UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

XML-PM : Um Método Eficiente para Identificação de Padrões no Processamento de Consultas a Dados XML

Elaine Sampaio Pinho de Castro

Prof. Dr. Javam de Castro Machado
orientador

Fortaleza, Janeiro de 2006

XML-PM : Um Método Eficiente para Identificação de Padrões no Processamento de Consultas a Dados XML

Elaine Sampaio Pinho de Castro
Universidade Federal do Ceará, Fortaleza, 2006.

Dissertação submetida para obtenção do grau de MESTRE em Ciência da Computação.

Orientador: Prof. Dr. Javam de Castro Machado

Aceitamos esta dissertação de acordo com os padrões exigidos.

Ana Carolina Salgado

Cláudia Linhares Sales

Javam de Castro Machado

Ao meu amado filhinho.

Agradecimentos

Agradeço em primeiro lugar a Deus, que me concedeu força, disciplina e motivação para completar esta jornada.

A toda a minha família. Em especial, aos meus pais, Jonas e Helena, meus irmãos, Daniel e Eveline, e minha vó, Benedita, por todo o amor e encorajamento durante toda a minha vida. Ao meu amado esposo, Marcos, por todo seu carinho, compreensão e apoio, principalmente nos momentos mais difíceis. Ao meu filhinho, Marcos Vinícius, por sua paciência. Aos meus sogros, Francy e Antônio, por toda ajuda que me deram no decorrer desta conquista.

A todos os professores da Universidade Federal do Ceará que contribuíram com minha formação acadêmica.

Aos professores que fizeram parte da banca examinadora, pelas valiosas sugestões e contribuições a esta tese. Em especial, ao meu orientador, Javam, por toda sua confiança e incentivo dedicados a mim.

A todos os amigos estudantes do Mestrado em Ciência da Computação, por todos os grupos de estudo que formamos e pela ótima convivência e companheirismo.

À FUNCAP pelo apoio financeiro.

Resumo

Em decorrência do volume de informações trocadas através de documentos XML, surge a necessidade de manipular estes documentos em sistemas de bancos de dados. No entanto, devido à natureza heterogênea dos dados XML, bancos de dados tradicionais não têm se mostrado adequados para essa tarefa. Esse fato tem despertado o interesse da comunidade acadêmica no desenvolvimento de técnicas voltadas ao tratamento específico de tais dados.

Sistemas de Gerenciamento de Bancos de Dados XML Nativos (SGBDXNs), são sistemas de gerenciamento de bancos de dados nos quais documentos XML são armazenados segundo uma estrutura lógica de grafo (ou árvore, caso este não contenha ciclos), onde os nós representam elementos e atributos e as arestas definem os relacionamentos elemento/sub-elemento ou elemento/atributo.

Uma consulta sobre documentos XML pode ser expressa por meio de uma árvore com anotações que estabelecem restrições com relação ao conteúdo ou à estrutura dos elementos desejados. Uma operação essencial na fase de avaliação de uma consulta consiste em achar todas as concordâncias distintas entre a árvore proveniente da consulta e a base de dados analisada. Este trabalho propõe o algoritmo XML-PM como um mecanismo eficiente para essa tarefa, dispensando o uso de decomposições, o que evita a necessidade de combinações, além de permitir que um reduzido número de resultados inválidos sejam montados. A principal desvantagem de trabalhos similares a este está na produção de resultados intermediários enormes frente aos finais, ou ainda ao fato de necessitarem de repetidas combinações de alguns sub-padrões. A fim de prover maior eficiência, índices sobre expressões de caminho simples são utilizados e a travessia do documento é realizada de forma *bottom-up*. A única restrição é que os elementos buscados não possuam sub-elementos com mesmo rótulo, o que não acarreta grandes empecilhos, pois, na prática, tal situação mostra-se pouco freqüente.

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 3

1.3 Organização do documento 3

2 Processamento de Consultas e XML **5**

2.1 Processamento de Consultas 5

2.2 Processamento de Consultas sobre Dados XML 14

2.2.1 Linguagens de Consulta XML 20

2.3 Consulta e Casamento de Padrões sobre Dados XML 23

2.4 Trabalhos Relacionados 24

2.5 XML-PM 28

2.6 Conclusões 29

3 Contexto da Abordagem **30**

3.1 Arquitetura Geral 30

3.2 Servidor de Páginas 31

3.3 Gerenciador de Armazenamento 31

3.3.1 Módulo de Armazenamento 31

3.3.2 Módulo de Indexação 34

3.4 Processador de Consultas 36

4 Árvore Genérica de Consulta **39**

4.1 GTPs Básicas 40

4.2 GTPs 43

4.2.1 Junção 43
4.2.2 Agrupamento, Agregação e Quantificadores 43
4.2.3 Consultas Aninhadas 45
4.3 Geração dos Planos Lógico e Físico 47
4.3.1 Traduzindo XQuery para GTP's 47
4.3.2 Traduzindo uma GTP para um Plano Físico 50
4.4 Conclusões 54

5 Identificação de Padrões sobre Árvores XML 55

5.1 Casamento de Padrões sobre Dados XML 55
5.2 Algoritmo XML-PM 56
5.2.1 Análise da Complexidade 64
5.2.2 Análise da Corretude 69
5.3 Experimentos 72
5.4 Conclusões 77

6 Conclusões 78

6.1 Trabalhos Futuros 79

Lista de Figuras

2.1 Etapas do Processamento de Consultas 7
2.2 Plano Lógico x Plano Físico 8
2.3 Exemplo de Consulta e sua representação simplificada 25

3.1 Arquitetura Geral do FoX 31
3.2 Documento XML 32
3.3 Árvore Correspondente ao Documento XML, Armazenado no FoX 33
3.4 Árvores Lógica e Física do Gerente de Armazenamento do FoX . 34
3.5 Arquitetura do Módulo de Indexação do FoX 35
3.6 *Layout* do Índice FoX 36
3.7 Arquitetura Processador de Consultas FoX 37
3.8 Gramática para um sub-conjunto da XQuery. 37

4.1 Consulta XQuery e sua consulta GTP correspondente. 41
4.2 Extensão da álgebra booleana para tratar o valor verdade *indefinido*. 42
4.3 Uma consulta com junção e sua GTP correspondente. 44
4.4 Uma consulta com quantificação universal e sua GTP correspondente. 44
4.5 Uma consulta envolvendo aninhamento e junção e sua GTP correspondente. 46
4.6 Algoritmo GTP. 49
4.7 Algoritmo para Tradução de uma GTP para um Plano de Execução. 52

5.1 Algoritmo para casamento de padrões 58
5.2 *Casamento de Padrões.* 60
5.3 Algoritmo para montagem dos resultados 63
5.4 GTP Básica 64
5.5 Plano Físico Gerado para a consulta da figura 5.4 usando XML-PM. 64
5.6 Plano Físico Gerado para a consulta da figura 5.4 usando junções estruturais. 65
5.7 Documento XML 66
5.8 Representaçao Gráfica das Consultas 73
5.9 $|$Entrada$|$ $\times$ $|$Acessos$|$ $\times$ $|$Saída$|$ para Q1 75
5.10 $|$Entrada$|$ $\times$ $|$Acessos$|$ $\times$ $|$Saída$|$ para Q2 75
5.11 $|$Entrada$|$ $\times$ $|$Acessos$|$ $\times$ $|$Saída$|$ para Q3 76
5.12 $|$Entrada$|$ $\times$ $|$Acessos$|$ $\times$ $|$Saída$|$ para Q4 76

Lista de Tabelas

2.1 Álgebra Lógica 13
2.2 Quadro comparativo de SGBDs XML Nativos 19

5.1 Consultas 73
5.2 Documentos 74

Capítulo 1

Introdução

1.1 Motivação

Em decorrência da ampla adoção da XML (*eXtensible Markup Language [22]*) como formato para troca de informações, surge a necessidade de manipulação destes dados em sistemas de gerenciamento de bancos de dados (SGBDs). No entanto, SGBDs tradicionais, em especial, relacionais, mostram-se pouco adequados para esta tarefa, pois possuem modelos de dados rígidos, o que vai de encontro à natureza semi-estruturada intrínseca ao modelo XML. A primeira desvantagem encontrada com isso é que uma camada responsável pela adaptação dos dados XML ao formato em questão é adicionada ao tratamento destes. Além disso, o fato de existirem técnicas de processamento de consultas maduras para o tratamento de dados relacionais não necessariamente implica que dados XML armazenados em relações também serão manipulados de forma eficiente.

Como exemplos de trabalhos que fazem uso de mapeamento temos [28], [27], [21], [43], [38]. No entanto, em geral, o mapeamento XML - relacional resulta em uma representação desnormalizada ou em um grande número de tabelas, o que nos leva freqüentemente a um grande número de junções, mesmo considerando um esquema XML simples, pois, freqüentemente, a informação estrutural de um documento XML é modelada por junções no mundo relacional. Além disso, mesmo ao obter um esquema relacional adequado, outro problema

surge: a estrutura dos dados pode evoluir rapidamente e, assim, tornar difícil a manutenção deste esquema relacional de forma correta e consistente. Mais ainda, a fase de otimização de consultas não considera características semânticas próprias de dados XML.

Outro problema identificado é a geração de campos nulos. Como um mesmo sub-elemento pode ocorrer múltiplas vezes em um elemento, em algumas técnicas de mapeamento, o esquema relacional gerado deve possuir tantas colunas quanto for o maior número de ocorrências desse sub-elemento em seu elemento pai.

Já outras abordagens armazenam o conteúdo XML em arquivos planos. Apesar dessa técnica ser bastante adequada para uma variedade de aplicações e ser a forma mais simples, essa estratégia sofre de limitações tais como rapidez no acesso e exploração da estrutura dos documentos. Esse mecanismo não oferece flexibilidade quanto a modificações do conteúdo, bem como, pode apresentar sérios problemas em relação ao seu tamanho.

Dessa maneira, podemos perceber que, a fim de obter melhores resultados no desempenho de consultas sobre massas de dados XML, técnicas de gerenciamento precisam ser desenvolvidas especialmente para este propósito, eliminando assim fases intermediárias e considerando as caracteríticas semi-estruturais desse formato. Diversos trabalhos têm sido desenvolvidos com o propósito de elaborar uma álgebra própria para dados XML, dentre os quais podemos destacar TAX - *Tree Algebra for XML* - [29] que provê uma álgebra capaz de expressar um grande subconjunto da XQuery [35].

Documentos XML, são, em geral, representados através de árvores (ou grafos, caso contenham ciclos). Além disso, para muitas linguagens de consulta propostas atualmente, inclusive a XQuery [35], também as consultas podem ser representadas através árvores com anotações, capazes de capturar as estruturas buscadas na massa de dados. A semântica de tais consultas é capturada pela noção de casamento de padrões. Dessa forma, grande tem sido o interesse da comunidade acadêmica em desenvolver estratégias capazes de obter todas as concordâncias distintas entre a árvore proveniente da consulta

e a base de dados analisada. No entanto, essa tarefa ainda se mostra bastante desafiadora, pois mesmo técnicas mais sofisticadas possuem alguns problemas, em especial, no tocante ao exagerado número de decomposições-combinações, além do tamanho, ainda grande, dos resultados intermediários frente aos finais.

1.2 Objetivos

Neste trabalho, propomos uma técnica de casamento de padrões sobre documentos XML. Tendo em vista a flexibilidade estrutural intrínseca ao modelo XML, o padrão buscado é uma árvore com anotações que permite capturar a natureza heterogênea dos dados em questão. Além disso, através de uma travessia *bottom-up* e do uso de índices sobre expressões de caminho simples, o algoritmo desenvolvido é capaz de recuperar as sub-estruturas requisitadas em apenas uma execução. A única restrição existente é que os elementos solicitados na árvore de consulta não podem conter sub-elementos com mesmo rótulo. Um segundo objetivo é a realização de experimentos capazes de comprovar a corretude do algoritmo, bem como sua complexidade.

1.3 Organização do documento

No capítulo 2 é dada uma introdução ao problema de processamento de consultas, mostrando seu funcionamento e as principais fases que o constituem. Em especial, características relacionadas ao tratamento de dados XML são abordadas. As principais dificuldades encontradas, bem como, abordagens propostas são identificados e analisados. Mais especificamente, trabalhos relacionados ao problema de identificação de padrões sobre documentos XML são examinados e uma solução para esta tarefa é proposta.

O sistema FoX [7] é apresentado como ambiente de experimentação de técnicas de manipulação de massas de dados XML no capítulo 3. O capítulo 4 aborda alguns conceitos que servem de alicerse para a solução proposta neste trabalho. No capítulo seguinte, a estratégia desenvolvida no presente trabalho,

para resolver a tarefa de identificação de padrões de consulta sobre documentos XML é descrita, bem como seus experimentos e análises de tempo e corretude. Finalmente, uma discussão sobre as conclusões obtidas e trabalhos futuros são apresentados no capítulo 6.

Capítulo 2

Processamento de Consultas e XML

Este capítulo destina-se a familiarizar o leitor com as etapas que envolvem o processamento de uma consulta, bem como identificar os principais obstáculos encontrados na manipulação de dados XML neste processo.

A seção 2.1 descreve como funciona o módulo Processador de Consultas, as fases que o constituem e principais operações desenvolvidas. Além disso, introduz brevemente algumas das dificuldades encontradas no tratamento de dados XML. Já a seção 2.2 apresenta algumas soluções propostas para este fim, bem como as principais linguagens de consulta utilizadas, em especial, a XQuery [35]. A noção de casamento de padrões é apresentada na seção 2.3, como uma idéia chave no tratamento de tais dados. Alguns dos principais trabalhos propostos são expostos na seção 2.4. O algoritmo XML-PM, alvo principal do presente trabalho, tem sua apresentação inicial na seção 2.5. O capítulo é encerrado com a seção 2.6, onde tem-se um breve resumo das principais conclusões obtidas.

2.1 Processamento de Consultas

Ao ser submetida ao sistema de banco de dados, uma consulta posta pelo usuário precisa passar por diversas etapas até que seu resultado possa ser

retornado. A esta seqüência de etapas dá-se o nome de *Processamento da Consulta* [25]. A tarefa de processar uma consulta pode ser dividida em duas fases: *compilação* e *execução*.

A fase de compilação pode ser dividida nos seguintes passos:

- *reconhecimento da consulta*: nesse passo é realizada a análise sintática e semântica dos termos que constituem a consulta, ou seja, aqui verifica-se se a consulta está escrita corretamente e se todas as suas referências são válidas. Uma árvore de expressões da consulta é criada e passada como entrada para o próximo passo.

- *geração do plano lógico*: criar o plano lógico envolve a escolha dos operadores algébricos, bem como a disposição (ordem de avaliação) desses na árvore lógica da consulta.

- *geração do plano físico*: envolve a escolha e ordenação dos métodos de execução correspondentes ao plano lógico gerado anteriormente. Operadores que não possuem correspondentes no plano lógico também podem ser inseridos. Este é o caso, por exemplo, quando a consulta do usuário faz restrições a respeito da disposição dos dados na saída.

As etapas de geração do plano lógico e geração do plano físico constituem o otimizador, que faz uso de metadados e estimativas de custo para auxiliar no desenvolvimento de suas tarefas. É tarefa do otimizador decidir em que seqüência as operações devem ser realizadas e os métodos pelos quais tais operações serão executadas. Uma vez otimizado o plano físico da consulta, o SGBD pode partir para a fase de execução, que consiste de uma aplicação de forma *bottom-up* dos operadores da árvore que formam este plano físico. Nessa fase os métodos selecionados são acionados e interagem com a base de dados a fim de obter o resultado da consulta. Na figura 2.1 temos uma representação gráfica das fases que constituem o processador de consultas.

O Módulo de Execução de consultas é basicamente composto por um conjunto de funções que implementam os algoritmos dos operadores físicos da

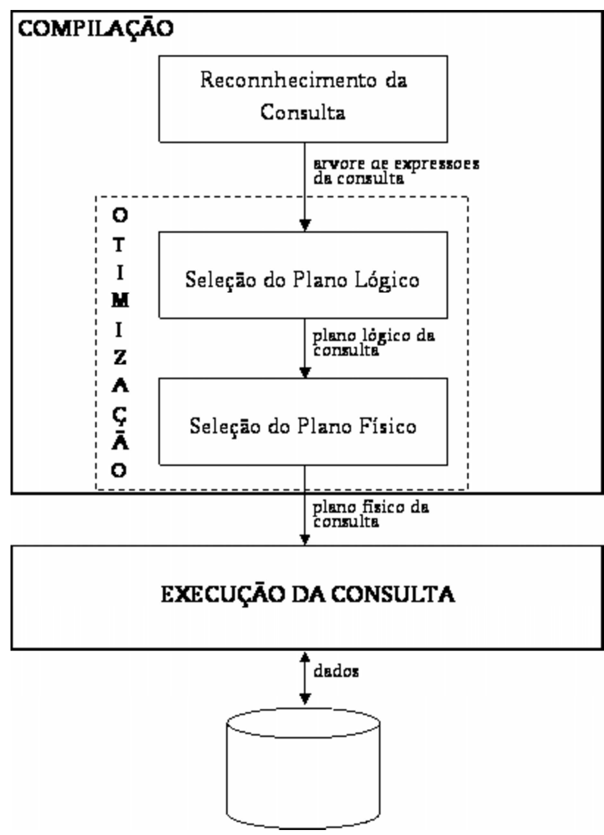
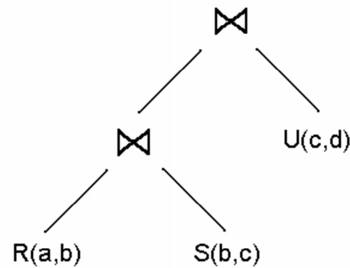
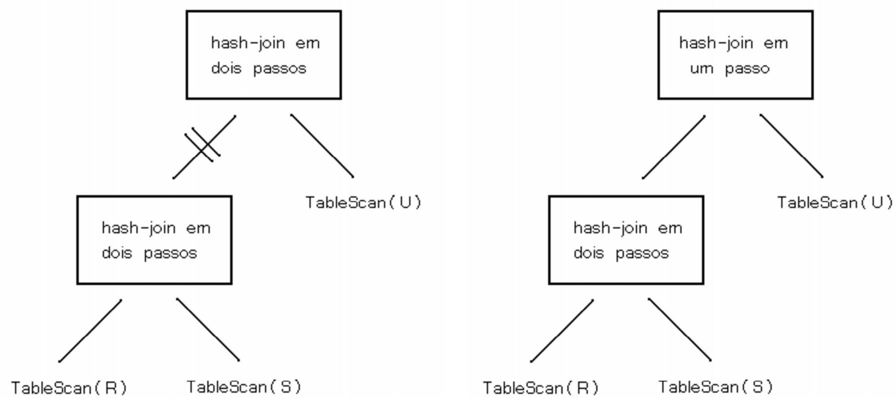


Figura 2.1: Etapas do Processamento de Consultas

álgebra, métodos de acesso aos dados, restrições físicas (tais como, ordenação dos resultados) e mecanismos de sincronização e comunicação entre esses [24]. A etapa de execução da consulta corresponde à fase em que as funções de implementação dos algoritmos dos operadores são acionadas para a execução da mesma. Essas implementações são responsáveis pela manipulação do banco de dados. A execução é iniciada a partir do plano de execução de consulta, também chamado de plano físico de consulta. Podemos definir o plano de execução de consulta como uma árvore algébrica de operadores, com anotações nos seus nós indicando os métodos de acesso para recuperação dos operandos e as implementações dos algoritmos que serão utilizadas [34].



(a) Plano Lógico



(b) Plano Físico A (c) Plano Físico B

Figura 2.2: Plano Lógico x Plano Físico

Exemplo: Considere a expressão de consulta *(R(a,b)⋈S(b,c))⋈U(c,d)*, onde sabe-se possuir as seguintes características a respeito de seu processamento:

1. R ocupa 5000 páginas do disco; S e U, 10000, cada um.

2. O resultado intermadiário de $R \bowtie S$ ocupa k páginas, para algum k.

3. As duas junções serão implementadas usando *hash-join*, em um passo ou dois dependendo do tamanho de k. Em operações binárias, quando algum dos operandos cabe na memória, apenas os elementos do operando restante, no caso, U, precisam ser recuperados diretamente do disco para que a operação seja realizada, algoritmos desenvolvidos para estas situações são conhecidos como *algoritmos de um passo*. Já para os casos onde os dois parâmetros não cabem na memória, é necessário, para ambos os argumentos, um passo anterior à operação em questão, uma ordenação ou a aplicação de uma função de *hash* sobre algum atributo, por exemplo. Algoritmos com tal estratégia são ditos *algoritmos de dois passos*.

Seja 2.2(a) o plano lógico correspondente à consulta especificada acima. As figuras 2.2(b) e 2.2(c) mostram duas opções de planos físicos para este, sendo: em 2.2(b) primeira junção é realizada através de um *hash-join* em dois passos, seu resultado materializado e, novamente, um *hash-join* em dois passos é usado, agora, para a segunda junção. Em contraste, em 2.2(c), tem-se uma opção para os casos em que o resultado de $R \bowtie S$ é, suficientemente, pequeno. A junção de $R \bowtie S$ com U pode ser realizada com apenas um passo e o fluxo entre estes pode ser *pipelined*, uma vez que, sendo k pequeno, tem-se blocos de memória disponíveis.

Os operadores da álgebra lógica definem a semântica das operações. Por se tratar de uma álgebra já madura, como exemplo de álgebra lógica, descreveremos a seguir os operadores que formam a álgebra relacional. A tabela 2.1 mostra uma lista contendo as principais operações desta álgebra, suas descrições, bem como as notações comumente utilizadas para estas. Considere R e S coleções de dados, onde o termo coleção é utilizado para identificar o universo de dados a ser analisado, podendo, inclusive, ser substituído por tabela.

- ***Operações sobre Conjuntos***

  São as operações de *união*, *interseção* e *diferença*, cuja semântica é ins-

pirada na teoria dos conjuntos [1]. No entanto, vale ressaltar, que não obrigatoriamente, a coleção de dados analisada forma um conjunto, pois, em tais coleções, pode acontecer a ocorrência de um mesmo dado diversas vezes. Dessa forma, a operação de *eliminação de duplicatas* se faz necessária a fim de evitar redundâncias.

- ***Seleção*** ($\sigma_{\text{condição de seleção}}$R)

  Esse operador seleciona itens de dados que satisfazem a um determinado predicado.

- ***Projeção*** ($\pi_{\text{subconjunto do esquema}}$)

  Seleciona itens de dados cuja estrutura pertence ao subconjunto especificado do esquema.

- ***Produto*** (R $\times$ S)

  Dadas duas coleções de entrada, constrói uma nova coleção através do produto cartesiano de seus dados.

- ***Junção*** (R $\bowtie_{\text{condição de junção}}$S)

  Existem vários tipos de junções, como demonstra a tabela 2.1, dentre as quais a mais comum é a junção natural. De forma simplificada, esse operador realiza um produto seguido de uma seleção sobre os itens que formam a condição de junção, algebricamente, tem-se $\pi_L(\sigma_C(R \times S))$, onde C é uma condição e L é uma lista contendo uma união dos esquemas de R e S (sem duplicações de nomes). O resultado dessa operação é uma coleção contendo as combinações de R e S que satisfazem à condição de junção.

- ***Agrupamento e Funções de Agregação*** $_{\text{itens de agrupamento}}\Im_{\text{lista de funções}}$R

  Usados para expressar funções de agregação matemática sobre coleções de valores da base de dados. Funções comuns aplicadas a coleções de valores numéricos são: SOMA, MÉDIA, MÁXIMO e MÍNIMO. Além dessas,

[1] `Considere R' e S' os esquemas das coleções R e S. X ⊆ S ∩ R e Y ⊂ R.`

a função de CONTAGEM é usada para contar itens ou valores. A lista *itens de agrupamento* é usada para especificar requisições que envolvem agrupamento dos itens aplicando uma função de agregação para cada grupo. Já a *lista de funções* é uma lista de pares <*função, item*>, onde *função* é uma das funções citadas anteriormente e *item* é um dos itens contidos na lista *itens de agrupamento*.

Em geral, cada operador definido na álgebra lógica possui uma ou mais implementações, os operadores físicos. Porém, existem operadores físicos que não têm correspondente nos planos lógicos, sendo específicos para outras tarefas que não são definidas na álgebra, como, por exemplo, recuperação de operandos em disco (operador *scan*). A figura 2.2 mostra uma consulta com seus respectivos planos lógico e físico.

Os algoritmos para implementação dos operadores podem variar bastante, diferenciando-se basicamente no que diz respeito às estratégias adotadas, tais como: *scanning*, *hashing*, *sorting* e indexação. Fatores relativos à disponibilidade de recursos são relevantes na escolha dos operadores físicos, como por exemplo, quantidade disponível de memória. Esta avaliação é feita durante a fase de compilação.

Ao se executar operadores, os resultados devem fluir entre os mesmos, de acordo com a estrutura da árvore. Durante a execução de um plano físico pode haver a necessidade de se materializar os resultados intermediários. No entanto, alguns operadores permitem a utilização da estratégia *iterator* ([25]), ou seja, fornecimento dos resultados para os acionadores dos operadores à medida que aqueles são obtidos, desta forma evitando a materialização e possibilitando um ganho de desempenho. A estratégia de controle de fluxo a ser adotada para cada operador também deve ser indicada na árvore de execução. Como indicado nas figuras 2.2(b) e 2.2(c), onde uma linha sem interrupções denota fluxo *pipeline* e uma linha cortada representa uma materialização.

Operadores relacionais, bem como as estratégias que os implementam, mostram-se adequados ao seu modelo de origem. Já para a manipulação de dados XML, a utilização do modelo relacional apresenta baixo desempenho se

comparado com os resutados obtidos em dados com formato adequado. Dessa forma, a fim de obter melhor desempenho no processamento de consultas sobre dados XML, novos operadores precisam ser desenvolvidos, além de novas estratégias de implementação. Isso se deve, principalmente, em virtude das seguintes características:

- *Modelo de dados*: documentos XML são representados por um modelo de dados baseado em grafo, o que adiciona maior complexidade à sua estrutura.
- *Heterogeneidade*: em um documento XML pode-se ter um mesmo subelemento omitido ou repetido várias vezes, para instâncias distintas.
- *Ausência de Esquema*: nem sempre informações a respeito da estrutura de um documento XML estão disponíveis.

Dessa forma, o uso de métodos especialmente desenvolvidos para o tratamento de dados XML se faz necessário para o bom desempenho de consultas sobre bases de dados nesse formato.

Operação	**Descrição**	**Notação**
Seleção	Seleciona todos os itens de R que satisfazem a condição de seleção.	σ<condição de seleção>(R)
Projeção	Produz uma nova coleção contendo somente os itens de R especificados na lista de itens e elimina itens duplicados.	π<lista de itens>
Theta Join	Produz todas as combinações de R e S que satisfazem a condição de junção.	R⋈<condição de junção>S
Equijoin	Produz todas as combinações de R e S que satisfazem a condição de junção apenas com comparações de igualdade.	R⋈<condição de junção>S ou R⋈(<itens de junção R>,<itens de junção S>S)
Junção Natural	Semelhante ao Equijoin, exceto que os itens de S contidos na condição de junção não são incluídos na coleção resultante. Se os itens de junção tem o mesmo nome não precisam ser especificados.	R*<condição de junção>S ou R*(<itens de junção R>, <itens de junção S>)S
União	Produz uma nova coleção com todos os itens de R e de S. R e S devem ser compatíveis.	R∪S
Interseção	Produz uma nova coleção contendo todos os itens que estão presentes tanto em R quanto em S. R e S devem ser compatíveis.	R∩S
Diferença	Produz uma coleção contendo todos os itens de R que não estão em S. R e S devem ser compatíveis.	R-S
Produto	Produz uma coleção cujo esquema é uma união dos esquemas de R e S e contem todas as possíveis combinações de R e S.	R×S
Divisão	Produz uma coleção T(X) que inclui todos os itens i[X] em R(Z) que aparecem em R em combinação com todo item de S, onde Z = X ∪ Y. Esse operador pode ser expresso como a seguinte seqüência: T1 ← π<Y>(R), T2 ← π<Y>((SxT1)-R), T ← T1 - T2.[2]	R÷S
Agrupamento e Agregações	Produz uma coleção contendo os itens especificados na lista itens de agrupamento, seguidos de cada uma das funções da lista funções de agregação aplicadas a seus respectivos itens.	<itens de agrupamento> $\Im$ <funções de agrupamento>
Ordenação	Ordena a coleção R de acordo com os itens especificados na lista de itens de ordenação.	τ<itens de ordenação>(R)

Tabela 2.1: Álgebra Lógica

2.2 Processamento de Consultas sobre Dados XML

Como mencionado na seção anterior, avaliar consultas sobre dados XML em sua forma nativa é uma tarefa mais desafiadora que em sistemas relacionais. Isso se deve, principalmente, à natureza semi-estruturada e ao modelo de dados baseado em grafo do formato XML.

O uso de índices sobre as expressões de caminho ou sobre os valores de elementos buscados, juntamente com estratégias eficientes para o desenvolvimento das operações chave no tratamento de consultas XML, em especial, casamento de padrões, pode trazer ganhos significativos no tempo de resposta dessas consultas. Além disso, técnicas de otimização tradicionais podem se mostrar pouco eficientes uma vez que não exploram a natureza semi-estruturada do formato XML.

O grande uso de XML nas trocas de dados tem levado a comunidade acadêmica a desenvolver técnicas de armazenamento, recuperação e manipulação de dados nesse formato. Nas seções a seguir apresentamos brevemente essas abordagens, alguns SGBDs que as utilizam em suas concepções, bem como suas soluções para a fase de avaliação de consultas.

Vale ressaltar que atenção especial vem sendo dada ao uso de técnicas capazes de avaliar dados XML em sua forma nativa. Como resultado desse esforço, surgem diversas propostas visando o desenvolvimento de operadores físicos eficientes para essa tarefa. A seção 4.3.2 exibe o conjunto de operações adotado em [10] como álgebra física.

SGBDs XML Habilitados

O termo *SGBD XML Habilitado* é utilizado para designar um SGDB tradicional no qual os dados XML são mapeados para o formato original do sistema a fim de aproveitar o poder de processamento já existente para esse modelo. No entanto, como vimos na seção 1.1, tais abordagens sofrem sérias limitações, em especial, no tocante ao desempenho de consultas.

Diversas técnicas de mapeamento têm sido propostas pela comunidade acadêmica, algumas das principais serão discutidas a seguir.

Florescu e Kossmann, em [21], apresentam alternativas simples para armazenar informações estruturais XML, a saber as abordagens *Edge*, *Binary* e *Universal*. Na abordagem *Edge*, uma tabela com os campos *source*, *ordinal* e *target* é criada, onde *source* é o nó pai do atributo, *ordinal* indica a ordem do elemento em relação aos seus irmãos e *target* indica o valor atômico do nó ou uma referência à outra tabela, na qual o valor desse elemento está armazenado. O principal problema dessa abordagem diz respeito à freqüente necessidade de *auto-joins* para obter o resultado de uma consulta. Já na abordagem *Binary*, para cada valor de *tag* diferente que ocorra no documento, uma tabela *Edge* é criada para armazenar elementos com esse rótulo. Os principais problemas dessa abordagem são o exagerado número de tabelas criadas e a quantidade de junções realizadas para responder uma consulta. Finalmente, a tabela *Universal* é criada a partir do *outer-join* das tabelas *Binary*. A tabela gerada nessa solução não é normalizada, além disso, apresenta problemas de redundância e perdas de desempenho. Para armazenamento dos nós folhas (nós de valores), duas estratégias são consideradas. Na primeira, os valores são armazenados de acordo com seus tipos, uma tabela para cada tipo. Já na segunda, os valores são armazenados na mesma tabela das estruturas.

Em [16], o mapeamento é realizado através da linguagem STORED (*Semistructured TO RElational Data*). A idéia proposta é identificar padrões em um documento XML a fim de determinar um esquema relacional básico. Sub-estruturas que não são definidas neste esquema são armazenadas em grafos de *overflow* e acessadas através de ponteiros. Além dos problemas citados na seção 1.1, nessa abordagem tem-se que a geração de um bom esquema relacional é uma tarefa NP-difícil, então um algoritmo heurístico de mineração de dados é utilizado para resolver essa tarefa. Como resultado, instâncias distintas de uma mesma entidade do mundo real, por conter estruturas diferenciadas, podem ser mapeadas em tabelas diferentes.

Em [28] uma metodologia para tratamento de dados XML é proposta

através da criação de uma *visão XML padrão*, gerada automaticamente através do XPERANTO [6]. Desse ponto, o algoritmo de reconstrução da visão XML age sobre a visão padrão construindo a visão de reconstrução XML. Assim, um único processador de consulta que saiba operar com essa visão de reconstrução XML pode ser aplicado a qualquer técnica de geração de esquema. Nessa mesma linha, tem-se também o sistema SilkRoute [18], onde a transformação relacional-XML é feita em duas fases. Inicialmente, uma visão XML virtual sobre o banco relacional é definida, através da linguagem RXL (*Relational to XML Transformation Language*). Uma vez que a visão virtual é definida, o SilkRoute recebe consultas XML-QL de usuários e as compõe automaticamente com a consulta RXL que descreve a visão XML sobre a base de dados. O resultado é uma outra consulta RXL que extrai apenas o fragmento dos dados relacionais que foi requisitado pela aplicação. O principal problema dessas metologias é a quantidade de camadas introduzidas, o que acarreta em degradação de desempenho.

O estudo aprofundado sobre as técnicas empregadas em SGBDs XML Habilitados está fora do escopo desse trabalho, maiores informações sobre o assunto podem ser obtidas em [28], [27], [21], [43], [38], [42], [16] e [18].

SGBDs XML Nativos

Segundo [17], SGBDs XML Nativos (SGBDXNs) são SGBDs que possuem uma interface capaz de mapear dados entre documentos XML e seus próprios modelos de dados internos. Por se tratarem de sistemas desenvolvidos visando manipulação de dados de estrutura irregular, essa categoria de SGBDs mostra-se mais adequada à execução de consultas sobre documentos XML, pois não sofrem com as limitações encontradas nos sistemas habilitados. Como exemplos dessa categoria podemos citar os sistemas Timber [41], Natix [19], OrientX [44], e Tamino [39, 20].

O processamento de consultas no Timber tem o propósito de ser independente da linguagem tanto quanto possível, possui *parsers* para XQuery [35], Quilt [14], XML-QL [1] e XQL [26]. No Timber, para cada nó é associada

uma tupla (*start, end, level*), que pode ser usada como identificador do nó, onde *start* e *end* definem o intervalo entre os rótulos de um dado nó e *Level* representa o nível de aninhamento de um determinado nó no documento. A álgebra física implementada tem como base os operadores da álgebra TAX (*Tree ALgebra for XML*) [29], onde a idéia de "árvores de padrões" é amplamente usada para a busca das expressões solicitadas. Maiores detalhes sobre estas implementações podem ser obtidas em [36]. A principal desvantagem dessa estratégia reside no fato de que seus operadores operam sobre árvores homogêneas, pouco adequadas ao tratamento de dados estruturados de forma flexível, como é o caso dos dados XML. Além disso, o uso do gerente de objetos Shore [31] aumenta o número de camadas em sua arquitetura, o que acarreta em perdas de desempenho. Outra característica interessante desse sistema é o uso de informações sobre o esquema para auxiliar no processamento da consulta, sempre que possível.

Já o sistema Natix destaca-se devido ao seu modelo de armazenamento, que leva em consideração a semântica dos relacionamentos entre os nós. Quanto maior o grau de relacionamento entre dois nós, mais próximos, fisicamente, esses são armazenados. Em seu modelo, dados XML são armazenados em forma de árvores hierárquicas, assim como as árvores de arquivos em sistemas operacionais. Teoricamente, o Natix pode ser classificado como um sistema híbrido, onde estruturas cuja granularidade é maior que o especificado são armazenadas numa parte estruturada do banco de dados, já estruturas cuja granularidade é menor são armazenadas como "objetos planos". O tamanho mínimo das sub-árvores pode ser limitado dinâmicamente e suporta inserção e remoção de fragmentos XML. Além disso, os objetos planos armazenados são, na verdade, grupos de nós clusterizados tratados como registros atômicos pelos níveis mais baixos do sistema. No entanto, informações de esquema são pouco aproveitadas e seu desempenho na execução de consultas apresenta problemas de eficiência. A fim de poder retornar documentos XML sob várias representações, o sistema provê a *Natix Virtual Machine* (NVM), componente especialmente desenvolvido para realizar essa tarefa. No entanto, a camada

NVM introduz complexidade à sua *engine* de execução, em especial devido à grande quantidade de programas requisitados.

OrientX apresenta como principal contribuição intelectual o uso de informações de esquema dos documentos XML para fornecer ganhos de processamento. Tais informações são utilizadas em todos os módulos desse sistema proporcionando:

- suporte a armazenamento clusterizado baseado no esquema;
- índices de caminho construídos de acordo com o esquema (a estrutura de índice proposta suporta consultas a um determinado *label* e a uma expressão de caminho simples [3] a partir da raiz do documento);
- informações estatísticas integrando esquema e histogramas e
- controle de acesso baseado nas permissões do usuário.

O avaliador de consultas suporta XQuery1.0 e utiliza a álgebra XAlgebra [44], similar à semântica XQuery definida em [15], diferindo desta nos seguintes aspectos:

- XAlgebra apenas decompõe uma expressão de caminho em uma série de caminhos simples que podem ser processados por técnicas maduras de *matching* de caminhos, ao invés de decompô-la em uma série de *for-nested-loops*;
- XAlgebra introduz alguns métodos de processamento, dentre os quais podemos destacar: *sort-merge join*, *predicate pushing-down* e *unnesting*;
- informações do esquema (estrutura e tipos) podem proporcionar ganhos de eficiência na checagem de tipos. Além disso, o otimizador leva em consideração a correlação entre os valores dos atributos e suas posições hierárquicas na estimativa de suas seletividades.

[3] Uma expressão de caminho simples é uma seqüência de nomes de elementos e/ou atributos XML, separados pelo operador /. Este operador indica uma relação pai-filho entre dois elementos ou entre um elemento e um atributo.

O principal problema dessa abordagem reside no alto consumo de espaço para o armazenamento das informações estruturais dos documentos XML, além de limitar a flexibilidade desses dados.

Tamino é um SGBDXN comercial, cuja álgebra opera sobre estruturas de tabela, necessitando, assim, de um operador de construção dos resultados, o que acarreta perdas no desempenho. Como linguagem de consulta, aceita um sub-conjunto da XQuery, tendo seu motor de execução formado por operações padrões, tais como, junção, ordenação e *scanning*, dentre outras das operações especificadas pela XQuery. Mantém índices sobre valores numéricos, textos e estruturas, entretanto, os índices explorados podem ser usados para determinar documentos contendo nós com determinadas propriedades, mas não para recuperar nós. Além disso, pouco se encontra na literatura a respeito de detalhes técnicos de sua arquitetura.

A tabela 2.2 apresenta um quadro comparativo entre os principais SGBDs XML Nativos com relação aos seus aspectos de armazenamento e álgebras lógica e física.

	Timber	**Natix**	**OrientX**	**Tamino**
Armazenamento	Shore	Próprio. Preocupa-se com a semântica dos dados XML.	Clusterizado com base no esquema.	Agrupa documentos com determinada sub-estrutura comum.
Álgebra Lógica	TAX - Opera sobre coleções de árvores homogêneas.	-	XAlgebra - Semanticamente semelhante à XQuery.	Baseada no uso de expressões FLWR.
Álgebra Física	Realiza o *matching* das árvores padrões utilizando a abordagem *iterator*.	NVM. Operações: scan, associação de variáveis e construção de resultados, via *iterator*.	Baseada em *matching*. Faz amplo uso das informações de esquema.	Opera sobre tabelas. Índices recuperam documentos. Abordagem *iterator*.

Tabela 2.2: Quadro comparativo de SGBDs XML Nativos

2.2.1 Linguagens de Consulta XML

Muitas linguagens de consulta têm sido propostas pela comunidade acadêmica, tais como XIRQL [23], XQL [26], Lorel [2], XML-QL [1], Quilt [14] e XQuery [35].

Atenção especial merece ser dada a XQuery, pois trata-se da linguagem de consulta recomendada pelo W3C. O modelo de dados utilizado define cada documento XML como uma árvore de nós, sendo também capaz de operar sobre coleções de documentos, fragmentos XML e coleções destes.

XQuery é uma linguagem funcional, onde cada consulta é uma expressão. As principais formas de expressões XQuery são [13]:

- **expressões de caminho**

  Baseiam-se na sintaxe do XPath e são especificadas de acordo com o seguinte modelo:

  Encontre todos os títulos de capítulos no documento books.xml:

  ```
  document("books.xml")//chapter/title
  ```

  Encontre todos os livros no documento bib.xml publicados pela Addison-Wesley após 1991:

  ```
  document("‘bib.xml")//book[publisher = "Addison-Wesley"AND @year >
  "1991"]
  ```

- **construtores de elemento**

  Usado nos casos em que a consulta precisa criar novos elementos, como, por exemplo:

  Gere um elemento <book> com atributo year cujo valor é o título do livro:

  ```
  <book>
     $b/@year
      $b/title
  </book>
  ```

  A variável `$b` é limitada em outra parte da consulta. Quando a consulta completa é executada, o construtor de elementos irá criar o seguinte resultado:

```
<book year="1992»
   <title>Advanced Programming in the Unix environment</title>
</book>
```

- **expressões FLWR (For-Let-Where-Return)**

  É análoga à construção SELECT-FROM-WHERE na SQL e consiste de:

  - **cláusula For**: usada para identificar as fontes de dados e inicializar as variáveis;
  - **cláusula Let**: associa variáveis a diferentes caminhos nos dados;
  - **cláusula Where**: especifica condições
  - **cláusula Return**: especifica o formato do resultado.

  Consultas com informação estrutural incompleta são formuladas com o auxílio de coringas, tais como *, ?, //, etc. Esses coringas podem ocorrer em qualquer lugar da expressão de caminho, já nas expressões FLWR podem ocorrer em qualquer lugar da cláusula For ou da cláusula Where.

  O exemplo a seguir retorna o título e o preço médio de todos os livros publicados pela Addison-Wesley:

```
<results>
   {
   FOR $t IN distinct(document("prices.xml")/prices/book/title)
   LET $p:=avg(document("prices.xml")/prices/book[title=$t]/price)
   WHERE (document("bib/xml")/book[title=$t]/publisher) =
         "Addison-Wesley"
   RETURN
      <result>
          $t
         <avg>
          $p
      </result>
   }
</results>
```

- **expressões que envolvem operadores e funções**

  XQuery provê a maioria dos operadores e funções que podem ser encontrados em outras linguagens, incluindo operadores aritméticos, de comparação, lógicos e operadores relacionados a seqüências. As funções internas incluem AVG, SUM, COUNT, MAX e MIN, além de funções

relacionadas ao documento XML e ao conjunto de nós como, por exemplo, DOCUMENT, EMPTY e DISTINCT.

No exemplo a seguir o preço mínimo de cada livro é retornado no elemento <minprice> que tem o título do livro como um atributo:

```
<results>
  {
  LET $doc := document("prices.xml")
  FOR $t IN distinct($doc/book/title)
  LET $p := $doc/book[title = $t]/price
  RETURN
     <minprice title= $t/text() >
        { min($p) }
     </minprice>
  }
</results>
```

Além disso, a XQuery também provê um mecanismo para especificar funções definidas pelo usuário.

- **expressões condicionais**

  XQuery também permite o uso de expressões IF-THEN-ELSE:

```
 <user>
    $u/userid
    $u/name
  {
  IF (empty($b))
  THEN <status>inactive</status>
  ELSE <status>active</status>
  }
</user>
```

- **expressões quantificadas**

  São as expressões *SOME* e *EVERY*. Através da expressão *SOME* é possível identificar se pelo menos um nó de um conjunto satisfaz o predicado. Já a expressão *EVERY* é usada para testar se todos os nós de um conjunto satisfazem um predicado.

  O seguinte exemplo lista os nomes dos usuários, se algum, que tem licitações em todos os seus itens:

```
<frequent_bidder>
  {
  FOR $u IN document("users.xml")//user_tuple
  WHERE
```

```
      EVERY $item IN document("items.xml")//item_tuple SATISFIES
         SOME $b IN document("bids.xml")//bid_tuple SATISFIES
            ($item/itemno = $b/itemno AND $u/userid = $b/userid)
   RETURN
      $u/name
   }
</frequent_bidder>
```

- **expressões que testam ou modificam os tipos dos dados**

  XQuery suporta tipos de dados "comuns" (com base no sistema de tipos XML Schema), bem como tipos de dados definidos pelo usuário. As expressões *INSTANCEOF* e *TYPESWITCH/CASE* são usadas para testar o tipo de dados de uma instância.

2.3 Consulta e Casamento de Padrões sobre Dados XML

Diferentemente do mundo relacional, nos sistemas XML nativos não podemos contar com o pleno conhecimento do esquema dos dados envolvidos em virtude da natureza mutante pertinente ao formato XML. Dessa forma, descobrir quais fragmentos de um documento XML obedecem às restrições impostas em uma consulta, operação núcleo no processamento de consulta, mostra-se uma tarefa mais complexa que em sistemas relacionais.

A fim de superar tais barreiras, muitos trabalhos têm sido propostos, no entanto, ainda apresentam deficiências em suas estratégias de manipulação de dados XML em seu formato nativo. A seção a seguir descreve brevemente sobre algumas das principais abordagens para o assunto.

Uma expressão XQuery pode ser representada através de uma árvore de padrões com nós rotulados. Dessa forma, uma maneira de obter o resultado de uma consulta é encontrando os fragmentos do documento que estão de acordo com a árvore solicitada, ou seja, realizando o casamento de padrões entre a árvore de consulta e a base de dados.

2.4 Trabalhos Relacionados

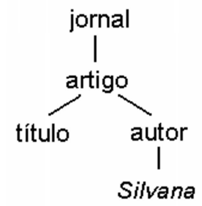
Soluções iniciais para a tarefa de identificar quais porções de um documento XML seriam relevantes para determinada consulta exploravam o amplo uso de junções estruturais, o que acarretava grandes problemas com relação ao tamanho dos resultados intermediários inválidos.

Considere o exemplo representado na figura 2.3, onde em 2.3(c) tem-se uma base de dados, e 2.3(b) exibe graficamente a consulta exposta em 2.3(a). Uma estratégia pouco sofisticada poderia, por exemplo, usar índices para recuperar para cada nó estrutural da árvore de consulta, todos os nós da base de dados que possuíssem mesma *Tag* que este e, posteriormente, realizaria junções estruturais sobre tais elementos. Pode-se perceber que os nós título(1, 3:5, 2) e título(1, 19:21, 2), são recuperados inutilmente, uma vez que apenas o nó *título*(1, 10:12, 3) - destacado em itálico na figura 2.3(c) - consta na resposta.

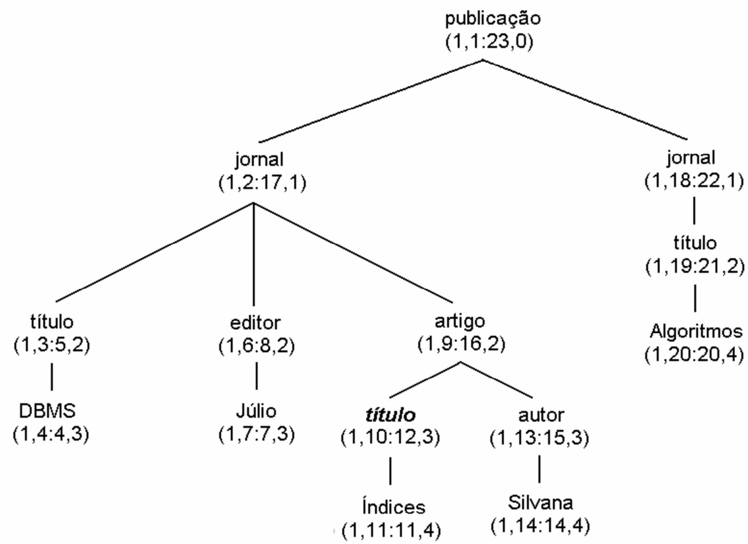
Jagadish et al., em [3], apresenta uma solução para o problema de junções estruturais através do desenvolvimento de um *merge-join* modificado, onde múltiplos acessos podem ser necessários a um dos operandos. Duas listas ordenadas são utilizadas para armazenar potenciais elementos sucessores e descendentes. O reconhecimento dos relacionamentos é realizado por meio de verificações das posições destes dentro do documento, a saber, (*DocId*, *startPos* : *EndPos*, *NumLevel*), onde *DocId* identifica o elemento a ser analisado, *LeftPos* e *RightPos* denotam, respectivamente, as posições inicial e final de cada elemento, e *LevelNum*, indica o tamanho do caminho entre o elemento e a raiz do documento. Li e Moon, em [30], e Chien et al. em [11], também propõem técnicas para a realização de junções estruturais, onde índices são usados para recuperar entradas não ordenadas. Li e Moon propõem um esquema de numeração flexível, capaz de identificar relacionamentos de forma eficiente. Já Chien et al. procuram reduzir a quantidade de soluções temporárias incorretas, mas só resolvem o problema pela metade, eliminando apenas descendentes inválidos. O principal problema no tratamento de junções estruturais diz respeito à necessidade de fases de decomposição e composição dos pares a serem

```
FOR $a IN document( "http://.../pub.xml") //jornal/artigo
     $b IN $a/título
WHERE $a/autor = "Silvana" RETURN
   <artigo> $b </artigo>
```

(a) Consulta



(b)



(c) Documento

Figura 2.3: Exemplo de Consulta e sua representação simplificada

emparelhados, além da construção de resultados intermediários enormes frente aos finais.

Bruno et al. [4] propõe os algoritmos *PathStack* e *TwigStack* que também utilizam o esquema (*DocId*, *LeftPos* : *RightPos*, *LevelNum*) para representar as posições dos elementos XML e valores de *strings*. Além disso, para cada *Tag*, uma lista (*stream*) T_{tag} de elementos com este rótulo é fornecida como entrada. A idéia por trás do algoritmo *PathStack* é repetidamente construir pilhas contendo respostas totais e parciais para o padrão de consulta, através da iteração dos nós de entrada, ordenados de acordo com os valores *LeftPos*. Uma vez construídas as pilhas, os relacionamentos são reconhecidos através da verificação de seus topos, com base no esquema de numeração adotado. Já o algoritmo *TwigStack* opera em duas fases. Na primeira, algumas soluções para caminhos da raiz à folha são computados. Na segunda, um *merge-join* dessas soluções é realizado para responder ao padrão solicitado. A diferença básica entre *PathStack* e a primeira fase do *TwigStack* é que antes de um nó h_q ser inserido em sua pilha correspondente, *TwigStack* assegura que: *i.)* h_q possui um descendente h_{qi} em cada uma das *streams* T_{qi}, para $q_i \in$ filhos(q), e , *ii.)* cada nó h_{qi}, recursivamente, satisfaz essa propriedade. No entanto, esses algoritmos não fazem a concatenação das sub-árvores resultantes. Além disso, *TwigStack* mostra-se mais apropriado para consultas que envolvem apenas relacionamentos antecessor-descendente. Em [8], Jagadish et al. faz uma extensão a este trabalho a fim de que as estruturas buscadas possam conter ciclos, interessante para os casos em que a estrutura do documento também pode contê-los. Para isso, uma estrutura chamada *partial solution pool* é criada para armazenar os nós retirados das pilhas que guardam os potenciais nós ancestrais de um elemento. Além disso, uma tabela é construída onde, para cada nó da árvore que representa o documento, uma entrada indicando seus pais é adicionada. Isso é feito para que múltiplas ascendências possam ser construídas, pois sempre que um é retirado de uma das pilhas, uma verificação é feita na tabela a fim de descobrir se este é potencialmente pai de algum outro nó. O trabalho desenvolvido por Chen, Lu e Ling em [9] também estende [4].

No entanto, o algoritmo desenvolvido é capaz de trabalhar sobre qualquer esquema de numeração, desde que as *streams* de entrada estejam ordenadas em pré-ordem. Além disso, três esquemas de numeração são analisados a fim de reduzir os resultados intermediários inválidos. De forma resumida tem-se: *i.) Tag Stream* - contém todos os elementos de um determinado documento que possuem mesma tag -; *ii.) Tag + Level Stream* - contém todos os elementos que possuem mesma *tag* e que estão no mesmo nível -; *iii.) Prefix-Path Stream* - contém todos os elementos que possuem mesmo prefixo e mesma *tag*. No entanto, sua estratégia também apresenta deficiências nos casos em que a consulta requisitada envolve os dois tipos de arestas (*antecessor-descendente* e *pai-filho*).

Classes de relacionamento são utilizadas em [33] a fim de prover a facilidade de trabalhar com árvores que possuem determinadas características comuns. No entanto, sua abordagem de *pattern matching* é implementada através do uso de junções de valores e junções estruturais.

Em [46], Zezula et al apresenta uma abordagem que ignora a ordem do documento durante sua identificação dos fragmentos de interesse. Sua estratégia é baseada no uso de estruturas semelhantes às árvores de padrões. Um menor número de operações é necessário para concatenar o resultado final. Porém, esta estratégia também exige um passo de composição dos resultados. Além disso, a estrutura de árvore utilizada não se mostra adequadamente flexível, uma vez que não permite a existência de elementos opcionais.

Em [45], Yao e Zhang propõem uma técnica de *matching* sobre árvores, cuja única restrição é que elementos internos não contenham sub-elementos com seu mesmo *label*. Tal limitação não oferece grandes restrições quanto ao uso dessa abordagem já que grande parte das consultas realizadas em situações reais não fazem uso desse tipo de estrutura. Porém, a abordagem empregada utiliza índices na recuperação de todos os nós envolvidos na consulta o que acarreta um amplo uso de junções estruturais desnecessárias, além de excessiva recuperação de nós que não levam a resultados válidos.

Dessa forma pode-se perceber que no atual estado da arte de processamento

nativo de consultas sobre bases XML, casamento de padrões, que envolve encontrar todos os fragmentos de um documento que possuem uma determinada sub-estrutura, tem se mostrado uma operação chave na manipulação de consultas sobre bases XML. No entanto, muitos ainda são os desafios a serem superados. Muitas das tecnologias desenvolvidas até o momento sofrem com o problema de decompor a estrutura desejada em nível de elementos, o que acarreta em ampla recuperação de elementos desnecessários, além de exigir maior esforço durante a fase de composição dos resultados. Uma proposta alternativa a este problema consiste na utilização das informações de esquema, no entanto, para documentos grandes e de estrutura complexa, tal abordagem requer grandes consumos de espaço para guardar tais informações, o que pode induzir a perdas significativas de desempenho.

2.5 XML-PM

A raiz dos problemas no tratamento de dados XML reside no fato de que dados XML são heterogêneos. A fim de superar tais desafios de realizar a identificação de estruturas mutantes, o algoritmo XML-PM foi desenvolvido. XML-PM faz uso de estruturas similares às GTPs, árvores flexíveis, propostas por Jagadish et al. em [10], capazes de capturar a natureza semi-estruturada do formato XML. Este trabalho estende o conceito de GTP, de acordo como descrito no capítulo 4, a fim de prover suporte no reconhecimento de relacionamentos antecessor-descendente e na identificação única dos padrões solicitados. Índices sobre as expressões de caminho buscadas e uma estratégia *bottom-up* de travessia do documento XML são utilizados para proporcionar melhores desempenhos. A principal vantagem da estratégia adotada frente às demais está na obtenção direta dos resultados finais, sem necessidade de diversas operações de combinação entre os fragmentos de uma saída potencialmente válida. Além disso, as expressões que servem de entrada aos índices utilizados para a recuperação dos elementos desejados são baseadas nos padrões descritos na árvore de consulta. Dessa forma, o número de elementos desnecessários recuperados é

bastante reduzido, quando comparado às demais abordagens. Maiores detalhes sobre o XML-pm serão dados no capítulo 5.

2.6 Conclusões

A necessidade de tratar dados XML em SGBDs resulta em inúmeros desafios, em especial, na fase de processamento de consultas sobre tais dados. Devido à natureza semi-estrutural do formato XML, SGBDs tradicionais apresentam limitações no tratamento de dados neste formato. No entanto, a necessidade de responder eficientemente consultas sobre tais dados tem despertado o interesse acadêmico no desenvolvimento de métodos capazes de realizar esta tarefa. Em especial, o reconhecimento da estrutura que satisfaz ao padrão solicitado em uma consulta tem se mostrado uma operação núcleo em sua fase de execução. Porém, apesar dos esforços, técnicas desenvolvidas especificamente para este fim ainda apresentam deficiências. Os principais problemas encontrados dizem respeito ao uso de fases para decomposição e combinação, além do número de resultados intermediários inválidos. Com isso, o operador proposto no presente trabalho é apresentado brevemente a fim de prover melhorias de desempenho na obtenção de fragmentos XML que respondam a uma consulta. Suas características principais são brevemente discutidas em comparação a outras abordagens para a solução deste problema e seu contexto de implementação descrito em maiores detalhes.

Capítulo 3

Contexto da Abordagem

Nesse capítulo apresentaremos o ambiente de experimentação de técnicas de armazenamento e consulta a dados XML, FoX. Em seu estágio atual, o FoX contempla as funcionalidades de Gerenciador de Armazenamento e Processador de Consultas. Nas seções a seguir, apresentaremos maiores detalhes sobre a arquitetura desse sistema, seus principais módulos, suas funções e como interagem entre si.

3.1 Arquitetura Geral

Atualmente, os principais componentes do FoX são: o Gerenciador de Armazenamento e o Processador de Consultas. Além dessas camadas temos ainda o nível de Servidor de Páginas, que fornece um conjunto de funções para armazenamento e recuperação de dados do disco, além de funcionalidades de *buffer* e *cache*. A arquitetura como um todo pode ser vista na figura 3.1.

O Gerente de Armazenamento interage, no nível mais baixo, com o servidor de páginas. Em seu atual estágio de desenvolvimento, essa camada é encarregada pelas funções de persistência e recuperação de sub-árvores XML.

O Processador de Consultas é responsável pelo reconhecimento, otimização e execução de uma consulta.

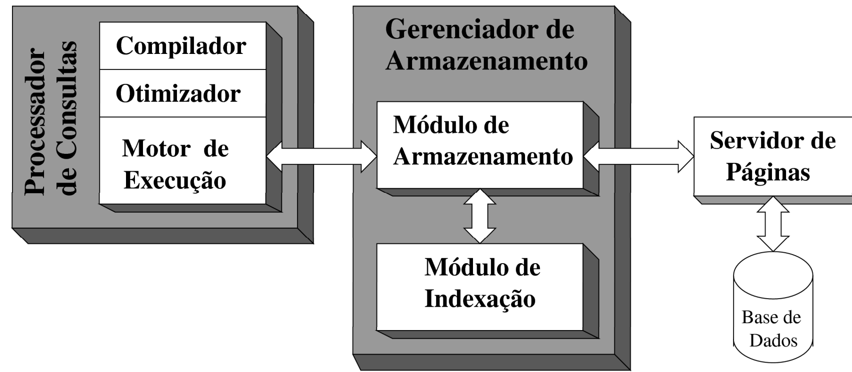


Figura 3.1: Arquitetura Geral do FoX

3.2 Servidor de Páginas

O Servidor de Páginas fornece às camadas superiores um conjunto de funções para armazenamento e recuperação dos dados da memória secundária, além de prover funcionalidades de *buffer* e *cache*. O número de *bytes* a serem tratados é limitado ao tamanho de página do FoX, parâmetro este configurado pelo usuário. Os dados processados nessa camada não possuem semântica, é tarefa dos níveis superiores provê-la.

Nessa fase de desenvolvimento do FoX, estamos utilizando o serviço de páginas implementado no Gerenciador de Objetos Armazenados GOA++ [32], desenvolvido pela COPPE, UFRJ.

3.3 Gerenciador de Armazenamento

O Gerenciador de Armazenamento é constituído de dois sub-sistemas principais, os Módulos de Armazenamento e Indexação.

3.3.1 Módulo de Armazenamento

Modelo Lógico de Armazenamento

O FoX adota o *modelo lógico de árvore* para armazenar documentos XML. Um documento XML $D = \{e_1, e_2, \ldots, e_n\}$, onde $e_i (1 \leq i \leq n)$ corresponde a um elemento ou atributo XML presente no documento D, pode ser representado

através de uma árvore $G_D = (V_{G_D}, E_{G_D})$, tal que:

- V_{G_D} é o conjunto de nós de G_D;
- E_{G_D} é o conjunto de arestas de G_D
- Cada nó $n_i \in V_{G_D}$ corresponde a um $e_i \in D$, e vice-versa;
- Existe uma aresta $a_{ij} \in E_{G_D}$ entre dois nós n_i e n_j (n_i, $n_j \in V_{G_D}$) se, e somente se, e_i é o elemento XML filho de e_j (e_i, $e_j \in D$)
- Cada nó n_i em V_{G_D} é rotulado com o nome do seu elemento ou atributo e_i correspondente e possui um identificador único na Base de Dados. Chamamos o rótulo dado a um elemento ou atributo e_i de *label*.
- O identificador de um nó $n_i \in V_{G_D}$ é formado pela quádrupla *<DocID, Início, Fim, Nível>*. Esses valores são armazenados no próprio nó.

Na figura 3.3, temos a representação em forma de árvore do documento XML da figura 3.2. Nesta figura, os identificadores são os rótulos dos nós e os valores textuais de cada nó, caso existam, são armazenados no próprio nó. Se D_1,D_2, . . . ,D_n são os documentos XML a serem gerenciados pelo FoX, então a Base de Dados $BD = G_{D_1}, G_{D_2}, \ldots, G_{D_n}$.

```
<library>
 <book>
  <title> ... </title>
  <author> ... </author>
  <publisher> ... </publisher>
  <date> ... </date>
  <edition> ... </edition>
 </book>
 <article>
  <title> ... </title>
  <author> ... </author>
  <journal> ... </journal>
  <date> ... </date>
  <volume> ... </volume>
  <number> ... </number>
  <pages> ... </pages>
 </article>
</library>
```

Figura 3.2: Documento XML

Estratégia de Armazenamento

A estratégia utilizada para armazenar as árvores lógicas em disco é baseada na estratégia do Natix [19]. Uma árvore lógica é materializada como uma árvore física, onde esta possui todos os nós lógicos e mais alguns nós de controle. Os nós de controle são necessários à manutenção da estrutura de árvores grandes que não cabem em uma página de disco.

A árvore física, por sua vez, é repartida em vários registros, onde cada registro armazena uma de suas sub-árvores. Uma página de disco pode conter um ou mais registros. A figura 3.4 mostra a divisão de uma árvore física em registros, onde os nós pontilhados são os nós de controle.

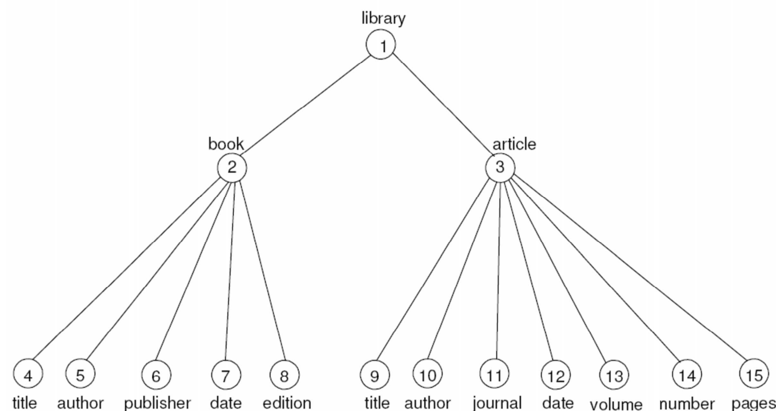


Figura 3.3: Árvore Correspondente ao Documento XML, Armazenado no FoX

Operações no Módulo de Armazenamento

Para resolver as expressões XPath enviadas pelo Processador de Consultas, o Módulo de Armazenamento, primeiramente, faz requisições ao Servidor de Páginas. Com isso, o Módulo de Armazenamento recupera todas as páginas de disco referentes à Base de Dados e, em seguida, monta as árvores lógicas correspondentes. Por fim, estas árvores são percorridas e as sub-árvores, cujas raízes são os nós que casam com as expressões de caminho XPath da entrada, são retornadas.

Para inserir um novo nó n em alguma árvore lógica do FoX, primeiramente é decidido onde, na árvore física, n deve ser inserido. Uma vez feito isto, adi-

cionamos n à sub-árvore correspondente a ele, armazenada em algum registro do FoX, digamos o registro r. Se após a inserção de n, a sub-ávore não cabe mais em r, ocorre um *split*, ou seja, um novo registro r_0 é criado e a sub-árvore correspondente a r é então dividida entre os dois registros r e r_0. Esta operação de *split*, no pior caso, propaga-se até o registro que contém a raiz da árvore física.

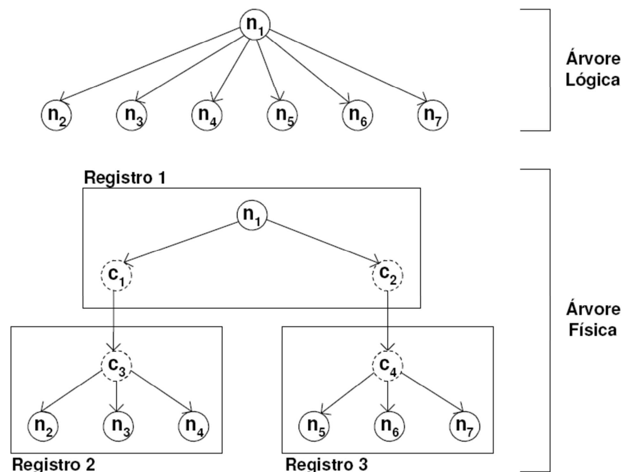


Figura 3.4: Árvores Lógica e Física do Gerente de Armazenamento do FoX

Para remover um novo nó n de alguma árvore lógica do FoX, primeiramente, localizamos onde, na árvore física, n está. Uma vez feito isto, removemos n da sub-árvore correspondente a ele, armazenada em algum registro do FoX, digamos o registro r. Se após a remoção de n, a sub-árvore em r é muito pequena - o tamanho mínimo de uma sub-árvore é configurado através de parâmetros extras -, ocorre um *merge*, ou seja, fundimos o registro r com o seu vizinho, digamos r_0, e eliminamos o registro r da árvore física. Esta operação de *merge*, no pior caso, propaga-se até o registro que contém a raiz da árvore física.

3.3.2 Módulo de Indexação

O Módulo de Indexação é o componente responsável por acelerar consultas na Base de Dados através da utilização de um índice, o Índice FoX [37], [5], cujas

chaves de busca são todas as expressões de caminho simples presentes na Base de Dados.

Dada uma expressão de caminho simples, o índice retorna todos os endereços de nós da Base de Dados XML que casam com aquela expressão. Escolhemos indexar expressões de caminho, e não valores de nós porque, por exemplo, a maior parte das consultas XQuery [35] fazem alguma referência à estrutura do documento, através de expressões XPath [12].

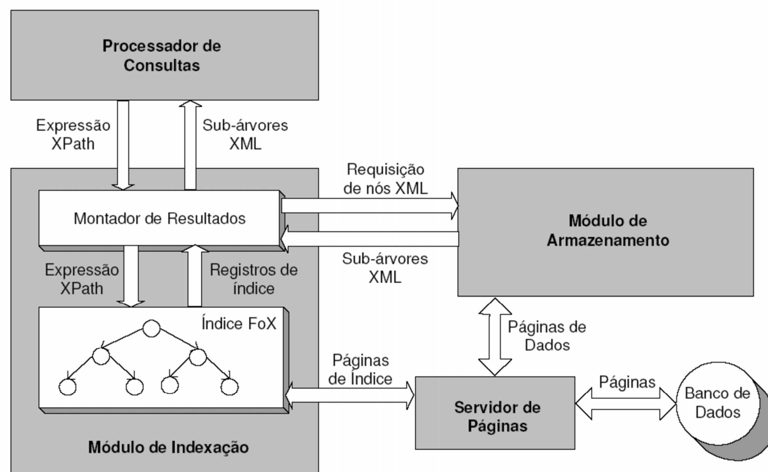


Figura 3.5: Arquitetura do Módulo de Indexação do FoX

O Módulo de Indexação é formado por dois componentes principais: (i) o sub-módulo referente ao índice que, por sua vez, é chamado de Índice FoX, e (ii) o Montador de Resultados. A figura 3.5 mostra a arquitetura do Módulo de Indexação, bem como a sua interação com o Módulo de Armazenamento, Servidor de Páginas e Processador de Consultas.

Primeiramente, o Processador de Consultas envia uma expressão de caminho e ao Montador de Resultados, que por sua vez, a repassa ao Índice FoX. Em seguida, páginas de índice são requisitadas, através de chamadas ao Servidor de Páginas. Estas páginas armazenam nós de índice, necessários à resolução de e. De posse destes nós de índice, realizamos uma busca e chegamos a um conjunto R de registros de índice. Cada registro em R corresponde ao endereço de um nó XML na Base de Dados, alcançável através de e. A figura 3.6 representa graficamente os nós e os registros do Índice FoX.

O conjunto R é repassado ao Montador de Resultados que, por sua vez, é responsável por recuperar os nós, cujos endereços estão nos registros, através de requisições ao Módulo de Armazenamento. Em seguida, o Módulo de Armazenamento retorna todas as sub-árvores XML na Base de Dados, cujas raízes são os nós requisitados no passo anterior. Finalmente, estas sub-árvores são retornadas ao Processador de Consultas.

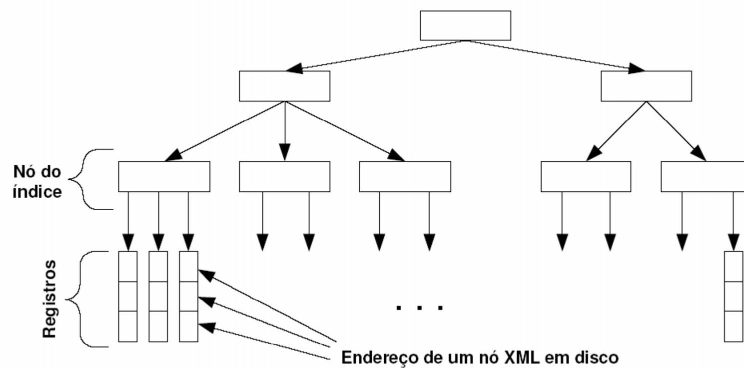


Figura 3.6: *Layout* do Índice FoX

No atual estágio do Índice FoX, apenas expressões de caminho simples são resolvidas. Maiores detalhes sobre o Módulo de Armazenamento e seus componentes podem ser encontrados em [37].

3.4 Processador de Consultas

Como vimos no capítulo 2, o Processador de Consultas é o módulo do sistema de banco de dados responsável por realizar as tarefas de reconhecimento da consulta, geração e otimização dos planos lógicos e físicos, além do acionamento dos métodos que implementam as operações especificadas no plano físico da consulta. O módulo de otimização está fora do escopo desse trabalho. A figura 3.7 mostra os principais módulos do processador de consultas do FoX, bem como sua interação com os demais módulos desse sistema.

O primeiro aspecto relevante no processamento de consultas diz respeito à escolha da linguagem de consulta adotada. A fim de prover maior conformi-

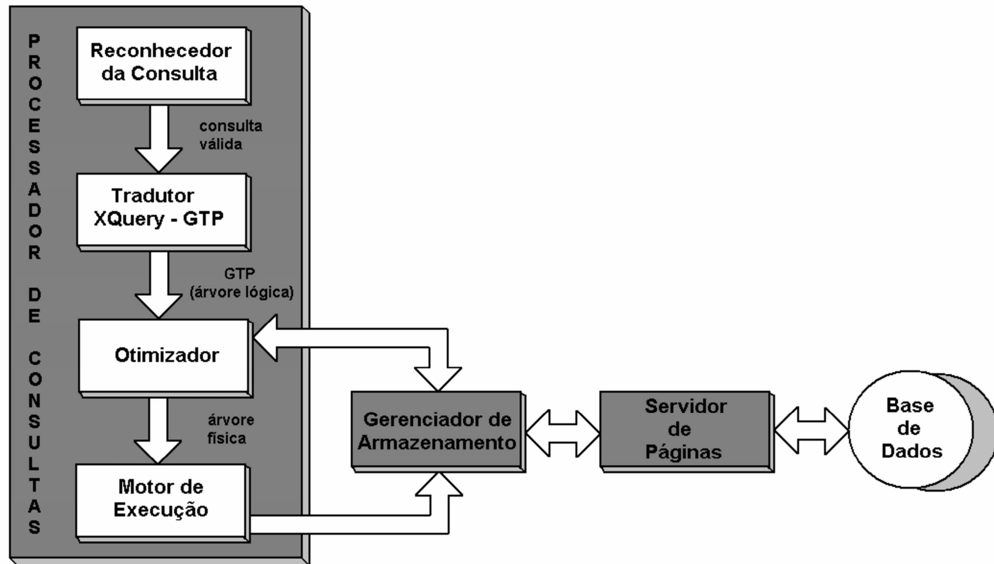


Figura 3.7: Arquitetura Processador de Consultas FoX

dade com as recomendações do W3C, além de ser uma linguagem amplamente aceita, o Processador de Consultas do FoX suporta como linguagem de consulta um subconjunto substancialmente expressivo da XQuery, cuja semântica é definida em [15].

No contexto do Processador de Consultas do FoX, o plano lógico de uma consulta é equivalente ao conceito de GTP (*Generalized Tree Patterns*), conforme definido nas seções 4.1 e 4.2. GTP's são árvores anotadas de padrões obtidas a partir de consultas XQuery. De forma geral, uma GTP contém a estrutura dos dados solicitados e é capaz de expressar consultas de acordo com a gramática definida na figura 3.8.

FLWR ::= (ForClause | LetClause)+ WhereClause ReturnClause.
ForClause ::= FOR $\$fv_1$ IN E_1, ..., $\$fv_n$ IN E_n.
LetClause ::= LET $\$lv_1$:= E_1, ..., $\$lv_n$:= E_n.
WhereClause ::= WHERE $\varphi(E_1, ..., E_n)$.
ReturnClause ::= RETURN $\{E_1\}...\{E_n\}$.
E_i ::= FLWR $|$ XPATH.

Figura 3.8: Gramática para um sub-conjunto da XQuery.

Por simplicidade, assumimos que a gramática é como segue:

- os predicados atômicos permitidos na fórmula booleana φ são aqueles que envolvem apenas operações de comparação, nomeadamente , $<$, $\leq$, $=$, $\neq$, $>$, $\geq$, ou o predicado *empty(*FLWR*)*. Os operandos de uma operação de comparação podem ser um dos seguintes: uma constante c, uma expressão XPath *XPE* ou uma *agg(XPE)*, onde *agg* é uma das funções de agregação.

- ciclos não são permitidos.

O capítulo 4 discute em maiores detalhes o conceito de GTP, bem como apresenta técnicas de tradução XQuery - GTP e GTP - plano físico.

Uma vez que a GTP correspondente à consulta de entrada é obtida e seu respectivo plano de execução é gerado, o Motor de Execução é acionado para solicitar ao Gerenciador de Armazenamento as páginas necessárias para a obtenção das informações requisitadas.

O primeiro grande desafio do Motor de Execução é realizar de forma eficiente o casamento de padrões entre os dados recebidos do Gerente de Armazenamento e as condições especificadas na GTP, parâmetro do operador de obtenção dos dados. Um método de casamento de padrões sobre árvores foi desenvolvido a fim de superar algumas deficiências encontradas em propostas similares e prover ganhos de desempenho. Além disso, índices sobre as expressões de caminho buscadas são amplamente usados.

Capítulo 4

Árvore Genérica de Consulta

Nesse trabalho utilizamos a idéia de árvore genérica de consulta (GTP - *Generalized Tree Pattern*), desenvolvida por Jagadish et al em [10] como mecanismo capaz de capturar a semântica de consultas XQuery. Para isso, o conceito inicial de *GTP Básica* foi estendido a fim de prover melhor suporte ao reconhecimento dos relacionamentos *antecessor-descendente* e *pai-filho*, aqui notacionados como *ad* e *pf*, respectivamente.

A seção 4.1 apresenta a definição de *GTP Básica*. GTPs Básicas são obtidas a partir de consultas XQuery e contém a estrutura dos dados solicitados. Comparativamente, no modelo de processamento utilizado (veja seção 3.4), uma GTP é equivalente ao plano de consulta, uma vez que é capaz de armazenar a semântica da consulta.

Respostas a estas árvores de consultas são obtidas através do casamento de padrões entre a estrutura solicitada na consulta e o documento XML.

GTPs Básicas são capazes de responder apenas ao sub-conjunto da XQuery definido na figura 3.8. No entanto, em [10] extensões são propostas a fim de capturar também a semântica de junções, agrupamentos, agregações e consultas aninhadas. Tais extensões são apresentadas na seção 4.2. Isso é muito importante, pois permite a construção de uma álgebra específica para tratamento de dados XML, além de fornecer base para uma implementação eficiente das demais operações da álgebra física. A seção 4.3 descreve como ocorre o processo de geração dos planos lógico e físico, bem como, apresenta os de-

mais operadores da álgebra física relacionada. Novamente, alguns operadores sofrem alterações com relação às versões inicialmente definidas em [10].

4.1 GTPs Básicas

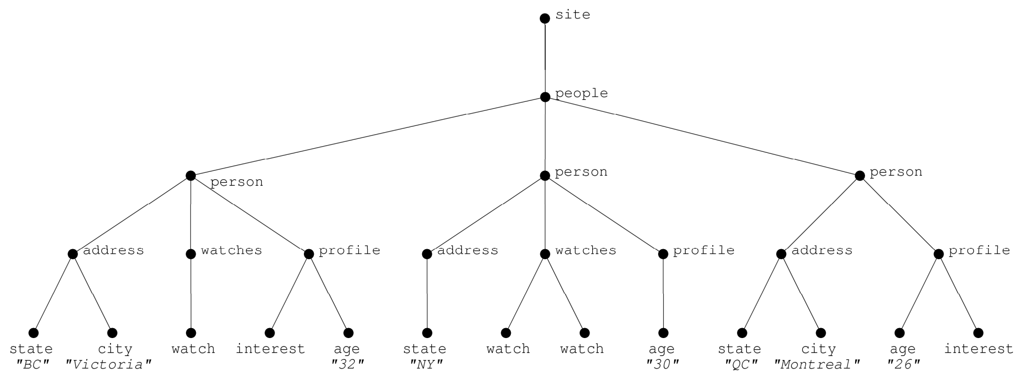
Definição 1: Uma *GTP Básica* é um par $G = (T,F)$, onde T é uma árvore e F é uma fórmula booleana, tal que:

1. cada nó de T é rotulado por uma variável distinta e tem um campo que indica o número do grupo ao qual pertence;
2. cada nó folha de T possui uma anotação, denotada por caminho, que indica se algum relacionamento do tipo *ad* ocorre em sua ascendência;
3. cada nó folha de T possui uma anotação que funciona como um identificador único. Tais campos são preenchidos numerando-se os nós em questão da esquerda para direita em ordem crescente.
4. cada aresta de T possui um par $<x,m>$, onde $x \in$ *{pf, ad}* representa o relacionamento entre os nós que formam a aresta e $m \in$ *{mandatório, opcional}* indica a relevância dessa aresta nas árvores resultantes. Sempre que o caminho entre um nó folha e a raiz de G contiver apenas arestas mandatórias, este será dito *mandatório*.
5. F é uma combinação booleana dos predicados aplicados aos nós.

Deste ponto em diante, a notação Q será utilizada para uma árvore de consulta GTP. O presente trabalho estende o conceito de GTP Básica proposto em [10] através do acréscimo dos tópicos 2 e 3 da definição acima. Tais extensões se fizeram necessárias para melhoria do suporte aos relacionamentos *ad* na estratégia adotada e para a identificação única dos padrões solicitados, respectivamente.

Considere a massa de dados exposta em 4.1(a). A figura 4.1(c) é um exemplo de GTP Básica para a consulta exposta em 4.1(b) Graficamente, as linhas

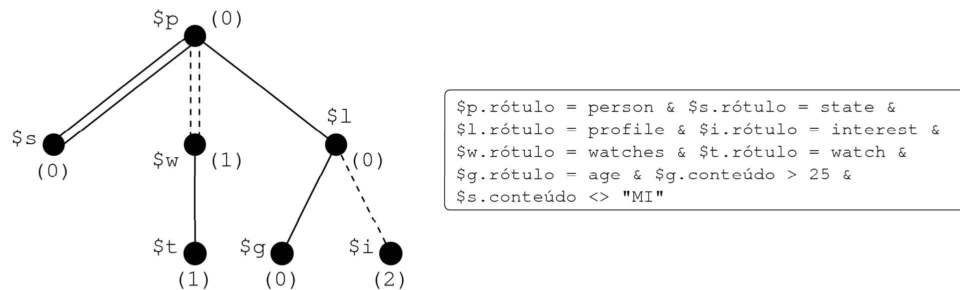
pontilhadas representam arestas opcionais, já linhas sólidas, mandatórias; linhas duplas indicam relacionamentos antecessor-descendente e simples, pai-filho. Na figura 4.1(c), o texto contido no retângulo ao lado da estrutura gráfica da GTP Básica trata-se de anotações que representam os predicados impostos pela consulta, este texto também recebe o nome de *fórmula da GTP* ou *fórmula da junção*, quando os predicados se referem a uma operação de junção.



(a) Documento

```
for $p in document(¨auction.xml¨)//person, $l in $p/profile
where $l/age > 25 and $p/state <> ¨MI¨
return <result> {$p//watches/watch} {$l/interest} </result>
```

(b) Consulta XQuery



(c) GTP Básica

Figura 4.1: Consulta XQuery e sua consulta GTP correspondente.

Um *grupo* é o conjunto máximo de nós de uma GTP que são conectados apenas por arestas mandatórias. Grupos são numerados aleatoriamente, mas, por convenção, o grupo dos nós contidos na cláusula FOR é o grupo 0. Na figura 4.1, o grupo de cada nó está indicado entre parênteses.

Seja G = *(T,F)* uma GTP e C uma coleção de árvores. O mapeamento parcial $h : T \rightarrow C$ é tal que:

- h é definida sobre todos os nós do grupo 0;
- se h está definida para um nó de um grupo, então ela necessariamente deve estar definida para os demais nós desse grupo
- h preserva os relacionamentos estruturais em G, ou seja, se h é definida sobre os nós u,v e se há uma aresta *(u,v)* em G do tipo *pf*, então *h(u)* é *filho* de *h(v)*;
- Da mesma forma, se h é definida sobre os nós u,v e se há uma aresta *(u,v)* em G do tipo *ad*, então *h(u)* é *descendente* de *h(v)*;
- h satisfaz F.

A função h é um casamento de padrões parcial, pois elementos conectados por arestas opcionais não são mapeados. A solução encontrada para resolver essa parcialidade foi estender os conectivos da álgebra booleana para tratar o valor verdade *indefinido*, denotado por $\perp$ e o atribuir para cada condição dependente sobre um nó não mapeado por h. A figura 4.2 mostra a extensão da álgebra booleana descrita. Resumidamente, a extensão trata $\perp$ como uma identidade para os conectivos $\wedge$ e $\vee$ e como um complemento para $\neg$. Dessa forma, h satisfaz F sss h avalia F como *verdadeira*. O casamento de padrões parcial de uma GTP é *válido* se ele satisfaz à fórmula booleana associada à GTP.

$\wedge$	$\perp$	1	0
$\perp$	$\perp$	1	0
1	1	1	0
0	0	0	0

$\vee$	$\perp$	1	0
$\perp$	$\perp$	1	0
1	1	1	1
0	0	1	0

$\neg$	$\perp$	1	0
	$\perp$	0	1

Figura 4.2: Extensão da álgebra booleana para tratar o valor verdade *indefinido*.

4.2 GTPs

Nas subseções a seguir serão descritas algumas extensões às GTPs Básicas a fim de tornar possível o tratamento de consultas envolvendo junções, agrupamentos, agregações, quantificadores e aninhamentos.

4.2.1 Junção

Para facilitar a compreensão, mostramos como GTPs capturam a semântica de junções através de um exemplo. Considere o exemplo da figura 4.3. Para todo par de elementos PERSON e OPEN_AUCTION que satisfazem às condições na cláusula *where*, precisamos encontrar os sub-elementos INTEREST correspondentes e os sub-elementos BIDDER do elemento OPEN_AUCTION quando este é referenciado pelo atributo OPEN_AUCTION do sub-elemento WATCH de PERSON. Para todo par (PERSON, OPEN_AUCTION), a existência de elementos correspondentes para o primeiro e segundo argumentos é independente. Essa lógica é capturada corretamente no par de GTPs da figura 4.3, um para cada operando da junção.

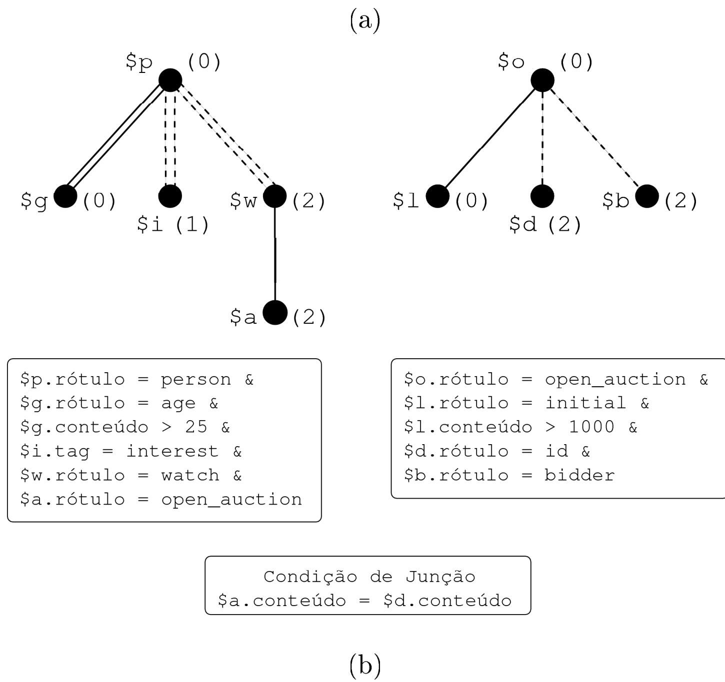
4.2.2 Agrupamento, Agregação e Quantificadores

Nessa seção discutiremos a respeito das extensões necessárias para que GTPs suportem agrupamento, agregação e quantificadores. Assumimos que não há expressão FLWR aninhada em expressões que envolvem quantificadores. Note que uma consulta envolvendo o quantificador *some* pode ser re-escrita em uma outra equivalente sem usá-lo. Por exemplo, a expressão "where some *$v* in *XPathExpression* satisfies *expression*" é equivalente, de acordo com a semântica XQuery [15], a "where *newExpression*", onde *newExpression* é expressão com todas as ocorrências de *$v* substituídas por *XPathExpression*.

Agregação convencional de valores é realizada diretamente pela construção de GTPs básicas; já agregações estruturais, onde coleções são agrupadas para formar novos grupos são tratadas através de consultas aninhadas, discutidas na seção 4.2.3.

```
for $p in document("auction.xml")//person,
    $o in document("auction.xml")//open_auction
where $p//age>25 and $o/initial>1000
return
    <result>
    {$p//interest}
    {$o[@id=$p//watch/@open_auction]/bidder}
    </result>
```

(a)

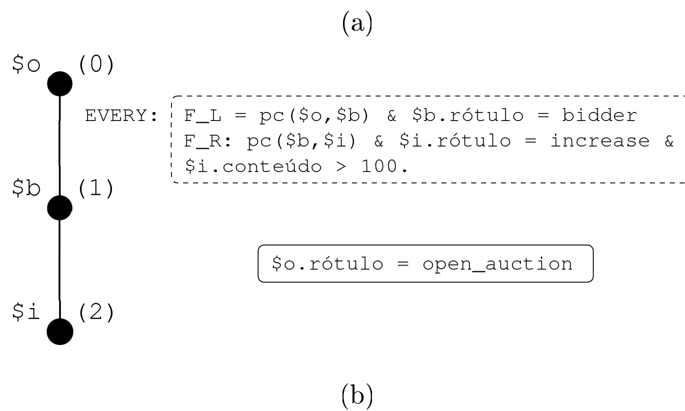


(b)

Figura 4.3: Uma consulta com junção e sua GTP correspondente.

```
for $o in document("auction.xml")//open_auction
where every $b in $o/bidder satisfies $b/increase > 100
return <result> {$o} </result>
```

(a)



(b)

Figura 4.4: Uma consulta com quantificação universal e sua GTP correspondente.

Resolver o quantificador *every* requer extensões às GTPs. Uma *GTP Universal* é uma GTP $G = (T,F)$, na qual algumas arestas podem ser rotuladas com 'EVERY'. Para isso é necessário que:

- a GTP inclua um par de fórmulas associado a uma aresta EVERY, por exemplo, F_L e F_R, que são combinações booleanas aplicáveis aos nós, inclusive os estruturais;
- nós abaixo da aresta EVERY que são mencionados em F_L devem ser separados em um grupo;
- nós abaixo da aresta EVERY que são mencionados em F_R , ou seja, não estão em F_L, devem ser separados em um novo grupo.

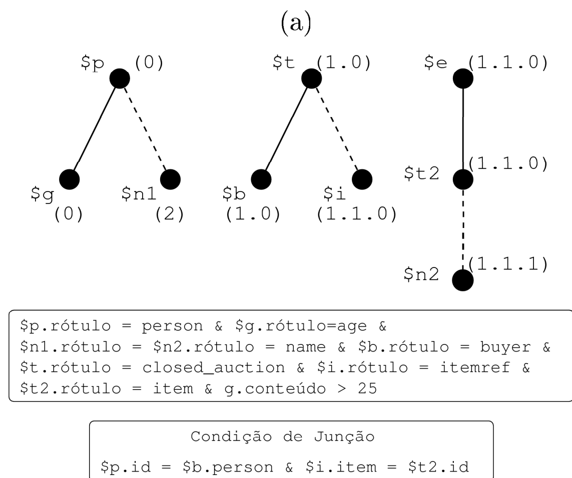
Na figura 4.4 temos um exemplo de GTP Universal. A GTP codifica a condição que para todo BIDDER *$b* que é sub-elemento do elemento OPEN_AUCTION *$o*, existe um sub-elemento INCREASE do BIDDER com valor superior a 100.

4.2.3 Consultas Aninhadas

A resolução de consultas aninhadas através de GTPs utiliza um dispositivo de esquema de numeração hierárquico dos grupos que é capaz de capturar a dependência entre os blocos de consulta (relacionamento consulta/sub-consulta). A idéia é adicionar um nível de hierarquia ao número do grupo cada vez que entrar em um novo bloco na construção da GTP, como ilustrado na figura 4.5.

```
for $p in document(¨auction.xml¨)//person
let $a :=
    for $t in document(¨auction.xml¨)//closed_auction
    where $p/@id=$t/buyer/@person
    return <item>
           {for $t2 in document(¨auction.xml¨)//europe/item
           where $t/itemref/@item=$t2/@id
           return {$t2/name}}
           </item>
where $p//age > 25
return <person name=$p/name/text()> {$a} </person>
```

(a)



(b)

Figura 4.5: Uma consulta envolvendo aninhamento e junção e sua GTP correspondente.

4.3 Geração dos Planos Lógico e Físico

Nesta seção apresentaremos brevemente o algoritmo desenvolvido em [10] para realizar a transformação de uma consulta XQuery em uma GTP conforme definida na seção 4.2, maiores detalhes sobre este podem ser obtidos em [10].

Jagadish et al em [10] também desenvolveu um algoritmo para a tradução de uma GTP em um plano físico. No entanto, descreveremos na seção 4.3.2 o algoritmo para a realização desta tarefa já adaptado pelo presente trabalho. Modificações a este algoritmo se fizeram necessárias pois o operador de casamento de padrões foi inserido na álgebra física. Além disso, no algoritmo original, índices são usados para recuperar todos os elementos da base que possuem mesmo rótulo que qualquer nó da GTP.

Antes de descrever os algoritmos mencionados, é importante ressaltar que a GTP obtida é, no contexto deste trabalho, equivalente ao plano lógico da consulta, por isso, uma vez de posse de uma GTP o próximo passo para o processamento da consulta é gerar o plano físico a partir desta. Inclusive, em [10], algumas técnicas de otimização são apresentadas, não somente para melhoria do plano lógico, mas também do físico.

4.3.1 Traduzindo XQuery para GTP's

O algoritmo GTP, figura 4.6, é capaz de traduzir o sub-conjunto XQuery especificado na figura 3.8 com as seguintes restrições:

- as cláusulas FOR precedem as cláusulas LET em uma mesma expressão FLWR;
- nenhuma expressão FLWR aparece no predicado φ;
- nenhuma expressão FLWR aparece nas cláusulas de quantificação (*some* e *every*).

O algoritmo possui um ambiente de *parsing* *ENV* para guardar a informação coletada do *parsing*. A função *buildTPQ(xp)*, constrói uma parte da

GTP a partir de *xp*, uma expressão XPath estendida de uma variável. Sempre que *xp* começa com uma função *document*, uma nova GTP é adicionada a *ENV*; caso *xp* inicie-se com uma variável, o nó padrão associado a esta variável é localizado e a nova parte resultante de *xp* inicia-se dele. A função examina cada elemento contido em *xp*, cria uma nova aresta e um novo nó, anota as arestas com o tipo de relacionamento associado e adiciona um predicado sobre o rótulo do nó e/ou suas propriedades. Quaisquer expressões de filtragem são tratadas de forma similar a uma cláusula WHERE simplificada. Os números produzidos para os nós GTPs são *strings* de números. O algoritmo aceita um número de grupo g como seu parâmetro, sendo este a *string* vazia em sua primeira execução. A notação g + *".x"* é usada para anexar o número x à *string* g e $g + 1$ para adicionar 1 ao número mais a direita de g.

De forma resumida, o algoritmo se desenvolve de acordo com os seguintes passos:

1. processa a cláusula FOR. Linhas 6-8. O *parsing* é realizado para cada variável que ocorre na cláusula FOR. Observe que o número de grupo permanece inalterado uma vez que elementos que ocorrem na cláusula FOR formam o primeiro grupo de nós existentes na consulta.

2. processa a cláusula LET. Linhas 10-13. Novamente o *parsing* é realizado, desta vez, para cada expressão, um novo número de grupo é construído.

3. processa a cláusula WHERE. Linhas 15-39. Aqui, predicados do tipo "*todo* E_L *satisfaz* E_R" têm seus parâmetros construídos separadamente, com novos números de grupo criados para cada um. Para os demais casos, cada expressão que ocorre no predicado é construído analogamente e quando agregações ocorrem, o número de grupo dos nós interligados ao nó onde a função é aplicada devem ser atualizados a fim de eliminar aninhamentos desnecessários.

4. processa a cláusula RETURN. Linhas 41-44. Nesta fase, cada uma das expressões existentes na cláusula é construída, gerando novos números de grupo para cada uma delas.

Maiores detalhes sobre este algoritmo podem ser obtidos em [10].

```
GTP
 1  Entrada: uma expressão FLWR Exp, um número de grupo g
 2  Saída: uma GTP ou GTPs com uma fórmula de junção
 3  if (último nível de g ≠ 0)
 4     then g ← g + ".0"
 5  /* fase 1 - processar cláusula FOR
 6  para cada ("for $fv ∈ E")
 7        do PARSE(E,G)
 8  ng ← g
 9  /* fase 2 - processar cláusula LET
10  para cada ("Let $lv ← E")
11        do
12           ng ← ng + 1
13           PARSE(E,NG)
14  /* fase 3 - processar cláusula WHERE
15  para cada predicado p ∈ φ
16        do if (p = "todo E_L satisfaz E_R")
17              then
18                    ng ← ng + 1
19                    parse(E_L, ng)
20                    seja F_L a fórmula associada ao padrão resultante de E_L
21                    ng ← ng + 1
22                    parse(E_R, ng)
23                    seja F_R a fórmula associada com o padrão resultante de E_R
24     else
25           para cada E_i como argumento de p
26                 do PARSE(E_i,G)
27           adicione p à fórmula da GTP ou à fórmula de junção
28           if (p = "count($n') > c" and c ≥ 0)
29              then
30                    g' ← grupo($n')
31                    if (g é prefixo de g')
32                       then
33                             atualizar o número de grupo de todos os nós em g' para g
34           if (p refere-se a max(min/média/soma)($n') e $n && grupo($n) == g)
35              then
36                    g' = grupo($n')
37                    if (g é prefixo de g')
38                       then
39                             atualizar o número de grupo de todos os nós em g' para g
40  /* fase 4 - processar a cláusula RETURN
41  para cada "E_i"
42        do
43  ng ← ng + 1
44  PARSE(E, NG)
45  end ALGORITMO
```

```
parse
1  Entrada: expressão FLWR ou expressão XPath E, número de grupo g
2  Saída: parte da GTP resultante de E
3  if (E é uma expressão FLWR)
4     then GTP(E,G)
5     else BUILDTPG(E)
6           end procedure
```

Figura 4.6: Algoritmo GTP.

4.3.2 Traduzindo uma GTP para um Plano Físico

A utilização de GTPs para a identificação das propriedades da consulta fornece base para uma implementação eficiente das demais operações da álgebra física. Antes de discutirmos a respeito do algoritmo para a geração do plano físico, descreveremos brevemente os demais operadores, especificados em [10], que compõem este conjunto.

Operadores Físicos

Index Scan - $IS_{exp}(S)$. Retorna os elementos da base de dados S cuja ascendência está de acordo com a especificada em *exp* através do uso de índices. *exp* é uma expressão de caminho simples.

Filtro - $F_p(S)$. Retorna os elementos de S que satisfazem o predicado p. A ordem da seqüência de entrada é preservada na saída.

Projeção - $P_l(S)$. Seleciona uma sub-árvore *S'* de S, onde para todo elemento e folha em *S'*, e pertence à lista l.

Ordenação - $O_b(S)$. Dada uma seqüência de árvores S, o operador ordena S de acordo com a lista base de ordenação, b. A ordem da saída reflete o procedimento de ordenação (por exemplo, por valor ou por identificador do nó especificado).

Junção por Valor - $J_p(S_1, S_2)$. Recebe duas seqüências de árvores como parâmetro e um predicado de junção. A operação de junção se baseia na comparação de valores, conforme especificado em p. A ordem de S_1 é mantida na saída. Um algoritmo similar ao *sort-merge join* utilizado em SGBDs tradicionais é usado, com exceção ao fato de que uma ordenação extra é incluída para que a saída siga a ordem de S_1. Como no mundo relacional, a álgebra física inclui também o *left-outer join* onde cada árvore de S_1 é retornada na saída se não combinar com nenhuma árvore de S_2.

Agrupamento - G_b(S). Recebe uma seqüência de árvores S, ordenada de acordo com a base de agrupamento b. O operador agrupa essas árvores conforme b. Para cada grupo ele cria uma árvore de saída contendo nós artificiais para servirem de raiz, sub-raiz e base de agrupamento, além dos nós pertencentes às árvores agrupadas. A ordem dada na entrada é mantida.

Agregação - $A_{in,\ on}$ (S, nome). Para cada árvore de S, aplica a função de agregação especificada em *nome* sobre o nó referenciado pela expressão *in* e armazena o resultado no nó referenciado por *on*. A ordem de S é preservada na saída.

Merge - M(S_1, ..., S_n). Recebe n seqüências de árvores de cardinalidade k como entrada e realiza um *n-way merge* dessas seqüências. Para cada $1 \leq i \leq k$, ele aglutina a árvore i de cada entrada sob uma raiz artificial e produz uma árvore de saída. A ordem é preservada.

Geração do Plano Físico

Esta seção apresenta um algoritmo para criação de um plano de execução a partir de uma árvore de consulta como definida nas seções 4.1 e 4.2.

Este trabalho acrescenta algumas modificações ao desenvolvido por Jagadish et al. em [10] uma vez que adiciona o operador responsável pela identificação de padrões de consultas e dispensa o uso do operador de junção estrutural. Além disso, em seu formato original, índices são usados para a captura de todos os elementos envolvidos no processo de execução da consulta. O grande problema dessa abordagem é o excessivo número de junções estruturais necessárias à obtenção do resultado, além da quantidade de nós inválidos recuperados. No contexto deste trabalho, índices recuperam apenas os elementos do documento que podem ser obtidos através de uma expressão de caminho e desde que e seja igual ao percurso estrutural de um nó folha de Q até a sua raiz.

PlanGen

```
1  Entrada: GTP G
2  Saída: plano físico para avaliar G
3  GRPs ← OrdemDeAvaliaçãoDosGrupos(G)
4  para cada grupo g em GRPs
5        do
6            GB ← groupBasis(g)
7            /* fase 1
8                  do exp ← achaExpressão(f, G)
9                      adicione IndexScan ao plano de execução, tendo como entrada exp
10           /* fase 2
11           C ← {p | p é um predicado na fórmula da GTP, p refere-se a um nó em g,
12           p é avaliável e p ainda não foi avaliado}
13           adicione o operador Filtro ao plano, tendo como argumentos a fórmula de C
14           e a saída do IndexScan relacionado
15           /* fase 3
16           T ← F ∪ IS - F, onde F e IS referem-se às saídas
17           dos operadores Filtro e IndexScan, respectivamente
18           adicione um operador de Ordenação ao plano de execução, tendo como entrada T
19           e a posição fim de seus elementos como chave de entrada
20           /* fase 4
21           adicione XML-PM ao plano de execução da consulta, tendo como entrada
22           a saída da fase anterior e a estrutura de G
23           /* fase 5
24           while (∃ predicado p na fórmula de junção e p refere-se a uma fórmula em g
25           e p é avaliável e p ainda não foi avaliado)
26                 do JC ← o conjunto dos predicados ps dependentes das mesmas entradas
27                     adicione VJ ao plano, tendo como argumento a fórmula de JC
28                     if (∃p ∈ JC e p refere-se a um nó em outro grupo)
29                        then atribua a VJ outer join
30                             torne a saída do passo precedente a stream de entrada da direita de VJ
31           /* fase 6
32           AG ← {agg($n) | $n em g e agg($n) na fórmula da GTP}
33           adicione o operador Agrupamento ao plano, tendo como entrada GB
34           e as agregações apropriadas
35           /* fase 7
36           AC ← {p | p é um predicado na fórmula da GTP e p refere-se a um nó em AG e
37           p é avaliável e p ainda não foi avaliado}
38           adicione o operador Filtro ao plano, tendo como argumentos a fórmula de AC
39           e a saída do passo precedente
40           /* fase 8
41           while (∃ predicado p na fórmula de junção e p refere-se a um nó em AG
42           e p é avaliável e p ainda não foi avaliado)
43                 do AJC ← conjunto dos predicados ps que dependem das mesmas entradas
44                     adicione VJ ao plano, tendo como argumento a fórmula de AJC
45                     if (∃p ∈ JC e p refere-se a um nó em outro grupo)
46                        then atribua a VJ outer join
47                             faça a saída do passo precedente ser a entrada da direita de VJ
48           /* fase 9
49           if (g tem um argumento de retorno)
50              then adicione o operador de agrupamento ao plano, tendo como entrada GB,
51                   se necessário
```

Figura 4.7: Algoritmo para Tradução de uma GTP para um Plano de Execução.

De forma resumida, o algoritmo executa os seguintes passos:

1. Recupera os nós do documento equivalentes às folhas da GTP. Linhas 8-9.

   Observe que os nós sempre são recuperados através do operador IndexScan, com base na expressão de caminho extraída da GTP.

2. Filtra os resultados com base nos predicados avaliáveis (um predicado é avaliável quando todos os seus nós de padrões dependentes são limitados ou as agregações já foram computadas). Linhas 11-14.

   Neste passo o operador FILTRO é inserido, tendo como operandos os elementos recuperados no passo anterior e o predicado associado a estes.

3. Ordena a saída da fase anterior de acordo com a posição *fim* dos elementos que a compõem. Linhas 16-19.

4. Calcula a identificação dos padrões requisitados sobre os elementos retornados da fase precedente. Linhas 21-22.

   Nesta fase o operador XML-PM é inserido no plano de execução da consulta. Observe que nenhum predicado da fórmula de G é utilizado, uma vez que, tal operador realiza apenas identificações estruturais.

5. Calcula junções de valor, se necessário. Linhas 24-30.

   Observe, nas linhas 28-29, que sempre que o predicado de junção se referir a elementos em grupos diferentes da GTP, um OUTER JOIN é inserido no plano em lugar da JUNÇÃO POR VALOR.

6. Calcula agregações, se necessário. Linhas 32-34.

   Acrescenta o operador de AGRUPAMENTO ao plano, tendo como parâmetros GB e as funções de agregação adequadas.

7. Filtra os resultados com base nos predicados dependentes dos valores de agregação, se necessário. Linhas 36-39.

   Insere o operador FILTRO tendo como operando a saída do passo anterior e um predicado p, é aplicado nos casos em que p inclui uma função de agregação.

8. Calcula junções de valor baseadas em valores de agregação, se necessário. Linhas 41-47.

Este caso é análogo ao passo de inserção de junções por valor, no entanto, se restringe aos casos em que o predicado solicitado envolve funções de agregação.

9. agrupa o argumento retornado, se houver. Linhas 49-51.

   Adiciona o operador de agrupamento tendo como entrada GB (base de agrupamento).

4.4 Conclusões

Em virtude da natureza heterogênea intrínseca aos dados XML e das limitações encontradas ao adaptá-los a outros formatos, é natural imaginarmos que as estruturas adequadas para manipulá-los devam ser capazes de gerenciar suas flexibilidades.

O conceito de GTP definido em [10] mostra-se adequado a essa tarefa uma vez que é capaz de expressar uma porção relevante da XQuery, das quais podemos destacar as noções de junção, agrupamento, agregações, quantificações e aninhamentos. Além disso, em [10] algoritmos para a tradução de uma consulta XQuery para sua representação GTP, bem como, para construção de um plano físico são apresentados. Mais, ainda algumas técnicas de otimização são sugeridas a fim de prover melhores tempos de resposta.

No capítulo seguinte, é descrito como GTPs são utilizadas no reconhecimento de padrões sobre o documento XML, bem como algumas adaptações necessárias ao aprimoramento dessas estruturas.

Capítulo 5

Identificação de Padrões sobre Árvores XML

Como discutido no capítulo 2.4, é grande o interesse no desenvolvimento de técnicas capazes de recuperar eficientemente sub-árvores XML contendo determinadas restrições, em especial, estratégias de casamento de padrões sobre os documentos XML têm sido bastante exploradas. Essa seção apresenta um método eficiente para o casamento de padrões entre a estrutura requerida em uma consulta e o documento XML que forma a base de dados. A solução aqui proposta adota o conceito de GTP, conforme descrito no capítulo 4, para expressar a semântica de uma consulta.

5.1 Casamento de Padrões sobre Dados XML

SGBDs Relacionais, por possuírem esquemas, possibilitam que os dados requisitados em uma consulta possam ser obtidos diretamente de suas bases através da filtragem dos valores especificados nos predicados, facilitando assim a localização dos atributos em análise. No entanto, no contexto de SGBDs XML Nativos, dispor de tamanha riqueza a respeito da estrutura dos dados torna-se extremamente dispendioso do ponto de vista do consumo de espaço. Dessa forma, dada uma consulta, reconhecer quais fragmentos do documento original a respondem, é tarefa fundamental durante sua execução.

Intuitivamente, como um documento XML pode ser representado através de uma árvore, é razoável imaginarmos que a estrutura adequada para sua manipulação também esteja neste formato. No entanto, a natureza heterogênea dos dados XML acrescenta dificuldades no tratamento destes, por isso, este trabalho utiliza árvores especiais que possuem uma estrutura suficientemente flexível capaz de capturar a semântica de consultas sobre dados XML.

Nas seções seguintes, apresentamos um mecanismo capaz de resolver eficientemente esse problema.

5.2 Algoritmo XML-PM

Primeiramente, considere como fonte de dados um documento XML D, cuja representação pode ser dada de acordo com o modelo lógico especificado na seção 3.3.1 e considere Q, como descrito em 4.1, uma árvore de consulta.

Através de índices sobre expressões de caminho simples, recuperamos os nós da base de dados equivalentes aos nós folhas da árvore de consulta Q. No exemplo da figura 5.2, os nós equivalentes ao nó b de Q são b_1, b_2, b_3, b_4 e b_4. Uma fila F_D contendo tais nós, ordenados de acordo com o campo *fim*, é fornecida como entrada para o algoritmo, bem como uma lista P contendo os identificadores dos caminhos rotulados como mandatórios na estrutura de Q. A ordenação é importante pois permite identificar quais elementos pertencem à mesma sub-árvore. Perceba que os campos *fim* da árvore poderiam ser vistos como a ordem de visita aos nós desta em um percurso em pós-ordem. A variável *Raiz* aponta para o elemento raiz da árvore resultante em construção. Inicialmente, seu valor é nulo. Para cada elemento e de F_D, verifica-se se o caminho entre ele e o elemento r equivalente à raiz de Q existe conforme especificado em Q. Para cada elemento verificado, se este concorda com o padrão solicitado, um ponteiro é inserido na pilha que armazena resultados temporariamente válidos, se seu campo *caminho* = *"SIMPLES"*, nenhuma verificação extra é necessária, sendo então este inserido na lista S para posterior montagem. Além disso, se o caminho identificado por e for mandatório , retira-se tal ocorrência de P e

armazena-se r em uma variável. Se $e.fim > r.fim$, temos que e não pertence à árvore cuja raiz é r. Nesse momento, se P estiver vazia, todas as arestas mandatórias terão sido encontradas. Logo, os elementos em S formam uma saída válida, que é, então, montada. Caso contrário, o processo deve ser reiniciado e os elementos de F_D anteriormente analisados, descartados. Observe que, através do uso da lista S, construções desnecessárias são evitadas uma vez que as ascendências dos elementos contidos nessa lista somente são montados quando o padrão é completamente satisfeito.

A seguir, maiores detalhes sobre o algoritmo são abordados, no entanto, algumas notações precisam ser estabelecidas antes disso.

- Seja F_D uma fila sendo cada um de seus elementos formado pela quádrupla $<n_D, c_Q, ids, exp>$, onde:
  - n_D é um elemento de D recuperado via índices sobre expressões de caminho simples. A expressão de caminho argumento dos índices é derivada a partir dos nós folhas de Q até a ocorrência de um relacionamento *ad*.
  - c_Q representa o campo *caminho* de um nó folha de Q e indica se o percurso dele até a raiz de Q possui alguma aresta do tipo *ad*.
  - *ids* representa os identificadores dos nós folhas de Q potencialmente correspondentes ao elemento em questão.
  - *exp* é usada para armazenar qual dos caminhos listados em *ids* corresponde ao elemento em questão.
- P é uma lista ordenada contendo os identificadores dos trajetos (*paths*) entre os nós folhas de Q até sua raiz, desde que estes sejam formados apenas por arestas mandatórias.
- S é uma lista usada para guardar os elementos de F_D que formam uma potencial saída válida.
- AchaRaizCaminho(n, *ids*, *exp*), retorna o elemento ascendente a n correspondente à raiz de Q, desde que o caminho entre estes esteja de

acordo com algum dos contidos na lista *ids*, sendo então o identificador desse caminho atribuído à entrada *exp*. Caso esse elemento não seja encontrado, o valor nulo é retornado. Além disso, os nós que formam este caminho são armazenados na pilha que guarda os elementos que formam uma saída potencialmente válida. O algoritmo para essa função é obtido trivialmente através da comparação dos elementos contidos no percurso frente aos caminhos especificados pela entrada *ids*. Dessa forma, sua complexidade depende do tamanho desse caminho e do tamanho da lista *ids*.

- MontaArvore(S, r, pilha) gera a árvore resultante do casamento de padrões, recebendo como entrada a lista S - que armazena nós folhas que fazem parte de uma árvore resposta da consulta - e a pilha que é utilizada para a construção de seus nós internos.

```
XML-PM(F_D, P, Q)
 1  S ← ∅
 2  Raiz ← Nulo
 3  RaizQ ← raiz de Q
 4  while F_D ≠ ∅
 5        do F_D Atual ← ExtraiPrimeiro(F_D)
 6           if Raiz = Nulo ou F_D Atual.c_Q ≠ "SIMPLES"
 7             then Aux ← AchaRaizCaminho(F_D Atual.n_D, RaizQ, F_D Atual.id)
 8                  if Aux ≠ Nulo
 9                    then if Aux ≠ Raiz e Raiz ≠ Nulo
10                           then if P = ∅
11                                  then execute MontaArvore(S, RaizQ,pilha)
12                                S ← ∅
13                                restaure P ao seu estado inicial
14                                Raiz ← Aux
15                         if <F_D Atual.id> ∈ P
16                           then retire de P F_D Atual.id
17                    else  F_D Atual ← Nulo
18             else  if Raiz ≠ Nulo
19                    then if F_D Atual.n_D.fim < Raiz.fim
20                           then insira F_D Atual em S, se possível
21                                if <F_D Atual.id> ∈ P
22                                  then retire de P F_D Atual.id
23                           else  if P = ∅
24                                  then execute MontaArvore(S, RaizQ, pilha)
25                                S ← ∅
26                                restaure P ao seu estado inicial
27                                Raiz ← AchaRaizCaminho(F_D Atual.n_D, RaizQ, F_D Atual.id)
28                                if Raiz ≠ Nulo
29                                  then if <F_D Atual.id> ∈ P
30                                         then retire de P F_D Atual.id
31  if P = ∅
32     then execute MontaArvore(S, RaizQ, pilha)
```

Figura 5.1: Algoritmo para casamento de padrões

A idéia do algoritmo consiste em percorrer os elementos de F_D verificando se as seguintes caracteríticas são respeitadas:

- suas ascendências estão de acordo com os padrões requisitados em Q: elementos cujo campo *caminho* não são rotulados como *"SIMPLES"*, precisam ser sempre verificados, uma vez que, seus caminhos não estão completamente especificados em Q.

- os elementos que formam uma potencial saída válida pertencem à mesma subárvore: como F_D é ordenada de acordo com a posição *fim* de seus elementos, para garantir que esta característica seja cumprida, antes de ser considerado parte de uma saída potencialmente válida, sua posição *fim* é comparada à da raiz dos elementos que a formam.

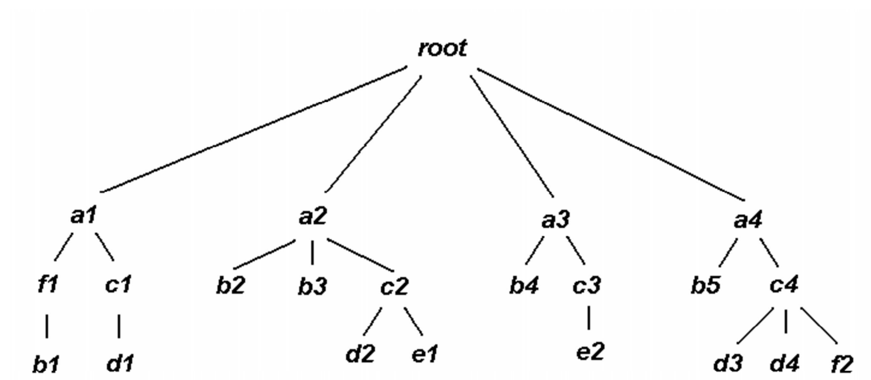
- sempre que um elemento possui ascendência de acordo com algum dos padrões estabelecidos em Q, o identificador relacionado à sua ascendência deve ser retirado de P, se possível.

Uma saída é montada apenas quando considerada de fato válida, ou seja, ao recuperar um elemento de F_D cujo campo *fim* seja superior ao contido em *Raiz*, se a lista P estiver vazia, os elementos contidos em S juntamente com os da pilha são base para compor uma saída válida.

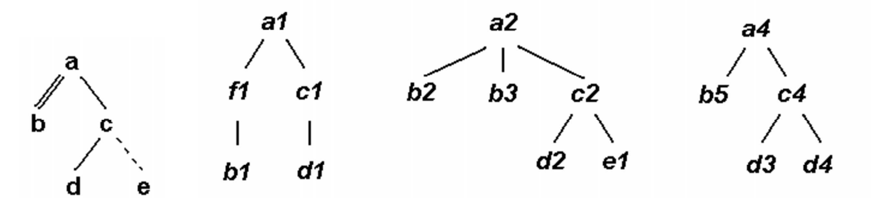
Exemplo: Considere D a fonte de dados exposta na figura 5.2(a) e Q uma consulta ilustrada através árvore ilustrada em 5.2(b).

Nesse caso, os elementos contidos em F_D são $\{b_1, d_1, b_2, b_3, d_2, e_1, b_4, e_2, d_4, b_5, d_3, d_4\ \}$ e $P = \{id_1, id_2\}$, onde $id_1 = <b\backslash\backslash a>$ e $id_2 = <d\backslash c\backslash a>$.

O processo é iniciado a partir de b_1. AchaRaizCaminho retorna a_1 à variável *Aux*, e , posteriormente, *Raiz* recebe este valor. Assim, b_1 é inserido em S e id_1 é retirado de P. Então o processo é reiniciado para d_1. Como o caminho associado a d_1 em Q é rotulado como "SIMPLES" e $Raiz = a_1$, o próximo passo é verificar se d_1 também é descendente de a_1 (linha 19). Assim, d_1 é inserido em S e id_2 retirado de P. O próximo elemento retirado de F_D é b_2. Novamente, AchaRaizCAminho é realizada, pois id_1 contém aresta dupla. Então, *Aux* assume o valor a_2 e, como P está vazia, MontaÁrvore é



(a) Fonte de dados



(b) Árvore de padrões simples

(c) Árvores Resultantes

Figura 5.2: *Casamento de Padrões.*

acionado. Depois disso, S e P voltam aos seus valores iniciais e $Raiz$ passa a valer a_2, indicando que agora esta será a raiz dos elementos a serem inseridos em S. Então, b_2 é acrescentado a S e id_1 retirado de P. Posteriormente, e de forma análoga à execução anterior, b_3 e d_2 são inseridos em S, também e_1 é inserido, diferenciando apenas a ausência de caminho algum em P relacionado a este. Assim, a saída cuja raiz é a_2 é considerada válida, montada e os valores de P e S restaurados. Os próximos elementos de F_D, b_4 e e_2, são recuperados, a raiz de b_4, a_3, localizada, *id1* retirado de P, a ascendência de b_4 e e_2 são inserido em S. No entanto, ao recuperar o elemento seguinte de F_D, b_5, como este possui seu campo *fim* superior ao de a_3 uma verificação sobre P é realizada. Como P não está vazia, os elementos de S são descartados e P restaurada ao seu estado inicial. Finalmente, a saída cuja raiz é a_4 é construída seguindo o mesmo processo.

A geração de resultados é realizada através do algoritmo da figura 5.3. Uma pilha recebida na entrada é usada para armazenar nós antecessores ao nó folha em análise pertencentes a subárvores ainda não montadas completamente na árvore resultante. A construção dessa pilha se dá de tal forma que, a qualquer momento, o campo *fim* do elemento topo sempre é o menor contido na pilha. Em decorrência dessa estratégia, assim como do fato de que sua entrada possui apenas nós que devem estar na saída, nenhum nó não pertencente ao resultado final é construído.

A idéia básica por trás do MontaArvore é a seguinte:

1. Os elementos de S são percorridos e suas ascendências geradas gradativamente com o auxílio da pilha.

2. Para cada elemento e na posição inicial de S, o valor de seu campo *fim* é verificado frente ao do topo t da pilha (linha 5). Esse passo é realizado a fim de verificar se e pertence à descendência do nó correspondente ao topo da pilha.

3. Caso seja inferior (linha 10) e deve ser retirado da fila, significando que e pertence à descendência do nó correspondente ao topo da pilha, e passa

a ser o nó atualmente em análise, denotado por n.

4. Caso contrário, se for igual (linhas 7-9) tanto t deve ser desempilhado, como e, extraído da pilha, uma vez que tratam-se do mesmo nó, passando então a ser o nó em análise; senão, t deve ser desempilhado e se tornar o nó n em análise, para que seu ascendente seja construído e analisado nos próximos passos.

5. Uma vez definido o nó a ser analisado, o pai (p) deste passa a ser verificado contra o topo da pilha (linhas 12-16).

6. Caso sejam iguais, p já está na árvore de saída e apenas uma aresta precisa ser criada (linha 16).

7. Caso contrário, p é adicionado à árvore de saída e um ponteiro para ele é empilhado a fim de que o processo se repita para seu ascendente (linhas 14-15).

8. Quando todos os elementos de S forem analisados, caso ainda exista algum elemento na pilha, sua ascendência deve ser construída até que o nó correspondente à raiz de Q seja inserido na árvore resultante (linhas 17-26).

```
MontaArvore(S, RaizQ, pilha)
 1  while S ≠ ∅
 2       do
 3          primeiroS ← Primeiro(S);
 4          if primeiroS.fim > pilha.topo.fim
 5            then noCorrente ← Desempilha(pilha);
 6            else  if primeiroS.fim = pilha.topo.fim
 7                    then noCorrente ← Desempilha(pilha);
 8                         noCorrente ← ExtraiPrimeiro(S);
 9                    else  noCorrente ← ExtraiPrimeiro(S);
10          CriaNo(noCorrente);
11          paiCorrente ← noCorrente.pai;
12          if paiCorrente.fim ≠ pilha.topo.fim
13            then CriaNo(paiCorrente);
14                 Empilha(paiCorrente);
15          CriaAresta(noCorrente, paiCorrente);
16  while pilha ≠ ∅
17       do noCorrente ← Desempilha(pilha);
18          paiCorrente ← noCorrente.pai;
19          if paiCorrente ≠ nulo
20            then CriaNo(paiCorrente);
21                 CriaAresta(noCorrente, paiCorrente);
22                 if paiCorrente.rotulo ≠ RaizQ.rotulo
23                   then Empilha(paiCorrente);
24                   else  raizSaida ← paiCorrente;
25            else  raizSaida ← noCorrente;
```

Figura 5.3: Algoritmo para montagem dos resultados

A única restrição para a fonte de dados é que, dada uma expressão E buscada, sua ascendência não pode conter tal expressão. Tal restrição não constitui grande limitação, pois, em situações reais, não é comum encontrarmos documentos com tal estrutura.

A principal desvantagem de outras propostas, tais como, [4] e [3], é o tamanho dos resultados intermediários. A técnica associada ao algoritmo proposto dispensa o uso de decomposições em sub-estruturas mais simples, o que evita a necessidade de operações de combinação, além de permitir que o número de resultados inválidos montados seja reduzido. A fim de prover eficiência, índices sobre expressões de caminho simples são utilizados e a travessia do documento analisado é realizada de forma *bottom-up*.

A fim de demonstrar o uso do operador de casamento de padrões em consultas XQuery triviais, a figura 5.5 exibe um possível plano de execução para a consulta exposta em 5.4. Na figura 5.6 temos o plano sugerido em [10] para a mesma consulta, onde junções estruturais (ou suas variações) são empregadas para solucionar o problema. Em especial, perceba que na figura 5.6 nenhuma condição de seleção é utilizada para reduzir o número de acessos a disco no mo-

mento da recuperação dos nós, ou seja, o operador INDEXSCAN recebe como entrada apenas o rótulo do elemento procurado. Já no algoritmo XML-PM, essa entrada é composta pelo caminho que começa em um nó folha de árvore de consulta (GTP) e termina ao atingir uma aresta antecessor-descendente, reduzindo assim, o número de nós recuperados inválidos.

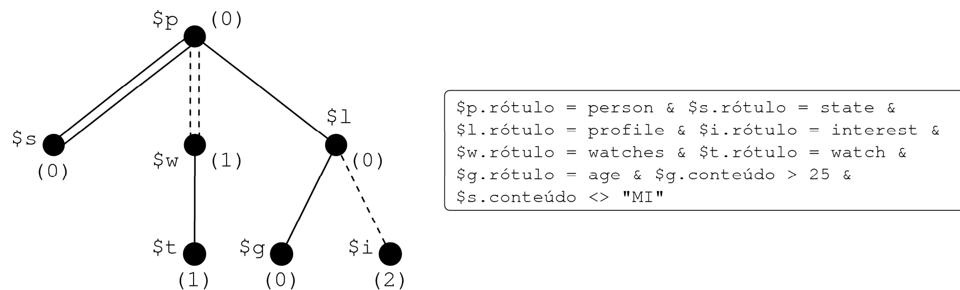


Figura 5.4: GTP Básica

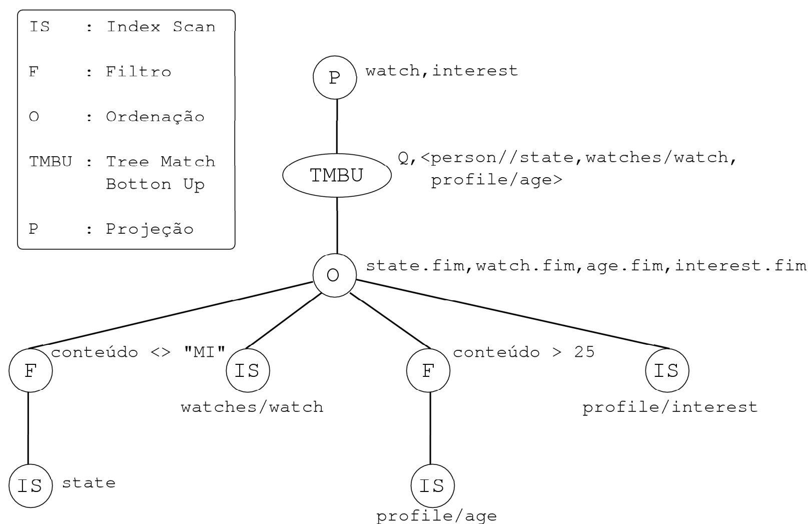
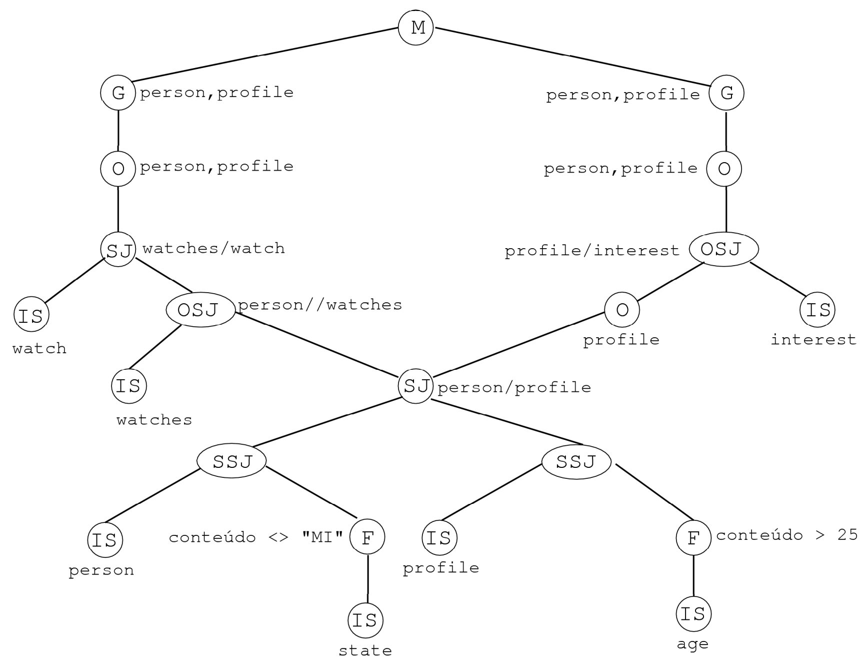


Figura 5.5: Plano Físico Gerado para a consulta da figura 5.4 usando XML-PM.

5.2.1 Análise da Complexidade

Esta seção apresenta uma análise de desempenho do algoritmo proposto para casamento de padrões sobre documentos XML representados como árvores. Para tanto, considere D um documento XML, como o da figura 5.7, h a altura da árvore de consulta, ou seja, o tamanho do maior caminho entre um elemento



F	: Filtro	OSJ	: Outer Structural Join
IS	: Index Scan	O	: Ordenação
SSJ	: Structural Semi-Join	M	: Merge
SJ	: Structural Join	G	: Agrupamento

Figura 5.6: Plano Físico Gerado para a consulta da figura 5.4 usando junções estruturais.

folha e a raiz de Q e seja m o número de elementos folhas da árvore que forma o documento D.

```
<library>
 <book>
  <title> ... </title>
  <author> ... </author>
  <publisher> ... </publisher>
  <date> ... </date>
  <edition> ... </edition>
 </book>
 <article>
  <title> ... </title>
  <author> ... </author>
  <journal> ... </journal>
  <date> ... </date>
  <volume> ... </volume>
  <number> ... </number>
  <pages> ... </pages>
 </article>
</library>
```

Figura 5.7: Documento XML

Além disso, observe que o pior caso ocorre quando se tem uma configuração com os seguintes itens:

1. O padrão solicitado em Q recupera todos os nós folhas de D e os armazena em F_D.

2. O caminho entre a raiz de Q e qualquer de suas folhas gera a mesma expressão de caminho parâmetro para o módulo de indexação.

3. Todas as arestas de Q são mandatórias.

4. O caminho entre qualquer folha e a raiz de Q inclui, pelo menos, uma aresta do tipo *ad*.

5. A distância entre qualquer folha e a raiz de Q é h.

Vale ressaltar que, a situação que acaba de ser descrita é um caso extremo para o qual outros operadores seriam mais eficientes, uma vez que se trata, na verdade, de uma recuperação do documento inteiro e não de uma seleção de fragmentos deste. Cabe ao otimizador delegar quais tarefas cada operador deve realizar. Além disso, o conhecimento prévio acerca da estrutura - esquema -

do documento analisado pode evitar comparações desnecessárias. O trabalho proposto em [44], OrientX, faz uso destas informações a fim de prover melhorias de desempenho durante a fase de execução de consultas.

O item 1 estabelece que F_D possui o maior tamanho possível, m, uma vez que contém unicamente cada uma das folhas de D, pois nós redundantes são evitados associando-se a cada nó em análise uma lista contendo os padrões que potencialmente o validam (concordam no casamento de padrões com sua ascendência). Essa lista é representada pelo campo *ids*, descrito na seção 5.2. Já o item 2 acrescenta comparações ao procedimento ACHARAIZCAMINHO(N, R, IDS, EXP), pois como exposto na seção 5.2, este procedimento verifica se a ascendência do nó n de entrada está de acordo com algum dos caminhos relacionados em *ids*. Assim, quanto maior o número de caminhos potenciais (listados em *ids*), maior a quantidade de verificações, no pior caso. O item 3, por sua vez, declara que a lista de pendências P, possui o maior tamanho possível. O item 4 força a execução de ACHARAIZCAMINHO(N, R, IDS, EXP) para todos os elementos de F_D. Finalmente, o item 5 indica que o caminho buscado por ACHARAIZCAMINHO(N, R, IDS, EXP) é sempre o maior possível, ou seja, a própria altura da árvore.

De acordo com o item 1, o laço da linha 4 do algoritmo da figura 5.1 irá iteragir *O(m)* vezes, uma vez que, em seu escopo, os elementos de F_D são apenas extraídos da fila. A cada iteração, os processos de retirada de um elemento de P e ACHARAIZCAMINHO(N, R, ID) são executados. Além disso, sempre que P se tornar vazia, o procedimento MONTAÁRVORE(S,R) é acionado para a montagem de uma das saídas do operador.

Considere l o número de caminhos distintos entre a raiz de Q e suas folhas. Por definição, P origina-se a partir dos caminhos entre a raiz de Q e suas folhas, onde apenas arestas rotuladas como mandatórias ocorrem. Dessa forma, o maior tamanho possível de P ocorre quando cada elemento de F_D é recuperado por uma expressão de caminho diferente, o que faz com que se tenha $l = m$.

O processo de retirada de um elemento de P gasta o tempo necessário para uma busca sobre uma lista encadeada, que é da ordem do tamanho desta, no

caso, m. Assim, as m iterações do laço da linha 4 resultarão em um gasto de tempo com o processo de retirada de um elementode P da ordem de:

$$O(m^2). \tag{5.2.1}$$

Já o método ACHARAIZCAMINHO(N, R, ID), como visto na seção 5.2, é executado em $O(mh)$, considerando $l = m$, sempre que é requisitado. Assim, o algoritmo XML-PM desprende, com este procedimento, tempo da ordem de:

$$O(m^2h). \tag{5.2.2}$$

No cálculo da complexidade de MONTAÁRVORE(S, R) alguns pontos precisam ser evidenciados:

1. da literatura, sabemos que os processos de retirada e inserção de elementos de filas e pilhas são realizados em $O(1)$, [40];

2. os procedimentos CRIANÓ(N) e CRIAARESTA(N_1, N_2) executam um número constante de passos de tempo constante, logo, também desempenham suas atividades em $O(1)$.

Dessa forma, a complexidade de MONTAÁRVORE(S, R) é determinada pelos laços das linhas 2 e 17 do algoritmo exposto na figura 5.3.

O laço da linha 2 irá iteragir um número de vezes da ordem do tamanho de S. Já o laço da linha 17 é executado em $O(k)$ passos, onde k denota o número de nós internos da árvore resultante da consulta contidos na pilha. Observe que, como o percurso é realizado de forma *bottom-up*, se o pai do nó em análise não estiver no topo da pilha significa que aquele ainda não foi visitado, pois por formação, dado um nó qualquer da pilha, todos os nós acima deste pertencem à sua sub-árvore, ou seja, o topo da pilha sempre está o mais próximo possível das folhas, no caso, a lista S. Mais ainda, isso limita o tamanho da pilha à altura da árvore, logo, no pior caso, $k = h$. Além disso, sempre que um nó interno é criado na árvore resultante do algoritmo, um ponteiro para este nó é inserido na pilha, linhas 14-15 e 23-24, exceto para o caso em que atinge a raiz. Perceba que estas inserções somente são realizadas quando o campo *fim*

do elemento topo da pilha for superior ao nó em questão, ou seja, quando a pilha contiver apenas elementos ascendentes ao nó em análise. Dessa forma, o maior tamanho que a pilha pode assumir é o número de nós contidos no caminho entre o nó em análise e a raiz de Q. Em outras palavras, no pior caso, o tamanho da pilha é a altura de D.

A fim de facilitar o cálculo da complexidade de MontaArvore, considere que todos os nós da lista F_D de entrada que concordam com o padrão solicitado são, em algum momento, inseridos na lista S. Dessa forma, de acordo com a situação descrita, MontaArvore consumirá em XML-PM, um tempo total dado por $O(m + c \times h)$, onde c é uma constante.

Dessa forma, a complexidade total do algoritmo apresentado na figura 5.1 para a tarefa de casamento de padrões pode ser dada por:

$$O(XML\text{-}PM) = O(m^2) + O(m^2 h) + O(m + c \times h)$$

$$\boldsymbol{O(XML\text{-}PM) = O\ (\ h\ (\ m^2 + c\ )\ ).} \qquad (5.2.3)$$

5.2.2 Análise da Corretude

Esta seção destina-se a comprovar que todas as saídas geradas por XML-PM são válidas. Para isso, considere a definição de saída válida dada a seguir.

Definição 1: *Uma saída do algoritmo* XML-PM *é dita válida quando:*

1. *em qualquer instante, para cada elemento* **e** *em S,* **e** *deve estar de acordo com algum padrão de sua lista* **ids**.
2. *ao executar o procedimento* MontaÁrvore(S, r), *todos os elementos em S pertencem à mesma sub-árvore.*
3. *todos os caminhos mandatórios especificados em Q devem ocorrer na saída.*

4. *o resultado de* MONTAÁRVORE(S, R) *é o sub-grafo de* D *induzido dos elementos de* S *até um nó com rótulo igual da raiz de* Q[1]*.*

Teorema 1: *Todas as árvores retornadas por* XML-PM *são saídas válidas.*

Prova: Seja A uma árvore qualquer resultante da execução de *XML-PM* sobre um documento D. Mostraremos que A respeita as condições impostas na definição 1.

A *cláusula 1* da *definição 1* deixaria de ocorrer quando um elemento de F_D, com ascendência discordante daquela solicitada na consulta, fosse inserido na lista S, linha 20. Além disso, considerando que todos os elementos com ascendências concordantes são inseridos em S, considere inserções imediatamente anteriores às verificações de P nas linhas 15 e 29. Como consta na seção 5.2, todos os elementos contidos na lista F_D são obtidos através do uso de índices sobre expressões de caminho simples. Tais expressões são obtidas percorrendo a árvore de consulta a partir de seus nós folhas até atingir a raiz da mesma ou uma aresta dupla. Sempre que a expressão de caminho que recupera um elemento inclui a raiz de Q, o campo *caminho* deste elemento é rotulado como *"SIMPLES"*. Dessa forma, não é possível que elementos com esta característica discordem do padrão solicitado. Assim, sempre que um elemento é recuperado de F_D, um teste é realizado, linha 6, e, se este for decorrente de um caminho que contém arestas duplas, a função ACHARAIZCAMINHO(N, R, IDS, EXP) é executada a fim de verificar se existe concordância com algum dos padrões especificados na lista *ids*. Caso a ascendência do elemento analisado discorde de todos os caminhos especificados em *ids*, o valor retornado é *"Nulo"* e o elemento em análise é descartado, linha 10.

Para mostrar a corretude do algoritmo em relação ao item 2 da definição 1, basta analisar em que situações um elemento é inserido em S, linhas 15, 20 e 29.

O primeiro caso, imediatamente anterior à checagem de P na linha 15, encontra-se no escopo da condição da linha 6, onde AchaRaizCaminho(n, r,

[1]novamente, considere que S contém todos os nós folhas da árvore resultante

ids, exp) é executada, retornando seu resultado na variável Aux. Sempre que Aux assume um valor não nulo, o elemento e de F_D em questão é inserido em S. No entanto, quando Aux possui um valor diferente de $Raiz$, significa que e pertence a uma nova sub-árvore, sendo então S reinicializada antes desta inserção.

Já as inserções demais inserções (linha 20 e 28) estão relacionadas, uma vez que, tratam-se de casos complementares. Ao recuperar um elemento e de F_D cuja ascendência é rotulada como *"SIMPLES"*, sabe-se que e está de acordo com algum padrão especificado em Q, cláusula 1 da definição 2. No entanto, é necessário verificar se e pertence à mesma sub-árvore dos nós contidos em S. O teste da linha 21 realiza tal checagem. Em caso afirmativo, e é inserido em S e o processo continua para o próximo elemento de F_D. Caso contrário, S é reinializado antes da inserção de e. Além disso, a variável $Raiz$ é atualizada para o valor correspondente a e, a fim de garantir que os próximos elementos a serem inseridos em S também pertençam à essa mesma sub-árvore.

Para que a *cláusula 3* ocorra, existem duas possibilidades: *i.)* MONTAÁRVORE(S, R, PILHA) é executado mesmo que P ainda contenha elementos; *ii.)* um elemento é retirado de P erroneamente. No entanto, *i.)* não pode ocorrer pois, as únicas chamadas a este método são antecedidas de testes sobre o tamanho de P, linhas 10-11 e 25-26. Por definição, P representa a parte obrigatória de Q. Logo, como S denota uma potencial saída válida, para que nenhum elemento seja retirado de maneira incorreta, basta que apenas os elementos contidos em S sejam testados contra P. Mais ainda, sempre que S for reinicializada, as remoções sobre P devem ser anuladas. De fato, todos os procedimentos de retirada de um elemento de P são precedidos de inserções em S. Além disso, sempre que S é reinicializada, P é restaurada ao seu estado inicial.

A *cláusula 4* diz respeito ao procedimento MONTAÁRVORE, que, através do uso de uma pilha, constrói uma subárvore, A, do documento analisado, onde as folhas correspondem aos elementos contidos em S, os nós internos são os nós ascendentes a estes em D e a raiz é o elemento correspondente à raiz

da árvore de consulta em D.

O procedimento MontaArvore faz uso de uma pilha para armazenar nós já criados de uma sub-árvore em construção, pois, exceto para os nós folhas, sempre que um nó é inserido em A este se torna o topo da pilha, linhas 14-15 e 23-24. Assim, para mostrar a corretude de MontaÁrvore, basta provar que, a qualquer instante, dentre os elementos da pilha, seu topo sempre é o elemento de menor valor no campo *fim*. Disso decorre que, a qualquer instante, todos os seus elementos são ascendentes de seu topo. Dessa forma, é necessário analisar as situações nas quais um elemento é inserido na pilha, linhas 15 e 24.

Na primeira iteração, o caso da linha 15 é trivial uma vez que a pilha encontra-se vazia. A condição da linha 5 faz uma verificação entre a próxima folha f da saída e elemento topo da pilha, estabelecendo que o nó a ser analisado deve ser o de menor valor no campo *fim*. Isso significa que, sempre que o topo da pilha não pertencer a ascendência de f (próxima folha da árvore de saída a ser construída), sua ascendência deve ser construída até atingir um nó n que seja antecessor de f, o que faz com que esta inserção na pilha respeite sua lei de formação.

No caso da linha 24 tem-se que o elemento a ser empilhado trata-se, na verdade, do pai do último elemento retirado da pilha, o que, pela construção do documento, respeita sua lei de formação.

5.3 Experimentos

Nesta seção é apresentada uma análise dos experimentos realizados a fim de avaliar o desempenho do algoritmo XML-PM.

Os algoritmos foram implementados no Delphi 7. As avaliações de desempenho foram realizadas em um AMD Athlon(tm) XP 2500+ com 1.84 GHz e 1GB de RAM rodando sobre windows XP. Um gerador de dados sintéticos foi desenvolvido para prover as massas de dados. As entradas para esse gerador são: altura da árvore, número máximo de filhos de um nó e o número máximo de rótulos que deve ser utilizado na árvore. A árvore é gerada, recursivamente,

através da escolha aleatória do número de filhos em cada nó, respeitando o limite máximo de chamadas. A tabela 5.2 descreve os documentos utilizados como massa de dados, apresentando seus valores de profundidade (máxima e média), número de filhos (máximo e médio), o número de rotúlos diferentes empregados, bem como, o seus tamanhos em número de nós.

A figura 5.8 exibe, graficamente, as consultas expostas na tabela 5.1 para as quais as medições foram realizadas. Dentre os diferentes tipos de consultas foram incluídas consultas envolvendo expressões de caminho e sub-árvores (*twig queries*). Além disso, os relacionamentos entre seus elementos também foram considerados e amostras de consultas com relacionamentos antecessor-descendente e pai-filho são também verificadas.

Consulta	**GTP**
Q1	//a / b // c / d
Q2	//a // b // c // d
Q3	//a / (b (/ d, / e) , c / f)
Q4	//a // (b (// d, // e) , c // f)

Tabela 5.1: Consultas

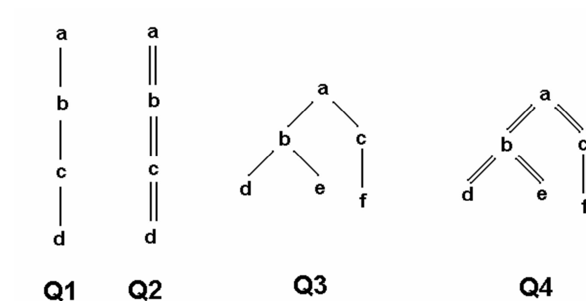


Figura 5.8: Representaçao Gráfica das Consultas

A fim de calcular o número de acessos a disco realizados por XML-PM, considere que uma página só pode conter um nó (elemento) do documento XML. Como o maior tamanho possível da pilha que armazena os resultados intermediários é a altura da árvore, dificilmente, esta não cabe na memória. Neste caso, manipulações de leitura e extração sobre a pilha não devem ser computadas como acessos a disco.

Consulta	Tamanho	Número de Nós	Profundidade Máxima	Profundidade Média	Número Máximo de Filhos	Número Médio de Filhos	Número de Rótulos
Doc1	6032K	50K	8	7.56	15	8.05	8
Doc2	11940K	100K	10	9.23	12	6.47	10
Doc3	25892K	200K	8	7.64	18	9.49	8
Doc4	37456K	300K	10	9.23	12	6.5	10
Doc5	52832K	400K	9	8.57	18	9.49	9

Tabela 5.2: Documentos

As figuras 5.9, 5.10 e 5.11, exibem os resultados obtidos com as consultas Q1, Q2 e Q3, respectivamente. Como pode ser visto na figura 5.11, Q3 envolve apenas relacionamentos pai-filho, o que reduz o número de nós recuperados inválidos. Já Q1 e Q2 (em especial, Q2, por conter apenas relacionamentos antecessor-descendente), geram um maior número de resultados temporários inválidos, figuras 5.9 e 5.10. No entanto, vale ressaltar, que, consultas onde todas as arestas originadas em seus nós folhas são do tipo antecessor-descendente implicam em um maior número de nós inválidos recuperados, uma vez que os parâmetros para as operações IndexScan são apenas os rótulos dos nós folhas, ou seja, na realidade, tratam-se do caso em que a lista de nós de entrada para o algoritmo é a maior possível. Ainda assim, perceba na figura 5.12) que Q4 mesmo contendo apenas arestas antecessor-descendente, apresenta baixo número de acessos desnecessários devido à estrátegia de reconhecimento empregada em XML-PM.

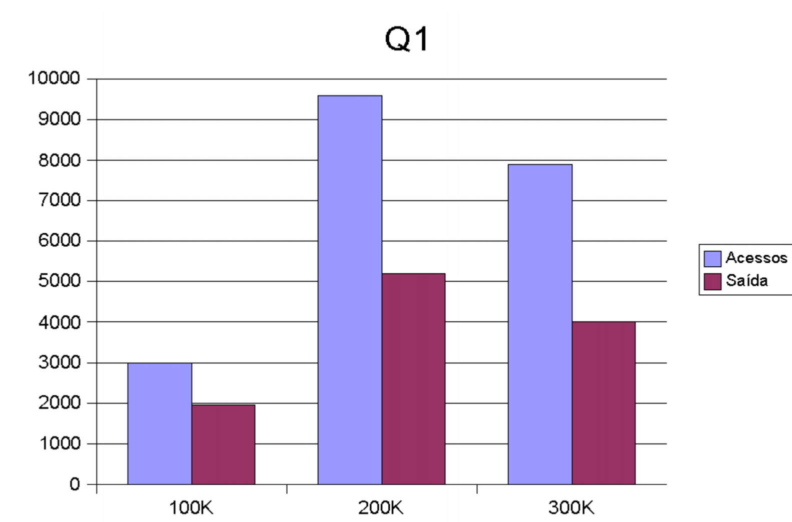


Figura 5.9: |Entrada| × |Acessos| × |Saída| para Q1

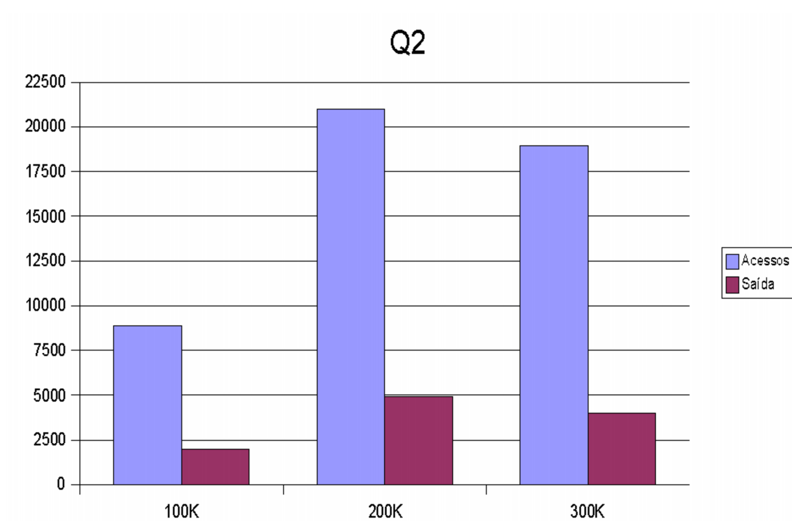


Figura 5.10: |Entrada| × |Acessos| × |Saída| para Q2

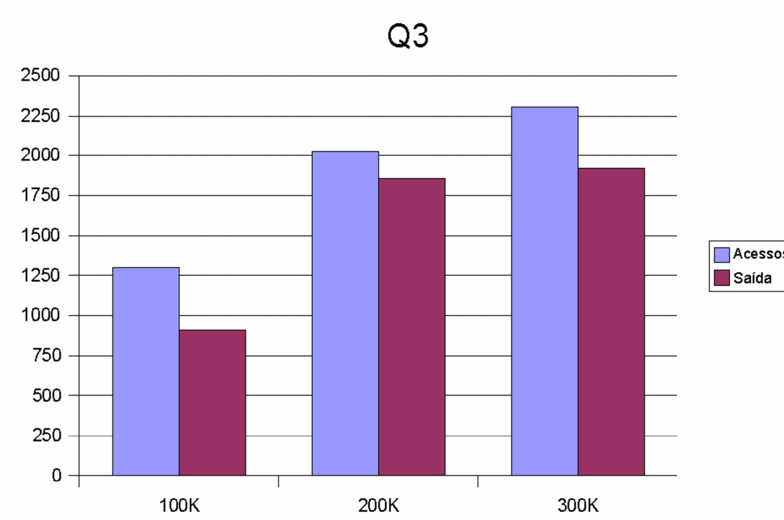


Figura 5.11: |Entrada| × |Acessos| × |Saída| para Q3

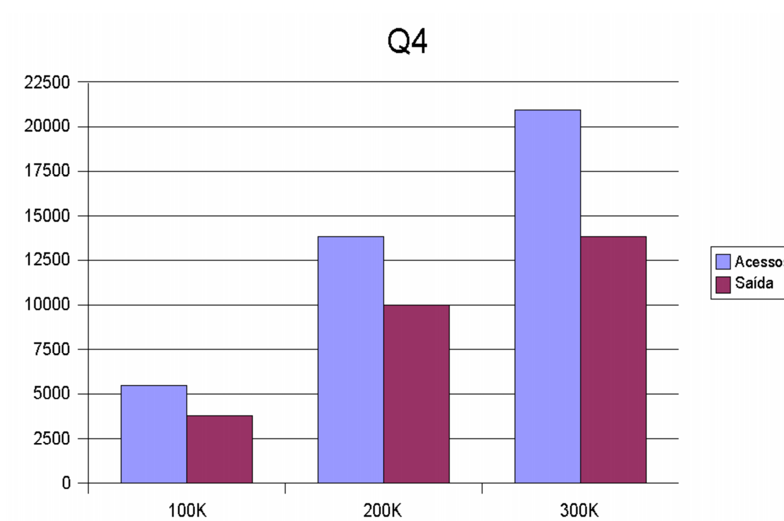


Figura 5.12: |Entrada| × |Acessos| × |Saída| para Q4

5.4 Conclusões

Neste seção foi apresentado o algoritmo XML-PM, principal contribuição do presente trabalho, como uma nova técnica desenvolvida para reconhecer fragmentos de um documento XML que satisfazem à estrutura especificada na consulta.

XML-PM é capaz de desenvolver sua execução recuperando apenas uma vez do disco os dados analisados, desde que a pilha utilizada para armazenar os nós ascendentes de um elemento caiba na memória principal. Como o maior tamanho possível desta pilha é a altura da árvore, tal situação mostra-se pouco freqüente. Além disso, índices sobre expressões de caminho simples são utilizados a fim de reduzir o número de nós na entrada que não satisfazem ao padrão solicitado.

Ademais, aspectos a respeito da corretude e complexidade são analisados e seus resultados comprovados através dos experimentos demonstrados na seção 5.3. XML-PM mostrou-se capaz de responder eficientemente às consultas adotadas, em especial, realizando um número de acessos bastante reduzido, se comparado ao obtido em técnicas similares.

Capítulo 6

Conclusões

A flexibilidade intrínseca ao modelo XML impõe significantes desafios no processamento de consultas sobre dados estruturados dessa forma. Em especial, reconhecer sub-estruturas de um documento XML que atendam às especificações de uma consulta, é uma função de extrema importância na fase de processamento desta.

Este trabalho apresenta uma técnica de casamento de padrões sobre documentos estruturados em formato de árvore. Para isso, adapta o conceito de *Generalized Tree Pattern* (GTP), proposto em [10] a fim de melhor resolver relacionamentos do tipo antecessor-descendente. Índices sobre expressões de caminho simples são utilizados a fim de reduzir o número de nós recuperados que não pertecem à saida.

Diferentemente de outras soluções para o problema, a estratégia adotada não precisa decompor o padrão buscado em sub-padrões que envolvam apenas relacionamentos simples. Além disso, através do uso de uma pilha e adotando como esquema de numeração a ordem de visita em pós-ordem, o algoritmo reduz significantemente o número de resultados intermediários inválidos. O método proposto não é aplicável quando um determinado elemento rotulado por e que ocorra no padrão solicitado contenha algum sub-elemento também com rótulo e. Tal limitação não implica grande deficiência, uma vez que, em dados reais, tal situação mostra-se pouco freqüente.

Os resultados obtidos com os experimentos foram bastante satisfatórios,

uma vez que, mesmo em situações adversas - consulta Q4, por exemplo - o algoritmo desempenhou um número de visitas considerado bom, se comparado aos valores obtidos em outras abordagens. Além disso, apresenta um número de resultados intermediários inválidos bastante inferior ao encontrado utilizando outros métodos nos casos em que a árvore de consulta contém apenas arestas pai-filho.

6.1 Trabalhos Futuros

Como trabalhos futuros, a serem realizados por outros membros do grupo de pesquisa, uma das primeiras tarefas pretendidas é o desenvolvimento do conjunto completo das implementações para os operadores suportados pela estrutura *GTP*, bem como a integração do Motor de Execução de Consultas com o restante do Módulo Processador do sistema FoX, pois isso provê uma base para diversos trabalhos de pós-graduação dentro do contexto do FoX.

Além disso, o desenvolvimento de uma estratégia de casamento de padrões que seja capaz de operar também sobre documentos XML que contenham ciclos é também uma tarefa almejada pelo grupo afim de ampliar o poder de processamento do sistema FoX.

Referências Bibliográficas

[1] D. Florescu A. Levy A. Deutsch, M. Fernandez and D. Suciu. A query language for xml. In *World Wide Web Conference*, 1999.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[3] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–, 2002.

[4] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[5] J. C. Machado C. P. Santiago. i-fox: Um Índice eficiente e compacto para dados xml. In *XIX Simpósio Brasileiro de Bancos de Dados*, pages 191–203, 2004.

[6] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. Xperanto: Publishing object-relational data as xml. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[7] G. M. Chaves and J. C. Machado. Fox-x: Um motor de execução de consultas xml. 2002. Projeto Financiado pela FUNCAP.

[8] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

[9] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD Conference*, pages 455–466, 2005.

[10] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB*, pages 237–248, 2003.

[11] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[12] World Wide Web Consortium. Xpath 1.0. *W3C Recommendation*, 1999. http://www.w3.org/TR/xquery/.

[13] D. Florescu M. Marchiori J. Robie. D. Chamberlin, P. Fankhauser. Xml query use cases. *W3C Candidate Recomendation*, 2005. http://www.w3.org/TR/xmlquery-use-cases.

[14] J.Robie e D. Florescu D. Chamberlin. Quilt: An xml query language for heterogeneous data source. In *Workshop on Web and Database*, 2000.

[15] M. Fernández A. Malhotra K. Rose M. Rys J. Siméon P. Wadler D. Draper, P. Fankhauser. Xquery 1.0 and xpath 2.0 formal semantics. *W3C Candidate Recommendation*, 2005. http://www.w3.org/TR/xquery-semantics/.

[16] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with stored. In *SIGMOD Conference*, pages 431–442, 1999.

[17] G. Dobbie. Databases, but not as we know them. In *15th Conference on Australasian Database*, volume 27, pages 11 – 13, 2004.

[18] Mary F. Fernandez, Wang Chiew Tan, and Dan Suciu. Silkroute: trading between relations and xml. *Computer Networks*, 33(1-6):723–745, 2000.

[19] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native xml base management system. *VLDB J.*, 11(4):292–314, 2002.

[20] Thorsten Fiebig and Harald Schöning. Software ag's tamino xquery processor. In *XIME-P*, pages 19–24, 2004.

[21] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[22] Tim Bray Jean Paoli C. M. Sperberg-McQueen Eve Maler François Yergeau, John Cowan. Extensible markup language (xml) 1.1. *W3C Recommendation*, 4th February 2004.

[23] Norbert Fuhr and Kai Großjohann. Xirql: An xml query language based on information retrieval concepts. *ACM Trans. Inf. Syst.*, 22(2):313–356, 2004.

[24] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2:73–170, 1993.

[25] J. Widom H. Garcia-Molina, J. D. Ullman. *Database System Implementation*. Prentice Hall, 2000.

[26] Hiroshi Ishikawa, Kazumi Kubota, and Yasuhiko Kanemasa. Xql: A query language for xml data. In *QL*, 1998.

[27] C. Zhang H. Gang D. J. DeWitt J. Shanmugasundaram, K. Tufte and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. *VLDB*, pages 302–314, 1999.

[28] J. Kiernan R. Krishnamurthy E. Viglas J. Naughton I. Tatarinov J. Shanmugasundaram, E. Shekita. A general technique for querying xml documents using a relational database system. *SIGMOD Record*, 2001.

[29] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *Proceedings of DBPL'01*, pages 149–164, 2001.

[30] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[31] M. J. Franklin N. E. Hall M. L. McAuliffe J. F. Naughton D. T. Schuh M. H. Solomon C. K. Tan O. G. Tsatalos S. J. White M. J. Carey, D. J. DeWitt and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Conference*, pages 383–394, 1994.

[32] R. C. Mauro. Aspectos de gerência de objetos persistentes: A implementação do goa++. Master's thesis, COPPE, UFRJ, 1998.

[33] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of xquery. In *SIGMOD Conference*, pages 71–82, 2004.

[34] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[35] M.F. Fernandez D. Florescu J. Robie J. Simeon S. Boag, D. Chamberlin. Xquery 1.0: An xml query language. *W3C Candidate Recommendation*, 2005. http://www.w3.org/TR/xquery/.

[36] H. V. Jagadish A. Nierman e Y. Wu S. Paparizos, S. Al-Khalifa. A physical algebra for xml. Technical report, 2001. http://www.eecs.umich.edu/db/timber.

[37] C. P. Santiago. Uma estratégia de indexação para dados xml. Master's thesis, MCC, UFC, 2004.

[38] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of xml documents. In *WebDB (Informal Proceedings)*, pages 47–52, 2000.

[39] Harald Schöning. Tamino - a dbms designed for xml. In *ICDE*, pages 149–154, 2001.

[40] R. L. Rivest C. Stein T. H. Cormen, C. E. Leiserson. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[41] The Timber Team. http://www.eecs.umich.edu/db/timber. The University of Michigan.

[42] P. Apers V. Mihajlovic, D. Hiemstra. On region algebras, xml databases, and information retrieval. *4th DutchBelgian Information Retrieval Workshop*, 2003.

[43] Iraklis Varlamis and Michalis Vazirgiannis. Bridging xml-schema and relational databases: a system for generating and manipulating relational databases using valid xml documents. In *ACM Symposium on Document Engineering*, pages 105–114, 2001.

[44] D. Luo S. Lu J. An Y. Chen X. Meng, Y. Wang and Y. Jiang. Orientx: A schema-based native xml database system. http://idke.ruc.edu.cn/OrientX/.

[45] Jingtao Yao and Ming Zhang II. A fast tree pattern matching algorithm for xml query. In *Web Intelligence*, pages 235–241, 2004.

[46] Pavel Zezula, Federica Mandreoli, and Riccardo Martoglia. Tree signatures and unordered xml pattern matching. In *SOFSEM*, pages 122–139, 2004.