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

ALVARISTO BERNARDES DO AMARAL PADILHA

Suporte a argumentos de consulta vagos através da linguagem XPath

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Carlos Alberto Heuser
Orientador

Porto Alegre, julho de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Padilha, Alvaristo Bernardes do Amaral

Suporte a argumentos de consulta vagos através da linguagem XPath / Alvaristo Bernardes do Amaral Padilha. – Porto Alegre: PPGC da UFRGS, 2005.

61 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientador: Carlos Alberto Heuser.

1. XML. 2. XPath. 3. Busca por similaridade. 4. Busca por palavra chave. I. Heuser, Carlos Alberto. II. Título.

“A vida pertence a Deus, pois a atividade da mente é vida e Ele é essa atividade. A pura auto-atividade da razão é a mais abençoada e eterna vida de Deus. Dizemos que Deus vive, eterno e perfeito, e que a vida contínua e eterna é de Deus, pois Deus é a vida eterna.”

— ARISTÓTELES

AGRADECIMENTOS

Gostaria de agradecer, primeiramente, a Deus sobre todas as coisas, pois sem Ele, nada é possível e eu acredito, do fundo do meu coração, que se eu estou concluindo este mestrado, é porque Ele propiciou o meu caminho até aqui. Em segundo lugar, eu gostaria de agradecer ao meu orientador o Prof. Dr. Carlos Alberto Heuser, pela confiança, pelo tempo e pelos ensinamentos a mim concedidos durante esses dois anos.

Sou muito agradecido também ao pessoal da sala 215, em especial a Carina Dorneles, que muito colaborou na execução desse trabalho. Ao Eduardo Kroth, a Raquel Stasiu ao Ségio Mergen e ao Andrei Lima, pelo apoio, solidariedade e a troca de idéias que tivemos no desenrolar desta pós. Obrigado pessoal!

Por outro prisma, devo agradecer a minha esposa Carla Lisiane Utzig do Amaral Padilha, que sempre compreendeu e me estimulou além dos pequenos sacrifícios que realizou para que eu concluisse a minha jornada. Aos meus pais, por todo amor e formação dada e aos meus sogros pela torcida constante.

Por último gostaria de agradecer ao Instituto de Informática pela oportunidade ímpar que ofereceram, juntamente como uma estrutura de altíssima qualidade e ao seu quadro de funcionários, sempre prestativos, bem como ao corpo doscente que é extremamente qualificado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

XML	Extensible Markup Language
GUI	Graphical User Interface
HTML	Hipertext Markup Language
IR	Information Retrieval
W3C	World Wide Web Consortium
XPath	XML Path Language
XQL	XML query language
DOM	Document Object Model
Jaxen	Java XPath Engine
SAX	Simple API for XML
SGBD	Sistema Gerenciador de Banco de Dados
ed	Edit distance
esp	Espaço
API	Aplication Programming Interface
IDF	Inverse Document Frequency
TFIDF	Term Frequency x Inverse Document Frequency
MAV	Metrics for Atomic Values
MCV	Metrics for Complex Values

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Abordagens clássicas de linguagens de consultas para bancos de dados possuem certas restrições ao serem usadas, diretamente, por aplicações que acessam dados cujo conteúdo não é completamente conhecido pelo usuário. Essas restrições geram um cenário onde argumentos de consultas, especificados com operadores boleanos, podem retornar resultados vazios. Desse modo, o usuário é forçado a refazer suas consultas até que os argumentos usados estejam idênticos aos dados armazenados no banco de dados.

Em bases XML, este problema é reforçado pela heterogeneidade das formas em que a informação encontra-se armazenada em diferentes lugares. Como solução, uma alternativa seria o uso de funções de similaridade na substituição de operadores boleanos, a fim de que o usuário obtenha resultados aproximados para a consulta especificada. Neste trabalho é apresentada uma proposta para suporte a argumentos de consulta vagos através da extensão da linguagem XPath. Para isso, são utilizadas expressões XPath que utilizam novas funções, as quais são, diretamente, adicionadas ao processador da linguagem de consulta. Além disso, é apresentada uma breve descrição das métricas de similaridade utilizadas para a criação das funções.

As funções que foram adicionadas a um processador XPath possuem uma ligação muito estreita com as métricas utilizadas. Como as métricas, as funções trabalham com valores simples (elementos atômicos) e compostos (elementos complexos). As funções que trabalham com elementos atômicos podem ser classificadas tanto pelo tipo de dado que será analisado, como pelo tipo de análise que será feita. As funções para elementos complexos comparam conjuntos de elementos atômicos de acordo com a forma do agrupamento (conjunto, lista ou tupla).

Palavras-chave: XML, XPath, busca por similaridade, busca por palavra chave.

TITLE: Suporting Vague Query Arguments Using XPath

ABSTRACT

The classical approaches for querying databases has some restrictions when applied to applications accessing data whose content the users are unaware of. This problem generates a scenario where queries having equality operators can lead to empty results. Thus the user is forced to retry queries until their arguments match data on the database.

In XML bases, this issue is reinforced by the heterogeneity of data types that can be found in different places. An alternative is to use similarity functions instead of boolean operators in order to allow the users to get approximate answers arranged in a ranking of results. In this work is presented an extension to XPath language for supporting vague query arguments. The extension consists on a set of similarity functions, added to the XPath engine, that can be evoked by the user when constructing a query. Furthermore, a brief description of the similarity metrics used for the functions is presented.

The functions, that where added to an XPath engine, have a close relation to the metrics used. Like the metrics, the functions works with simple values (atomic elements) and complex values (complex elements). The functions which work with atomic elements can be classified by the data type that will be analyzed as well as the kind of parse that will be done. The functions for complex elements match sets of atomic elements in agreement with the way they are put together (set, list or tuple).

Keywords: XML, XPath, similarity search, key-word search.

1 INTRODUÇÃO

Com a larga utilização de XML como fonte de dados para troca de informações em ambientes heterogêneos, faz-se premente a utilização de linguagens de consulta que facilitem a manipulação dos dados. As linguagens clássicas para manipulação de dados foram desenvolvidas para dar suporte a consultas com argumentos de busca que utilizem o-peradores booleanos, ou seja, objetos são selecionados, ou não, de acordo com o predicado da consulta. Na forma como tais linguagens são definidas atualmente, o usuário deve especificar consultas cujos valores das condições de busca correspondam exatamente aos valores armazenados no banco de dados, caso contrário, o retorno a essas consultas será vazio. Dessa forma, torna-se um desafio para certas aplicações fornecerem suporte a consultas em que o usuário tem dúvidas com relação à forma na qual a informação se encontra na base de dados. Esse desafio torna-se ainda maior em bancos de dados disponíveis através da Web.

Um bom exemplo, envolvendo a Web, são aplicações de integração de dados de diferentes fontes onde são encontradas diferentes representações do mesmo objeto do mundo real. Em um sistema de integração de citações bibliográficas, como o Citeseer (http://citeseer.ist.psu.edu/cis), variações de nome para a mesma conferência são bastante comuns, tais como "Simpósio Brasileiro de Banco de Dados", "Simpósio Brasileiro de BD" ou "SBBD". Esse tipo de aplicação deveria recuperar como resultado todas as representações de um determinado objeto, independente do argumento de consulta utilizado. Além disso, é possível encontrar múltiplas convenções em atributos, como nome e endereço, onde a ordem das palavras pode estar invertida, dependendo da base a ser pesquisada.

Seguindo a mesma linha de raciocínio, poderiam existir objetos inteiros em documentos XML distintos, cuja única diferença é a forma com a qual os valores estão armazenados. Da mesma forma, o usuário pode desejar conhecer uma determinada informação fornecendo uma lista ou conjunto de valores. Por exemplo, um usuário poderia querer saber quais artigos foram escritos pelos autores Serge Abiteboul e Dan Suciu. Se a ordem em que os autores aparecem é importante, então os argumentos da consulta formam uma lista senão, um conjunto. É importante notar que, além de utilizar dois argumentos, a consulta exige que esses argumentos encontrem-se agrupados (lista ou conjunto), não obstante o fato de que eles podem ter sido informados com erros de digitação ou podem estar de forma abreviada na base.

Existem casos em que a cadeia de caracteres, digitada na formulação da consulta, seja um padrão de fato, ou seja, não contenha nenhum tipo de erro. O erro pode estar na base a ser pesquisada. Diversos tipos de erros podem ser encontrados como, por exemplo, erros de datilografia, fonéticos ou na transmissão de dados. A ênfase está em consultar dados semi-estruturados, ou seja, melhorar as consultas sobre documentos XML centra-

dos nos dados e não no documento. Algumas propostas procuram casar os métodos de consulta tradicionais em SGBD com buscas por similaridade como em (MOTRO, 1988; GRAVANO, 2001a), porém não existe proposta abrangente que estenda uma linguagem de consulta tradicional para XML com comparações por similaridade. Além disso, pode haver a necessidade de localizar-se um registro inteiro ou uma tupla. Como isso poderia ser feito? Neste trabalho serão mostrados métodos de consulta por similaridade que casam elementos atômicos com complexos (tuplas, listas, etc).

A adoção de funções de similaridade em substituição aos operadores booleanos pode ser considerada uma solução para o problema. Usando essa abordagem é possível que as linguagens forneçam suporte a argumentos de consultas cujos valores são aproximados àquilo que o usuário de fato deseja, ou ainda que o argumento corresponda a uma das formas dentre as diversas representações possíveis armazenadas no banco de dados. Com o uso de funções de similaridade, é possível gerar um escore, que indica quão semelhante é cada objeto na base em relação ao argumento de consulta, e retornar ao usuário uma classificação ordenada (*ranking*) com as respostas mais próximas à consulta solicitada.

Neste trabalho, é proposta uma extensão da linguagem XPath (W3C, Acesso em: 23 abr. 2005), a qual é utilizada para localizar elementos em um documento XML. Mais detalhes sobre essa linguagem podem ser vistos na seção 3.1. Essa extensão foi batizada com o nome SimXPath (Similaridade em XPath), que permite o uso de funções de similaridade em expressões de caminho. Tais funções são definidas com base em algumas das métricas de similaridade para elementos XML previamente definidas em (DORNELES et al., 2004). Essas métricas se aplicam tanto a elementos com valores atômicos, quanto a elementos com valores complexos.

As funções para elementos atômicos citadas comparam alguns tipos básicos de dados (cadeia de caracteres, data e número) através de métricas de similaridade, podendo haver mais de uma para cada tipo. As funções para elementos complexos consideram o tipo de estrutura em que os elementos atômicos podem ser encontrados. Através do casamento da estrutura contida na consulta, a qual pode ser uma lista, um conjunto ou uma tupla, é feita a comparação dos elementos atômicos, contidos nestas estruturas, por meio das métricas de similaridade atômicas. Para listas e conjuntos, não só os tipos de dados devem ser o mesmo, como também o tipo do elemento, enquanto que, para tuplas, ambas as restrições não são necessárias.

A intenção da proposta aqui descrita não é modificar a sintaxe da XPath, pois se deseja utilizar o processador já existente. Dessa forma, a proposta consta da extensão do conjunto de funções já disponíveis na linguagem. O presente trabalho apresenta, de forma geral, duas contribuições:

- suporte a consultas com condições de busca aproximada, através da adição de novas funções para uma linguagem;
- definição de uma extensão, sem alteração da sintaxe, para uma linguagem padrão já adotada pela comunidade (XPath).

Apesar da XPath não ser uma linguagem de consulta completa, ela foi escolhida por sua simplicidade e por ser utilizada como núcleo de outras linguagens de consulta mais completas, como XQuery. Além disso, a linguagem XPath, com seu poder de expressão, consegue resolver a maior parte das consultas com pouco esforço. A intenção deste trabalho é estender o seu conjunto de funções, já existentes, com as funções de similaridade.

Este trabalho está organizado como segue. O capítulo 2 apresenta os trabalhos estudados como base para a elaboração desta dissertação. Através desses trabalhos, é apresen-

tada uma visão geral das métricas de similaridade existentes, bem como de mecanismos de busca que, de alguma forma, utilizam similaridade. O capítulo 3 descreve como um processador de consulta da linguagem XPath foi estendido com as funções de similaridade. No capítulo 4, são apresentadas as funções adicionadas a linguagem XPath, juntamente com exemplos de expressões SimXPath para auxiliar na sua compreensão. O capítulo 5 apresenta o protótipo desenvolvido e formas de utilizá-lo. O capítulo 6 conclui e aponta os trabalhos futuros.

2 TRABALHOS RELACIONADOS

Existem várias propostas que apresentam um aperfeiçoamento aos mecanismos de busca ou linguagens de consultas de modo a trazer resultados mais precisos. Alguns trabalhos possuem enfoque em melhorar ou apresentar novas métricas ao comparar, por similaridade, conjuntos de valores, sejam de caracteres ou numéricos, no aperfeiçoamento de consultas em SGBD. Outros propõem o aprimoramento dos mecanismos de busca em bases heterogênias com enfoque na Web. Nesses, além de métricas de similaridade, são utilizados mecanismos de busca por palavra-chave, geralmente utilizando, como base, documentos HTML ou XML. Não obstante, haverão alguns trabalhos que tratam das duas abordagens.

O estudo do primeiro grupo de trabalhos visou estabelecer as métricas de similaridade para elementos atômicos, ou seja, indivisíveis, de modo a aproveitar aquelas que estejam mais aprimoradas e se adequem melhor a este trabalho. A estratégia utilizada não foi estudar ostensivamente toda a literatura sobre o assunto, mesmo porque a quantidade de trabalhos relacionados é muito ampla. Porém foram estudados os principais artigos, dentro das métricas escolhidas, e também artigos que apresentavam um estado da arte como (NAVARRO, 2001a) e (LIMA, 2002).

Métricas de similaridade entre cadeias de caracteres podem ser utilizadas sob dois enfoques: i) na comparação de duas cadeias de caracteres, e; ii) na filtragem de textos. No segundo caso as métricas utilizam palavras representativas, geralmente retirada de um dicionário, ou basiam-se em mecanismos de busca por palavra-chave. Nesse caso, é necessário que o texto a ser filtrado, passe por um algum tipo de indexação, onde os termos mais representativos são selecionados e contados. Só depois desse procedimento é que são utilizadas métricas de filtragem.

É importante salientar que o foco está na pesquisa de métricas que se enquadram no primeiro caso. Isto ocorre porque a linguagem de consulta utilizada não realiza um pré-processamento do texto pois o tipo de base escolhida (repositórios de documentos XML) são normalmente pesquisado online além de, em alguns casos, sofrerem constantes atualizações.

2.1 Métricas de similaridade

A base das métricas de similaridade deste trabalho foi extraída de (DORNELES et al., 2004), que apresenta um amplo conjunto de métricas, tanto para elementos atômicos, como para complexos, comprovado por um conjunto de testes realizados, que demonstram a eficiência das métricas utilizadas. Algumas dessas métrica foram coletadas em trabalhos existentes na comunidade científica, como *Edit Distance* e *N-grams*, e outras em um trabalho de conclusão de curso, como *acronym*.

O relatório expõe as métricas para elementos atômicos de acordo com o tipo de dado e com a parte semântica da informação. Por exemplo, dentre as métricas para conjuntos de caracteres existe uma que trata da similaridade para acrônimos. Desse mesmo trabalho foram aproveitados os conceitos para elementos complexos (conjunto, lista e tupla). Porém, as consultas baseadas nesses elementos utilizavam um objeto de consulta (*query-object*), o qual não era apropriado para a extensão da linguagem proposta. As consultas com objetos complexos serão abordadas posteriormente neste trabalho.

Após escolher o conjunto de métricas para elementos atômicos nas quais se baseia este trabalho, foi feito um estudo para verificar a sua origem e o seu estado da arte. Sob esse prisma, (LIMA, 2002) faz um apanhado de diversas métricas. Não se pretende desenvolver todas as possibilidades de consulta por similaridade. Antes disso, será apresentado um estudo com várias métricas, de onde um conjunto bem genérico será extraído, para compor as funções que farão parte da nova linguagem.

2.1.1 Edit Distance

Em Navarro (2001), são abordadas algumas métricas de similaridade de cadeia de caracteres, sendo a maior ênfase dada na *edit distance*, trazendo uma visão geral sobre o assunto (*survey*), bem como um histórico bem detalhado da evolução da métrica. Em (NAVARRO, 2001b), Navarro apresenta um resumo de seu artigo anterior (NAVARRO, 2001a), focando as métricas que indexam o texto para ampliar a busca por similaridade, dividindo as existentes abordagens em duas dimensões: estrutura dos dados e métodos de busca.

A métrica *Edit distance* é uma das soluções mais completas e poderosas para medir a diferença entre duas cadeias de caracteres. Quando ela possui diversas variantes, como pesos diferentes para operações de inserção, alteração ou exclusão, ou para determinados caracteres, é chamada *general edit distance*. Caso contrário, se o peso de todas as operações possuem valor 1, ela é chamada *simple edit distance* ou apenas *edit distance* (ed). Algumas das variantes dessa função são *Hamming distance* (SANKOFF; KRUSKAL, 1983), *Longest Common Subsequence distance* (NEEDLEMAN; WUNSCH, 1970; APOSTOLICO; GUERRA, 1987) e *Episode distance* (DAS, 1997). Como essas funções se aplicam a casos específicos, não serão levadas em consideração neste trabalho, tendo em vista a abordagem genérica que está sendo aplicada. Desta forma, a função utilizada é a *Edit distance* ou Levenshtein (LEVENSHTEIN, 1965). Essa função se caracteriza por medir a distância entre duas cadeias de caracteres, contando a quantidade de operações de inserção, exclusão e substituição de caracteres que são necessários realizar sob a primeira cadeia de caracteres, para deixá-la igual segunda. Cada operação tem peso 1, e os caracteres não influenciam no peso das operações como acontece em algumas de suas variantes. Considere duas cadeias de caracteres *s* e *t* e s_i o *i*-ésimo caracter, onde *s*=$s_1 s_2 ... s_{|s|}$. Para computar a distância entre duas cadeias de caracteres ($ed(s,t)$), é necessário, primeiramente, preencher uma matriz C, cujas dimensões serão de acordo com o tamanho das cadeias de caracteres. A tabela 2.1 ilustra a função *ed*("perder", "perdoar").

A quantidade de caracteres em *s* ($mod\ s$) indicará o número de linhas e a quantidade de caracteres em *t* ($mod\ t$), o número de colunas. A métrica baseada em programação dinâmica utiliza a seguinte fórmula para preencher esta matriz:

$$C'_i = min(C_{i-1,j-1} + \delta(s_i, t_i), C_{i-1,j} + 1, C_{i,j-1} + 1)$$

		p	**e**	**r**	**d**	**o**	**a**	**r**
	0	1	2	3	4	5	6	7
p	1	**0**	1	2	3	4	5	6
e	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
d	4	3	2	1	**0**	1	2	3
e	5	4	3	2	1	**1**	**2**	3
r	6	5	4	3	2	2	2	**2**

Tabela 2.1: Tabela gerada pela comparação entre "perder" e "perdoar" através da métrica *Edit distance*

onde $\delta(s_i, t_i)$ será 0 se $s_i = t_i$ caso contrário será 1. Na última célula da matriz teremos o valor da distância ($ed(s,t) = C_{|s|,|t|}$). Porém teremos $ed(s,t) >= 0$ e $ed(s,t) <= max(|s|,|t|)$, onde será igual a 0 quando s=t, caso contrário será maior. Para que o valor retornado esteja entre 0 e 1, ele deve ser normalizado, dividindo a distância pela maior cadeia de caracteres:

$$Sim = \frac{ed(s,t)}{max(|s|,|t|)}$$

Essa alternativa também é vista em (CHAUDHURI, 2003). Vários trabalhos utilizam essa métrica para realizar a comparação entre cadeias de caracteres. Em (FRENCH; POWELL; SCHULMAN, 1997) ela é utilizada para casar registros de referências bibliográficas de fontes heterogêneas de dados utilizando clusterização. Em (GRAVANO, 2001a), ela é utilizada para aumentar a capacidade de processamento de SGBD relacionais, na tentativa de localizar registros com cadeias de caracteres similares às informadas na consulta. Em (GRAVANO, 2001b), Gravano utiliza a mesma técnica de casamento de cadeias de ca-racteres (*Q-gram*), para realizar a junção (*join*) entre tabelas que possuam registros similares, enquanto em (JIN; LI; MEHROTRA, 2002), ela é usada para determinar se dois registros devem ser considerados uma duplicata.

Em (CHAUDHURI, 2003), a métrica *ed* é comparada com uma função chamada casamento de similaridade *fuzzy* (*fms*). O princípio dessa função é realizar o casamento de tuplas considerando o peso dos *tokens* através da função de peso *IDF* (BAEZA-YATES; RIBEIRO-NETO, 1999). Com isso, o autor pretende resolver os problemas onde a comparação entre cadeias de caracteres, com vários *tokens*, pode acarretar em um falso casamento. Essa função é proposta para ser utilizada em bancos de dados relacionais, principalmente no suporte a análise em *data warehouses*, realizando o trabalho de limpar registros que tratam do mesmo objeto, porém possuem pequenos erros de digitação, campos faltando, etc. Apesar de, na comparação direta entre *ed* e *fms*, o artigo ter mostrado a função *fms* ser mais eficiente, os autores utilizam a métrica *ed* para a comparação entre os tokens.

2.1.2 N-grams

A técnica de *n-grams* consiste basicamente em se dividir uma cadeia de caracteres em subcadeias de caracteres de tamanho n e compará-la com as subcadeias de caracteres de uma outra palavra a ser pesquisada (ULLMANN, 1977). Porém essa técnica foi proposta com as restrições de que as palavras não poderiam ter mais de 2 diferenças (inserção,

alteração ou exclusão de caracteres para transformar uma em outra) e que todas elas seriam comparadas a um dicionário com palavras de 6 caracteres. Mais adiante, em 1992, Ukkonen (UKKONEN, 1992) elabora uma técnica conhecida como *q-grams*, que visa localizar, com agilidade, cadeias de caracteres em grandes conjuntos de caracteres. Essa é uma das abordagens existentes para filtragem de textos e consiste em separar o texto em vários grânulos de tamanho q. Posteriormente, separa-se também uma palavra padrão nesses mesmos grânulos e então se procura por ocorrências dos grânulos dessa palavra no texto. A localização de m-q+1 grânulos do padrão no texto significa um casamento (match), onde m representa o tamanho da palavra, em caracteres.

Em (UKKONEN, 1992), o casamento dos grânulos da palavra padrão com o texto pode ocorrer fora de ordem. Caso algum casamento seja feito passa-se a procurar os grânulos ao redor, para tentar fazer uma identificação positiva. Em (SUTINEN; TARHIO, 1995) a busca é feita procurando-se localizar os grânulos em seqüência. Além disso é considerada a posição relativa em que eles se encontram dentro do texto, sendo que os grânulos não podem estar a uma distância *K* maior que a definida.

Embora a técnica de *n-grams* esteja envolvida em similaridade, neste trabalho ela está enquadrada juntamente com diversas outras, em técnicas de filtragem, cuja função é a de informar quando duas cadeias de caracteres não são similares através da comparação de seus grânulos. Desta forma ela é muito utilizada em conjunto com métricas de comparação por similaridade, como acontece em (GRAVANO, 2001b), onde, após ser aplicado um filtro sobre o texto e terem sido localizadas possíveis palavras candidatas, é aplicada a função *edit distance* sobre as mesmas para averiguar a sua similaridade.

Neste trabalho, essa abordagem foi empregada como está apresentada em (LIMA, 2002), com algumas especializações. Como prevê a técnica original (ULLMANN, 1977), as cadeias de caracteres são comparadas no intuito de verificar a sua similaridade, não sendo essa métrica utilizada para filtragem. A métrica ainda utiliza, além das duas cadeias de caracteres que serão comparadas, o tamanho do grânulo (*gram*), que é o valor de n, e o número de caracteres especiais que serão colocados no início e no fim das cadeias de caracteres, com a finalidade de melhorar a quantificação da similaridade.

Através de vários testes feitos, foi adotado para n o tamanho padrão 3. Quanto menor o tamanho de n, maior são os valores de similaridade encontrados. Se for usado um n=1, por exemplo, pode acontecer de a métrica informar que duas cadeias de caracteres são iguais (Sim=1.0), quando essa informação não é verdadeira. Isso acontece porque a função verificará a existência dos grânulos, que, nesse caso, são do tamanho de um caractere, da primeira cadeia na segunda cadeia de caracteres. Como a comparação não é posicional, basta que uma cadeia de caracteres possua o mesmo conjunto de caracteres da outra, para a métrica considerá-las idênticas.

Com n=3, esse comportamento não é observado, pois os caracteres não são comparados individualmente, mas sim em subconjuntos de 3. Para n=2, há o inconveniente de algumas cadeias de caracteres, que possuem apenas palavras fora de ordem, receberem Sim=1.0. Isso ocorre, porque o espaço que separa duas palavras em uma cadeia de caracteres irá fazer parte do grânulo e, com n=2, não haverá nenhum grânulo composto pela intersecção das duas palavras, o que não ocorre com n=3. Por exemplo, suponha-se que se deseje encontrar o seguinte texto nos nodos de um documento XML: "Porto Alegre, RS". Em determinado ponto do documento em questão, existe um nodo com a seguinte ocorrência: "RS- Porto Alegre". É importante salientar que a função remove todos os caracteres especiais como ponto, vírgula, traço, etc, considerando somente caracteres alfanuméricos com apenas um espaço de separação entre as palavras, quando for o caso.

Com n=2, seria criado o seguinte conjunto de grânulos (o caracter "_" significa espaço em branco) :

Porto Alegre, RS															
_p	po	or	rt	to	o_	_a	al	le	eg	gr	re	e_	_r	rs	s_
RS- Porto Alegre															
_r	rs	s_	_p	po	or	rt	to	o_	_a	al	le	eg	gr	re	e_

Tabela 2.2: Formação dos grânulos com n=2

Como se pode observar, todos os grânulos existentes na primeira cadeia de caracteres também se encontram na segunda, resultando em duas cadeias de caracteres idênticas, através do método *n-grams*. Embora se saiba, conceitualmente, que se trata do mesmo objeto, não é interessante adotar esta estratégia como padrão, pois em outras situações isso pode não corresponder à realidade. Se n=3 for considerado, será obtido o seguinte conjunto de grânulos:

Porto Alegre, RS														
_po	por	ort	rto	to_	o_a	_al	ale	leg	egr	gre	re_	**e_r**	_rs	rs_
RS- Porto Alegre														
_rs	rs_	**s_p**	_po	por	ort	rto	to_	o_a	_al	ale	leg	egr	gre	re_

Tabela 2.3: Formação dos grânulos com n=3

Nos grânulos acima, existem dois grânulos diferentes: i)"e_r", no primeiro conjunto e; ii)"s_p"no segundo conjunto de grânulos. Essa diferença é suficiente para retornar um alto grau de similaridade (Sim=0.9333) e, ao mesmo tempo, diferenciar as duas cadeias de caracteres.

O tipo de caractere especial, adotado para ser inserido no início e no final das cadeias de caracteres a serem comparadas, foi o espaço em branco, como pode ser visto em (LIMA, 2002). Embora outros textos apresentem outros tipos de caracteres como *#* e *$* como em (GRAVANO, 2001b), a inserção de espaços em branco coincide com possíveis espaços localizados dentro do texto, para separar as palavras. Como essa separação ocorre por, no máximo, um espaço, o valor padrão para essa variável é 1. Com isso, aumenta-se o número de grânulos iguais caso as palavras estejam apenas fora de ordem.

Consulta	**Base**	**0 esp**	**1 esp**	**2 esp**	**3 esp**
Porto Alegre, RS	RS- Porto Alegre	0.769	0.933	0.823	0.842
Los Angeles, California	California- Los Angeles	0.85	0.954	0.875	0.884
Serge Abiteboul	Abiteboul, Serge	0.769	0.933	0.823	0.842
Carlos Alberto Heuser	Heuser, Carlos Alberto	0.842	0.952	0.869	0.88

Tabela 2.4: Tabela de comparação dos valores gerados com a inserção de espaços através da técnica de *n-grams*

A tabela 2.4 mostra o resultado de algumas consultas efetuadas sobre uma base utilizando a métrica *n-grams*. Para essas consultas, foi considerado um valor de $n = 3$. A primeira observação é que os graus mais altos foram atingidos com o número de espaços igual a 1. A segunda observação é que, acima de 2 espaços, o grau começa a aumentar

após a queda natural de 1 para 2. Esse comportamento fica claro, pois quanto mais caracteres idênticos forem inseridos na mesma posição em ambas as cadeias de caracteres, maior será a similaridade entre elas.

2.1.3 Acronym

Para tentar solucionar a comparação entre duas cadeias de caracteres, onde uma delas pode estar de forma abreviada, foram pesquisadas métricas que solucionassem o problema de acrônimos. Uma solução para esse problema foi encontrada em (LIMA, 2002), onde a métrica proposta, primeiramente, executa uma varredura em cada cadeia de caracteres eliminando caracteres, que não sejam alfanuméricos. Posteriormente, a(s) cadeia(s) de caracteres que não possuem abreviaturas são reduzidas a seus acrônimos, ou seja, ficam apenas as iniciais. Finalmente, é aplicada a métrica de *Guth* sobre as cadeias de caracteres, e então um escore de similaridade é retornado. Esta última métrica será detalhada mais adiante.

Para formar os acrônimos, são levadas em consideração as letras maiúsculas que iniciam as palavras, bem como palavras maiores que dois caracteres (caso a cadeia de caracteres tenha mais de uma palavra). Com esta última medida, evita-se a ocorrência das preposições mais comuns, em nomes próprios, e de artigos na composição de uma abreviatura. Caso a cadeia de caracteres seja composta por apenas uma palavra, a métrica a considerará um acrônimo.

Essa métrica também foi mencionada em (DORNELES et al., 2004), dentro do escopo das métricas para elementos atômicos.

2.1.4 Guth

A métrica criada por Gloria Guth (GUTH, 1976) compara duas cadeias de caracteres, letra a letra, procurando um casamento. Quando duas letras, na mesma posição, são diferentes, ela procura por essa letra em outras posições. Esse algoritmo é ideal para comparar nomes próprios e surgiu como alternativa para métricas que comparavam duas palavras levando em consideração o seu conteúdo fonético. No entanto, essa métrica retorna o número de casamentos feitos entre as duas cadeias de caracteres. Para que ele servisse ao propósito deste trabalho, o seu valor de retorno passou a ser a proporção entre a quantidade de casamentos pelo tamanho do maior acrônimo (normalização).

2.1.5 Datas

Da mesma maneira que a similaridade para acrônimos, praticamente não existem trabalhos na tentativa de solucionar o problema de similaridade entre datas. Uma das poucas alternativas encontradas em (LIMA, 2002) verifica a similaridade entre datas, mesmo estando em diferentes formatos. Existem algumas variações, *yearSim*, *monthSim* e *daySim*, que usam a mesma métrica de datas completas, entretanto aplicadas respectivamente a ano, mês e dia somente.

A métrica que realiza a comparação entre as datas, primeiramente, faz o reconhecimento da data, que se encontra na base e pode estar em formatos diferentes (figura 2.1), transformando-a em um padrão reconhecido pela linguagem. É importante ressaltar que a data no formato "mm/dd/aaaa" só será reconhecida nesta forma se o dia for maior que 12, senão o padrão é "dd/mm/aaaa". Posteriormente, ambas as datas, juntamente com a data atual, são passadas para o calendário Juliano, através de um algoritmo que pode ser visto em (BAUM, Acesso em: 30 jun. 2005). Dessa forma todas as datas que serão utilizadas

<data>2005/3/14</data>
<data>14 de janeiro de 2005</data>
<data>14 janeiro, 2005</data>
<data>january 14, 2005</data>
<data>4/14/2005</data>
<data>12-28-2005</data>
<data>2000.12.10</data>
<data>31/07/2005</data>

Figura 2.1: Formatos reconhecidos de datas

na métrica de similaridade para datas possuem um valor numérico linear.

Considerando que *d1* e *d2* são duas datas a serem comparadas, e *dBase*, a data atual a fórmula que indicará a similaridade entre *d1* e *d2* é a seguinte:

$$Sim(d1, d2) = \frac{dBase - menor(d1, d2)}{dBase - maior(d1, d2)}$$

A grande vantagem dessta métrica é reconhecer uma data entre os diversos formatos, nos quais ela pode ser encontrada em bases XML heterogêneas com a adição de duas formas importantes, onde é levada em consideração uma estrutura complexa para armazenamento da data. Na figura 2.2, ambos os objetos são reconhecidos pela métrica como datas válidas, embora o nome dos nodos de um dos objetos esteja em inglês. Embora não seja a finalidade deste trabalho, fica aberta a possibilidade de fazer-se este reconhecimento da estrutura, utilizando-se outros meios como *Thesaurus* ou casamento de estruturas. A definição, com experimentos, pode ser encontrada em (LIMA, 2002).

```
<data>                          <date>
  <dia>14</dia>                   <day>14</day>
  <mes>janeiro</mes>              <month>January</month>
  <ano>2005</ano>                 <year>2005</year>
</data>                         </date>
```

Figura 2.2: Formatos complexos reconhecidos de datas

2.1.6 Números

Uma função de similaridade para números foi encontrada apenas em (DORNELES et al., 2004). Essa função calcula a similaridade entre dois números (positivos ou negativos), dividindo a sua diferença absoluta pelo $average$, que é a média dos números encontrados dentro do conjunto de números em questão. A fórmula que calcula a similaridade para dois números $n1$, $n2$ é definida da seguinte forma:

$$sim(n1, n2) = \frac{abs(max(n1, n2) - min(n1, n2))}{average}$$

Caso um número positivo esteja sendo comparado com um número negativo, o $average$ é multiplicado por dois, pois os números encontram-se em domínios diferentes.

2.2 Outras métricas

As funções até aqui descritas foram as utilizadas neste trabalho, porém existem diversas outras que foram estudadas, mas não foram aproveitadas. Este trabalho não se limita apenas às funções apresentadas, sendo essas apenas um conjunto inicial proposto com a finalidade de abranger o maior número possível de casos de consultas imprecisas. Novas métricas de similaridade podem ser adicionadas, porém isso ficará para trabalhos futuros.

2.2.0.1 Métricas de ligação de registros

Métricas de ligação de registros visam comparar um grande conjunto de dados com outro em busca de registros que façam parte de ambos. Para isso são considerados os pesos dos campos dos registros (ex. CPF possui um peso maior do que idade), bem como a possibilidade de existirem pequenos erros tipográficos.

Monge e Elkan, em 96 (MONGE; ELKAN, 1996), propuseram um método híbrido de consulta, que casa a pesquisa em registros com uma função de distância. Em seu esquema, duas cadeias de caracteres longas, *S* e *T*, são comparadas recursivamente. Primeiramente, elas são quebradas em subcadeias de caracteres $S = S_1...S_{|S|}$ e $T = T_1...T_{|T|}$ e cada subcadeia de caracteres de *S* é comparada com cada subcadeia de caracteres de *T*. A função de similaridade fica da seguinte forma:

$$sim(S,T) = \frac{1}{|S|}\sum_{i=1}^{|S|}\max_{j=1}^{|T|} sim(S_i,T_j)$$

Onde sim é uma função recursiva que possui uma complexidade quadrática. Segundo Elkan, essa função resolve o problema de abreviações e acrônimos, porém, em seu artigo, não ficou claro como isto acontece.

A métrica de Jaro-Winkler (WINKLER, 1999) é um aprimoramento da métrica de Jaro (JARO, 1995). Em seu trabalho, Jaro propôs um método de ligação probabilístico, com a finalidade de comparar dois arquivos para localizar registros iguais ou com alto grau de similaridade, e foi elaborado para realizar o casamento dos arquivos da saúde pública dos Estados Unidos. Resumidamente, para duas cadeias de caracteres *s* e *t*, considere $s_{i,j}$ o conjunto de caracteres em *s* que também estão em comum em *t* e considere $t_{i,j}$ os caracteres em *t* os quais estão em comum em *s*. Um caractere *a* em *s* está em comum em *t* quando ele está próximo à posição em que se encontra em *t*.

Considere $T_{s_{i,j},t_{i,j}}$ a medida do número de transposições de caracteres em $s_{i,j}$ relativos a $t_{i,j}$. A métrica de similaridade de Jaro para *s* e *t* é:

$$Jaro(s,t) = \frac{1}{3} * (\frac{|s_{i,j}|}{|s|} + \frac{|t_{i,j}|}{|t|} + \frac{|s_{i,j}| - T_{s_{i,j},t_{i,j}}}{2|s_{i,j}|})$$

Winkler propôs uma extensão à métrica de Jaro, inserindo *P* como o tamanho do maior prefixo comum entre as cadeias de caracteres *x* e *y*, ficando da seguinte forma:

$$JaroWinkler(s,t) = Jaro(s,t) + \frac{P}{10} * (1 - Jaro(s,t))$$

2.2.0.2 *Métricas de casamento de coeficientes*

Algumas métricas são baseadas em coleções de palavras (ou *tokens*). Essas técnicas se destacam por serem aplicadas em cadeias de caracteres formadas por várias subcadeias de caracteres. Elas baseiam-se na distância vetorial das palavras e são muito utilizadas na comunidade de recuperação de informação. A métrica mais simples é o coeficiente de casamento (*Matching Coefficient*). Essa métrica baseia-se em contar o número de palavras em comum, sem levar em consideração o tamanho das cadeias de caracteres. Outra métrica que se destaca é a similaridade Jaccard, cuja função entre os conjuntos de palavras *S* e *T* se resume em:

$$Jaccard(S,T) = \frac{|S \bigcap T|}{|S \bigcup T|}$$

Essa métrica utiliza o conjunto de palavras das instâncias comparadas para testar a similaridade. Diferentemente da similaridade Jaccard, a similaridade Dice penaliza um número menor de entradas compartilhadas, embora o número penalizado por Jaccard seja pequeno. Essa métrica retorna um valor entre 0 e 1, onde a similaridade é definida pelo dobro do número de palavras em comum, dividido pelo número total de termos de ambos os conjuntos.

$$Dice(S,T) = 2 * \frac{|S \bigcap T|}{|S| + |T|}$$

Se o resultado for 1, significa que os vetores são iguais, ou seja, trata-se do mesmo conjunto de palavras.

A métrica de similaridade Cosine é similar a Dice e é largamente utilizada na comunidade cientifica (COHEN; RAVIKUMAR; FIENBERG, 2003). Ela se baseia no tamanho do ângulo formado entre os vetores utilizados na comparação. Quanto menor o ângulo, maior a similaridade entre as palavras. O tamanho dos vetores pode ser reduzido com a retirada das *stop words* (palavras que não identificam um documento como artigos, preposições, etc). A sua fórmula, considerando o ângulo entre dois vetores, é:

$$cos\theta(S,T) = \frac{|S \bigcap T|}{|S|^{1/2} * |T|^{1/2}}$$

A similaridade Overlap também é bastante similar a Dice e tem como objetivo verificar se um conjunto de palavras é subconjunto do outro. Caso positivo, o método considera que foi realizado um casamento completo, ou seja $S = T$.

$$Overlap(S,T) = \frac{|S \bigcap T|}{min|S|, |T|}$$

De acordo com (RIJSBEGEN, 1979) essas são a métricas mais comumente utilizadas. Aqui elas foram apresentadas sem tecer considerações sobre extensões e especializações como por exemplo a similaridade Jaccard, que apareceu, inicialmente, em (JACCARD, 1912) e, posteriormente, foi aperfeiçoada por vários outros trabalhos.

2.2.0.3 Métricas linguísticas

As métricas linguísticas pretendem resolver erros da compreensão humana das palavras, tanto no modo como elas são escritas (*Smith* ou *Smyth*), como na sua formação fonética (*Sinclair* ou *St Clair*) sendo muito utilizadas no auxílio à identificação de nomes próprios. Aqui serão descritas, brevemente, algumas destas métricas. Maiores detalhes podem ser encontrados em (LAIT; RANDELL, 1993) e (LIMA, 2002).

A métrica Russel Soundex foi desenvolvida para resolver problemas fonéticos no casamento de nomes da língua inglesa. Ela converte cada nome em um código de quatro caracteres, sendo o primeiro uma letra e os outros três, números. O código é utilizado para identificar nomes equivalentes. Esse método possui adaptações para outras línguas como francês e alemão porém não foram encontradas adaptações para o português.

O método de codificação Metaphone baseia-se em casar palavras parecidas e utilizando regras comuns da pronuncia da língua inglesa. Para isso ele ignora as vogais, a não ser que elas sejam a primeira letra, e reduzem o número de consoantes a 16 (B, X, S, K, J, T, F, H, M, N, P, R, W, Y, 0), sendo que *0* substitui o som de "th"e *X* o som de "sh".

O programa FONEM foi criado para solucionar problemas de ortografia em nomes franceses e possui 64 regras. Dessas, 53 são regras de transcrição e 11 são regras auxiliares, que estabelecem a ordem quando mais de uma regra pode ser aplicada a um mesmo nome. O algoritmo pretende reduzir o número de variações ortográficas em 18%.

2.3 Mecanismos de consulta

Nesta seção são abordados trabalhos que apresentam ferramentas que auxiliam na busca por similaridade/proximidade. As abordagens tratam, basicamente, de bancos de dados tradicionais e de fontes de dados heterogêneas. Outro enfoque mostrado nas pesquisas são os tipos de busca feito nessas bases que podem ser por similaridade/aproximação de palavras/textos, por buscas por palavras-chave ou trabalhos que utilizam ambas as abordagens, entre outros.

Motro propôs, em 1988 (MOTRO, 1988), uma interface para bancos de dados relacionais que permitia a execução de consultas com conteúdo vago. A esse sistema ele deu o nome de VAGUE e ele funciona com um comparador de seleção chamado *similar-to*. Esse comparador determina a similaridade entre os dados determinando a distância entre os mesmos por meio de um modelo vetorial. Porém o artigo não especifica que tipos de métricas ele utiliza para obter a similaridade entre os atributos, dando um maior foco em como essa ferramenta foi implementada no topo de um SGBD e como funciona o processamento das consultas.

Yianilos desenvolveu, em 1997, uma ferramenta que realiza a comparação entre duas cadeias de caracteres, uma proveniente dos argumentos de uma consulta e outra de uma base de dados. O núcleo de sua ferramenta é uma função baseada em casamentos bipartidos. Essa função devolve uma indicação numérica da similaridade e funciona para os casos em que uma das cadeias de caracteres possui erros tipográficos, quando o texto está incompleto ou fora de ordem. Essa ferramenta tornou-se comercial e, em (YIANILOS; KANZELBERGER, 1997), encontram-se detalhes de como ela pode ser utilizada na forma de uma API para aumentar o poder de busca das consultas.

Bollacker, Lawrence e Giles (GILES; BOLLACKER; LAWRENCE, 1998; LAWRENCE; GILES; BOLLACKER, 1999) criaram um protótipo de biblioteca digital chamada *CiteSeer*. O princípio de funcionamento dessa biblioteca é a indexação automática de citações,

que é feita através da localização automática de artigos, extração e identificação de citações, que ocorrem em diferentes formatos e identificação do contexto das citações no interior dos artigos.

A identificação do contexto é utilizada para simular o conceito que os usuários possuem sobre similaridade de documentos. Para localizar documentos semelhantes são usadas medidas de distância semântica no corpo do texto para averiguar sua relação. Uma dessas medidas é a *edit distance* em conjunto com o calculo da frequência das palavras. Com um mecanismo de busca por palavra-chave, o protótipo pode localizar artigos que se enquadram com as palavras da consulta. Após o usuário selecionar um dos artigos, o *CiteSeer* pode localizar outros artigos relacionados a ele através de informações de citações comuns ou similaridade de palavras. Para a procura por similaridade, são utilizadas comparações de distância entre os cabeçalhos dos artigos e vetores de palavras, utilizando o esquema TFxIDF para localizar artigos com palavras similares.

Goldman (GOLDMAN, 1998), em 1998, apresentou um *framework* para tentar resolver o problema da busca por proximidade em banco de dados arbitrários. Na sua abordagem, a base de dados era considerada um conjunto de objetos e baseava-se na relevância da proximidade destes objetos. O modelo desenvolvido reconhece a base como um grafo onde a informação está nos vértices e as arestas são os relacionamentos. O modelo de proximidade foi implementado sobre o sistema de banco de dados Lore que utiliza um modelo de dados que facilita a troca de dados heterogêneos. Por meio de palavras-chaves *Find* e *Near*, o sistema realiza a busca em objetos próximos a um objeto considerado como relevante, considerando a distância das arestas.

Alguns pesquisadores dedicaram seus esforços na tentativa de facilitar as consultas em documentos XML, realizando buscas e trazendo resultados, onde a estrutura dos objetos selecionados é similar a estrutura da consulta, tendo em vista que ambos podem ser facilmente transformados em árvores (figura 2.3). A motivação dessa abordagem está no fato de que, em uma consulta XML, tanto o conteúdo, como a estrutra, devem ser casados com os elementos do documento a ser pesquisado. Por exemplo, para a consulta abaixo retirada de (SCHLIEDER, 2001):

```
/catalog/cd[composer="Rachmaninov" and title="Piano concerto"]
```

As expressões `/catalog/cd/composer` e `/catalog/cd/title` correspondem ao casamento estrutural, enquanto `[composer="Rachmaninov"]` e `[title="Piano concerto"]` correspondem ao casamento de conteúdo.

Nesse sentido, Baeza-Yates apresentou uma proposta de implementação de uma linguagem XML (XQL (LAPP; ROBIE; SCHAC, 1998)). Nesta proposta (BAEZA-YATES; NAVARRO, 2002) ele mostrou como XQL pode ser implementado utilizando um modelo de nodos próximos (*proximal nodes* (NAVARRO; BAEZA-YATES, 1997)). O modelo resolve consultas baseado no conteúdo dos dados (palavras), na estrutura a ser retornada e na combinação dos dois. Para isso, texto e estrutura, numa forma hierárquica, são indexados. As entradas em cada índice indicam as posições no texto nas quais determinada palavra ocorre. Na hierarquia, cada nó indica a posição no texto do componente estrutural ao qual está associado. Ao realizar a combinação da busca da palavra com a busca da estrutura, o processamento da consulta é acelerado porque somente os nós mais próximos na lista da palavra indicada na consulta são pesquisados de cada vez. Pode ocorrer que um nó, mesmo não participando da consulta, seja retornado, desde que esteja próximo.

Torsten desenvolveu uma linguagem de consulta para bases XML chamada approXQL (SCHLIEDER, 2001). Em sua abordagem, tenta resolver o problema de consultas que não

são contempladas por nenhum resultado, retornando uma classificação ordenada com os objetos mais similares ao solicitado. Para traçar esta similaridade, é feito um cálculo de quantas transformações um objeto deve sofrer para ficar igual ao outro. Essas transformações são operações de troca de nome, inserção e exclusão de um grupo restrito de elementos. Um exemplo de transformação são as duas folhas inseridas abaixo do elemento `title` da segunda árvore da figura 2.3. Elas representam a divisão em palavras da sequência de texto a qual formava o valor do elemento. A aproximação de um objeto consulta, representado pela seguinte expressão:

```
cd[title["piano" and "concerto"] and composer["rachmaninov"]],
```

com um pedaço de uma árvore transformada, pode ser vista na figura 2.3.

Bastante similar a abordagem de Torsten, Ciaccia (CIACCIA; PENZO, 2002a) aborda a similaridade estrutural, apresentando uma nova função, que atribui um valor aos resultados, levando em consideração o grau de correção e completude de acordo com uma dada consulta, bem como o grau de coesão dos dados recuperados. Grau de correção é um valor, normalizado no intervalo [0, 1], que indica o quão o casamento, entre os nodos da estrutura da consulta com os da estrutura dos dados, está correto enquanto que o grau de completude atribui um outro valor para a quantidade de casamentos realizados.

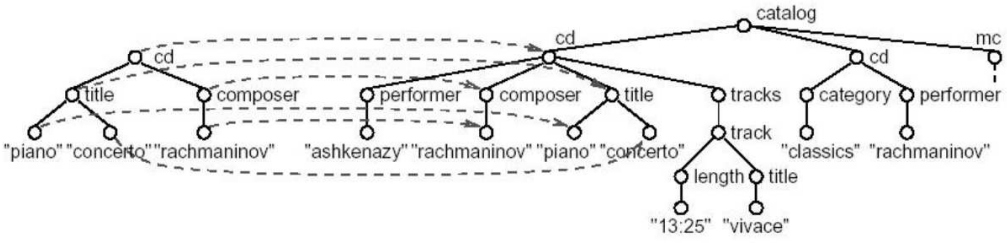

Figura 2.3: "Encaixe"entre uma árvore de consulta e uma árvore de dados ((SCHLIEDER, 2001)).

Posteriormente, em (CIACCIA; PENZO, 2002b), é apresentada uma proposta de encaixe aproximado de uma consulta do usuário em coleções de documentos XML. A motivação para essa abordagem encontra-se no fato de que a mesma informação pode ser encontrada em diferentes formatos estruturais em documentos distintos. Para realizar o encaixe aproximado, é apresentado um subconjunto da linguagem XQuery 1.0, onde tanto a consulta, como os documentos XML são transformados em árvores. Depois um algoritmo realiza um casamento dos nodos de forma "relaxada", ou seja, objetos em documentos XML que não possuem a mesma estrutura contida na consulta podem ser retornados após receberem uma pontuação, que leva em consideração semântica e integralidade estrutural, e a correção dos resultados como também a coesão dos dados. Cada um desses itens possui uma fórmula que, após computadas, são combinadas para indicar uma medida de peso global do objeto.

XSEarch (COHEN, 2003) é outra ferramenta que pretende realizar consultas utilizando similaridade estrutural. Oferece uma sintaxe simples que pode ser utilizada por usuários inexperientes formada por termos que podem ser palavras-chave, rótulos ou uma combinação "rótulo:palavra-chave". As respostas devem conter fragmentos XML ao invés de documentos inteiros, onde esses fragmentos devem estar semanticamente relacionados e retornados de forma ordenada de acordo com o seu valor. Por "semanticamente

relacionados" entende-se daqueles elementos que estão muito próximos (fortemente interconectados) como por exemplo um autor e seu artigo. Para elaborar a classificação ordenada dos objetos três fatores são importantes:

- a similaridade entre a consulta e o resultado;
- o peso dos rótulos que aparecem no resultado e;
- as características da árvore resultado.

Para calcular a similaridade é criado um esquema similar ao TFIDF: o TFILF (*Term Frequency* x *Inverse Leaf Frequency*), ou seja, a frequência do termo no documento vezes o número de folhas contendo um termo da consulta, dividido pelo número de folhas contidos no corpo de documentos considerados para a consulta.

XRANK (GUO, 2003) é um mecanismo que realiza buscas por palavra-chave, de forma conjuntiva, tanto em documentos HTML como em XML. No caso de documentos XML, ele retorna uma coleção de elementos, juntamente com seus descendentes, os quais contenham as palavras chaves ordenadas de acordo com sua relevância. Essa ordenação leva em consideração os *hyperlinks* (referenciamentos) e a distância ancestral/descendente entre os elementos.

Em (GRAVANO, 2003), Gravano utiliza a métrica de similaridade *cosine* para realizar a junção de documentos provenientes de fontes de dados da web. O intuito é buscar a mesma informação que pode estar representada em diferentes esquemas ou até escrita em formatos diferentes (com abreviações, erros de digitação, etc). Primeiramente, os dados a serem integrados são "baixados" e armazenados em SGBD relacionais. Posteriormente, são transformados em vetores de *tokens* ou *q-grams*, onde então a métrica de similaridade é utilizada para verificar a semelhança dos registros.

Em (JIN; LI; MEHROTRA, 2002), são utilizadas métricas de casamento de coeficiente juntamente com métricas de casamento de cadeias de caracteres, como *edit distance* e *q-grams*, para realizar a ligação de grandes conjuntos de dados. Primeiramente, dois registros são combinados em um espaço multi-dimensional. Depois os objetos cuja distância estiverem dentro de um *threshold* são separados e, como já foi dito em 2.1.1, as métricas para cadeias de caracteres são utilizadas para determinar se esses objetos são duplicados.

Outra tentativa de melhorar as respostas para consultas com conteúdo vago pode ser vista em (LALMAS; RöLLEKE, 2004). Nesse artigo, é descrita uma estratégia orientada a relevância para representação e recuperação de documentos XML, considerando não apenas conteúdo vago, mas também estrutura vaga. Para calcular a relevância de um objeto, são atribuídos pesos aos elementos e as palavras, as quais formam o conteúdo de um elemento considerando sua frequência (TF). A atribuição de pesos é um esforço na tentativa de identificar quais os elementos e palavras são mais significativos para descrever o elemento ascendente.

A partir desse ponto, passam a ser relacionados aqueles trabalhos que apresentam uma preocupação maior em localizar elementos através do conteúdo de documentos XML. O enfoque principal está naqueles trabalhos que inserem similaridade na comparação dos valores dos elementos, ao invés de realizar uma simples busca booleana, porém ainda aparecem alguns trabalhos que implementam busca por palavra-chave em linguagens de consulta. Um exemplo disto pode ser visto em (FLORESCU; KOSSMANN; MANOLESCU, 2000).

Nesse artigo, a linguagem de consulta XML-QL (DEUTSCH, 1999) é estendida para suportar busca por palavra-chave em documentos XML, sem considerar a sua estrutura. Para isso é inserido um predicado chamado *contains*, que se parece com uma função onde são passados parâmetros com a palavra a ser buscada e a profundidade máxima a ser atingida (descendentes). Esta extensão é feita em conjunto com o suporte de um SGBD relacional (Oracle8i), que é utilizado, entre outras coisas, para armazenar os arquivos invertidos e até para mostrar como toda a busca por palavra-chave pode ser realizada em um banco de dados relacional, se os dados XML estiverem replicados em seu interior.

XIRQL (FUHR; GROSJOHANN, 2001) estende uma linguagem de consulta para XML (XQL (LAPP; ROBIE; SCHAC, 1998)), a qual é centrada na visão dos dados, para permitir a busca de objetos baseados em estrutura e em conteúdos vagos. Isso é feito utilizando-se elementos de IR e criando-se índices de termos com pesos para conteúdos textuais. Além disso é apresentada uma álgebra relacional, que serve como base para o plano de consulta para a nova linguagem criada (XIRQL). Em um dos capítulos, ele aborda a execução de consultas, utilizando tipos de dados e predicados vagos, onde a similaridade deve ser aplicada de acordo com o tipo do dado o qual deve ser fornecido por um esquema XML. Entre os tipos de similaridade são citados exemplos de similaridade para números, datas, textos em diferentes línguas e nomes próprios. Porém, em nenhuma parte do artigo, foram apresentadas estas funções, como também não foi dito que tipo de métricas foram utilizadas.

Em (THEOBALD; WEIKUM, 2001) uma linguagem de consulta chamada XXL (*flexible XML search language*), baseada em XML-QL, é criada e estendida para suporte a busca por similaridade. Esse suporte possibilita a comparação, tanto dos nomes dos nodos de uma estrutura, quanto dos valores que eles possuem. Além de diversos *wildcards*, que aumentam a abrangência da consulta, é inserido um operador de similaridade "$\sim$", que pode anteceder tanto o nome de um elemento, como o seu valor na formulação da consulta. A colocação desse operador indica que, para aqueles elementos ou valores, deve ser aplicada uma função de similaridade, a qual irá retornar um valor entre 0 e 1, que indica a relevância do objeto.

Para realizar testes, foi implementado um protótipo sobre o Oracle8i interMedia, que é um sistema próprio para recuperação de textos. Para localizar cada registro na base, são usadas funções baseadas em conhecimento ontológico de palavras (*thesaurus*), que se apoiam em dicionário, e funções baseadas em palavras, juntamente com uma função *SCORE*, que retorna a probabilidade de relevância para cada registro. Como métrica de similaridade para as palavras foi, primeiramente, utilizado Levenshtein (ver 2.1.1) e, posteriormente, *thesaurus*.

Em (DORNELES et al., 2004), Dorneles apresenta um conjunto de métricas de similaridade para manipular coleções de valores em XML. Para calcular a similaridade entre as coleções, são apresentados dois grupos distintos de métricas: as *MAV* (*metrics for atomic values*) e *MCV* (*metrics for complex values*). As *MAVs* são utilizadas para calcular a similaridade entre elementos atômicos, como um nome ou uma data, enquanto as *MCVs* são utilizadas para elementos complexos, como, por exemplo, uma lista de autores. Após o calculo da similaridade, o resultado é apresentado na forma de uma classificação ordenada. Ao final, são mostrados os experimentos realizados em fontes de dados reais, salientando-se a eficácia dos métodos utilizados através de um gráfico, onde aparece a curva de revocação e precisão.

Esse último artigo está intimamente ligado a este trabalho, pois foi a partir dele e de (DORNELES et al., 2004) que foram extraídas as métricas utilizadas para criar uma

linguagem de consulta que suportasse argumentos vagos. Outro aspecto importante é que, em ambos, aparecem experimentos que demonstram a eficiência dos métodos adotados.

2.4 Conclusões

Como pode ser observado, a maioria dos trabalhos apresentados tratam a similaridade entre elementos atômicos de forma muito restrita. Não há uma proposta que apresente várias métricas de similaridade, tanto para tipos de dados diferentes, como para diferentes formatos, que a mesma informação pode assumir, para um mesmo tipo de dado (como, por exemplo, acrônimos ou inversão de palavras).

Por outro lado, alguns trabalhos realizam a aproximação de estruturas para tentar satisfazer a uma consulta. Normalmente, são selecionados objetos que possuam estruturas similares àquela contida na consulta submetidada pelo usuário. Desta forma, o usuário não consegue definir, de forma fixa, como seus dados devem estar organizados dentro do objeto.

Além disso foram apresentadas abordagens que utilizam SGBD relacionais, dentre eles alguns comerciais, como base para criação de um protótipo, criando uma estreita relação com a capacidade do *hardware* utilizado, bem como da necessidade de aquisição de licenças, no caso de sua utilização comercial.

3 PROCESSAMENTO XPATH

O presente capítula apresenta detalhes de como um mecanismo de consulta para XPath foi extendido. Primeiramente, a linguagem de consulta XPath é conceituada e explicada com alguns exemplos tendo em vista sua importância para este trabalho.

3.1 Características da linguagem XPath

Conceitualmente, a linguagem XPath é utilizada no endereçamento de partes de um documento XML. As expressões XPath possibilitam a navegação e seleção de elementos em um documento, o que é feito através dos caminhos de localização (*location path*), muito similares aos utilizados em sistemas de arquivos. O retorno de uma consulta XPath pode ser na forma de valores ou de elementos. No caso de mais de um elemento, a resposta vem sempre na forma de conjunto (*node-set*), não sendo possível gerar uma classificação ordenada dos resultados, ou seja, colocá-los em sequência de acordo com o valor de algum elemento ou de alguma função. Isso ocorre, pois, dentro dessa linguagem, não existe clausula ou função que em ordem os objetos de acordo com o valor de um determinado elemento. Por não existir a possibilidade de ordenar o resultado, o tipo do valor de retorno das funções, definidas no capítulo 4, poderia ser booleana, ou seja, poderia-se passar o valor do ponto de corte (*threshold*) como parâmetro e a função resolveria se o elemento é ou não simililar. Isso seria suficiente para localizar elementos em expressões de caminho. Entretanto um retorno booleano não possibilita a implementação de uma classificação ordenada dos resultados, nem a utilização dessa funcionalidade em linguagens que incorporam a linguagem XPath, o que torna a abordagem limitada.

Os caminhos de localização também podem ser utilizados como parâmetros de entrada das funções. Nesse caso, eles representam um *node-set* e possuem o mesmo significado em todas as funções em que aparece:

- *node-set* é um argumento com as mesmas características definidas em (W3C, Acesso em: 23 abr. 2005), porém deve ser um caminho relativo ao nodo contexto da expressão. Como o *node-set* é um conjunto de nodos representado por uma expressão de caminho, em seu último nodo, deve conter um valor (elemento atômico) ou outro elemento (elemento complexo). Ele serve para indicar o elemento da base que fará parte da comparação por similaridade.

Basicamente, o *node-set* é um conjunto de nodos selecionado por um caminho de localização. Se não houver nenhum predicado, limitando quais nodos devem ser selecionados, todos os nodos, relativos àquele caminho, serão selecionados. Considere a base apresentada na figura 4.2 e o seguinte caminho de localização:

```
/movies/film/crew
```

Todos os nodos `crew`, juntamente com todos os seus subelementos, farão parte do resultado. No caso do argumento *node-set*, pode-se ter, por exemplo, `direction/name` como um caminho relativo ao caminho de localização acima. Nesse caso, o nodo `name` é o nodo que será utilizado na comparação por similaridade. É importante salientar que, por ser um caminho relativo, o valor passado através do argumento *node-set* não começa com uma "/".

O caminho de localização pode ser formado por diversos passos (*location steps*), os quais são separados por uma "/". Um *location step* pode ter zero ou mais predicados. Os predicados são expressões arbitrárias utilizadas para refinar o conjunto de nodos selecionados por um *location step*.

O predicado é equivalente a clausula *where* em comando SQL, pois serve para filtrar o conjunto de nodos relativos àquele *location step* para produzir um novo conjunto. Para cada nodo, a expressão do predicado é avaliada e, caso o resultado seja verdadeiro, o nodo em questão é incluído no conjunto de nodos do resultado.

Elementos do tipo atômico possuem apenas um valor atribuído, não possuindo subelementos em seu contexto. Embora na especificação da XML só existam dados do tipo PCDATA, XPath possui outros tipos de dados em suas funções. Os argumentos passados como parâmetros nas funções são convertidos para esses valores. Na conversão dos argumentos é feito um reconhecimento do tipo. Tipos PCDATA são convertidos para *string* de forma diteta. Para valores do tipo *date* ou *number* é necessário identificar se os números e as datas são válidas. Esses tipos de dados serão usados nas funções que serão descritas, não só na passagem de parâmetros, como também no retorno do resultado da função.

Os tipos dos argumentos, passados como parâmetros das funções específicas da linguagem SimXPath, possuem a mesma convenção adotada em (W3C, Acesso em: 23 abr. 2005), ou seja, os argumentos podem ser dos tipos (*string*, *number*, *boolean* e *node-set*). O tipo de retorno, de todas as funções, é *number*.

3.2 Extensão do processador

Para extensão, foi escolhido um mecanismo de consulta baseado na XPath 1.0 (W3C, Acesso em: 23 abr. 2005). Atualmente, a W3C publicou uma versão rascunho (Working Draft) de como deveria ser a XPath 2.0 (W3C, Acesso em: 17 mar. 2005). Esse rascunho deixa a XPath muito similar a XQuery 1.0, pois acrescenta novos tipos de dados, sequências e quantificadores. Entretanto, XPath 1.0 já está consagrada na comunidade científica, possuindo diversos mecanismos disponíveis e, além disso, serve de núcleo para outras linguagens como a XQuery. Desse modo, a extensão proposta para XPath 1.0 foi feita de uma forma que não altera a sintaxe da linguagem e pode ser utilizada nas linguagens, nas quais XPath é um componente, aumentando seu poder de expressão.

O mecanismo adotado para incluir-se as modificações foi o Jaxen (CODEHAUS, Acesso em: 30 jun. 2005), o qual foi desenvolvido em java. Esse processador foi escolhido por diversas razões destacando-se entre elas:

- utiliza uma licença de código aberto similar a do Apache (ASF/BSD), que é uma das menos restritivas existentes;

- provê um ponto único de avaliação de uma expressão XPath, independente de qual modelo de objeto está sendo usado;

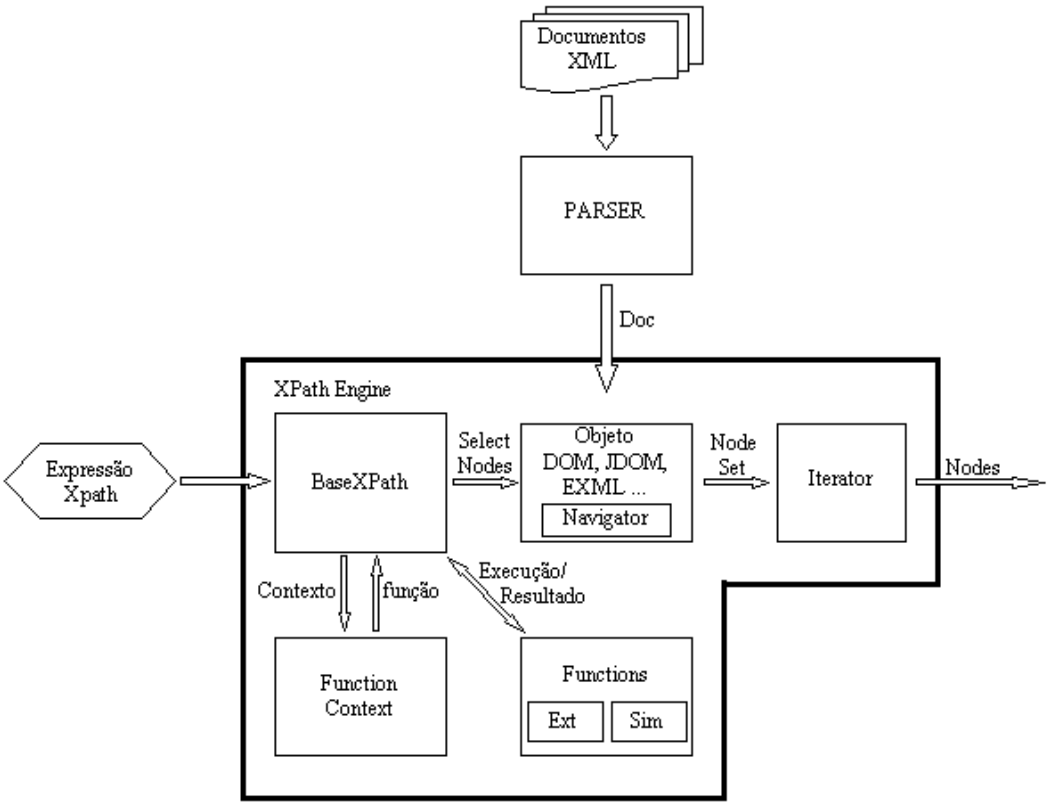

Figura 3.1: SimXPath Engine

- é compatível com diversos modelos de objetos como DOM, dom4j e EXML, entre outros;

- possui uma equipe que desenvolve seu código-base o que aumenta a sua robustez.

A figura 3.1 mostra a arquitetura básica desse mecanismo. Primeiramente, é necessário instanciar um objeto por itermédio de um parser (SAX), que irá realizar a análise gramatical do documento XML a ser consultado (1). É importante ressaltar que o objeto SAX não faz parte do mecanismo Jaxen. Posteriormente, deve-se escolher com qual modelo de objeto se deseja trabalhar (DOM, JDOM, ...). Em (2) um objeto DOM é construído com o documento oriundo da análise. Através de uma cadeia de caracteres, contendo uma expressão XPath, um objeto *BaseXPath* é instanciado (3) recebendo como entrada o objeto DOM criado.

Utilizando o método *SelectNodes()*, o objeto *BaseXPath* vai percorrendo a árvore XML, selecionando todos os nodos identificados na expressão de caminho inserindo-os no conjunto de nodos do resultado (4). Se, em alguns dos passos da seleção, houver um predicado e dentro deste predicado houver uma função, o método *SelectNodes()* irá pesquisar qual função está sendo solicitada. Isso é feito por intermédio da classe *FunctionContext* que, recebendo o contexto da expressão, retornará a função a ser executada, já instanciada (5). A execução é então passada para a função que se encontra referenciada na expressão (6). Após o processamento da função, o objeto *BaseXPath* avalia o resultado do predicado e caso seja verdadeiro, inclui o nodo corrente no conjunto de nodos do resultado (7). A execução da consulta continua normalmente do ponto onde havia parado e ao final, a classe *Iterator* percorre o conjunto de nodos (*node-set*) do resultado, retornando

nodo por nodo (8).

A classe que possui o contexto de todas as funções aponta para um pacote onde as mesmas estão depositadas. É exatamente nesse pacote que são inseridas as novas funções de similaridade (na figura 3.1 a caixa *Sim* dentro da caixa *Functions*). Foi preciso também atualizar a classe que mapeava estas novas funções (*FunctionContext*). Desse modo, a linguagem foi estendida, mantendo-se as suas características.

4 FUNÇÕES DE CONSULTA POR SIMILARIDADE

Nesta seção, é apresentada a forma como as funções propostas para a SimXPath implementam as métricas descritas na seção 2.1. As funções possuem o mesmo nome e funcionam de acordo como foram definidas nas métricas, com algumas adaptações, para que se adequassem ao padrão da linguagem XPath. As funções serão apresentadas enfatizando-se o modo como elas foram implementadas e qual o seu comportamento dentro das consultas. Também foi acrescentada mais uma consulta atômica ao conjunto já existente, a fim de torná-lo mais completo. No momento oportuno, essas consultas serão salientadas.

Os exemplos mostrados são construídos sobre uma base real de dados XML (filmes e citações bibliográficas). A base de filmes originou-se do site *The Collection of Computer Science Bibliographies* (http://liinwww.ira.uka.de/bibliography/Database/index.html). Os dados sobre filmes foram extraídos manualmente da Web a partir das seguintes fontes de dados: *Blockbuster* USA, UK e Ca, *The Internet Movie Database (IMDb)*, *Movies.com* e *Yahoo Movies*, totalizando 295 instâncias. Em ambos os casos, supôs-se que as instâncias da base de dados encontravam-se armazenadas em um único documento XML. Parte dessas bases é apresentada na figura 4.2, e a representação do seu esquema pode ser vista em forma de árvore na figura 4.1.

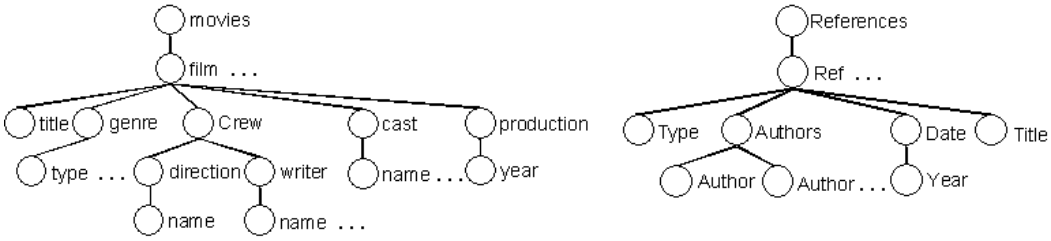

Figura 4.1: Esquema da base de filmes e de citações bibliográficas.

As funções XPath, em geral, são utilizadas dentro de predicados. Para que as funções de similaridade possam ser utilizadas com eficiência, deve-se adotar essa mesma abordagem. Como o resultado dessas funções é um valor entre 0 e 1, quando elas forem referenciadas dentro de um predicado, também devem ser comparadas com um valor entre 0 e 1. Por exemplo, se deseja-se saber se um determinado autor, cujo nome foi escrito com possíveis erros ortográficos, faz parte de uma coleção de artigos, deve-se comparar se o valor de retorno de uma das funções de similaridade para cadeia de caracteres, com esse nome, é maior que algum número estipulado entre 0 e 1. Se for estipulado o valor 0.7, significa que o nome passado como parâmetro deve ser, no mínimo, 70% similar ao nome que se encontra na base.

Filmes

```
<movies>
   <film>
      <title>Anger Management</title>
      <genre><type>Comedy</type></genre>
      <crew>
         <direction><name>Peter Segal</name></direction>
         <writer><name>David Dorfman</name></writer>
      </crew>
      <cast>
         <name>Adam Sandler</name>
         <name>Jack Nicholson</name>
      </cast>
      <production><year>2003</year></production>
   </film>
   <film>
      <title>Bad Boys</title>
      <genre>
        <type>Action</type>
        <type>Comedy</type>
      </genre>
      <crew>
         <direction><name>Michael Bay</name></direction>
         <writer>
            <name>George Gallo</name>
            <name>Michael Barrie</name>
         </writer>
      </crew>
      <cast>
         <name>Martin Lawrence</name>
         <name>Will Smith</name>
      </cast>
      <production><year>1995</year></production>
   </film>
   <film>
      <title>Chicago</title>
      <genre>
         <type>Musical</type>
         <type>Drama</type>
      </genre>
      <crew>
         <direction><name>Rob Marshall</name></direction>
         <writer>
            <name>Maurine Dallas Watkins</name>
            <name>Bob Fosse</name>
         </writer>
      </crew>
      <cast>
         <name>Renée Zellweger</name>
         <name>Catherine Zeta Jones</name>
         <name>Richard Gere</name>
      </cast>
      <production><year>2002</year></production>
      <language>English</language>
      <description>sexual content and dialogue</description>
   </film>
</movies>
```

Citações Bibliográficas

```
<References>
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>Abraham Silberschatz</Author>
         <Author>Henry F. Korth</Author>
         <Author>Sudarshan/Author>
      </Authors>
      <Date>
         <Year>1997</Year>
      </Date>
      <Title>Database System Concepts</Title>
   </Ref>
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>Abraham Silberschatz</Author>
         <Author>Peter Galvin</Author>
      </Authors>
      <Date>
         <Year>1998</Year>
      </Date>
      <Title>Operating Systems Concepts, Fifth Edition</Title>
   </Ref>
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>Serge Abiteboul</Author>
         <Author>Peter Buneman</Author>
         <Author>Dan Suciu</Author>
      </Authors>
      <Date>
         <Year>1999</Year>
        <Month>apr</Month>
      </Date>
     <Title>Data on the Web</Title>
   </Ref>
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>H. Korth</Author>
         <Author>S. Mehrotra</Author>
         <Author>R. Rastogi</Author>
      </Authors>
      <Date>
        <Year>1999</Year>
        <Month>apr</Month>
      </Date>
      <Title>Ensuring Consistency on...</Title>
   </Ref>
</References>
```

Figura 4.2: Coleção de instâncias de filmes (film) e de citações bibliográficas (ref)

Pensando em uma futura implementação em um processador XQuery, as funções foram definidas com valor de retorno do tipo numérico. Esse valor significa o escore de similaridade entre o argumento de busca, passado como parâmetro na consulta, e os elementos XML da base, estando contido no intervalo [0,1]. Quanto mais próximo de 1, maior a similaridade entre os elementos.

4.1 Elementos Atômicos

As funções para elementos atômicos visam comparar, de forma exclusiva, os valores que se encontram nas folhas de um documento XML. Para padronizar, todas as funções para elementos atômicos possuem dois parâmetros de entrada, sendo:

1. a expressão de caminho que indica a localização do elemento ou atributo a ser comparado, representado por *node-set*;

2. o argumento de comparação usado pela função, representado pelo seu tipo *string* ou *number*.

A expressão de caminho passada como argumento em uma função (o *node-set*) deve ser relativa ao nodo contexto que está fora do critério de seleção (representado em XPath por "[ ]"). Por exemplo, na expressão `/references/ref/date[year="2000"]`, o nodo contexto é `date` e o argumento `[year="2000"]` é o critério de seleção).

Essas funções seguem a mesma classificação usada para as *A-metrics*, encontradas em (DORNELES et al., 2004), para os tipos de dados *cadeia de caracteres*, *data* e *número*. As *A-metrics* são métricas para elementos atômicos e tratam da similaridade utilizando funções conhecidas como *edit distance*, *n-grams* e outras elaboradas como *dateSim*. As métricas que compõem as *A-metrics* foram estudadas e descritas na seção 2.1. A partir dessas métricas foram definidas as funções que são apresentadas nas próximas subseções.

4.1.1 Funções para cadeias de caracteres

A implementação das funções está baseada no que foi exposto na seção 2.1. Para cadeia de caracteres, as funções comparam dois valores do mesmo tipo. Embora essas funções se apliquem ao mesmo tipo de dado, cada função cumpre um propósito diferente.

4.1.1.1 editSim

Função: *number* **editSim***(node-set, string)*

Essa função compara cadeias de caracteres que possuam erros tipográficos. Deve ser utilizada, basicamente, quando não se sabe como a palavra está escrita ou quando a fonética da mesma induz ao erro. Ela se baseia na métrica *edit distance* e foi implementada como está proposto na seção 2.1.1. Essa função não leva em consideração o peso dos *tokens* como visto em (CHAUDHURI, 2003).

Exemplo 1 *Localizar as citações bibliográficas que possuam autor com um nome parecido com "Abraham Silberchats".*

```
/References/Ref[type="book" and editSim(Authors/Author,"Abraham
Silberchats")>0.7]
```

No exemplo 1, o mecanismo da XPath irá percorrer todos os elementos `/References/Ref` que existem no documento. Para cada um desses elementos, ela irá realizar as comparações booleanas que se encontram no predicado ([ ]). Porém, antes da comparação ser realizada, o mecanismo irá executar a função de similaridade **editSim**, onde `Authors/Author` é um elemento descendente do nodo contexto (`/Ref`), e o nome do autor é um valor impreciso. Um escore de similaridade é retornado e apenas serão selecionados os elementos `Ref` cujo escore de similaridade de `Author` for maior do que o definido na consulta. Nesse exemplo o nome do autor está com uma letra ausente e uma trocada. Na comparação com o nome original, contido no documento, através da função, é retornada uma similaridade de 0.9. Como 0.9 é maior que 0.7, todas as referências que tiverem o nome "Abraham Silberschatz" serão consideradas como resultado da consulta.

```
<References>
   <Ref>
      <Type>BOOK</Type>
      <Authors>Abraham Silberschatz</Author>
```

```
      <Author>Henry F. Korth</Author>
      <Author>Sudarshan/Author>
   </Authors>
   <Date>
      <Year>1997</Year>
   </Date>
   <Title>Database System Concepts</Title>
</Ref>
<Ref>
   <Type>BOOK</Type>
   <Authors>
      <Author>Abraham Silberschatz</Author>
      <Author>Peter Galvin</Author>
   </Authors>
   <Date>
      <Year>1998</Year>
   </Date>
   <Title>Operating Systems Concepts, Fifth Edition</Title>
</Ref>
<Ref>
 .
 .
 .
```

4.1.1.2 acronymSim

Função: *number* **acronymSim***(node-set, string)*

Essa função auxilia na construção de consultas em que se deseja pesquisar um acrônimo ou abreviações. Como em todas as funções, os argumentos de pesquisa deverão ser, primeiramente, o conjunto de nodos e a segunda parte a cadeia de caracteres a ser comparada. Porém, o acrônimo pode estar em qualquer um dos lados, ou seja, pode-se pesquisar a palavra inteira para se localizar o acrônimo ou pode-se procurar pelo acrônimo tentando-se localizar a palavra. A função não faz distinção de qual lado se encontra parte abreviada. Entretanto, como esta é uma função específica para consulta de acrônimos, ela não levará em conta a similaridade das palavras caso ela apareça por extenso em ambos os lados. É importante salientar que uma cadeia de caracteres (sem espaços) será considerada um acrônimo. Portanto essa função não deve ser utilizada para a abreviatura ou acrônimo de uma palavra apenas.

Exemplo 2 *Localizar as citações bibliográficas que possuam um autor com as iniciais "HK".*

```
/References/Ref[type="book" and
                acronymSim(Authors/Author,"HK")>0.6]
```

Nesse exemplo, procura-se um autor que possua as iniciais HK. Na base da figura 4.2 existe apenas um autor cujo nome se aproxima destas iniciais ("Henry F. Korth"). Se, no argumento de pesquisa, um dos nomes houvesse sido escrito por extenso (ex. "Henry K"), o resultado seria o mesmo, pois a função leva em consideração apenas as iniciais de cada argumento. Porém, se o parâmetro utilizado tivesse sido apenas "Henry" a função entenderia que estaria se procurando um elemento com as iniciais H.E.N.R.Y. e, dessa forma, a referência desejada não seria localizada.

4.1.1.3 ngramsSim

Função: *number* **ngramsSim***(node-set, string)*

A função **ngramsSim** (ULLMANN, 1977) é derivada do algorítmo *n-grams* e tem a finalidade de calcular a similaridade entre cadeias de caracteres que podem estar fora de ordem. Pode ser utilizada em documentos XML onde algumas palavras podem estar fora de ordem dentro de um texto. Um caso típico é o armazenamento de nomes próprios, onde, algumas vezes, o sobrenome vem antes do primeiro nome. Outro exemplo é quando países/estados podem estar junto com suas capitais, porém a ordem é desconhecida ("Porto Alegre, RS"ou "RS- Porto Alegre"). Esses dois casos servem apenas para ilustrar alguns entre outros vários casos em que ele pode ser utilizado. Deve-se levar em consideração, também, que as palavras envolvidas podem conter pequenos erros tipográficos que ainda assim o escore de similaridade pode ser alto. Entretanto não é apropriado utilizar essa função na comparação direta entre cadeias de caracteres que possam conter erros. Embora ela também sirva para essa finalidade, a função **editSim** é mais precisa.

Exemplo 3 *Localizar os filmes dirigidos pelo diertor "Segal, Peter" depois do ano 2000.*

```
/movies/film[production/year>2000 and
             ngramsSim(crew/direction/name, "Segal, Peter")>0.8]
```

Apesar do nome estar fora de ordem em comparação com o nome que se encontra na base XML, a função **ngramsSim** é capaz de compreender esta desordem e, no caso do exemplo descrito, devolver um valor de similaridade de 0.9. Para o calculo da similaridade, a métrica leva em consideração apenas caracteres alfanuméricos. Logo, para a consulta feita, a vírgula é desconsiderada e obteria-se o seguinte resultado:

```
<movies>
   <film>
      <title>Anger Management</title>
      <genre><type>Comedy</type></genre>
      <crew>
         <direction><name>Peter Segal</name></direction>
         <writer><name>David Dorfman</name></writer>
      </crew>
      <cast>
         <name>Adam Sandler</name>
         <name>Jack Nicholson</name>
      </cast>
      <production><year>2003</year></production>
    .
    .
    .
```

4.1.1.4 containsSim

Função: *number* **containsSim***(node-set, string)*

As funções até então definidas utilizavam, em suas comparações, todos os *tokens* contidos no valor de um elemento e já haviam sido criadas e descritas pela comunidade científica. Para suprir a necessidade de se fazer consultas por similaridade em apenas parte do conteúdo de um elemento foi criada a função **containsSim**. Essa é a única função, aqui apresentada, que não se encontra na seção 2.1. Ela possui o mesmo conceito que a função **contains** definida em (W3C, Acesso em: 23 abr. 2005). Para cada *token* da cadeia de caracteres da consulta, ela calcula a similaridade entre os *tokens* contidos dentro do elemento através da função **editSim**. Posteriormente, essa função faz a média do somatório da maior similaridade encontrada para cada *token* da cadeia de caracteres passada como argumento na função.

Exemplo 4 *Localizar os filmes que contenham a palavra "dialogo" em sua descrição.*

```
/XML/movies/film[containsSim(description, "dialogo")>0.7]
```

Para essa consulta a função retorna um valor de similaridade de 0.75, e o resultado é o que se pode ver a seguir.

```
    .
    .
    .
   <film>
      <title>Chicago</title>
      <genre>
         <type>Musical</type>
         <type>Drama</type>
      </genre>
      <crew>
         <direction><name>Rob Marshall</name></direction>
         <writer>
            <name>Maurine Dallas Watkins</name>
            <name>Bob Fosse</name>
         </writer>
      </crew>
      <cast>
         <name>Renée Zellweger</name>
         <name>Catherine Zeta Jones</name>
         <name>Richard Gere</name>
      </cast>
      <production><year>2002</year></production>
      <language>English</language>
      <description>sexual content and dialogue</description>
   </film>
</movies>
```

Convém resaltar que os *tokens* da cadeia de caracteres passada como parâmetro não precisam ser consecutivos. Se a mesma consulta acima fosse feita para a cadeia de caracteres "sexual dialogue", utilizando a função **contains**, primitiva da XPath, nenhum resultado seria retornado, pois "sexual" e "dialogue" não são palavras consecutivas. Entretanto a função **containsSim** não leva em consideração a distância em que essas palavras se encontram no texto, devido ao fato da função não saber se a comparação feita, para cada palavra, se trata de um casamento positivo.

4.1.2 Funções para números

A métrica de similaridade para números, embora não seja tão atraente, pois normalmente consultas envolvendo quantidades numéricas utilizam comparadores de $<$, $>$ ou ambos para determinar um intervalo, são interessantes no sentido de que podem ser utilizadas juntamente com as métricas para elementos complexos. Como será visto mais adiante, funções como **tupleSim** só admitem métricas de similaridade em sua composição não trabalhando com operadores booleanos.

Função: *number* **sameNumSim***(node-set, number)*

Essa função encontra a similaridade entre dois números positivos ou dois números negativos, usando a fórmula da subseção 2.1.6.

4.1.3 Funções para Data

Essas funções foram criadas para localizar datas aproximadas a um determinado período dado. A imprecisão em se saber o formato em que a data pode estar, bem como

o dia, o mês ou o ano corretos, faz dessa função uma excelente ferramenta no auxílio a busca por similaridade. Dessa forma, foram criadas as seguintes funções:

Função: *number* **dateSim***(node-set, string)*
Função: *number* **yearSim***(node-set, string)*

Sendo que a primeira leva em consideração datas completas e a segunda apenas o ano. Analogamente, poderiam ter sido desenvolvidas funções para mês e dia, como proposto em (DORNELES et al., 2004), porém isso não foi feito neste trabalho.

Exemplo 5 *Localizar as citações bibliográficas escritas por volta de maio do ano 2000.*

```
/References/Ref[dateSim(Date, "01/05/2000")>0.75]
```

```
    .
    .
    .
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>Serge Abiteboul</Author>
         <Author>Peter Buneman</Author>
         <Author>Dan Suciu</Author>
      </Authors>
      <Date>
         <Year>1999</Year>
        <Month>apr</Month>
      </Date>
     <Title>Data on the Web</Title>
   </Ref>
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>H. Korth</Author>
         <Author>S. Mehrotra</Author>
         <Author>R. Rastogi</Author>
      </Authors>
      <Date>
        <Year>1999</Year>
        <Month>apr</Month>
      </Date>
      <Title>Ensuring Consistency on...</Title>
   </Ref>
</References>
```

No exemplo 5, foi utilizada a função para datas completas embora no documento os objetos que contém dados sobre data não possuem o dia. Nesses casos, a função atribui o primeiro dia do mês ao valor de dia. Se, da mesma forma, estivesse faltando o valor de mês, seria considerado o primeiro mês do ano. A data informada no parâmetro *string* deve ser do tipo "dd/mm/aaaa". Essa mesma consulta poderia ter sido realizada utilizando-se apenas o ano, ficando da seguinte forma:

```
/References/Ref[yearSim(Date, "2000")>0.75]
```

A grande dificuldade de criar uma função de similaridade para datas é a variedade de formatos em que elas podem ser encontradas. Procurando solucionar este problema, a função de similaridade para datas **dateSim** compara uma data de entrada, no formato "dd/mm/aaaa" com datas nos formatos da seção 2.1.5. O primeiro formato é idêntico ao formato do parâmetro de entrada, devendo o dia anteceder o mês.

O segundo formato, leva em consideração o mês escrito por extenso ou de forma abreviada (no mínimo 3 letras). O mês pode estar escrito tanto em inglês, quanto em português. Nesse formato, caracteres especiais não são levados em consideração bem como a ordem em que se encontram o dia, o mês e o ano dentro do elemento, porém o ano, deve ser composto de 4 caracteres. O terceiro formato considera o elemento passado através do argumento *node-set* como um elemento composto por elementos atômicos cujos nomes devem ser `dia`/`day`, `mes`/`month` e `ano`/`year` independente de ordem.

4.2 Elementos Complexos

As funções de similaridade para elementos complexos foram elaboradas para auxiliar o usuário a localizar elementos atômicos agrupados em estruturas complexas. A comparação dos elementos atômicos é feita através das métricas de similaridade atômicas. Esta comparação obedece a forma na qual os elementos estão agrupados (lista, conjunto ou tupla) informada na elaboração da consulta. Definida a estrutura a ser pesquisada, são localizados na base todos os objetos que possuem a mesma forma contida na consulta, para realizar a comparação atômica de seus elementos.

Para todas as estruturas, um alto escore de similaridade é alcançado se, além dos escores de similaridade de seus componentes ser alto, alguns requisitos específicos de cada estrutura for alcançado. As seguintes descrições, para essas estruturas, foram apresentadas em (DORNELES et al., 2004).

a) *Tupla*. A métrica *tupleSim* compara duas tuplas de valores. Um escore alto é conseguido se o número e o nome dos componentes de cada tupla for igual.

b) *Coleções*

i) Conjunto. A métrica *setSim* compara dois conjuntos de valores. Um escore alto é alcançado se os componentes dos dois conjuntos tiverem todos o mesmo nome e forem em mesmo número, não importando se estão na mesma ordem ou não;

ii) *Lista*. A métrica *listSim* compara duas listas de valores. Um escore alto é atingido se os componentes das duas listas tiverem todos o mesmo nome, forem em mesmo número e estiverem na mesma ordem;

Lista de autores	Conjunto de instituições	Tupla de conferencia

`<publicacao>`
`  <autores>`
`  <autor>Mehrotra, Sharad</autor>`
`  <autor>Rastogi, Rajeev</autor>`
`  <autor>Korth, Henry</autor>`
`  <autor>Silberschatz, Abraham</autor>`
`  </autores>`
`</publicacao>`
	`<instituicoes>`
	`  <nome>Unicamp</nome>`
	`  <nome>UFRGS</nome>`
	`  <nome>UFRJ</nome>`
	`  <nome>UFAM</nome>`
	`</instituicoes>`
		`<conferencia>`
		`  <nome>Simpósio Italiano de Sistemas`
		`  de Banco de Dados</nome>`
		`  <endereco>Itália</endereco>`
		`  <area>Sistemas de Banco de Dados</area>`
		`  <ano>2001</ano>`
		`</conferencia>`

Figura 4.3: Exemplos de instâncias de uma base XML

Através dessas funções é possível realizar consultas que, até mesmo em SQL tradicional, são muito difíceis, como, por exemplo, localizar os trabalhos que tenham

sido escritos por um determinado grupo de autores. Nessa consulta, ainda pode estar implícita a ordem em que se deseja encontrar os autores e até mesmo localizar obras que foram feitas, exclusivamente, por dois ou mais autores. A seguir será visto, mais detalhadamente, como este tipo de consulta pode ser facilmente escrita utilizando-se SimXPath.

4.2.1 Coleções

Coleções são agrupamentos de dados que possuem as mesmas características. As coleções podem ser divididas em listas e conjuntos, onde a diferença básica é que, nas listas, a ordem em que os elementos aparecem dentro do documento faz diferença. Em SimXPath, as funções que lidam com coleções trabalham com dois argumentos de entrada, os quais possuem o mesmo significado em todas essas funções. Os tipos de dados desses parâmetros estão de acordo com os tipos definidos em (W3C, Acesso em: 23 abr. 2005) e já mencionados na seção 3.1. Os argumentos definidos como parâmetros de entrada são:

- o caminho relativo que contém o elemento que se refere à coleção, representado por *node-set* e;
- os valores dos componentes da coleção, representados por (*string, string**).

Como um exemplo para a definição acima, na figura 4.3, o elemento `instituicoes` é o nome do elemento que possui, dentro dele, os componentes de uma coleção (na mesma figura, o elemento `nome`), formando a coleção `/instituicao/nome`. Ainda na mesma figura 4.3, os valores dos componentes da coleção de instituições seriam (Unicamp, UFRGS, UFRJ, UFAM).

Para uma estrutura complexa ser considerada uma coleção, em SimXPath, não é necessário que os elementos atômicos dessa coleção sejam irmãos. Basta apenas que eles sejam descendentes do mesmo objeto complexo e que estejam dentro do mesmo grau dentro da árvore XML. Por exemplo, algumas referências bibliográficas possuem o nome do autor, dividido em nome e sobrenome (*pref* e *last*). Poderia-se executar uma consulta pelo sobrenome dos autores, agrupando-os em um conjunto, embora esses elementos não sejam irmãos. Considere o seguinte trecho de documento XML abaixo:

```
<Ref>
   <Type>BOOK</Type>
   <Authors>
      <Author>
        <Last>Abiteboul</Last>
        <Pref>Serge</Pref>
      </Author>
      <Author>
        <Last>Buneman</Last>
        <Pref>Peter</Pref>
      </Author>
      <Author>
        <Last>Suciu</Last>
        <Pref>Dan</Pref>
      </Author>
   </Authors>
   <Date>
      <Year>1999</Year>
     <Month>apr</Month>
   </Date>
```

```
    <Title>Data on the Web</Title>
  </Ref>
```

A seguinte consulta retornaria o elemento `Ref` da instância acima:

```
/References/Ref[editListSim(Authors/Author/Last, "Abiteboul",
                                                 "Buneman",
                                                 "Suciu")>0.8]
```

Embora o elemento `Last` não possua irmãos idênticos (o irmão de `Last` é `Pref`), o tipo *node-set* considera o caminho relativo `Authors/Author/Last` à expressão de caminho `/References/Ref`, uma coleção de `Last` dando, assim, flexibilidade às consultas.

4.2.1.1 Funções para listas

As funções para listas são ideais para serem utilizadas quando se deseja localizar objetos que possuam elementos que aparecem repetidas vezes dentro dele. Um exemplo bastante comum é a busca por livros ou artigos que tenham sido escritos por dois ou mais autores. Outro exemplo é a procura de filmes que tenham sido interpretados por dois ou mais atores. Nesse último caso, a ordem pode ser importante pois, o primeiro elemento da lista, indicaria qual o principal o principal ator, principalmente em se tratando de documentos XML.

Essas funções estão intimamente ligadas ao conceito de lista (DORNELES et al., 2004) associado às métricas de similaridade atômicas. Isso significa que, nessas funções, a estrutura de lista está ligada à comparação por similaridade dos objetos atômicos que ela contém. A similaridade atômica utilizada será de acordo com o tipo de elemento que se está comparando como visto na seção 4.1.

O nome dessas funções é formado pelo nome da função atômica (*edit*, *ngrams*, *date*, etc), o nome do tipo de coleção (no caso *List*) e o sufixo *Sim*. Para cada função atômica existirá uma função de lista.

i) **Lista:** as funções definidas para listas de coleções são as seguintes:
   Função: *number* **editListSim***(node-set, string, string*)*
   Função: *number* **acronymListSim***(node-set, string, string*)*
   Função: *number* **ngramsListSim***(node-set, string, string*)*
   Função: *number* **containsListSim***(node-set, string, string*)*
   Função: *number* **dateListSim***(node-set, string, string*)*
   Função: *number* **yearListSim***(node-set, string, string*)*
   Função: *number* **numListSim***(node-set, string, string*)*

As funções para lista comparam cada argumento *string* com o elemento XML da instância equivalente ao argumento *node-set*, fornecido na mesma ordem encontrada na colocação dos parâmetros *string* ou seja, o primeiro parâmetro *string* será comparado com o primeiro nodo do conjunto de nodos, o segundo parâmetro com o segundo nodo e assim por diante. Com todos os escores obtidos, a métrica para conjuntos é aplicada.

Exemplo 6 *Localizar as citações bibliográficas que possuam os seguintes autores, nesta ordem: "Henry Korth", "Mehrotra Sharad"e "Rastogi Rajeev".*

```
/References/Ref[acronymListSim(Authors/Author, "Henry Korth",
                                               "Mehrotra Sharad",
                                               "Rastogi Rajeev")>0.7]
```

```
    .
    .
    .
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>H. Korth</Author>
         <Author>S. Mehrotra</Author>
         <Author>R. Rastogi</Author>
      </Authors>
      <Date>
        <Year>1999</Year>
        <Month>apr</Month>
      </Date>
      <Title>Ensuring Consistency on...</Title>
   </Ref>
</References>
```

No Exemplo 6, para cada elemento `Ref` encontrado, a função *acronymListSim* compara cada valor dado como entrada para os argumentos do tipo *string* "Henry Korth", "Mehrotra Sharad" e "Rastogi Rajeev" com cada elemento `Authors/Author`, na mesma ordem em que se encontram na instância XML. Só são selecionados pelo processador os elementos `Ref` do documento XML que possuem como descendentes, os elementos `Author`, na mesma ordem em que foram solicitados e cuja média de similaridade seja maior do que 0.7. O argumento *node-set* indica que `Authors` é uma lista que contém elementos atômicos `Author`. Neste exemplo, em *node-set* apareceram o elemento que contém a lista bem como o nome do elemento do qual a lista é feita, embora a única exigência é que, no caminho relativo, apareça o elemento do qual a lista é feita. Desta forma as seguintes construções também estão corretas:

Para selecionar `References`

```
/References[acronymListSim(Ref/Authors/Author, "Henry Korth",
                                               "Mehrotra Sharad",
                                               "Rastogi Rajeev")>0.7]
```

Para selecionar `Authors`

```
/References/Ref/Authors[acronymListSim(Author, "Henry Korth",
                                               "Mehrotra Sharad",
                                               "Rastogi Rajeev")>0.7]
```

Como se pode observar, as funções para coleções têm um formato diferente das funções para tuplas. Ao contrário da *tupleSim*, em coleções, não faz sentido fornecer, como entrada da função, o retorno das funções de similaridade de cada componente. Isso porque, como os componentes têm o mesmo nome, o processador XPath compara cada argumento de entrada apenas com o primeiro elemento na instância XML. Passando o caminho relativo que contém o elemento que se refere à coleção mais

o elemento componente através do *node-set*, e os seus valores através dos argumentos do tipo *string*, pode-se garantir que os valores serão comparados na ordem em que aparecem, no caso de listas, ou serão comparados todos com todos, no caso de conjuntos. Além disso, a definição de uma métrica de coleção para cada métrica de similaridade atômica existente é necessária, para definir qual métrica de similaridade deve ser usada sobre os elementos atômicos da coleção. O nome da função para coleções é formado pela métrica de similaridade atômica (*edit, sig,* etc), seguido pelo tipo de coleção (*Set* ou *List*) e o sufixo *Sim*.

4.2.1.2 Funções para conjuntos

Similarmente às funções para listas, as funções para conjunto também tratam com elementos atômicos de mesmo tipo, porém a comparação de cada argumento *string* é feita com todos os elementos na instância XML equivalentes ao caminho fornecido pelo argumento *node-set* e, para cada comparação, retorna o maior escore de similaridade. Com todos os escores obtidos, a métrica para conjuntos é aplicada. Essas funções possuem nomenclatura bastante similar às funções para listas, alterando-se apenas o radical *List* pelo *Set*.

i) **Conjunto**: as funções definidas para conjunto de dados são as seguintes:
Função: *number* **editSetSim***(node-set, string, string*)*
Função: *number* **acronymSetSim***(node-set, string, string*)*
Função: *number* **ngramsSetSim***(node-set, string, string*)*
Função: *number* **containsSetSim***(node-set, string, string*)*
Função: *number* **dateSetSim***(node-set, string, string*)*
Função: *number* **yearSetSim***(node-set, string, string*)*
Função: *number* **numSetSim***(node-set, string, string*)*

Essas funções são ideais para localizar objetos através do conteúdo de suas listas, onde a ordem em que os ítens aparecem não tem importância. Podem ser usadas para localizar remédios que contenham determinadas substâncias, artigos com certas citações e até filmes interpretados por algum conjunto de atores, entre outras várias possibilidades.

Exemplo 7 *Localizar os filmes que tenham participado os atores "Wil Smit" e "Martin Laurence".*

```
/movies/film[editSetSim(cast/name, "Wil Smit",
                                   "Martin Laurence",)>0.7]
```

```
   .
   .
   .
  <film>
     <title>Bad Boys</title>
     <genre>
       <type>Action</type>
       <type>Comedy</type>
     </genre>
     <crew>
        <direction><name>Michael Bay</name></direction>
        <writer>
           <name>George Gallo</name>
```

```
            <name>Michael Barrie</name>
         </writer>
      </crew>
      <cast>
         <name>Martin Lawrence</name>
         <name>Will Smith</name>
      </cast>
      <production><year>1995</year></production>
   </film>
    .
    .
    .
```

No exemplo 7, a função *editSetSim* compara cada valor do argumento *string*, "Wil Smit" e "Martin Laurence" (que foram escritos com erros tipográficos), com todos os elementos na instância XML equivalentes ao caminho `cast/name` fornecido. Para cada comparação, retorna o maior escore de similaridade. Com todos os escores obtidos, é calculada a média de acordo com o número de elementos encontrados na instância. Só são selecionados pelo processador os elementos `film` do documento XML que possuírem, como descendentes, no mínimo, dois elementos `name` e cuja média de similaridade seja maior do que 0.7. Caso existam mais de dois elementos, o valor da média cai de forma equivalente à quantidade a mais de elementos que houver.

4.2.1.3 Funções para subgrupos

Pode-se efetuar uma consulta com apenas parte dos elementos contidos em uma estrutura. Para isso, foram definidas funções que tratam sub-estruturas de elementos complexos. Para as funções que trabalham com listas e conjuntos, basta substituir o tipo da estrutura *Set* ou *List*) por sua sub-estrutura (*SubTupla*, *SubSet* ou *SubList*), originando funções como **editSubSetSim**, **editSubListSim**, **acronymSubSetSim** e assim por diante. Para todas as funções de lista e conjunto, existem as respectivas funções de sublistas e subconjuntos.

As funções de listas e conjuntos visam coleções que possuem exatamente o mesmo número de parâmetros passados nos argumentos *string*; caso contrário, os valores de similaridade seriam muito baixos. As funções para subgrupos visam localizar os objetos, mesmo que o número de parâmetros seja menor que os elementos contidos dentro do grupo encontrado na base.

Exemplo 8 *Localizar as citações bibliográficas que possuam, pelo menos, os seguintes autores, nesta ordem: "Buneman, Peter" e "Suciu, Dan".*

```
/References/Ref[ngramsSubListSim(Authors/Author, "Buneman, Peter",
                                                 "Suciu, Dan")>0.8]
```

```
    .
    .
    .
   <Ref>
      <Type>BOOK</Type>
      <Authors>
         <Author>Serge Abiteboul</Author>
         <Author>Peter Buneman</Author>
         <Author>Dan Suciu</Author>
      </Authors>
      <Date>
```

```
      <Year>1999</Year>
     <Month>apr</Month>
   </Date>
  <Title>Data on the Web</Title>
</Ref>
 .
 .
 .
```

No exemplo 8, a função *ngramsSubListSim* compara cada valor dado como entrada para os argumentos cadeias de caracteres "Buneman, Peter" e "Suciu, Dan" com o elemento `Authors/Author` de ordem equivalente na instância XML. Para as funções de sublistas, é importante que os elementos se encontrem de forma consecutiva na base. No exemplo acima, a função *ngramsSubListSim* retornaria um baixo valor de similaridade, caso os nomes fossem "Abiteboul, Serge" e "Suciu, Dan". Neste caso, deveria ser usada a função *acronymSubSetSim*.

O que muda, em relação às funções que não consideram subestruturas, é a forma como os argumentos de entrada são comparados pelas métricas. As funções *setSim* e *listSim* comparam duas estruturas de elementos XML e, para que o escore de similaridade seja alto, elas deve ser formadas pelo mesmo número de componentes, sejam componentes de tuplas ou coleções. Já as funções *subSetSim* e *subListSim* fornecem a flexibilidade de que uma das estruturas pode possuir um número menor de componentes.

4.2.2 Tuplas

As funções para tuplas têm, como objetivo principal, retornar objetos, cuja principal característica é possuir elementos filhos que sejam diferentes entre si, não só na nomenclatura, mas também nos tipos dos dados que eles contêm. Dessa forma, é proposta uma estrutura de similaridade que se adeque a qualquer tipo de objeto complexo e leva em consideração os tipos de dados dos elementos atômicos que estão contidos nela. A função para tuplas é definida da seguinte forma:

Função: *number* **tupleSim***(number, number*)*

A função utilizada para elementos complexos, que são considerados tuplas, possui, como parâmetro de entrada, valores numéricos, representados por (*number, number**). Cada *number* corresponde ao retorno de uma função de similaridade atômica, que foi previamente aplicada a cada componente da tupla. O valor de *number* está contido no intervalo [0,1], correspondendo aos escores de similaridade dos componentes.

Outro detalhe importante nessa função é que, diferentemente de todas as outras funções, tanto atômicas quanto complexas, o caminho relativo não faz parte dos parâmetros de entrada da função. Isso ocorre porque o objeto complexo que se deseja encontrar não é utilizado, diretamente, pela função de similaridade. Entretanto ele deve ser o último objeto antes do predicado, para que os caminhos referenciados dentro das funções atômicas, cujo retorno servirá de entrada para a **tupleSim**, sejam relativos a esse objeto. Na figura 4.3, `conferencia` é uma tupla, pois possui diferentes subelementos subordinados a ela (`nome`, `endereco`, `area` e `ano`). É, em

`conferencia`, que deve-se iniciar o predicado, e a função de tupla deve ser aplicada em seus subelementos.

A quantidade de parâmetros dessa função não precisa ser na mesma quantidade dos subelementos do objeto complexo. Assim não existe uma função **subTupleSim** como visto para conjuntos. Ao escrever uma consulta, utilizando a função **tupleSim**, a métrica assume que os argumentos de entrada podem conter parte ou todos os elementos filhos do objeto pesquisado, devolvendo o resultado equivalente. Considere o seguinte exemplo:

Exemplo 9 *Localizar a referência que tenha como autor "Henry Korth", tenha sido publicada por volra do ano "2000" e que contenha, em seu título, a palavra "Consystenci".*

```
/References/Ref[tupleSim(acronymSim(Authors/Author, "Henry Korth"),
                         yearSim(Date/Year, 2000),
                         containsSim(Title, "Consystenci"))>0.7]
```

```
   .
   .
   .
  <Ref>
     <Type>BOOK</Type>
     <Authors>
        <Author>H. Korth</Author>
        <Author>S. Mehrotra</Author>
        <Author>R.Rastogi</Author>
     </Authors>
     <Date>
       <Year>1999</Year>
       <Month>apr</Month>
     </Date>
     <Title>Ensuring Consistency on...</Title>
  </Ref>
</References>
```

Para a consulta do exemplo 9, o processador aplica cada função de elemento atômico para cada elemento especificado, `Authors/Author`, `Date/Year` e `Title`, retornando um escore de similaridade para cada um. Cada escore equivale a um parâmetro *number* da função *tupleSim*. Com os escores retornados, a função *tupleSim* calcula a média aritmética. Nessa consulta, os valores passados através do argumento *node-set* das funções para elementos atômicos são relativos ao elemento `Ref`. Só serão selecionados pelo processador os elementos `Ref` do documento XML que possuírem como filhos, os três elementos solicitados na consulta `Authors/Author`, `Date/Year` e `Title`, e cuja média dos escores de similaridade seja maior do que 0.7.

Essa mesma consulta poderia ter sido escrita, comparando-se cada métrica atômica, diretamente, com o grau de similaridade desejado. Porém o que se deseja é agrupar esses valores, de forma a fornecer um grau de similaridade para o objeto, no intuito de se poder gerar, em uma linguagem mais sofisticada (como a XQuery), uma classificação ordenada com os objetos mais similares. Além disso, se alguma das métricas atômicas retornasse um valor um pouco inferior ao *threshold*, o objeto inteiro seria rejeitado, ocorrendo um falso descarte.

4.3 Expressividade da linguagem SimXPath

Para dar alguns exemplos da expressividade da linguagem, serão apresentadas algumas consultas que podem ser realizadas em SimXpath. O exemplo a seguir mistura várias métricas em uma só consulta:

```
/References/Ref[tupleSim(ngramsListSim(Authors/Author, "Henry
                Korth", "Mehrotra Sharad", "Rastogi Rajeev"),
                containsSim(Title, "Enshuring"),
                dateSim(Date, "15/03/2000"))>0.7]
```

Também é possível realizar comparações com elementos localizados remotamente, conectando-se o seu *namespace*. Para o exemplo em questão, considere-se a declaração do seguinte *namespace*:

```
bib="http://metropole.inf.ufrgs.br/\~{}abernardes/xml\_bibsources",
```

onde pode-se desejar localizar determinados artigos, usando como base um subconjunto de autores da base exemplo. A consulta ficaria da seguinte forma:

```
/bib:References/bib:Ref[ngramsSubSetSim(bib:Authors/bib:Author,
                 /References/Ref[1]/Authors/Author[1]/text(),
                 /References/Ref[1]/Authors/Author[2]/text())>0.8]
```

Desta forma, o primeiro e o segundo autor da base exemplo seriam comparados com todos os conjuntos de autores de cada referência (`Ref`) da base remota. Seriam retornadas apenas aquelas referências cuja similaridade do subconjunto de autores seja maior que 0.8.

4.3.1 Retornando classificação ordenada com XQuery

XPath baseia-se em uma abordagem navegacional para especificar os nodos que serão selecionados. Dada uma expressão, ela navega através de uma árvore XML por meio dos *axes* (filho, descendente, pai, ancestral, anterior/próximo irmão ou atributo), verificando se esses nodos satisfazem as condições do *nodetest* e do filtro contido dentro do predicado, quando for o caso. Sendo assim, XPath é apenas um mecanismo de endereçamento e não uma linguagem completa, como, por exemplo, XML-QL.

Embora SimXPath, devido a suas características e a não possuir comandos de ordenação, não possa retornar os resultados de uma consulta de similaridade na forma de uma classificação ordenada, ela é utilizada como base de outras linguagens como XQuery.

Ao incorporar a SimXPath, poderia-se realizar consultas, utilizando funções de similaridade, e obter os resultados de forma ordenada de acordo com a relevância. Isso é possível, porque o resultado das funções de similaridade é numérico, indicando o grau de similaridade entre os elementos da consulta e os elementos contidos na base. Para o seguinte exemplo:

Exemplo 10 *Localizar os livros que tenha como autor "Henry Korth" e tenham sido publicados por volta do ano "2000".*

```
FOR $ref IN /References/Ref,
LET $sim = tupleSim(ngramsSim($ref/Authors/Author, "Korth Henry"),
                    yearSim($ref/Date/Year, 1998))
WHERE $sim > 0.5
RETURN <Book>$ref/Title,<sim>$sim</sim></Book>
SORTBY $sim DESCENDING
```

Teria-se o seguinte resultado:

```
<Book><Title>Ensuring Consistency on...</Title><sim>0.89</sim></Book>
<Book><Title>Database System Concepts</Title><sim>0.75</sim></Book>
```

5 PROTÓTIPO IMPLEMENTADO

Como já foi dito no capítulo 3, um mecanismo foi utilizado para implementar as funções propostas nesse trabalho. Porém para poder realizar testes e para observar, de modo prático, os resultados obtidos, o mecanismo proposto foi incorporado a um aplicativo que executa uma dada consulta sobre um documento XML e exibe a resposta através de uma interface visual (GUI). O nome desse aplicativo é XPE (acrônimo para XPath Explorer). Esse aplicativo possui o código aberto e pode ser localizado através da URL "http://www.purpletech.com/xpe".

Os documentos XML podem ser acessados por meio da digitação de um URL ou através de uma janela de diálogo para arquivos. Após carregado o documento, basta digitar uma expressão XPath, que a interface irá exibir os elementos ou atributos que casam com a expressão. Além disto, é mostrado o valor da expressão (cadeia de caracteres, número ou booleano), e a árvore XML inteira do documento original com os nodos, resultados da consulta, salientados (figura 5.1).

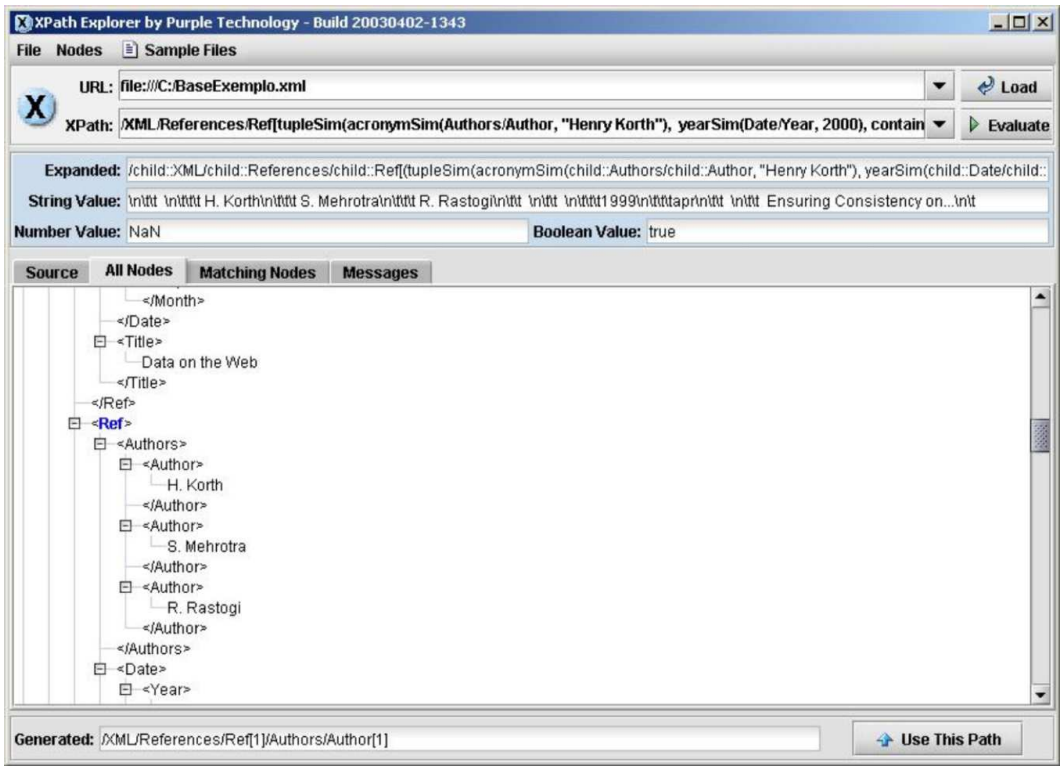

Figura 5.1: Resultado de uma expressão SimXPath.

Outra funcionalidade interessante desse aplicativo é que, ao clicar em um nodo, ele gera, automaticamente, a expressão XPath para aquele nodo. Essa expressão irá aparecer na caixa *Generated*. Esse aplicativo se torna bastante atraente para aqueles que necessitam testar o funcionamento de expressões XPath, ou até mesmo para quem está tendo dificuldades em escrever uma.

A SimXPath pode ser usada também, diretamente, em aplicativos java ou em programas auxiliares em java (jsp, *applets*, etc). Para executar uma expressão XPath sobre um documento XML e extrair os valores dos nodos resultantes da consulta, utilizando um programa java, é bastante simples, como pode ser visto no código abaixo:

```
public static void readResponse(InputStream dadosXML)
 throws IOException, SAXException, TransformerException,
 ParserConfigurationException, JaxenException
{
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);
  DocumentBuilder builder = factory.newDocumentBuilder();

  InputSource data = new InputSource(dadosXML);
  Node doc = builder.parse(data);

  XPath expression = new org.jaxen.dom.DOMXPath(
  "/XML/References/Ref[tupleSim(acronymSim(Authors/Author, "Henry Korth"),
                                yearSim(Date/Year, 2000),
                                containsSim(Title, "Enshuring Consystenci"))>0.7]");

  Navigator navigator = expression.getNavigator();

  List results = expression.selectNodes(doc);
  Iterator iterator = results.iterator();
  while (iterator.hasNext())
  {
    Node result = (Node) iterator.next();
    String value = StringFunction.evaluate(result, navigator);
    System.out.println(value);
  }
}
```

Onde, resumidamente, lê-se um documento XML recebido por intermédio do parâmetro *dadosXML*. Após executar um *parse* sobre o mesmo, é instanciado um objeto DOM (poderia ser também JDOM, dom4j, ou ElectricXML) para uma expressão XPath. Posteriormente, o navegador da XPath irá selecionar os nodos que serão passados para um objeto iterador, que irá percorrer o conjunto dos nodos que fazem parte do resultado, extraindo seus valores. Maiores detalhes podem ser vistos em (CODEHAUS, Acesso em: 30 jun. 2005).

Para ligar um *namespace*, basta adicionar o seguinte comando após a declaração da variável `expression`:

```
expression.addNamespace("bib",
 "http://metropole.inf.ufrgs.br/~abernardes/xml_bibsources");
```

5.1 Implementação

Após o estudo das métricas de similaridade, chegou-se a um impasse de como elas deveriam ser incorporadas a um processador de consultas. A proposta inicial era

de que os objetos de consulta deveriam possuir um atributo que indicasse o tipo de similaridade a ser utilizada. O processador, ao ler esse objeto, identificaria a similaridade e aplicaria a métrica correspondente ao objeto em questão. Porém esse tipo de abordagem foge aos padrões das consultas XPath além de ser necessário uma grande adaptação no processador para entender essa nova forma de consulta. Posteriormente pensou-se em inserir novos comandos à linguagem de consulta, como acontece em (THEOBALD; WEIKUM, 2001), entretanto isso acarretaria na alteração da sintaxe da linguagem bem como no modo de execução do processador.

Após esses estudos, descobriu-se que os dados dos elementos poderiam ser manipulados através de funções. Como esse é um recurso nativo dos processadores XPath e já é previsto na especificação da linguagem (ver (W3C, Acesso em: 23 abr. 2005)), foi decidido extender um processador utilizando funções que incorporassem métricas de similaridade. Desse modo, a extensão da linguagem XPath através do mecanismo Jaxen foi feito em duas partes:

- criação e empacotamento das funções de similaridade e;
- atualização da classe que mapeava as funções.

Primeiramente era necessário localizar o pacote onde estavam localizadas as funções nativas na linguagem XPath. Estudando o mecanismo Jaxen, verificou-se um pacote chamado *Function*, o qual continha as funções nativas linguagem. Descoberto onde deveriam ficar as novas funções foi feita uma análise da inteface que as mesmas deveriam ter. A interface de todas as funções é da seguinte forma:

```
public interface Function {
   Object call(Context context, List args) throws FunctionCallException;
```

Onde o parâmetro *context* é o nodo contexto no ponto da expressão em que a função é chamada e o parâmetro *args* é a lista de argumentos fornecidos durante a chamada da função. Dessa forma, quando uma função de similaridade é chamada, têm-se: i)a métrica de similaridade a ser utilizada que já é intrínseca a própria função; ii)o nodo contexto, ou seja em que ponto da árvore XML está a execução da expressão no momento em que foi chamada a função e; iii)o dado a ser comparado e com qual nodo deve ser comparado. É claro que a informação contida neste último ítem deve ser retirado da lista contida em *args*. Na figura 5.2, pode ser visto a implementação da métrica *Edit Distance*.

Para que o processador Jaxen localizasse as novas funções (as quais foram empacotadas em um pacote chamado *sim* dentro do pacote *function*), foi pesquisado o local em a referência a elas ocorria. Essa referência ocorria dentro da função *FunctionContext*. Essa função implementa o padrão *Singleton* (LARMAN, 2001) e utliza uma variável do tipo *HashMap* para armazenar a referência a todas as funções. A seguir está um trecho do código onde essa variável é inicializada:

```
.
.
.
registerFunction( null,  // namespace URI
                  "sum",
                  new SumFunction() );
```

```
registerFunction( null,  // namespace URI
                  "true",
                  new TrueFunction() );

registerFunction( null,  // namespace URI
                  "translate",
                  new TranslateFunction() );


// register similarity functions

registerFunction( null,  // namespace URI
                  "ngramsSim",
                  new NgramsSimFunction() );

registerFunction( null,  // namespace URI
                    "editSim",
                  new EditSimFunction() );

registerFunction( null,  // namespace URI
                    "acronymSim",
                  new AcronymSimFunction() );
.
.
.
```

Onde *registerFunction* é um método da classe *SimpleFunctionContext* que contém a variável *functions* do tipo *HashMap*. Os parâmetros correspondem ao URI *namespace* da função a ser registrado dentro do contexto, a cadeia de caracteres que identifica a função e o objeto implementado que deve ser usado quando a função for executada. 0 segundo parâmetro deve estar de acordo com o nome previsto na sintaxe da linguagem, pois é esse nome que será extraído da expressão XPath para realizar a busca dentro da variável *functions*.

```
public class EditSimFunction implements Function{

    public Object call(Context context, List args) throws FunctionCallException
    {
        String strMatch1;
        String strMatch2;
        double sim = 0;
        double maior = 0;

        if (args.size() == 2)
        {
            Object obj = (Object) args.get(0);
            if ( args.get(1) instanceof String)
            {
                strMatch2 = (String) args.get(1);
            }
            else return new Double( 0 );

            if (obj instanceof List)  //verifica se o node-set eh lista
            {
            Iterator nodeIter = ((List)obj).iterator();
            //tests the similarity of all elements of the list
            //and keep the biggest in sim
            while (nodeIter.hasNext())
              {
                strMatch1 = StringFunction.evaluate(nodeIter.next(), context.getNavigator());
                maior = evaluate(strMatch1, strMatch2).doubleValue();

                if (maior > sim)
                    sim = maior;
              }
            }
          else //if it is not list it's atomic element
          {
            strMatch1 = StringFunction.evaluate(args.get(0), context.getNavigator());
            sim = evaluate(strMatch1, strMatch2).doubleValue();
          }
            return new Double(sim);
        }
        throw new FunctionCallException("editSim() requires two arguments.");
    }

    public static Number evaluate(String strMatch1, String strMatch2)
    {
        strMatch1 = normalizeText(strMatch1);
        strMatch2 = normalizeText(strMatch2);
        int Custo[][] = new int[strMatch1.length() + 1][strMatch2.length() + 1],i, j;
        for (i = 0; i <= strMatch1.length(); i++)
        {
            Custo[i][0] = i;
        }
        for (j = 0; j <= strMatch2.length(); j++)
        {
            Custo[0][j] = j;
        }
        for (i = 1; i <= strMatch1.length(); i++)
        {
            for (j = 1; j <= strMatch2.length(); j++)
            {
               Custo[i][j] = Math.min(Math.min(Custo[i - 1][j], Custo[i][j - 1]) + 1, Custo[i - 1][j - 1] +
               (strMatch1.charAt(i - 1) == strMatch2.charAt(j - 1) ? 0 : 1));
            }
        }
        if (strMatch1.length() > 0 || strMatch2.length() > 0)
        {
            return new Double(1 - Custo[strMatch1.length()][strMatch2.length()] /
            (double)Math.max(strMatch1.length(), strMatch2.length()));
        }
        else
        {
            return new Double(0.0);
        }
    }

    private static String normalizeText(String str)
    {
        String letters = "qwertyuioplkjhgfdsazxcvbnm1234567890 ";
        int index = 0;
        String strFinal = "";

        // put to lower case
        str = str.toLowerCase();
        for (index = 0; index < str.length() ;index++)
        {
            if (letters.indexOf(str.charAt(index)) != -1)
            {
                strFinal = strFinal + str.charAt(index);
            }
        }
        return (strFinal.trim());
    }
```

Figura 5.2: Código da função de similaridade EditSim

6 CONCLUSÃO E TRABALHOS FUTUROS

A utilização de métricas de similaridade, visando a melhoria dos resultados obtidos em consultas em bases de dados, não é nenhuma novidade. Várias métricas já foram utilizadas tanto em SGBD relacionais, como foi visto em (MOTRO, 1988; GOLDMAN, 1998), como também em documentos XML, conforme foi apresentado em (FLORESCU; KOSSMANN; MANOLESCU, 2000; FUHR; GROSJOHANN, 2001; THEOBALD; WEIKUM, 2001) entre vários outros trabalhos. Porém, até então, nenhum dos trabalhos estudados apresentou um conjunto de métricas que fossem aplicáveis a situações diferentes. Na maioria, era apresentado apenas uma métrica, que poderia ser para nomes próprios ou para abreviaturas e acrônimos entre outros.

Com esse trabalho pretendeu-se apresentar uma proposta que trataria da similaridade na maior parte dos casos. Para tanto, procurou-se tratar de tipos diferentes de dados, como data e números além das representações que uma informação pode assumir em uma cadeia de caracteres (abreviaturas, ordem trocada de palavras, etc.).

Além disso, no intuito de apresentar uma ferramente mais completa, procurou-se encaixar as consultas de similaridade em estruturas complexas, de forma a tratar da similaridade de um objeto inteiro. Dessa forma foram utilizadas estruturas bastante conhecidas, como tuplas, conjuntos e listas. A utilização dessas estruturas possibilitou consultas que, até então, não poderiam ser escritas em XPath, mostrando uma nova abordagem na manipulação de estruturas complexas. Essa abordagem é um paradoxo aos trabalhos que procuram fazer um casamento aproximado da estrutura da consulta, com as estruturas dos objetos encontrados nas bases, como acontece em (BAEZA-YATES; NAVARRO, 2002; SCHLIEDER, 2001). Ela apresenta um casamento estrutural fixo e pré-determinado. Elementos complexos, que não se encaixam na forma estrutural definida pela consulta, não serão selecionados.

Assim, esse trabalho contempla uma ferramenta para consulta de documentos XML, com suporte a argumentos vagos. É importante ressaltar que grande parte da proposta foi baseada no trabalho de Dorneles (DORNELES et al., 2004), sendo a maior contribuição a transformação dos objetos de consulta lá existentes em consultas XPath. O processador utilizado é de código aberto e desenvolvido em java. Isso significa que qualquer um pode estudar as modificações feitas, além de adicionar novas funções, sendo possível executá-lo em qualquer plataforma. O processador utilizado não é um SGBD, portanto, não possui as qualidades intrínsecas deste sistema. Por esse motivo, a inserção de funções de similaridade se torna fácil, pois não há a necessidade de se preocupar com índices, pois não existe um pré-processamento

dos documentos. Os documentos XML, normalmente, são carregados e consultados sob demanda. Também não foi alterado o plano de consulta, como acontece em (SCHLIEDER, 2001), e nem foi necessária a alteração das bases a serem consultadas, com a atribuição de pesos aos elementos mais significativos como acontece em (COHEN, 2003) e (GUO, 2003).

No entanto, ainda existe muito a ser melhorado. A métrica de similaridade *Levenstein* ou *edit distance* já é consagrada na comunidade científica, porém o seu campo de atuação é um tanto limitado. Para simples comparação entre cadeias de caracteres que possuam apenas pequenos erros tipográficos, ou no auxilio a consultas em que o usuário demonstra incerteza no modo como se escreve a informação, ela se mostrou bastante eficiente. Porém, como métrica de casamento de cadeias de caracteres, na tentativa de identificar se dois objetos provenientes de instâncias vindas de bases heterogêneas são similares, ela mostrou algumas deficiências. Como visto em (CHAUDHURI, 2003) e (GRAVANO, 2003), essa métrica possui falhas ao tentar casar cadeias de caracteres com mais de um *token*, pois um pequeno desarranjo entre eles pode fazer com que a métrica retorne como falso um casamento positivo, ou como verdadeiro um casamento negativo. Também no caso de abreviaturas e acrônimos, ela demonstrou problemas. A técnica de *Q-grams*, aplicada em (GRAVANO, 2001b), demonstrou ser bastante eficiente nos caso de trocas de ordem entre as palavras e até em abreviaturas, porém não em acrônimos.

Fica visível a necessidade de elaborar uma métrica que possua um espectro mais amplo, a fim de cobrir todos os casos, sem, no entanto, elevar os custos computacionais. Nesse caso, podem ser feitos estudos que levem em consideração a utilização de ontologias ou de técnicas que utilizam o modelo vetorial.

Outro problema diz respeito ao *threshold*. Nesse trabalho foram considerados *thresholds* aleatórios, sendo que as consultas a serem formuladas pelos usuários necessitam de *thresholds* adequados. Porém o que é um *threshold* adequado? Um valor muito baixo pode trazer uma quantidade muito grande de respostas, enquanto um muito alto pode deixar de fora resultados úteis. Na grande maioria dos trabalhos estudados, não é a-presentada uma solução para esse problema, sendo, apenas em alguns, apresentada uma tentativa de melhorar o *threshold* fornecido pelo usuário.

Ainda há a necessidade de melhorar a atribuição de valores às métricas complexas, principalmente na métrica para o cálculo de tuplas. O simples cálculo da média não leva em consideração a relevância de determinados elementos dentro da tupla. Esses elementos poderiam servir como identificadores do objeto e, portanto, deveriam possuir um maior peso no momento do cálculo. A própria forma de escrever a consulta poderia ser melhorada, podendo haver o reconhecimento automático da similaridade a ser utilizada para cada tipo de dado.

No entanto a SimXPath pode se tornar uma ferramenta de grande valia nas mãos de usuários experimentados. Ela pode ser usada para auxiliar na construção de sítios na web, que propiciem aos usuários consultas mais flexíveis, ou para ser incorporada a linguagens de consulta mais completas como XQuery ou XSLT, fornecendo um suporte àqueles usuários que possuem dúvidas nos argumentos que serão utilizados para formular uma consulta.

REFERÊNCIAS

APOSTOLICO, A.; GUERRA, C. The Longest Common Subsequence Problem Revisited. **Algorithmica**, [S.l.], v.2, p.316–336, 1987.

BAEZA-YATES, R. A.; NAVARRO, G. XQL and proximal nodes. **Journal of the American Society for Information Science and Technology (JASIST)**, [S.l.], v.53, n.6, p.504–514, 2002.

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

BAUM, P. **Date Algorithms**. 1998. Disponível em: <http://vsg.cape.com/ pbaum/date/date0.htm>. Acesso em: 30 jun. 2005.

CHAUDHURI, S. et al. Robust and efficient fuzzy match for online data cleaning. In: SIGMOD CONFERENCE, 2003, San Diego, California. **Proceedings...** [S.l.: s.n.], 2003. p.313–324.

CIACCIA, P.; PENZO, W. Adding Flexibility to Structure Similarity Queries on XML Data. In: INTERNATIONAL CONFERENCE ON FLEXIBLE QUERY ANSWERING SYSTEMS, FQAS, 2002, Copenhagen, Denmark. **Proceedings...** [S.l.: s.n.], 2002.

CIACCIA, P.; PENZO, W. Relevance Ranking Tuning for Similarity Queries on XML Data. In: WORKSHOP EEXTT AND CAISE 2002 WORKSHOP DTWEB ON EFFICIENCY AND EFFECTIVENESS OF XML TOOLS AND TECHNIQUES AND DATA INTEGRATION OVER THE WEB, VLDB, 2002. **Proceedings...** Berlin: Springer-Verlag, 2002. p.22–34. (Lecture Notes in Computer Science, v.2590).

CODEHAUS. **Jaxen XPath Implementation**. 2003. Disponível em: <http://jaxen.org/>. Acesso em: 30 jun. 2005.

COHEN, S. et al. XSEarch: a semantic search engine for xml. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2003, Berlin, Germany. **Proceedings...** [S.l.]: Morgan Kaufmann, 2003. p.45–56.

COHEN, W.; RAVIKUMAR, P.; FIENBERG, S. A Comparison of String Metrics for Matching Names and Records. In: KDD-2003 WORKSHOP ON DATA CLEANING AND OBJECT CONSOLIDATION, 2003, Washington, DC. **Proceedings...** [S.l.: s.n.], 2003. p.13–18.

DAS, G. et al. Episode Matching. In: ANNUAL SYMPOSIUM ON COMBINATORIAL PATTERN MATCHING, CPM, 1997. **Proceedings...** Berlin: Springer-Verlag, 1997. p.12–27.

DEUTSCH, A. et al. A query language for XML. **Computer Networks**, [S.l.], v.31, n.11–16, p.1155–1169, 1999.

DORNELES, C. F.; HEUSER, C. A.; LIMA, A. E. N.; SILVA, A. S. da; MOURA, E. S. de. Measuring similarity between collection of values. In: ACM INTERNATIONAL WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT, WIDM, 2004, Washington DC, USA. **Proceedings...** New York: ACM Press, 2004. p.56–63.

DORNELES, C. F.; LIMA, A. E.; HEUSER, C.; SILVA, A. da; MOURA, E. de. **Acessing XML data by allowing imprecise query arguments**. [S.l.]: UFRGS, 2004. (RP-342).

FLORESCU, D.; KOSSMANN, D.; MANOLESCU, I. Integrating keyword search into XML query processing. In: WORLD WIDE WEB CONFERENCE ON COMPUTER NETWORKS, 2000, Amsterdam, The Netherlands. **Proceedings...** Netherlands: North-Holland Publishing Co., 2000. p.119–135.

FRENCH, J. C.; POWELL, A. L.; SCHULMAN, E. Applications of Approximate Word Matching in Information Retrieval. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, CIKM, 1997, Las Vegas, Nevada. **Proceedings...** New York: ACM Press, 1997. p.9–15.

FUHR, N.; GROSJOHANN, K. XIRQL: a query language for information retrieval in XML documents. In: INTERNATIONAL ACM SIGIR CONFERENCE ON RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL, NEW ORLEANS, USA, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.172–180.

GILES, C. L.; BOLLACKER, K. D.; LAWRENCE, S. CiteSeer: an automatic citation indexing system. In: ACM INTERNATIONAL CONFERENCE ON DIGITAL LIBRARIES, DL, 1998, Pittsburgh, PA, USA. **Proceedings...** New York: ACM Press, 1998. p.89–98.

GOLDMAN, R. et al. Proximity Search in Databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1998, New York, USA. **Proceedings...** San Francisco: Morgan Kaufmann, 1998. p.26–37.

GRAVANO, L. et al. Using q-grams in a DBMS for Approximate String Processing. **IEEE Data Engineering Bulletin**, [S.l.], v.24, n.4, p.28–34, Dec. 2001.

GRAVANO, L. et al. Approximate String Joins in a Database (Almost) for Free. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2001, San Francisco, CA, USA. **Proceedings...** [S.l.]: Morgan Kaufmann, 2001. p.491–500.

GRAVANO, L. et al. Text joins in an RDBMS for web data integration. In: WORLD WIDE WEB, 2003, Budapest, Hungary. **Proceedings...** New York: ACM Press, 2003. p.90–101.

GUO, L. et al. XRANK: ranked keyword search over xml documents. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2003, San Diego, California. **Proceedings...** New York: ACM Press, 2003. p.16–27.

GUTH, G. Surname Spellings and Computerizaed Record Linkage. **Historical Methods Newsletter**, [S.l.], v.10, n.1, p.10–19, 1976.

JACCARD, P. The distribution of flora in the alpine zone. **The New Phytologist**, [S.l.], v.11, n.2, p.37–50, 1912.

JARO, M. Probabilistic Linkage of Large Public Health Data Files. **Statistics in Medicine**, [S.l.], v.14, n.(5-7), p.491–498, 1995.

JIN, L.; LI, C.; MEHROTRA, S. **Efficient Similarity String Joins in Large Data Sets**. [S.l.]: UCI ICS, 2002. (TR-DB-02-04).

LAIT, A.; RANDELL, B. **An Assessment of Name Matching Algorithms**. [S.l.]: Department of Computing Science of University of Newcastle upon Tyne, 1993. Technical report.

LALMAS, M.; RöLLEKE, T. **Modelling Vague Content and Structure Querying in XML Retrieval with a Probabilistic Object-Relational Framework**. Berlin: Springer-Verlag, 2004. p.432 – 445 (Lecture Notes in Computer Science, v.3055).

LAPP, J.; ROBIE, J.; SCHAC, D. XML query language (XQL). In: THE QUERY LANGUAGES WORKSHOP, QL, 1998, Boston, USA. **Proceedings...** [S.l.: s.n.], 1998.

LARMAN, C. **Applying UML and Patterns**: an introduction to object-oriented analysis and design. 2nd ed. Upper Sadle River: Prentice Hall, 2001.

LAWRENCE, S.; GILES, C. L.; BOLLACKER, K. Digital Libraries and Autonomous Citation Indexing. **IEEE Computer**, [S.l.], v.32, n.6, p.67–71, 1999.

LEVENSHTEIN, V. Binary codes capable of correcting spurious insertions and deletions of ones. **Problems of Information Transmission**, [S.l.], v.1, n.1, p.8–17, 1965.

LIMA, A. E. N. **Similaridade em elementos XML**. [S.l.]: UFRGS, 2002. Trabalho de conclusão (graduação/Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MONGE, A. E.; ELKAN, C. The Field Matching Problem: algorithms and applications. In: KNOWLEDGE DISCOVERY AND DATA MINING, KDD, 1996. **Proceedings...** [S.l.: s.n.], 1996. p.267–270.

MOTRO, A. VAGUE: a user interface to relational datbases that permits vague queries. **ACM Transactions on Office Information Systems**, [S.l.], v.6, n.3, p.187–214, July 1988.

NAVARRO, G. A guided tour to approximate string matching. **ACM Computing Surveys**, [S.l.], v.33, n.1, p.31–88, 2001.

NAVARRO, G. et al. Indexing Methods for Approximate String Matching. **IEEE Data Engineering Bulletin**, [S.l.], v.24, n.4, p.19–27, 2001.

NAVARRO, G.; BAEZA-YATES, R. A. Proximal Nodes: a model to query document databases by content and structure. **Information Systems**, [S.l.], v.15, n.4, p.400–435, 1997.

NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. **Journal of Molecular Biology**, [S.l.], v.48, p.444–453, 1970.

RIJSBEGEN, C. J. van. **Information Retrieval**. 2nd ed. [S.l.]: Butterworths, 1979.

SANKOFF, D.; KRUSKAL, J. **Time Warps, String Edits, and Macromolecules**: the theory and practice of sequence comparison. [S.l.]: Addison-Wesley, 1983.

SCHLIEDER, T. Similarity search in XML data using cost-based query transformations. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2001, Santa Barbara, USA. **Proceedings...** [S.l.: s.n.], 2001.

SUTINEN, E.; TARHIO, J. On Using q-Gram Locations in Approximate String Matching. In: ANNUAL EUROPEAN SYMPOSIUM, ESA, 1995, Corfu, Greece. **Proceedings...** [S.l.: s.n.], 1995. p.327–340. (Lecture Notes in Computer Science, v.979).

THEOBALD, A.; WEIKUM, G. **Adding Relevance to XML**. [S.l.: s.n.], 2001. (Lecture Notes in Computer Science, v.1997).

UKKONEN, E. Approximate String Matching with q-grams and Maximal Matches. **Theoretical Computer Science**, [S.l.], v.92, n.1, p.191–211, 1992.

ULLMANN, J. R. Binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words. **Computer Journal**, [S.l.], v.20, n.2, p.141–147, 1977.

W3C. **XML Path Language (XPath) 2.0 Working Draft**. 2003. Disponível em: <http://www.w3.org/TR/2003/WD-xpath20-20031112(12.nov.2003)>. Acesso em: 17 mar. 2005.

W3C. **XML Path Language (XPath) 1.0 Recomendation**. 1999. Disponível em: <http://www.w3.org/TR/xpath(16.nov.1999)>. Acesso em: 23 abr. 2005.

WINKLER, W. E. **The state of record linkage and current research problems**. Wachington, DC: Statistical Research Division, U.S. Bureau of the Census, 1999. (R99/04).

YIANILOS, P. N.; KANZELBERGER, K. G. **The LikeIt Intelligent String Comparison Facility**. [S.l.]: NEC Research Institute, 1997. Technical report.