databases. **World Wide Web**, Hingham, MA, v.8, n.3, p.233-259, Sept. 2005.

WEIS, M.; NAUMANN, F. Detecting Duplicate Objects in XML Documents. In: INTERNATIONAL WORKSHOP ON INFORMATION QUALITY IN INFORMATION SYSTEMS, IQIS, 2004. **Proceedings…** [S.l.:s.n.], 2004. p.10-19.

WEIS, M.; NAUMANN, F. Detecting Duplicates in Complex XML Data. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 22., 2006. **Proceedings…** [S.l.:s.n.], 2006. p. 109-109.

WESTFECHTEL, B.; MUNCH, B. P.; CONRADI, R. A Layered Architecture for Uniform Version Management. **IEEE Transactions on Software Engineering**, New York, v.27, n.12, p.1111-1133, 2001.

WINKLER, W. E. **The state of record linkage and current research problems**. Washington, D.C.: Statistical Research Division, U.S. Bureau of the Census, 1999. (Report N.: RR1999/04).

XML Path Language (XPath) 2.0. Disponível em: <http://www.w3.org/TR/xpath20/>. Acesso em: jan. 2008.

XQuery 1.0: An XML Query Language. Disponível em: <http://www.w3.org/TR/xquery>. Acesso em: jan. 2008.

XYLEME, L. A Dynamic Warehouse for XML Data of the Web. **IEEE Data Engineering Bulletin**, [S.l.], v.24, n.2, p.40-47, 2001.

YANG, B.; GARCIA-MOLINA, H. Efficient Search in Peer-to-Peer Networks. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, ICDCS, 2002, Vienna, Austria. **Proceedings…** [S.l.:s.n.], 2002.

YANG, J.; CHEUNG, W. ; CHEN, X. Learning the Kernel Matrix for XML Document Clustering. In: IEEE INTERNATIONAL CONFERENCE ON E-TECHNOLOGY, E-COMMERCE AND E-SERVICE, 2005, Hong Kong. **Proceedings...** Washington: IEEE Computer Society, 2005. p.353-358.

ZHANG, K.; STATMAN, R.; SHASHA, D. On the Editing Distance between Unordered Labeled Trees. **Information Processing Letters**, Amsterdan, v.42, n.3, p.133-139, May 1992.

ZHANG, H. The Optimality of Naive Bayes. In: INTERNATIONAL FLAIRS CONFERENCE, 17., 2004, Florida, USA. **Proceedings…** [S.l.:s.n.], 2004.

ZHU, Y.; WANG, H.; HU, Y. Super-Peer Based Lookup in Structured Peer-to-Peer Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS, PDCS, 16., 2003, Reno, Nevada. **Proceedings…** [S.l.:s.n.], 2003. p. 465-470.

# ANEXO A IMPLEMENTAÇÃO EM JAVA PARA O USO DO ALGORITMO MD5

```
   import java.io.FileInputStream;
   import java.io.IOException;
   import java.security.MessageDigest;
   import java.security.NoSuchAlgorithmException;

   public class MD5FileTest {

   public static void main(String[] args) throws Exception {
       System.out.println("Arquivo:" + args[0] + ", Hash: "+ getFileHash(args[0]));
   }

   public static String getFileHash(String fileName) throws NoSuchAlgorithmException,
IOException{
       MessageDigest md = MessageDigest.getInstance("MD5");
       FileInputStream in = new FileInputStream(fileName);
       byte[] buffer = new byte[8192];
       int length;
       while ((length = in.read(buffer)) != -1)
           md.update(buffer, 0, length);
       byte[] raw = md.digest();
       return StringUtil.hexEncode(raw);
   }
   }
```

Código em Java para a execução do algoritmo MD5 (*MD5FileTest.java*)

```
public class StringUtil {
    public static String hexEncode(byte[] aInput) {
       StringBuffer result = new StringBuffer();
       char[] digits = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c',
       'd', 'e', 'f' };
       for (int idx = 0; idx < aInput.length; ++idx) {
       byte b = aInput[idx];
       result.append(digits[(b & 0xf0) >> 4]);
       result.append(digits[b & 0x0f]);
       }
       return result.toString();
    }
}
```

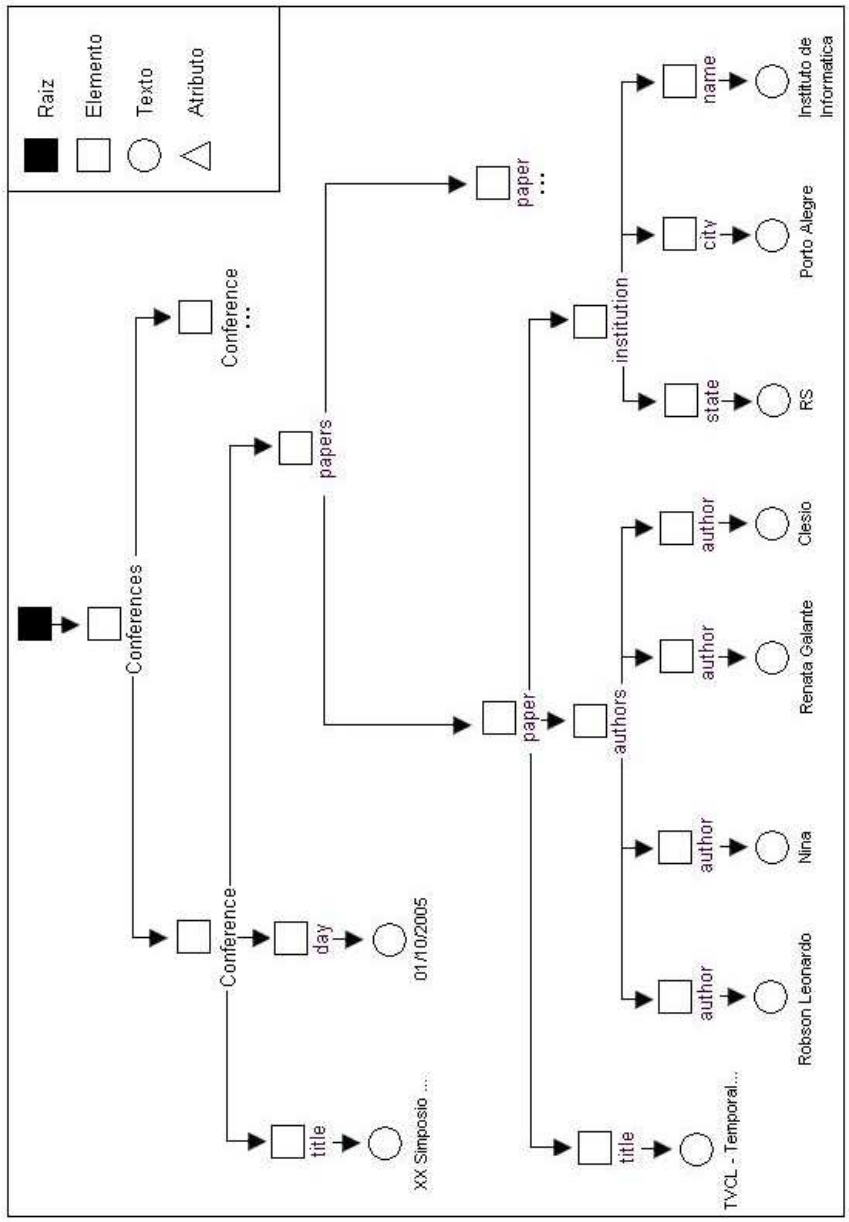
Código em Java para a conversão do resultado *hash* para formato hexadecimal (*StringUtil.java*)

# ANEXO B ARQUIVO XML EXEMPLO

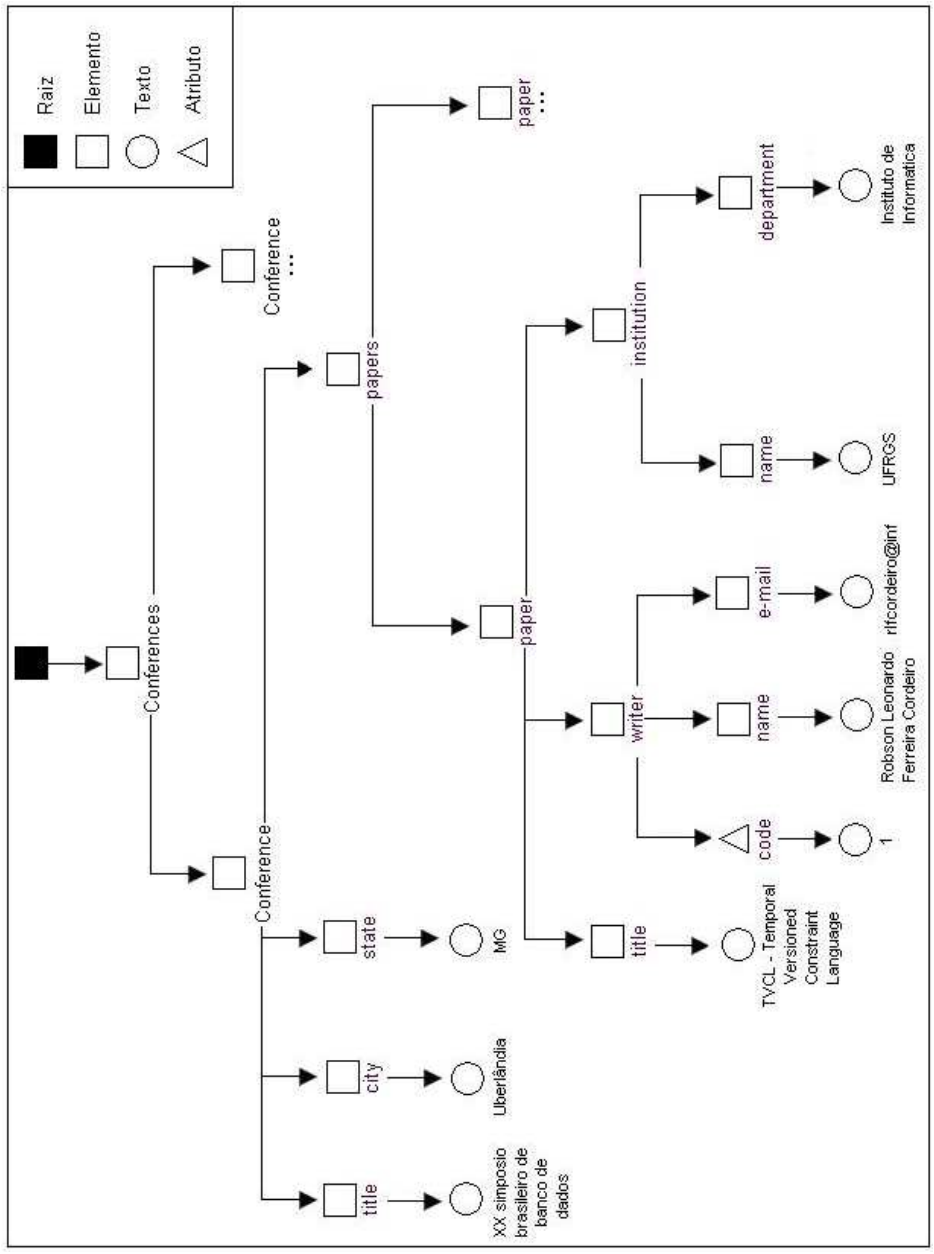
```
<?xml version="1.0" encoding="UTF-8"?>
<resume>
  <header>
    <name>
      <firstname>Jo</firstname>
      <surname>Doe</surname>
    </name>
    <!-- This person is using the old, but compatible US-style
    address. -->
    <address>
      <street>123 Elm #456</street>
      <city>Garbonzoville</city>
      <state>NX</state>
      <zip>99999-9999</zip>
    </address>
    <contact>
      <phone>555.555.5555</phone>
      <email>doe@doe.doe</email>
      <url>http://doe.com/~doe/</url>
    </contact>
  </header>

  <objective>
    <para>I really, really, really, <emphasis>really</emphasis> want a
      job.
    </para>
  </objective>

  <history>
    <job>
      <jobtitle>Senior Toilet Cleaner</jobtitle>
      <employer>HM Secret Service</employer>
      <period>
<from>
  <date>
    <month>August</month>
    <year>1943</year>
  </date>
</from>
<to>
  <present/>
</to>
      </period>
      <description>
<para>Cleaned out public lavoratories using my toothbrush.
  Assissted with enterprise resource planning.
</para>
      </description>
    </job>

    <job>
```

```
      <jobtitle>Junior Bedpan Cleaner</jobtitle>
      <employer>Framingham Palace</employer>
      <period>
<from>
  <date>
    <month>October</month>
    <year>1633</year>
  </date>
</from>
<to>
  <date>
    <month>October</month>
    <year>1634</year>
  </date>
</to>
      </period>
      <description>
<para>Analyzed bedpan cleanliness strategies for royal
  family.  Learned how to make souffles.
</para>
      </description>
    </job>
  </history>

  <academics>
    <degrees>
      <degree>
<level>BA</level>
<major>Renaissance Bedpan Design</major>
<date>
  <month>February</month>
  <year>1631</year>
</date>
<institution>Fishbaum del Schlozberg</institution>
<annotation>
  Graduated with lowest honors.
</annotation>
      </degree>
    </degrees>
  </academics>

  <skillarea>
    <title>Special Skills</title>
    <skillset>
      <skill>Speak multiple European languages with no discernable
        accent
      </skill>
      <skill>Able to predict earthquakes</skill>
    </skillset>
  </skillarea>

  <copyright>
    <year>2009</year>
    <legalnotice>
      <para>So there!</para>
    </legalnotice>
  </copyright>

</resume>
```

# ANEXO C DOCUMENTO DE ENTRADA A



# ANEXO D DOCUMENTO DE ENTRADA B



# ANEXO E DOCUMENTO DE ENTRADA C