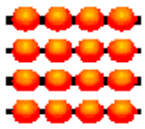
UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO



DISSERTAÇÃO DE MESTRADO

UMA ESTRATÉGIA DE INDEXAÇÃO PARA DADOS XML

Cynthia Pinheiro Santiago

Prof. Dr. Javam de Castro Machado
Orientador

Fortaleza, Abril de 2004

Para minha família.

Agradecimentos

Ao longo deste trabalho, algumas pessoas desempenharam papéis fundamentais. Gostaria de expressar, de forma particular, a minha gratidão:

- Ao meu pai, Jurandir, o grande incentivador de minha vida;
- À minha mãe, Maria do Carmo, por seu carinho e orações;
- À minha irmã, Sabrina, pelo companheirismo e por sempre torcer por mim;
- A Ricardo Melo e família, pelo apoio e incentivo;
- Aos meus tios, primos e avós, pelos momentos agradáveis;
- Ao professor Javam Machado, pela orientação e por compartilhar seu conhecimento e experiência, fundamentais ao bom desenvolvimento deste trabalho;
- Aos integrantes do projeto FoX, pelo excepcional trabalho de equipe;
- Aos colegas do Mestrado em Ciência da Computação da Universidade Federal do Ceará, pelo apoio, paciência e, principalmente, pelas horas de descontração ao longo destes meses de Mestrado;
- Aos professores do Mestrado em Ciência da Computação da Universidade Federal do Ceará, por terem fornecido o embasamento teórico necessário a este trabalho;
- À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo incentivo financeiro;
- Às professoras Marta Mattoso e Rossana Andrade, que gentilmente compuseram a minha banca de Mestrado;
- A todas as outras pessoas, não mencionadas anteriormente, que direta ou indiretamente contribuíram para a evolução deste trabalho;
- E finalmente, a Deus, por sempre guiar os meus passos.

Resumo

Com o aumento significativo da quantidade de documentos XML, surgiu a necessidade de tratá-los também como Base de Dados, sobre a qual consultas poderiam ser feitas. Atualmente, existem diversas formas de gerenciar documentos XML, desde as mais simples, utilizando um sistema de arquivos, até as formas mais sofisticadas, através de Sistemas Gerenciadores de Banco de Dados (SGBDs). Estes SGBDs, por sua vez, podem ser relacionais, baseados em objetos ou XML nativos.

Nos SGBDs XML Nativos, os dados XML são persistidos em disco segundo uma estrutura lógica de árvore ou grafo, onde elementos e atributos são representados pelos nós da árvore e as arestas são definidas através das relações de parentesco entre os elementos. Neste contexto, apresentamos o FoX, um protótipo de Banco de Dados XML Nativo, em fase de desenvolvimento na Universidade Federal do Ceará. Este Banco é formado por vários componentes, entre eles o Gerenciador de Armazenamento, responsável por persistir as estruturas lógicas de árvore em disco e, além disso, recuperar elementos ou atributos requisitados através de consultas.

Para acelerar as consultas em qualquer Banco de Dados, estruturas de índice são construídas a partir de campos que identificam unicamente um item na Base de Dados. O objetivo deste trabalho é desenvolver um índice eficiente para dados XML e implementá-lo dentro do FoX, através do desenvolvimento da arquitetura e das funcionalidades de um Módulo de Indexação, sendo este parte integrante do Gerenciador de Armazenamento.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

O XML é um padrão amplamente utilizado. Por causa de sua flexibilidade e simplicidade, adotá-lo significa uma boa escolha como, por exemplo, na representação de dados na Internet e como formato padrão para troca de informações entre diferentes aplicativos, entre outros.

É esperado que, em breve, um volume muito grande de documentos XML exista e então surge a necessidade de tratá-los também como Bases de Dados, sobre as quais consultas possam ser feitas. Existem muitas maneiras de transformar documentos XML em Bases de Dados, entre elas aquelas que fazem uso da estratégia nativa de armazenamento, descrita a seguir.

Podemos representar um documento XML como um grafo (ou árvore, caso este não possua ciclos) direcionado onde os nós representam elementos e atributos e onde uma aresta de um nó v para um nó v' representa uma relação pai/filho (elemento/sub-elemento ou elemento/atributo) entre v e v'. Quando um documento XML é persistido em disco de acordo com esta estrutura de dados, dizemos que ele é armazenado segundo a *estratégia nativa de armazenamento*, ou ainda segundo o *armazenamento nativo*.

A idéia da estratégia de armazenamento nativa é defendida pelo Lore [1] e pelo Natix [2]. Outras estratégias para a persistência dos dados XML incluem tranformá-los em tabelas de Bancos de Dados relacionais [3, 4, 5, 6, 7]. Existem ainda outras abordagens como em [8], onde o documento XML é armazenado como um arquivo de texto.

Segundo [9], *Bancos de Dados XML Nativos* são sistemas construídos especialmente para o gerenciamento de dados XML, ou seja, possuem a capacidade de definir, criar, armazenar, manipular, publicar e recuperar documentos ou fragmentos de documentos

XML. Os Bancos de Dados XML Nativos aos quais nos referimos aqui são aqueles que persistem o documento XML segundo a estratégia nativa de armazenamento e que, além disso, não possuem nenhum tipo de esquema para os dados. Portanto, de agora em diante, quando mencionarmos *Bancos de Dados XML Nativos*, estaremos nos referindo à esta especialização dos mesmos.

Com relação a tais bancos, o fato de não haver mapeamento dos dados XML para dados relacionais (ou objeto-relacionais) e reescrita de consultas é um ponto positivo a seu favor, pois isto elimina o risco de uma eventual perda de semântica durante a conversão e acarreta uma diminuição no tempo de resposta das consultas. O armazenamento nativo torna mais fáceis as consultas sobre a estrutura hierárquica dos dados XML, enquanto que nos modelos relacionais, isso acarreta uma grande quantidade de processamento para efetuar operações de junção.

Para acelerar ainda mais as consultas em Bancos de Dados XML Nativos, estruturas de índice são construídas a partir de campos que identificam unicamente um nó da Base de Dados. Um índice sobre dados XML, armazenados segundo a estratégia nativa, é a proposta deste trabalho.

1.2 Objetivos

O nosso objetivo é desenvolver um índice eficiente e compacto para dados XML. Um segundo objetivo é implementá-lo dentro do FoX, um Banco de Dados XML Nativo em desenvolvimento nesta Universidade, através do desenvolvimento da arquitetura e das funcionalidades de um Módulo de Indexação para o mesmo.

1.3 Organização

O restante desta dissertação está organizado da seguinte forma: o Capítulo 2 analisa algumas das diversas estratégias de indexação para dados XML existentes. No Capítulo 3, detalhamos as principais funcionalidades do FoX, um protótipo de Banco de Dados XML Nativo, que é o contexto no qual o índice deste trabalho – denominado *Índice FoX* – está inserido. O Capítulo 4 é dedicado à estrutura de índice *String B-Tree*, na qual nos baseamos para desenvolver o *Índice FoX*. No Capítulo 5, apresentamos detalhes sobre a indexação no FoX: as modificações feitas na estrutura da *String B-tree* de modo a adaptar suas características à indexação de dados XML, os algoritmos utilizados no *Índice FoX*, a estruturação das páginas do índice, a interação entre o índice e os demais componentes do

FoX e um estudo de caso para demonstrar a eficiência de nossa estratégia de indexação. Finalmente, o Capítulo 6 conclui esta dissertação, apresentando as contribuições e trabalhos futuros.

Capítulo 2

Indexação de Dados XML

A exemplo do que ocorre em uma biblioteca, onde não é desejável olhar todo o acervo em busca de um livro específico, também não é desejável vasculhar todo o documento XML em busca de um elemento ou atributo específicos. Para acelerar a busca em ambos os casos, índices são criados e aplicados sobre valores que identifiquem os itens em questão. No caso da biblioteca, valores a serem indexados seriam, por exemplo, o título ou o autor de cada livro. No caso de um documento XML, o que devemos indexar são nomes de elementos ou atributos.

Nos Bancos de Dados relacionais, índices são criados, geralmente, sobre o conjunto de valores de um campo de dados. Durante o procedimento de busca no índice, uma *chave de busca* é passada como parâmetro e, rapidamente, obtemos as tuplas das quais esta chave faz parte.

No entanto, o mesmo não ocorre quando indexamos dados XML. A grande maioria das consultas sobre tais dados faz referência à *estrutura* do documento XML, e não a valores de elementos e atributos. Logo, faz-se necessário um índice que considere, como chaves de busca, dados relevantes a respeito desta estrutura.

Uma das formas de representar a estrutura de um documento XML é através de *expressões de caminho*. Logo, boas chaves de busca para um índice, construído sobre dados XML, seriam os elementos do conjunto formado por todas as expressões de caminho identificadas na Base de Dados.

Qualquer expressão de caminho pode ser descrita através da gramática XPath [10]. No entanto, existem dois tipos de expressões de caminho que serão bastante referenciadas nas seções seguintes: as *expressões de caminho simples* e as *expressões regulares de caminho*.

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.

Uma *expressão regular de caminho* é uma expressão que, além do operador '/', possui também os operadores '//', que indica a relação ancestral-descendente entre dois elementos ou entre um elemento e um atributo, e '|', operador de disjunção entre dois elementos. Além disso, uma expressão regular de caminho prevê a existência de curingas para elementos, simbolizados pelo caracter '*'.

Ainda a respeito de expressões de caminho, dizemos que o *tamanho* de uma expressão de caminho A é a quantidade de nomes de elementos e atributos, não necessariamente distintos, presentes em A. O tamanho de A é representado por $|A|$. Esse conceito será importante nas sub-seções seguintes.

A literatura a respeito de índices sobre dados XML é extensa e várias abordagens foram propostas. As mesmas podem ser divididas em cinco categorias principais: (i) indexação em sistemas comerciais, (ii) indexação por sumários, (iii) indexação por regras de numeração, (iv) indexação com estrutura de dados otimizada para busca e (v) estratégias de indexação híbridas.

2.1 Indexação em Sistemas Comerciais

Nesta categoria, estão as estratégias de indexação para dados XML presentes nos Bancos de Dados comerciais relacionais ou objeto-relacionais. As estratégias apresentadas a seguir são aquelas implementadas no Oracle e no DB2, que são os Bancos mais adiantados na tecnologia de gerenciamento de dados XML.

2.1.1 Oracle

Durante o armazenamento de um documento XML em um Banco Oracle, caso exista um XML Schema [11] correspondente, o documento é analisado e decomposto em um conjunto de objetos SQL. Em seguida, estes objetos são armazenados em uma tabela do tipo XMLType, definida a partir do esquema. O texto do documento também é armazenado, de modo que o Oracle possa acessar seu conteúdo. Caso não exista um XML Schema, o documento é armazenado como uma *stream* de *bytes* em uma coluna do tipo XMLType [12].

Três tipos de índices podem ser criados sobre os dados XML no Oracle: a árvore B convencional, o índice baseado em texto e o índice funcional. Os dois últimos são criados sobre qualquer tipo de documento XML presente no Banco, seja ele armazenado através

de tabelas ou não. A árvore B só pode ser construída sobre dados estruturados em tabelas, ou seja, sobre os objetos SQL obtidos a partir do armazenamento de um documento XML com esquema.

O Oracle utiliza expressões de caminho XPath [10] para definir índices do tipo árvore B sobre tabelas XMLType. Neste caso, é criado um índice para cada expressão de caminho XPath, pois o que é indexado neste caso são valores de elementos e atributos, como ocorre em Bancos relacionais. Caso a expressão XPath possa ser reescrita em SQL objeto-relacional, a árvore B, correspondente aos elementos ou atributos XML alcançáveis através daquela expressão, é criada. Se isto não for possível, um índice funcional é criado em seu lugar.

2.1.2 DB2

O DB2 [13] oferece uma extensão, chamada *XML Extender*, para gerenciar documentos XML. Um documento XML pode ser representado: (i) como uma *coluna XML*, onde o documento é inteiramente armazenado como uma *stream* de *bytes* em uma coluna do tipo XMLCLOB, XMLVARCHAR ou XMLFile ou (ii) como uma *coleção XML*, onde o documento é dividido em um conjunto de tabelas.

As colunas do tipo XMLCLOB e XMLVARCHAR armazenam o documento como CLOB e VARCHAR, como o próprio nome já diz. Já quando a coluna é do tipo XMLFile, o documento é tratado como um arquivo em um sistema local de arquivos. Os elementos e atributos mais freqüentes no documento são mapeados, através do DAD (*Data Access Definition*), em tabelas auxiliares para indexação. Em seguida, índices são criados sobre os dados nas tabelas auxiliares [14].

Uma *coleção XML* é um conjunto de tabelas relacionais que armazenam os dados XML, de acordo com o mapeamento proprietário do DB2. Os índices possíveis de serem implementados tanto nas tabelas da coleção, como nas tabelas auxiliares obtidas através do DAD, são aqueles normalmente aplicados sobre as tabelas do DB2. Logo, nenhum índice é desenvolvido especialmente para os dados XML.

2.2 Indexação por sumários

Um sumário é um "resumo" da estrutura e dos dados de um documento XML. Foi a estratégia de indexação adotada pelos primeiros desenvolvedores do armazenamento nativo. Com o sumário, pretendiam alcançar dois objetivos: acelerar consultas e prover

uma espécie de esquema para o documento armazenado na Base de Dados XML.

A representação dos dados XML, neste caso, é feita através de um modelo baseado em grafos chamado OEM [15], com pequenas modificações para alguns tipos de sumário. Neste modelo, o documento pode ser visto como um grafo direcionado, onde os nós representam elementos e atributos e as arestas indicam se há uma relação pai-filho entre dois elementos ou entre um elemento e um atributo. Além disso, as arestas são rotuladas com os nomes de seus respectivos nós destino e os nós têm identificadores únicos, na Base de Dados. Um exemplo de grafo OEM pode ser visto na figura 2.1.

```
<library>
  <book>
    <title>Book A</title>
    <author>Author B</author>
  </book>
  <book>
    <title>Book C</title>
    <author>Author D</author>
  </book>
  <article>
    <title>Article E</title>
    <author>Author F</author>
  </article>
</library>
```

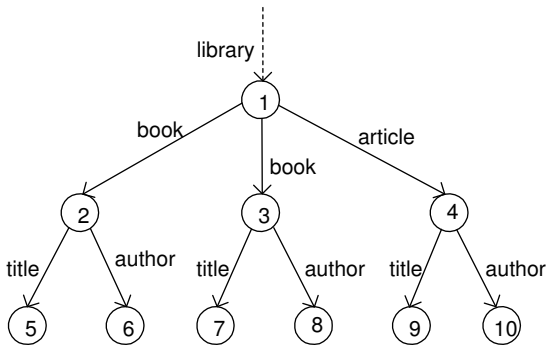


Figura 2.1: Base de Dados XML e seu respectivo grafo OEM

Seja $G = (V_G, E_G)$ o grafo OEM que representa uma Base de Dados XML, onde V_G é o conjunto de nós e E_G é o conjunto das arestas de G. Formalmente, um *sumário* $H = (V_H, E_H)$, correspondente a G, é também um grafo direcionado onde cada nó $u \in V_H$ é, na verdade, um subconjunto $V'_G \subset V_G$. Todos os nós em V'_G possuem o mesmo nome. O conceito de *sumário* será exemplificado na sub-seção seguinte.

Dizemos que cada nó $u \in V_H$ é uma *extensão de nós* de V_G. Dados dois nós v e v' de V_G, se existe uma aresta entre v e v' em G, então também existe uma aresta entre a extensão de v e a extensão de v' em H.

Exemplos conhecidos de sumários são o *DataGuide*, o *ToXin*, o *T-index*, o *A(k)-index* e o *APEX*.

2.2.1 DataGuide

O *DataGuide* [16] foi o índice que primeiro introduziu a definição de sumário e foi implementado no Lore [1], SGBD desenvolvido na Universidade de Stanford.

Um *DataGuide* pode ser descrito da seguinte forma: Dado que G é o grafo OEM correspondente à Base de Dados XML, um *Dataguide* H é um *sumário* de G tal que (i) toda expressão de caminho simples em G é representada em H uma única vez e (ii) cada expressão de caminho simples em H é também uma expressão de caminho simples em G. Um exemplo de *DataGuide* pode ser visto na figura 2.2.

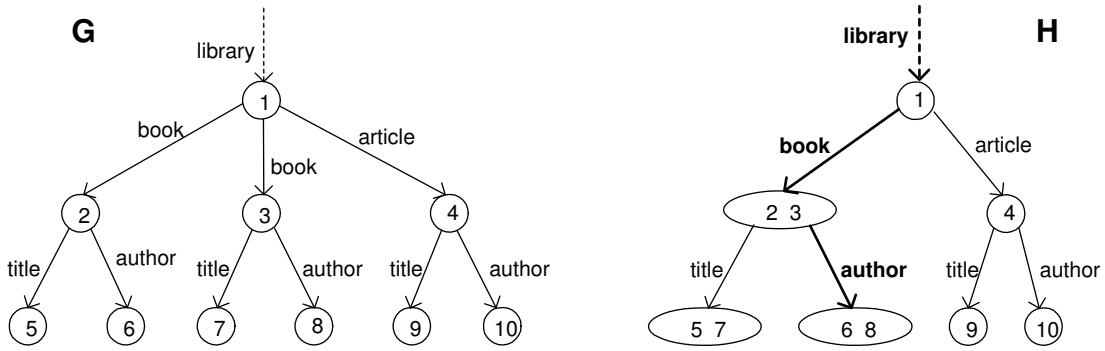


Figura 2.2: Base de Dados OEM e seu respectivo *Dataguide*

Note que, seguindo a definição de sumário, se G é um grafo OEM e H é o *Dataguide* correspondente a G, cada nó u em V_H é um *subconjunto*, ou *extensão de nós*, V'_G de V_G. Além disso, os nós em cada extensão do *DataGuide* possuem o mesmo nome. Note ainda que, se existe uma aresta entre dois nós v e v' em G existe também uma aresta entre as extensões de v e v' em H.

A busca por uma expressão de caminho simples A no *DataGuide* H da figura 2.2 significa visitar alguns nós de H, sempre fazendo o casamento de padrões de nomes de elementos ou atributos de A com os rótulos presentes nas arestas. Por exemplo, se $A =$ 'library/book/author', as arestas a serem percorridas são aquelas em negrito na figura 2.2 e os nós a serem retornados são aqueles cujos identificadores são 6 e 8.

Para resolver expressões de caminho mais sofisticadas, como expressões regulares de caminho, ou que não se originam na raiz do documento, o *DataGuide* precisa de alguns índices auxiliares, como por exemplo o *Vindex* (*Value Index*), que localiza um nó com um determinado valor, o *Tindex* (*Text Index*) que localiza nós contendo uma palavra ou grupo de palavras específicos e o *Lindex* (*Link Index*), que retorna o pai de um dado nó [17].

O tamanho do *DataGuide* não é maior que tamanho da Base de Dados, caso esta não possua ciclos. No entanto, um *DataGuide* pode ser tão grande quanto, no caso em que a Base de Dados apresenta uma estrutura muito irregular. Caso haja ciclos na Base de

Dados, o tempo de criação e o tamanho de um *DataGuide* podem ser exponenciais no tamanho de G [16].

2.2.2 ToXin

O ToXin [18] é uma estratégia de indexação formada por dois índices: o *Path Index*, que sumariza todas as expressões de caminho simples em um documento XML e que é utilizado para navegar na estrutura do documento em qualquer sentido e a partir de qualquer nó, e o *Value Index*, que suporta predicados sobre valores de objetos e atributos.

O *Path Index* é formado por dois componentes: uma árvore de índice H – um *DataGuide* [16] – e por um conjunto de *funções de instância* (*instance functions*), uma para cada aresta em E_H. Cada função de instância é armazenada em duas tabelas *hash* redundantes: uma chamada *forward instance table*, para a navegação no sentido pai-filho, e outra chamada *backward instance table* para navegação no sentido contrário. É a *forward instance table* que elimina a limitação imposta pelos *DataGuides* de só resolver expressões de caminho que se iniciam na raiz do documento. A *backward instance table* permite a navegação no sentido filho-pai, fazendo o papel do índice *Lindex*, mencionado na seção 2.2.1.

O *Value Index* consiste em um conjunto de relações, denominadas *value relations*, que armazenam identificadores dos nós XML na Base de Dados e seus respectivos valores. Existe uma *value relation* para cada aresta em E_H cujo destino é um nó folha. Essas relações são implementadas como árvores B+, cujas chaves são os valores que, por sua vez, são tratados sempre como *strings*.

Toda aresta da árvore de índice H possui uma função de instância, uma *value relation* ou ambas. Essas estruturas podem ser melhor visualizadas na figura 2.3. Por simplicidade, as *backward instance tables* não estão sendo exibidas.

O *ToXin* resolve expressões regulares de caminho com predicados sobre valores, tal como aquelas definidas na gramática XPath [10]. Para resolver a expressão de caminho A = '//book[title="Book A"]/author', primeiro a *forward instance table* **FI 1** é consultada. Em seguida, descemos no *DataGuide* e acessamos a tabela **TV 1** para resolver o predicado '[title="Book A"]'. No passo seguinte, consultamos a *backward instance table* **FI' 3** para recuperar o nó pai do elemento cujo *oid* é 5. Por último, obtemos o resultado da expressão através de uma consulta à **FI 4**. Com isso, o resultado da expressão é o nó da Base de Dados cujo *oid* é 6.

Uma observação importante sobre esta estratégia de indexação é que a mesma não leva em consideração IDREFs – seu modelo de dados é uma árvore e, portanto, não possui

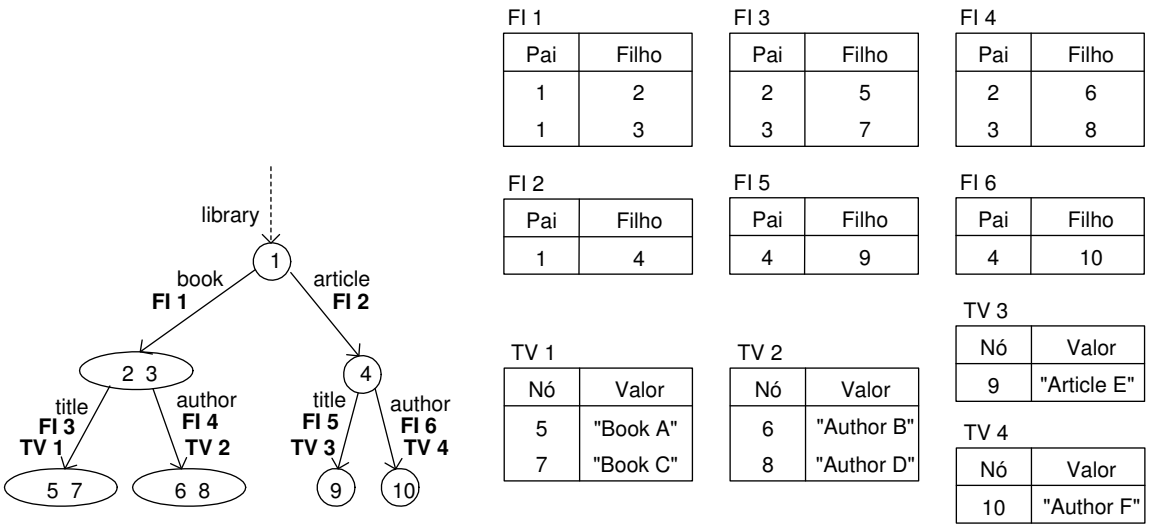


Figura 2.3: Um índice *ToXin* para a Base de Dados da figura 2.2

ciclos – ou a ordem dos elementos. O tamanho do índice é linear no tamanho da entrada, uma vez que a árvore H é um *DataGuide*.

2.2.3 T-index

O *T-index* (*Template Index*) [19] limita os tipos de expressões de caminho resolvidos pelo sumário, como forma de reduzir o tamanho deste. Para tanto, *templates* ou modelos são definidos e somente as expressões de caminho da Base de Dados que se enquadram nestes modelos são referenciadas no índice.

Um *template* envolve elementos e atributos da Base de Dados e o símbolo $\boxed{P}$, que pode ser substituído por qualquer expressão regular de caminho. Um exemplo de template para o grafo OEM da figura 2.1 é $\boxed{P}$/author. Um exemplo mais genérico seria $\boxed{P}/x$, onde x representa um elemento ou atributo qualquer.

O *1-index* (figura 2.4), cujo modelo é $\boxed{P}/x$, é um caso particular do *T-Index*. Este índice, em especial, resolve todas as expressões regulares de caminho sobre a Base de Dados e, por isso, é o maior índice da classe dos *T-indexes*. É também o mais utilizado como estratégia de indexação.

Seja G o grafo OEM que corresponde à Base de Dados. Para construir o *1-index*, utilizamos o conceito de *bissimilaridade*. Dizemos que dois nós $v, v' \in V_G$ são *bissimilares* se o conjunto de todas as expressões de caminho simples que levam a v é exatamente igual ao conjunto de todas as expressões de caminho simples que levam a v'. Através do

conceito de bissimilaridade, os elementos de V_G são agrupados em classes de equivalência (extensões de nós) e cada uma dessas classes corresponde a um nó do *1-index*.

Na figura 2.4, mostramos uma Base de Dados G, o *1-Index* correspondente a G e, para efeito de comparação, o *DataGuide* correspondente a G. Note que caso não existam ciclos na Base de Dados, o *1-index* é exatamente igual ao *DataGuide*. A busca por uma expressão de caminho simples A no *1-index* é similar à busca em um *DataGuide*, ou seja, se H é o grafo OEM correspondente ao *1-index*, percorremos os nós em V_H, sempre fazendo o casamento de padrões de nomes de elementos ou atributos em A com os rótulos presentes nas arestas em E_H. A busca por expressões de caminhos mais complexas não é discutida em [19].

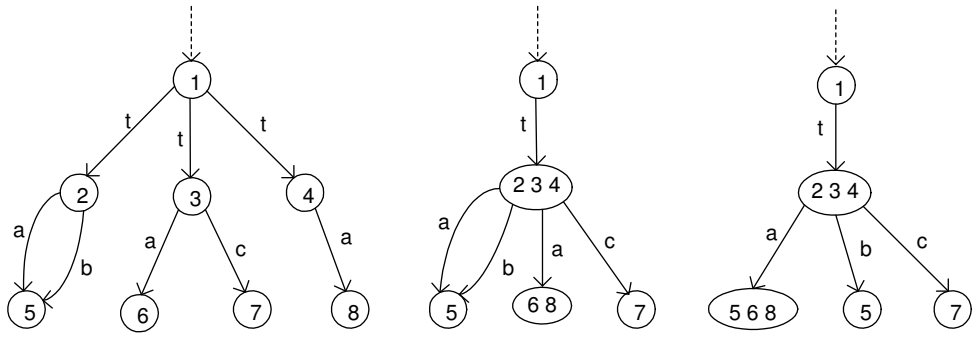


Figura 2.4: Base de Dados, *1-index* e *DataGuide* correspondentes

Uma grande vantagem do *1-index* em relação ao *DataGuide* é que, devido ao seu algoritmo de construção, no pior caso ele terá o mesmo tamanho da Base de Dados. Além disso, o mesmo é computado em $O(|E_G| \lg |V_G|)$.

2.2.4 A(k)-index

A principal observação por trás do *A(k)-index* [20] é que, dada uma expressão de caminho A em uma consulta XML, $|A|$ não é muito grande, na maioria dos casos. Logo, um índice preciso para expressões de caminho de tamanho até um certo t pode ser tão eficaz quanto um índice preciso para todas as expressões de caminho da Base de Dados.

Quanto mais próximo de 1 é o tamanho das expressões suportadas pelo índice, mais agrupados estarão os nós da Base de Dados em extensões do sumário. Com isso, o índice ocupa menos espaço. Se $A = x_1/x_2/\ldots/x_t$, onde x_i $(1 \leq i \leq t)$ é nome de elemento ou atributo, denotamos por $k = t-1$ o parâmetro de A que indica o número de arestas entre x_1 e x_t. É baseado em k que o *A(k)-index* é construído.

Para calcular o *A(k)-index*, o conceito de *k-bissimilaridade* [20] é utilizado. Este conceito é muito parecido com a definição de bissimilaridade vista na seção 2.2.3: Dado que G é o grafo OEM que representa a Base de Dados, dizemos que dois nós $v, v' \in V_G$ são *k-bissimilares* se: (i) v e v' possuem o mesmo nome e (ii) o conjunto de todas as expressões de caminho simples $A = x_1/x_2/\ldots/v$, com k arestas entre x_1 e v, é exatamente igual ao conjunto de todas as expressões de caminho simples $A' = x'_1/x'_2/\ldots/v'$, com k arestas entre x'_1 e v'. Através deste conceito, classes de equivalência em V_G são definidas e as mesmas correspondem a extensões de nós no *A(k)-index*.

À medida que o parâmetro k aumenta, mais o *A(k)-index* vai se assemelhando ao *1-index*. De fato, esse índice pode ser considerado como uma generalização do *1-index*. Na figura 2.5 vemos uma Base de Dados G e uma seqüência de *A(k)-indexes* correspondentes a G.

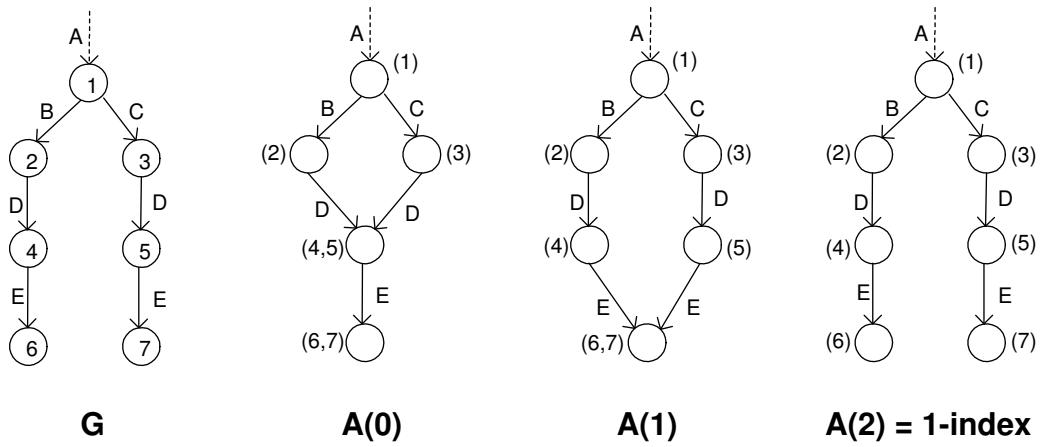


Figura 2.5: Base de Dados e *A(k)-indexes*

Para resolver expressões de caminho que não se iniciam na raiz do documento, um índice auxiliar, chamado *label-map* é utilizado. Este índice é implementado como uma tabela *hash*, onde suas entradas são todos os nomes de elementos e atributos presentes na Base de Dados. A busca por um elemento ou atributo de nome x nesta tabela retorna os *oids* dos nós do *A(k)-index* correspondentes a x.

2.2.5 APEX

No *APEX* (*Adaptative Path indEX*) [21], um algoritmo de *data mining* é utilizado para extrair o conjunto $\mathcal{P}$, formado pelas expressões de caminho mais freqüentes em consultas

sobre a Base de Dados. A estratégia de indexação consiste de um sumário, denominado G_{APEX}, e de uma *hash tree* chamada H_{APEX}.

O modelo de dados OEM, neste caso, sofre uma pequena alteração: o nó raiz existe e é representado por um vértice especial denominado *xroot*, cujo identificador é zero. Além disso, uma expressão de caminho aqui é definida como uma seqüência do tipo $A =$ '$//x_1/x_2/\ldots/x_n$', onde $x_1, x_2, \ldots, x_n$ são nomes de elementos ou atributos. O conjunto $\mathcal{P}_0$ é formado por todas as expressões de caminho presentes na Base de Dados XML de tamanho um.

No H_{APEX} estão representadas todas as expressões em $\mathcal{P}_0$ e $\mathcal{P}$. Estruturalmente, o mesmo consiste de nós, chamados de *hnodes*, onde cada *hnode* é uma tabela *hash*. Cada entrada em uma tabela *hash*, correspondente a um *hnode*, possui um ponteiro para outro *hnode* ou para um nó em G_{APEX}.

O G_{APEX} inicial é construído sobre as expressões em $\mathcal{P}_0$. Em seguida, baseado nas expressões em $\mathcal{P}$, nós são inseridos ou removidos do G_{APEX}. Analogamente, o H_{APEX} inicial é construído sobre as expressões em $\mathcal{P}_0$ e depois, baseado nas expressões em $\mathcal{P}$, entradas são inseridas ou removidas das tabelas *hash* do H_{APEX}. O algoritmo de *data mining* é executado de tempos em tempos para atualizar o índice. Por isso esta estratégia é dita "adaptativa".

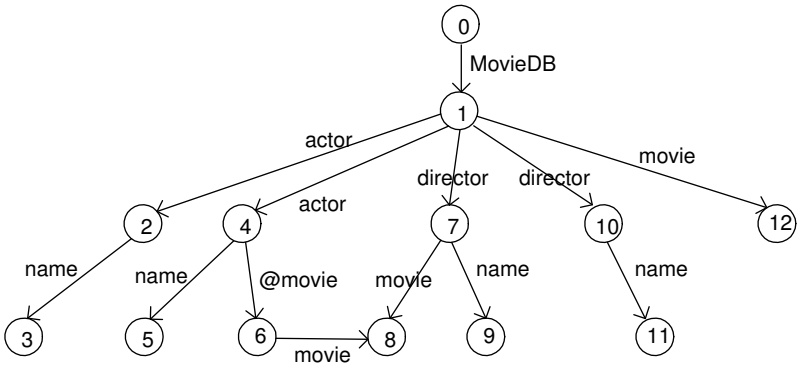


Figura 2.6: Base de Dados XML

Um exemplo de índice *APEX* para a Base de Dados da figura 2.6 pode ser visto na figura 2.7, onde $\mathcal{P} = \{$'//director/movie', '//@movie/movie', '//actor/name'$\}$. Entre parênteses, estão os identificadores dos nós na Base de Dados pertencentes a cada extensão do G_{APEX}. Se desejarmos resolver a expressão '//director/movie' através do índice, primeiro buscamos a entrada em T_1 correspondente a 'movie' e em seguida descemos na *hash tree* até o *hnode* T_2 e buscamos a entrada correspondente a 'director', o que nos leva à

extensão cujo identificador é 6. O nó na Base de Dados a ser retornado como resultado da busca é aquele cujo identificador é 8.

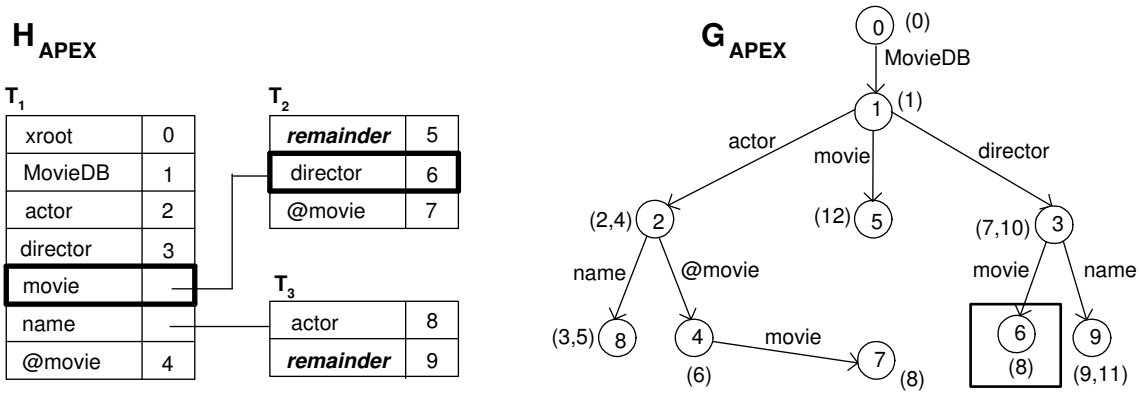


Figura 2.7: Índice APEX

Esta estratégia de indexação é bastante eficiente para resolver expressões de caminho que não se iniciam na raiz do documento por causa das facilidades de navegação no H_{APEX}. O sumário G_{APEX} também é geralmente menor que um *DataGuide* ou *1-index* porque o mesmo só se compromete em representar as expressões de caminho mais freqüentes em consultas à Base de Dados, não todas. No entanto, para resolver uma expressão de caminho não prevista no H_{APEX} uma grande quantidade de *joins*, proporcional ao tamanho da expressão de caminho e do G_{APEX}, precisa ser feita.

2.3 Indexação utilizando esquemas de numeração

Os *esquemas de numeração* de dados XML surgiram como uma alternativa para se avaliar eficientemente *structural joins* [22] entre elementos e atributos XML, ou seja, para encontrar todos os pares de elementos que satisfazem a relação pai-filho ou a relação ancestral-descendente, simbolizadas respectivamente pelos operadores '/' e '//', dentro de uma determinada expressão de caminho.

Ao contrário das estratégias de indexação por sumário, esta classe de índices não exige um modelo de dados em especial, podendo o mesmo ser tabelas, árvores ou mesmo arquivo de texto, desde que se consiga identificar unicamente um elemento ou atributo XML na Base de Dados.

Um *esquema de numeração* atribui números a elementos e atributos dos documentos na Base de Dados XML. Geralmente estes números são representados por uma tupla do tipo

(`StartPos` : `EndPos`, `LevelNum`) para cada elemento ou atributo, onde o último parâmetro identifica sempre o grau de aninhamento do elemento ou atributo dentro do documento. Os valores de `StartPos` e `EndPos` variam, de acordo com o esquema de numeração.

Um esquema de numeração bastante simples, denominado *Esquema de Numeração por Coordenadas Absolutas* [23], é aquele onde, dado um elemento ou atributo x, $StartPos_x$ e $EndPos_x$ indicam a posição do primeiro e último caracter de x com relação ao início do documento XML. Este esquema de numeração pode ser visto na figura 2.8.

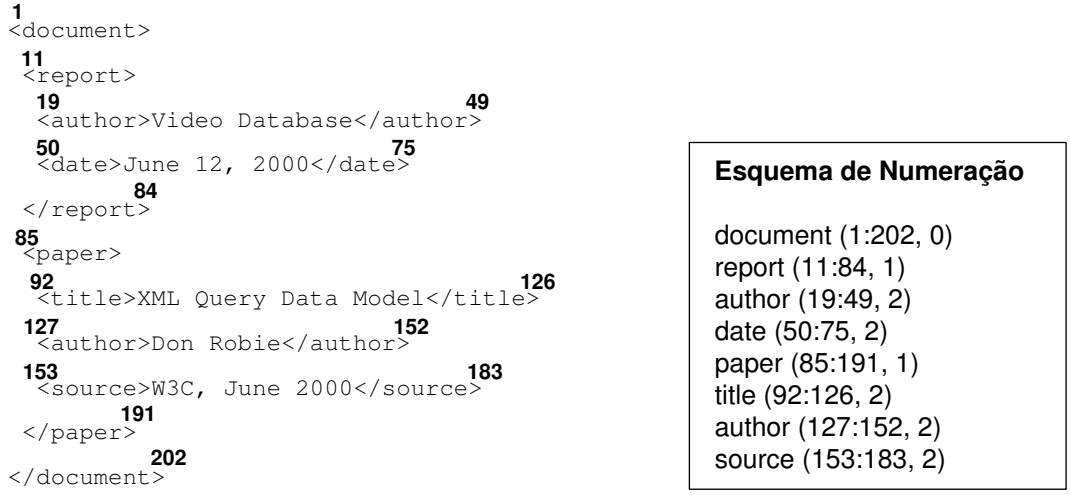


Figura 2.8: Estratégia de Numeração por Coordenadas Absolutas

Em um outro exemplo, se o documento XML é representado por uma árvore G, os números do *Esquema de Numeração de Dietz* [24] são obtidos a partir do percurso de G: Para todo nó $v \in V_G$, $StartPos_v$ e $EndPos_v$ identificam a ordem em que v foi visitado segundo, respectivamente, o algoritmo de pré-ordem e pós-ordem [25]. A figura 2.9 ilustra este esquema de numeração.

Na maioria dos esquemas de numeração é possível determinar a relação ancestral-descendente entre quaisquer pares de elementos ou atributos x e x' da seguinte forma: x é ancestral de x' se $StartPos_x < StartPos_{x'}$ e $EndPos_x > EndPos_{x'}$. Além disso, dizemos que x é pai de x' se $LevelNum_{x'} = LevelNum_x + 1$.

Em um cenário de consulta, para resolver a expressão de caminho A = 'report/author' por exemplo, é necessário ter duas listas, uma com todos os elementos da Base de Dados cujo nome é 'report' e outra com todos os elementos da Base de Dados cujo nome é 'author', onde cada item destas listas armazena o identificador do elemento na Base de Dados e os números obtidos através do esquema de numeração. Um elemento da lista

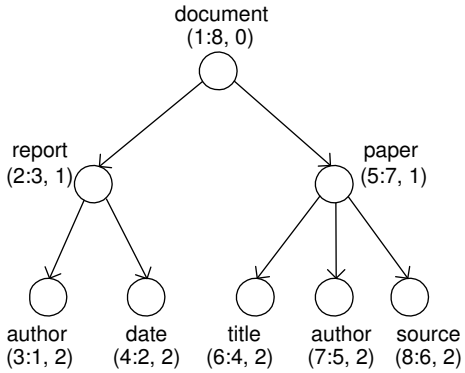


Figura 2.9: Esquema de Numeração de Dietz

'report' é comparado a um elemento da lista 'author' a cada vez, verificando-se se a relação pai-filho existe entre ambos através da comparação entre os seus respectivos números. A forma de percorrer estas listas – que podem ser muito grandes, na prática – e de escolher quais elementos devem ser comparados a cada momento estão intimamente ligados com a eficiência da estratégia de indexação. Por isso, existem alguns trabalhos como [22] e [26], dedicados a otimizar este processo.

Para obter rapidamente as listas de elementos mencionadas no parágrafo anterior, uma estrutura de índice – geralmente uma árvore B+ [27] – é construída a partir dos nomes de elementos e atributos presentes na Base de Dados.

Como exemplos de estratégias de indexação baseadas em esquemas de numeração, apresentamos o *XISS* e o *XR-Tree*.

2.3.1 XISS

O *XISS* [28] resolve expressões regulares de caminho através de uma combinação de esquema de numeração e algoritmos de junção de caminhos (*path-joins*).

Uma limitação existente nos esquemas de numeração vistos anteriormente é que a inserção de novos elementos ou atributos exige que novos números sejam calculados para cada nó na Base de Dados. Como forma de minimizar este problema, o *XISS* propõe um novo esquema de numeração descrito da seguinte forma: Se G é a árvore que representa a Base de Dados, um algoritmo de pré-ordem visita os nós de G e atribui a cada um deles dois inteiros, um que especifica a *ordem* em que o nó foi visitado e outro que indica o *tamanho* do mesmo. Mais especificamente, o par $<$***order, size***$>$ é tal que:

- Para um nó v' e seu ancestral v, $order(v) < order(v')$ e $order(v') + size(v') \leq$

$order(v) + size(v)$. Ou seja, o intervalo $[order(v'), order(v') + size(v')]$ está contido no intervalo $[order(v), order(v) + size(v)]$.

- Para dois nós irmãos v e v', se v é visitado antes de v' no algoritmo de pré-ordem, então $order(v) + size(v) < order(v')$.

Logo, temos que para um nó $v \in V_G$, $size(v) \geq \sum_{v'} size(v')$ para todo v' filho de v. Logo $size(v)$ pode ser um inteiro arbitrário maior que o número total de nós-folha descendentes de v, de maneira a poder acomodar futuras inserções. Um exemplo desse esquema de numeração pode ser visto na figura 2.10.

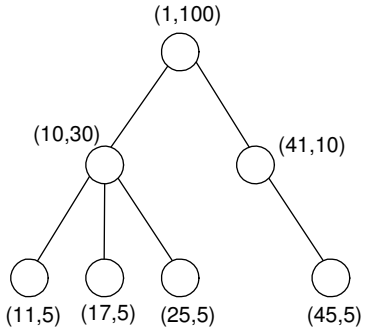


Figura 2.10: Esquema de numeração do XISS

O *XISS* é formado por três índices, todos implementados como árvores B+: o *element index*, o *attribute index* e o *structure index*. As chaves de busca para o *element index* é uma combinação de identificador de documento com nomes de elementos, de maneira que as folhas da árvore B+ apontam para listas de elementos com mesmo nome e pertencentes ao mesmo documento. Em cada item destas listas são armazenados o par <***order, size***>, o nível do elemento em G e um ponteiro para o seu nó pai em G. O *attribute index* tem basicamente a mesma estrutura, com apenas um campo a mais que é um ponteiro para o seu conteúdo, armazenado numa tabela à parte chamada *value table*. O *structure index* tem como chaves de busca identificadores de documentos e é uma coleção de vetores onde cada vetor correspondente a um documento D da Base de Dados. Em cada vetor estão dados relevantes, com relação ao esquema de numeração, de todos os elementos e atributos em um dado D.

Dada uma expressão regular de caminho A, esta é decomposta em um conjunto de sub-expressões que, por sua vez, são processadas pelos algoritmos de junção: (i) $\mathcal{EA}$-Join, cuja entrada é uma lista de elementos e outra de atributos, (ii) $\mathcal{EE}$-Join, cuja entrada são duas listas de elementos e (iii) $\mathcal{KC}$-Join, que processa uma expressão regular de caminho que

representa zero, uma ou mais ocorrências de uma sub-expressão (por exemplo, 'chapter*' ou 'chapter+'). Maiores detalhes sobre os algoritmos em [28].

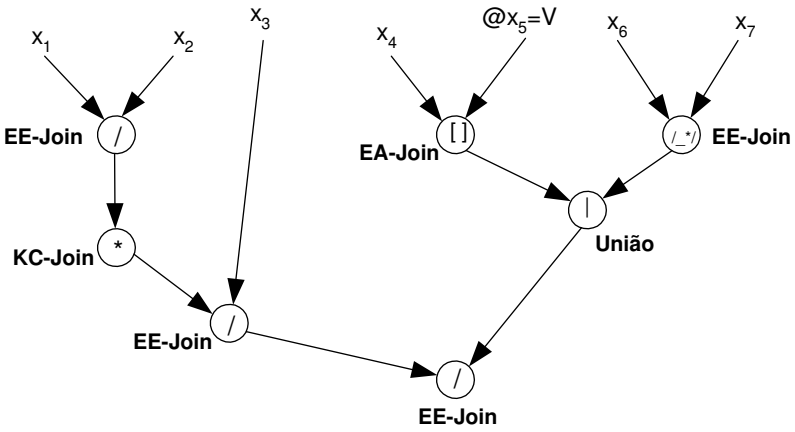


Figura 2.11: Decomposição de uma expressão regular de caminho no XISS

Para exemplificar, seja $A = '(x_1/x_2) * /x_3/((x_4[@x_5 = V]) \mid (x_6/\_ * /x_7))'$ a expressão regular de caminho a ser processada pelo *XISS*. A decomposição de A em sub-expressões pode ser vista na figura 2.11, assim como o algoritmo utilizado para juntar os resultados intermediários em cada ponto.

2.3.2 XR-Tree

O esquema de numeração utilizado para o índice *XR-Tree* [29] baseia-se no percurso da árvore G, correspondente aos documentos XML na Base de Dados, segundo o algoritmo de pré-ordem, que seqüencialmente atribui números a cada nó visitado. Cada nó $v \in V_G$ recebe dois inteiros $start_v$ e end_v que indicam quando v foi visitado, respectivamente, pela primeira e pela última vez. Na figura 2.12, vemos parte de uma árvore XML e os números atribuídos a cada um de seus nós. Este esquema de numeração é semelhante ao esquema de Dietz e segue as mesmas regras para identificar a relação ancestral-descendente e pai-filho entre dois pares de nós.

Cada nó em $v \in V_G$ é representado por três inteiros $(key_v, start_v, end_v)$, onde key_v é a chave de v, tal que $start_v \leq key_v \leq end_v$. Por definição, uma árvore *XR-Tree* é uma árvore B+ modificada com chaves de busca complexas – obtidas a partir de algum processamento sobre os valores de key_v, $start_v$ e end_v – e com uma lista de intervalos associada a cada um de seus nós internos.

Uma *XR-Tree* é construída para cada lista de elementos ou atributos presentes na

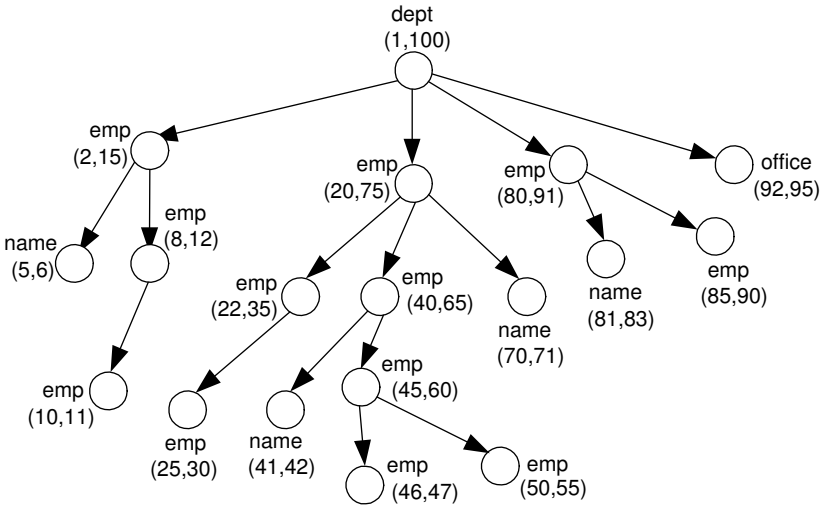


Figura 2.12: Esquema de Numeração utilizada no *XR-Tree*

Base de Dados. O objetivo deste índice é identificar eficientemente não apenas a relação ancentral-descendente, mas também o inverso, ou seja, a relação descendente-ancestral.

A *XR-Tree* correspondente à lista de elementos 'emp' na figura 2.12 pode ser vista na figura 2.13. As chaves estão destacadas nos nós internos e nos nós-folha. Nas folhas em particular, para cada nó $v \in V_G$, $key_v = start_v$. Maiores detalhes sobre as propriedades e operações de atualização e busca deste índice em [29].

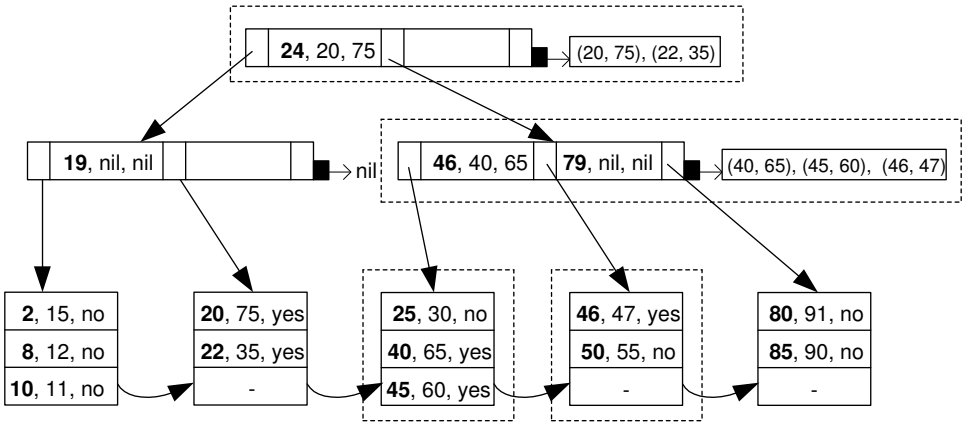


Figura 2.13: Árvore *XR-Tree* e os nós visitados durante uma busca por descendentes

Na figura 2.13, os nós de índice destacados representam os nós visitados durante a busca por todos os 'emp' descendentes do nó da Base de Dados cujo intervalo é (40, 65).

O resultado são todos os nós v ($v \in V_G$), representados nos nós-folha destacados do índice, tais que $start_v > 40$ e $end_v < 65$, ou seja, obtemos como resultado os nós da Base de Dados cujos intervalos são (45, 60), (46, 47) e (50, 55).

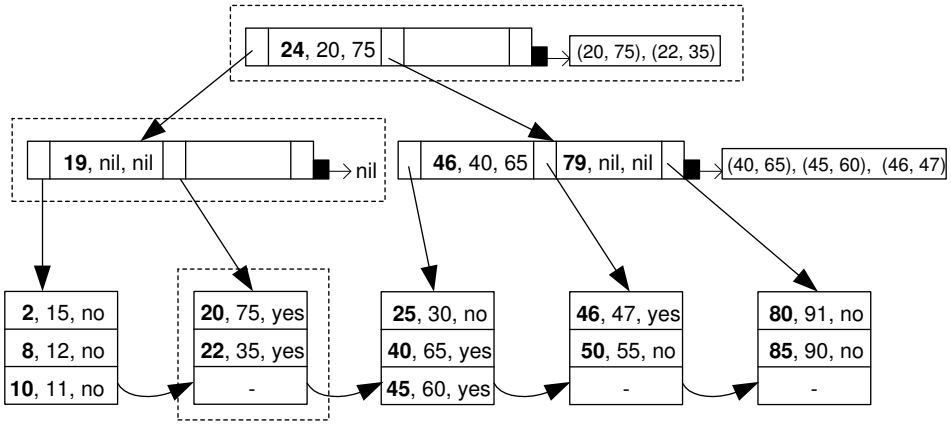


Figura 2.14: Árvore *XR-Tree* e os nós visitados durante uma busca por ancestrais

A figura 2.14 ilustra quais nós de índice são visitados durante a busca por todos os 'emp', ancestrais do nó da Base de Dados cujo intervalo é (40, 65). O resultado da busca é composto por: (i) todos os nós $v \in V_G$ representados nas listas dos nós internos destacados no índice, tais que $start_v < 40 < end_v$ e (ii) todos os nós $v' \in V_G$ representados no nó-folha destacado do índice, tal que $start_{v'} < 40 < end_{v'}$ e tal que o *flag* de v' tenha o valor 'no'. Logo, esta busca retorna como resultado o nó da Base de Dados cujo intervalo é (20, 75).

Estes dois procedimentos de busca são, por fim, combinados no algoritmo *XR-stack*, responsável por processar *structural joins* entre duas listas de elementos, uma de ancestrais e outra de descendentes, indexadas através de *XR-trees*.

2.4 Indexação com estruturas otimizadas para busca

A terceira classe de estratégias de indexação diz respeito àquelas que codificam as expressões de caminho presentes na Base de Dados XML em *strings*. Em seguida, utilizam estas expressões codificadas como chaves em estruturas otimizadas para busca de *strings*, como árvores B+ [27], árvores sufixo [30], árvores patrícias [31] ou variações.

Uma busca por uma expressão de caminho A qualquer envolve dois passos: (i) codifica-se A segundo o mesmo algoritmo utilizado para codificar as expressões de caminho da Base de Dados e (ii) percorre-se os nós da árvore de índice até que se chegue a uma folha que,

por sua vez, aponta para o conjunto de elementos ou atributos XML alcançáveis através de A.

Ao contrário dos sumários, esta classe de índices não exige nenhum tipo de armazenamento XML em especial, podendo ser adotado aqui qualquer modelo lógico.

Uma estrutura de dados otimizada para busca retorna os itens de dados desejados em tempo constante, minimizando os acessos a disco. Um exemplo de tal estratégia de indexação é o *Index Fabric*.

2.4.1 Index Fabric

O *Index Fabric* [32], uma versão balanceada de árvores patrícias, tem como chaves de busca uma combinação entre (i) expressões de caminho simples que originam-se na raiz do documento XML e (ii) valores de elementos e atributos.

As expressões de caminho são codificadas através da união de caracteres especiais ou pequenas *strings* denominadas *designators*, onde cada nome de elemento ou atributo tem o seu *designator* correspondente. O mapeamento entre ambos é mantido através de uma tabela chamada *designator dictionary*, armazenada no Banco de Dados. Um exemplo desta codificação é mostrada na figura 2.15. O fragmento XML mostrado na figura 2.15 seria inserido no *Index Fabric* como '**IBN**ABC Corp'.

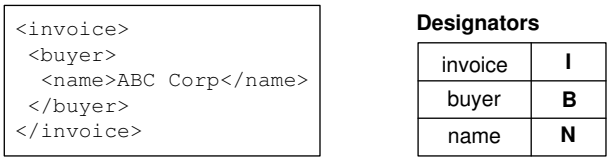


Figura 2.15: Fragmento XML e seus *designators*

Uma árvore patrícia é desbalanceada e, portanto, inadequada à indexação em memória secundária. No entanto, a estrutura da árvore patrícia usada no *Index Fabric* é dividida em camadas, de maneira que qualquer consulta ao *Index Fabric* requer um número constante de acessos a disco. Isto ocorre porque, caso a árvore não caiba em uma página de disco, a mesma é dividida em sub-árvores – cujos tamanhos não são maiores que uma página em disco – que, por sua vez, são indexadas por uma segunda árvore, originando assim mais uma camada de índice. Esse processo continua até que a nova camada criada seja do tamanho de uma página em disco. A figura 2.16 mostra esta divisão.

O *Index Fabric*, a exemplo do *DataGuide*, armazena todas as expressões de caminho simples com origem na raiz do documento, assim como os valores de elementos e atributos.

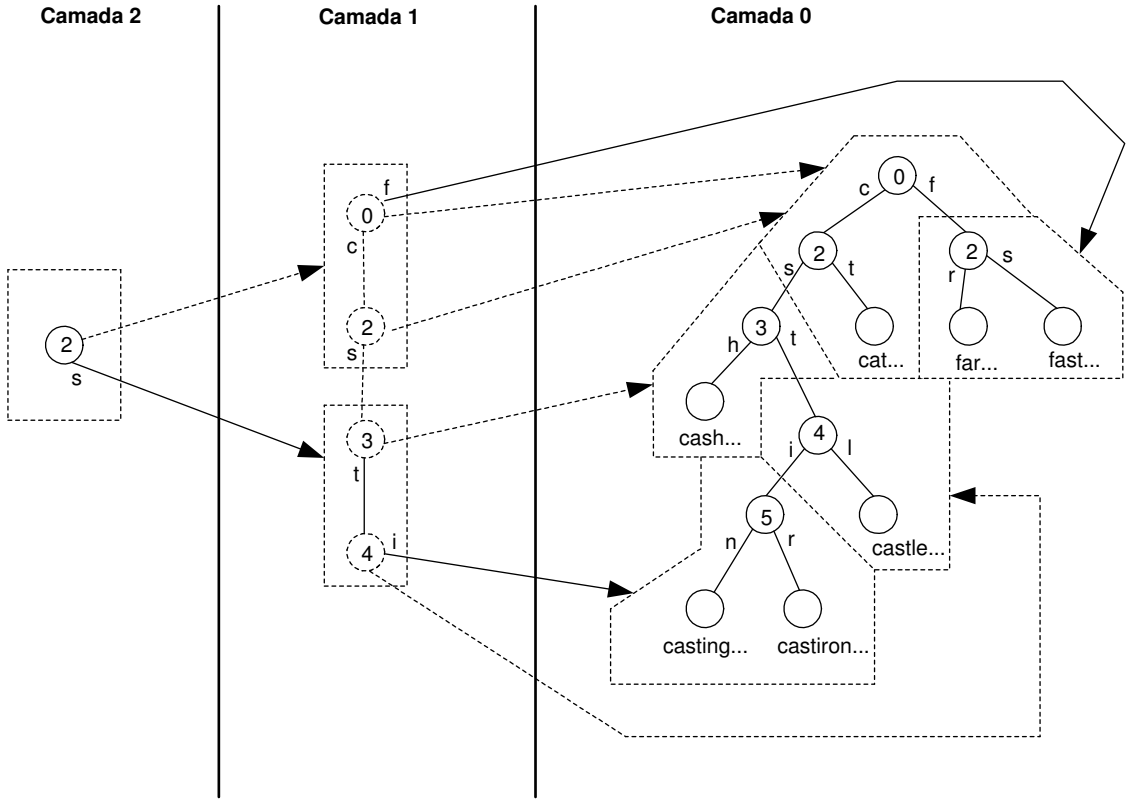


Figura 2.16: Camadas do *Index Fabric*

Por essa característica, há quem o considere como um índice da classe dos sumários. No entanto, como definimos a princípio, um sumário é um grafo OEM – com uma ou outra modificação, dependendo da estratégia de indexação utilizada – e uma vez que uma árvore patrícia não se enquadra nesta definição e, além disso, possui algumas características muito peculiares [31], achamos mais adequado incluir o *Index Fabric* nesta classe de índices.

2.5 Estratégias de Indexação Híbridas

As estratégias de indexação híbridas são aquelas que apresentam características de duas ou mais das classes de indexação vistas anteriormente. São índices mais complexos e mais eficientes do ponto de vista de tempo de processamento de consultas e número de acessos a disco. Além disso, são capazes de processar uma variedade maior de expressões de caminho, notadamente de expressões de caminho XPath [10]. Como exemplo de tais índices, temos o *RIST* e o *PathGuide*.

2.5.1 RIST

O *RIST* (*Relationships Indexed Suffix Tree*) [33] possui características de índices com esquema de numeração e de índices que utilizam estruturas de dados otimizada para busca de *strings*.

Dado um documento D armazenado na Base de Dados, os elementos e atributos de D e seus respectivos valores são codificados como uma seqüência de caracteres C_D. A busca por uma expressão de caminho A em D envolve dois passos: (i) A é codificada em uma seqüência C_A, seguindo o mesmo algoritmo utilizado para codificar D, e (ii) é feito o casamento de sub-expressões de C_A sobre C_D.

A codificação de um documento D é feita da seguinte forma: percorre-se a árvore G_D, correspondente a D, em pré-ordem e representa-se cada nó $v_i \in V_{G_D}$ por um par do tipo (***símbolo***, ***prefixo***) onde ***símbolo*** é o nome de v_i e ***prefixo*** representa a expressão de caminho simples que vai da raiz de G_D até v_i.

Um exemplo de codificação pode ser visto na figura 2.17. Nesta figura, por simplicidade, todos os nomes de elementos e atributos foram substituídos por letras e os valores dos mesmos foram substituídos por inteiros i_j, onde i_j é o resultado de uma função *hash* h.

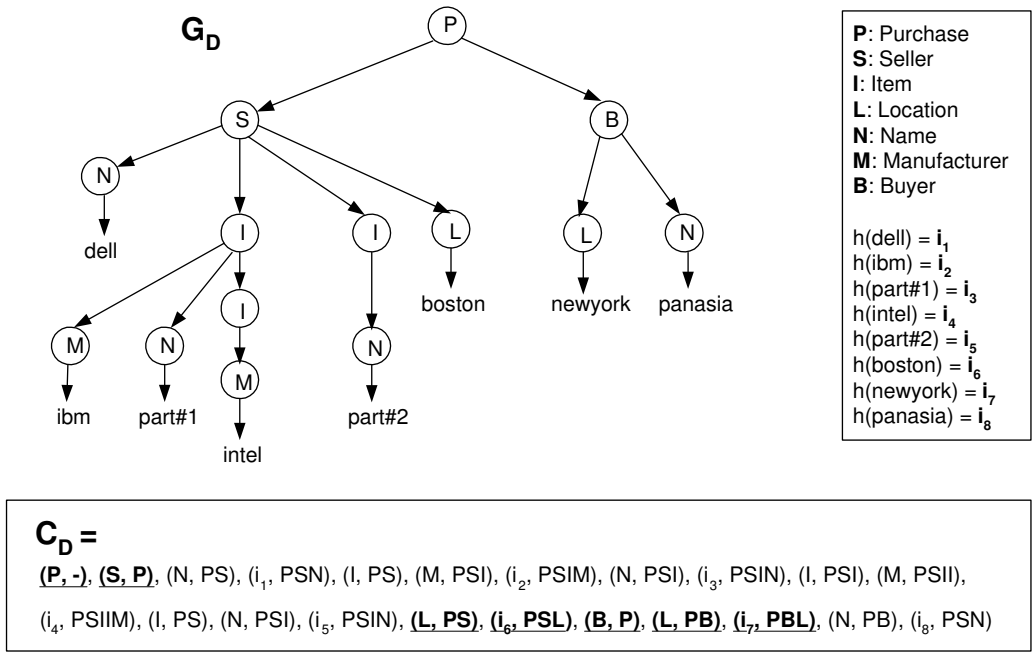


Figura 2.17: Codificação de um documento XML no RIST

A expressão de caminho A = 'Purchase[Seller[Location = boston]]/Buyer[Location = newyork]' pode ser codificada como C_A =(P,−), (S,P), (L,PS), (i_5,PSL), (B,P), (L,PB), (i_7,PBL). A seqüência assinalada na figura 2.17 indica o casamento de padrões entre C_A e C_D.

Uma vez que obtemos a codificação $C_{D_1}, C_{D_2}, \ldots, C_{D_n}$, referentes aos documentos $D_1, D_2, \ldots, D_n$ da Base de Dados, indexamos todos os C_{D_i} ($1 \leq i \leq n$) através de uma árvore sufixo [30], como mostrado na figura 2.18. Cada elemento da seqüência C_{D_i}, para todo $1 \leq i \leq n$, corresponde a um conjunto de elementos ou atributos na Base de Dados XML. Uma vez que os mesmos estão envolvidos em duas árvores distintas – a árvore do documento e a árvore sufixo – existem dois tipos de relação ancestral-descendente a se considerar: a relação ancestral-descendente (i) entre nós da árvore do documento, chamada de *D-Ancestorship*, e (ii) entre nós da árvore sufixo, denominada *S-Ancestorship*. Segundo esta relação dizemos que, por exemplo, (S,P) é um *D-ancestor* de (L,PS) e que (i_1,PSN) é um *S-ancestor* de (L,PS).

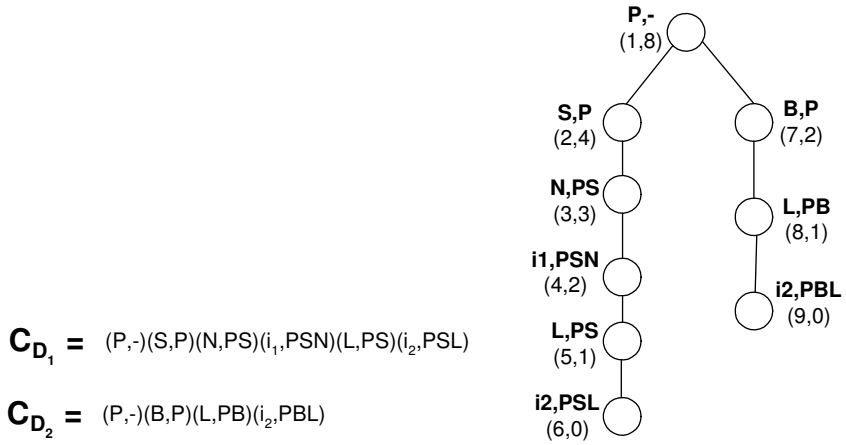


Figura 2.18: Codificação dos documentos D_1 e D_2 e árvore sufixo correspondente

O índice *RIST* indexa *D-ancestors* e *S-ancestors* através de árvores B+ [27]. Para tanto cada nó z da árvore sufixo é rotulado pelo par de inteiros $< order_z, size_z >$, onde $order_z$ é a ordem em que z foi visitado segundo um algoritmo de pré-ordem e $size_z$ é o número de descendentes de z. Um nó z é *S-ancestor* de z' se e somente se $order_{z'} \in (order_z, order_z + size_z]$. Este esquema de numeração está ilustrado na figura 2.18.

A árvore B+ que determina a relação *D-Ancestorship*, chamada de *D-Ancestor B+ Tree*, tem como chaves de busca todos os rótulos do tipo (*símbolo*, *prefixo*) presentes nos nós da árvore sufixo. Cada folha desta árvore B+ aponta para outras árvores B+,

denominadas *S-Ancestor B+ Trees*, que determinam a relação *S-Ancestorship*. As *S-Ancestor B+ Trees* têm como chaves de busca os números $order_z$, atribuídos a cada nó z da árvore sufixo. Estas árvores B+ estão ilustradas na figura 2.19.

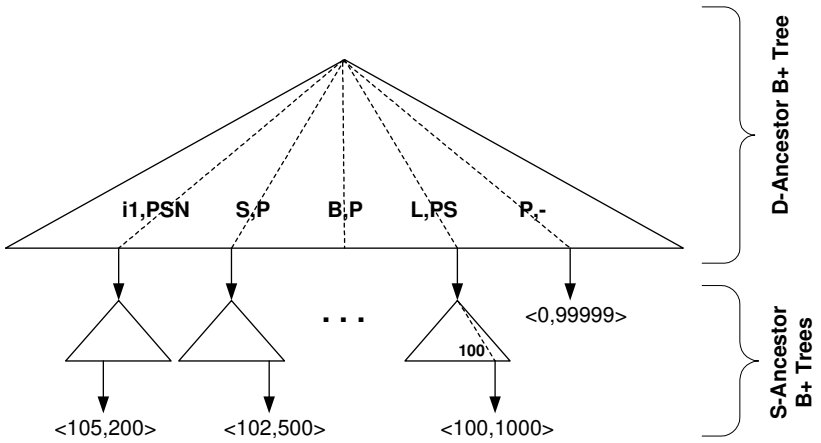


Figura 2.19: *D-Ancestor B+ Tree* e *S-Ancestor B+ Trees*

Seja a expressão de caminho A a ser resolvida pelo índice *RIST*, onde $C_A = q_1, \ldots, q_j$ e q_i (para todo $1 \leq i \leq j$) é do tipo (*símbolo*, *prefixo*). Suponha que o nó z, cujos números são $< order_z, size_z >$, seja um nó resultante do casamento de padrões $q_1, \ldots, q_{i-1}$ na árvore sufixo. Para casar o próximo elemento q_i, consultamos *D-Ancestor B+ Tree* usando q_i como chave de busca, o que nos leva a uma *S-Ancestor B+ Tree*. Em seguida fazemos uma busca por todas as chaves k nesta *S-Ancestor B+ Tree* tal que $order_z < k < size_z$. Para cada nó da árvore sufixo correspondente a algum k retornado nesta busca, usamos o mesmo processo para casar o elemento q_{i+1}, até que se chegue a q_j.

Uma evolução desta estratégia de indexação é o *ViST* (*Virtual Suffix Tree*) [33] que utiliza um esquema de numeração dinâmico, de maneira a suportar futuras inserções, o que não acontece com o esquema do *RIST*. O esquema de numeração do *ViST* baseia-se em estimativas obtidas a partir de informações estatísticas e semânticas dos dados XML. Neste caso, a árvore sufixo nem mesmo precisa ser construída. Maiores detalhes em [33].

2.5.2 PathGuide

O *PathGuide* [34] é uma estrutura de dados para indexação construída a partir de árvores sufixo e que se utiliza de esquemas de numeração, possibilitando a resolução de expressões de caminho mais complexas. O modelo de dados usado pelo *PathGuide*,

ilustrado na figura 2.20, é uma versão simplificada do modelo DOM [35].

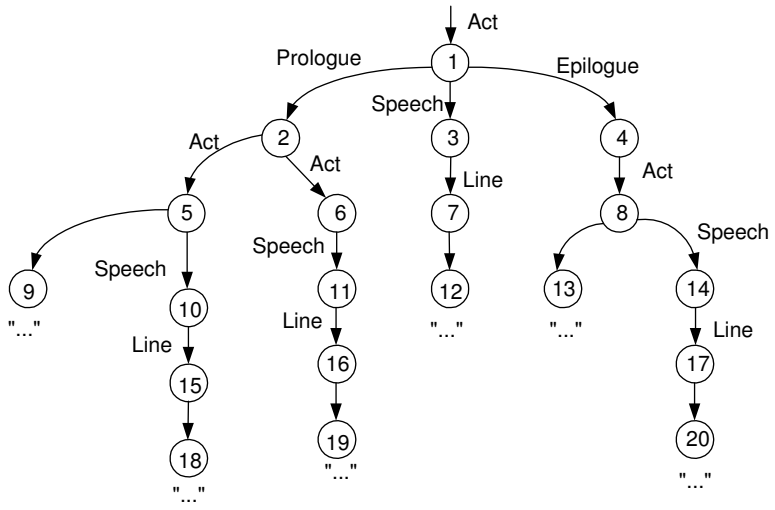


Figura 2.20: Modelo de Dados adotado no *PathGuide*

A construção do índice é feita em 4 etapas: (i) Extrai-se todo nó v que representa texto, juntamente com a aresta cujo destino é v, da árvore de dados da figura 2.20; (ii) substitui-se o rótulo das arestas na árvore resultante por símbolos denominados *designators*; (iii) funde-se todos os nós irmãos v e v', se as arestas cujos destino são v e v' possuem o mesmo rótulo e (iv) gera-se a árvore sufixo da árvore obtida no passo anterior, de maneira que todas as expressões de caminho simples, absolutas e relativas, sejam codificadas na árvore sufixo [34]. Uma *expressão de caminho simples absoluta* é do tipo '$/x_1/x_2/\ldots/x_n$' e uma *expressão de caminho simples relativa* é representada por '$//x_1/x_2/\ldots/x_n$', onde $x_1, x_2, \ldots, x_n$ são rótulos de arestas.

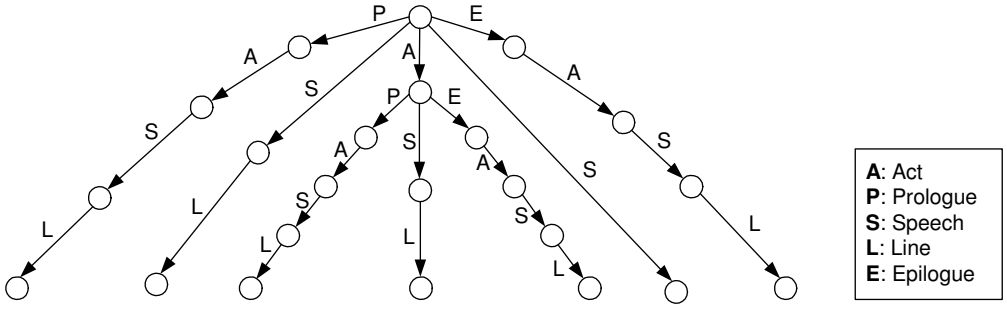


Figura 2.21: *PathGuide*

A árvore obtida no passo 4 para a Base de Dados da figura 2.20 pode ser vista na figura 2.21. Uma vez que o *PathGuide* armazena identificadores de elementos em cada um de seus nós, o mesmo resolve eficientemente qualquer expressão de caminho simples absoluta ou relativa. Para resolver expressões de caminho mais sofisticadas, que envolvam o eixo ancestral-descendente ou predicados sobre elementos, um esquema de numeração semelhante ao apresentado no *XISS* [28] é utilizado. Uma extensão dos algoritmos em [28, 22] suporta o processamento de expressões de caminho como '$x_i//x_j$', '$x_j[x_k$=V]', '$x_i[@x_j$=V]', '$x_j[//x_k$=V]', onde V é uma *string*.

2.6 Comparações

Podemos distinguir pontos positivos e negativos em cada classe de estratégias de indexação. Nesta seção faremos uma análise comparativa entre as mesmas.

Nas estratégias de indexação por sumário, os documentos XML são armazenados na Base de Dados segundo um modelo lógico de grafos ou árvores. Um sumário também é um grafo ou uma árvore que tem por objetivo "resumir" a estrutura dos documentos XML em disco. O mesmo representa expressões de caminho, onde seus nós referem-se a um ou mais elementos ou atributos de mesmo nome.

Alguns sumários – aqueles que representam todas as expressões de caminho simples da Base de Dados – como o *DataGuide* [16], *ToXin* [18] e *1-Index* [19] também oferecem uma espécie de esquema dinâmico, de maneira a permitir que consultas possam ser feitas sem nenhum conhecimento prévio sobre os dados XML armazenados. Estes índices resolvem muito bem expressões de caminho simples que se iniciam na raiz de documentos XML. No entanto, expressões de caminhos mais complexas, quando suportadas pela estratégia de indexação, exigem índices auxiliares e várias operações de junção entre os nós do sumário.

Uma vez construído o sumário, nenhum acesso adicional à Base de Dados precisa ser feito pois, além de representarem a estrutura dos dados XML, os nós do índice também armazenam seu conteúdo. Isso faz com que um sumário seja, em alguns casos, tão grande quanto a Base de Dados. Na estratégia do *DataGuide*, o mesmo pode ser inclusive maior que a Base de Dados, caso esta possua ciclos [16].

Em uma tentativa de diminuir o tamanho dos sumários, estratégias como o *APEX* [21] e o *A(k)-index* [20] surgiram. O *APEX* [21] indexa somente as expressões de caminho mais frequentes em consultas sobre a Base de Dados, obtidas através de um algoritmo de *Data Mining*. O *A(k)-index* indexa somente as expressões de caminho de tamanho até um certo t. Mesmo assim, no caso do *A(k)-index* por exemplo, o índice ainda fica

muito grande (mais de 40% do tamanho da Base de Dados em uma das cargas testadas) [20]. Além disso, para resolver expressões de caminho não previstas no índice uma grande quantidade de operações de junção entre nós do sumário precisa ser feita, tornando o custo de execução de consultas alto.

As estratégias de indexação baseadas em esquemas de numeração, por sua vez, oferecem outras alternativas para resolver expressões de caminho, principalmente as que envolvem o operador ancestral-descendente ('//') e coringas ('*'). Através dos esquemas de numeração, os nós da Base de Dados são divididos em regiões ou intervalos tal que, a grosso modo, dados dois nós v e v', v é ancestral de v' se o intervalo de v' está contido no intervalo de v.

Nesta classe de estratégias de indexação, listas de elementos ou atributos com mesmo nome são indexados através de índices como árvores B+ [27], de maneira que resolver a expressão 'x_i/x_j' ou '$x_i//x_j$' significa comparar os números de todos os elementos cujo nome é 'x_i' com os números de todos os elementos cujo nome é 'x_j'. Esta operação de junção é chamada de *structural join* [22].

Um dos pontos negativos desta estratégia de indexação é que, à medida que inserções vão ocorrendo na Base de Dados, o gerenciamento dos intervalos dos nós torna-se mais complexo. Em alguns casos, a inserção de nós exige que novos intervalos sejam atribuídos aos elementos e atributos, causando a completa reconstrução do índice. O *XISS* [28] apresenta um esquema de numeração mais flexível. No entanto, o mesmo suporta apenas um número limitado de inserções e, caso a Base de Dados seja freqüentemente atualizada, o custo de manutenção do índice torna-se muito caro.

Um outro tópico que merece atenção nesta estratégia de indexação são os algoritmos utilizados nos *structural joins*. Como estas operações são freqüentes, é fundamental que os algoritmos de junção sejam eficientes. O *XISS* apresenta uma série de algoritmos de junção, que o permite resolver inclusive expressões regulares de caminho. Já o *XR-Tree* [29] propõe uma nova estrutura de indexação, baseada em árvores B+, para nomes de elementos e atributos que permite determinar eficientemente tanto a relação ancestral-descendente como a relação descendente-ancestral entre elementos de duas listas.

A terceira estratégia de indexação diz respeito àquelas que utilizam estruturas otimizadas para busca de *strings*, cujas chaves são expressões de caminho codificadas. Como único representante desta classe de índices, temos o *Index Fabric* [32]. Este índice resolve eficientemente expressões de caminho simples que se iniciam na raiz dos documentos presentes na Base de Dados. Para resolver expressões de caminho mais complexas, uma série de operações deve ser feita sobre os nós na Base de Dados. Alternativamente, o

Index Fabric permite que o usuário defina expressões de caminho mais complexas e/ou mais frequentes chamadas *refined paths* [32], a serem representadas no índice. No entanto, é preciso que o usuário (i) tenha um conhecimento prévio sobre as expressões mais frequentes em consultas sobre a Base de Dados e (ii) que as codifique manualmente.

A última classe de estratégias de indexação mistura conceitos de duas ou mais das estratégias resumidas nos parágrafos anteriores, notadamente as que utilizam esquemas de numeração e estruturas otimizadas para busca de *strings*. As duas estratégias incluídas nesta classe, o *RIST* [33] e o *PathGuide* [34], codificam expressões de caminho e as indexam através de árvores sufixo [30].

O *RIST* aplica um esquema de numeração sobre os nós da árvore sufixo e, baseado neste esquema e na codificação das expressões de caminho da Base de Dados, uma complexa estrutura de indexação, baseada em árvore B+, é apresentada. Com ela é possível resolver qualquer expressão regular de caminho sem que nenhuma de operação de junção seja necessária. Um problema com esta estratégia está no fato de que a mesma utiliza um esquema de numeração estático, o que impede futuras inserções na árvore sufixo.

O *PathGuide* [34] constrói a árvore sufixo de maneira a representar todas as expressões de caminho simples absolutas e relativas presentes na Base de Dados. Logo, estas expressões são resolvidas eficientemente apenas através de uma consulta à árvore, uma vez que cada nó da mesma armazena identificadores de nós na Base de Dados. Para resolver expressões de caminho mais complexas, um esquema de numeração é utilizado e os números também são armazenados nos nós da árvore sufixo.

Através deste esquema de numeração, que por sua vez é baseado no esquema do *XISS*, é possível resolver eficientemente expressões regulares de caminho. As desvantagens desta estratégia são: (i) a estratégia de indexação em si e o problema das inserções e (ii) a utilização de uma árvore sufixo como índice. Uma árvore sufixo não é adequada ao armazenamento em memória secundária, pois é desbalanceada e, além disso, se a estrutura dos documentos na Base de Dados é muito irregular, a árvore sufixo sofre o mesmo problema que os sumários: a mesma pode ser tão grande quanto a Base de Dados.

Capítulo 3

FoX

O FoX é um projeto de Banco de Dados XML Nativo, independente de esquema, e cujo armazenamento é nativo, desenvolvido nesta Universidade. Neste capítulo, apresentaremos a sua arquitetura e as funcionalidades dos seus principais sub-sistemas.

O restante deste capítulo está organizado da seguinte forma: na seção 3.1 apresentamos a arquitetura geral do FoX. Em seguida, na seção 3.2, discutimos o Servidor de Páginas, que é o componente responsável pela interface entre o FoX e o disco. Na seção 3.3, apresentamos o Gerenciador de Armazenamento – o componente em atual fase de desenvolvimento – e seus dois módulos: o Módulo de Armazenamento (seção 3.3.1) e o de Indexação (seção 3.3.2). Dentro da seção referente ao Módulo de Armazenamento, fazemos ainda uma breve introdução ao modelo lógico (seção 3.3.1.1) e à estratégia de armazenamento (seção 3.3.1.2) utilizados pelo FoX, bem como as operações que este módulo desempenha.

3.1 Arquitetura Geral

O FoX compreende dois sub-sistemas principais: o Processador de Consultas e o Gerenciador de Armazenamento (figura 3.1).

O Processador de Consultas, baseado na álgebra XML do W3C [36], envolve o Compilador, o Otimizador e o Motor de Execução de Consultas. As consultas são, portanto, escritas em XQuery [37] e submetidas ao Processador de Consultas que, após reconhecimento e otimização, gera os planos de execução. Esses planos são, por sua vez, argumentos para o Motor de Execução que os executa utilizando-se dos serviços oferecidos pelo Gerenciador de Armazenamento. Este último interage, no nível mais baixo, com a camada de persistência de páginas denominada Servidor de Páginas, responsável pelo serviço de

escrita e leitura de dados no disco.

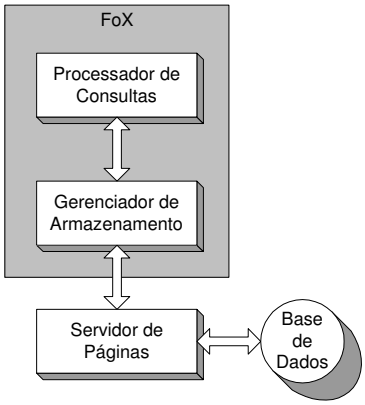


Figura 3.1: Arquitetura geral do FoX

No estágio atual de desenvolvimento, está o Gerenciador de Armazenamento, responsável por: (i) persistir em disco, através do Servidor de Páginas, os dados XML de acordo com a estratégia nativa de armazenamento e (ii) recuperar do disco sub-árvores XML, resultantes de consultas sobre a Base de Dados.

3.2 O Servidor de Páginas

O Servidor de Páginas oferece um conjunto de funções para recuperar (armazenar) *streams* de *bytes* do (no) disco. O número de *bytes* a serem recuperados ou armazenados é limitado pelo tamanho de página do FoX, que é um parâmetro configurável. Nesta camada, os *bytes* não têm semântica alguma. Os mesmos só a adquirem depois que são interpretados pelo Gerenciador de Armazenamento.

O Servidor de Páginas também oferece as funcionalidades de gerenciamento de *buffer* e *cache*, tornando o acesso a dados mais rápido e eficiente. Neste módulo do FoX, estamos utilizando o serviço de páginas implementado no Gerenciador de Objetos Armazenados GOA++ [38], sistema desenvolvido na COPPE, UFRJ.

3.3 O Gerenciador de Armazenamento

O Gerenciador de Armazenamento é o sub-sistema responsável pelo armazenamento e indexação do conteúdo XML. Na arquitetura desse sub-sistema, temos como núcleo os

Módulos de Armazenamento e Indexação (figura 3.2).

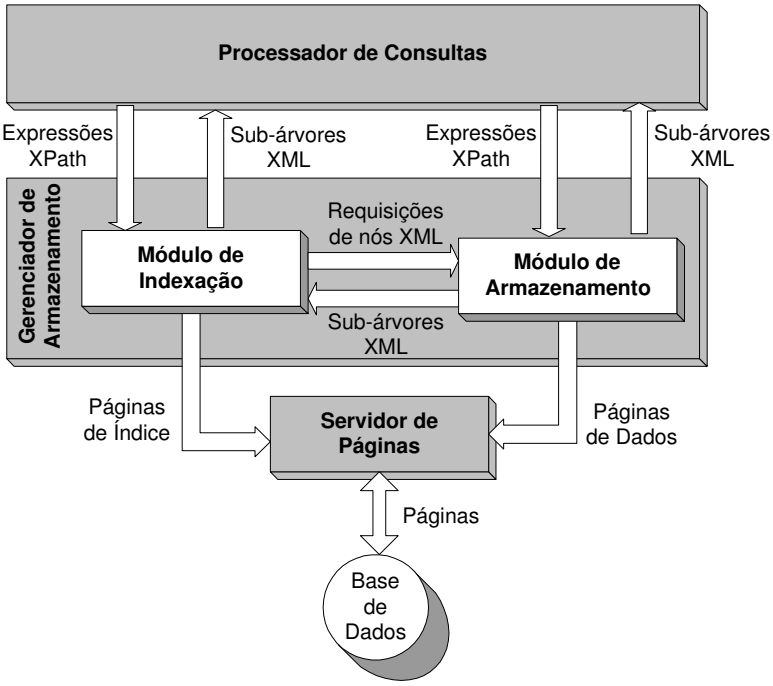


Figura 3.2: Arquitetura do Gerenciador de Armazenamento do FoX

O Processador de Consultas – mais especificamente o Motor de Execução de Consultas, responsável pela implementação dos operadores físicos da álgebra XQuery – interage com o Gerenciador de Armazenamento através de expressões de caminho XPath [10], extraídas das consultas XQuery. Estas expressões de caminho podem ser enviadas diretamente ao Módulo de Armazenamento ou, alternativamente, ao Módulo de Indexação, a fim de otimizar o acesso. Em qualquer um destes módulos, a expressão XPath é avaliada e sub-árvores XML, presentes na Base de Dados, são retornadas como resultado.

Para resolver as expressões de caminho XPath, o Gerenciador de Armazenamento faz chamadas ao Servidor de Páginas. Com isso, é possível recuperar dados XML, no caso em que as expressões de caminho são repassadas ao Módulo de Armazenamento, ou as estruturas do índice, caso as expressões sejam enviadas ao Módulo de Indexação. Além disso, a fim de recuperar os dados representados nas folhas do índice, o Módulo de Indexação interage com o Módulo de Armazenamento, passando endereços de nós XML e recebendo, em troca, as sub-árvores cujas raízes são estes nós. Em seguida, estas sub-árvores XML são repassadas às camadas superiores do FoX.

3.3.1 O Módulo de Armazenamento

O Módulo de Armazenamento[1] desempenha as funções de recuperação, inserção e remoção do conteúdo XML de acordo com as consultas e atualizações feitas sobre o mesmo, na Base de Dados. Antes de explicar como isto é feito, detalharemos o modelo lógico e a estratégia de armazenamento adotados no FoX.

3.3.1.1 Modelo Lógico de Armazenamento

O FoX adota o *modelo lógico de árvores* para armazenar documentos XML. Um documento XML $D = \{x_1, x_2, \ldots, x_n\}$, onde x_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 $v_i \in V_{G_D}$ corresponde a um $x_i \in D$, e vice-versa;
- Existe uma aresta $e_{ij} \in E_{G_D}$ entre dois nós v_i e v_j $(v_i, v_j \in V_{G_D})$ se, e somente se, x_i é o elemento XML pai de x_j $(x_i, x_j \in D)$ e
- Cada nó v_i em V_{G_D} é rotulado com o nome do seu elemento ou atributo x_i correspondente e possui um identificador único na Base de Dados.

```
<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 3.3: Documento XML

[1]Um outro trabalho de Mestrado desta Universidade

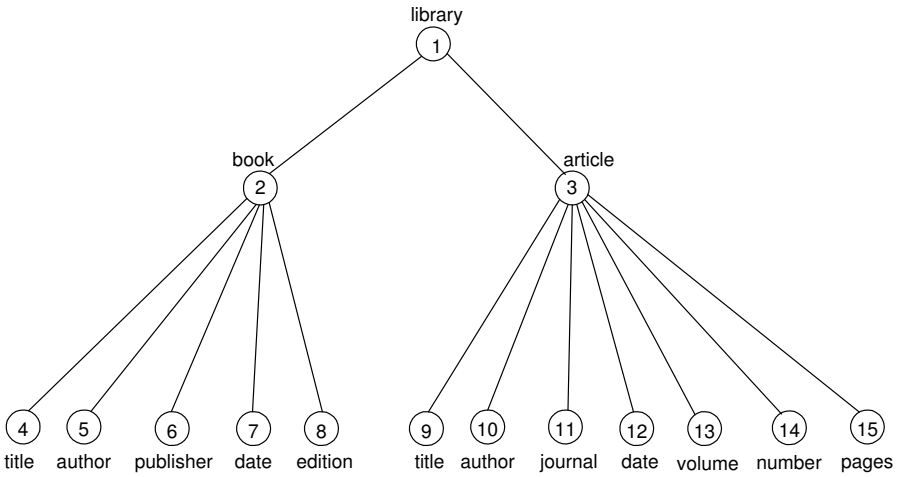


Figura 3.4: Árvore correspondente a um documento XML, armazenada no FoX

Na figura 3.4, temos a representação em forma de árvore do documento XML da figura 3.3. Nesta figura, os identificadores são os rótulos dos nós e os valores textuais de cada nó, caso existam, são armazenados no próprio nó. Se $D_1, D_2, \ldots, D_n$ são os documentos XML a serem gerenciados pelo FoX, então a Base de Dados $BD = \{G_{D_1}, G_{D_2}, \ldots, G_{D_n}\}$.

3.3.1.2 Estratégia de Armazenamento

A estratégia utilizada para persistir as árvores lógicas em disco é baseada na estratégia do *Natix* [2]. 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 de dados*, onde cada registro $\hat{r}$ armazena uma das sub-árvores desta árvore. Uma página de disco pode conter um ou mais registros. A figura 3.5 mostra a divisão de uma árvore física em registros de dados, onde os nós pontilhados são os nós de controle.

3.3.1.3 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

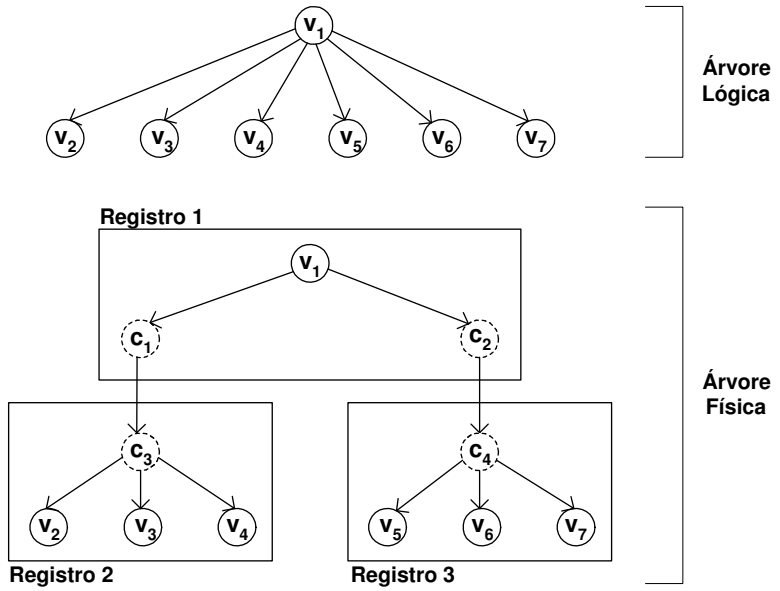


Figura 3.5: Árvores lógica e física do FoX

caminho XPath da entrada, são retornadas.

Para inserir um novo nó v em alguma árvore lógica do FoX, primeiramente é decidido onde, na árvore física, v deve ser inserido. Uma vez feito isto, adicionamos v à sub-árvore correspondente a ele, armazenada em algum registro do FoX, digamos o registro $\hat{r}$. Se após a inserção de v, a sub-árvore não cabe mais em $\hat{r}$, ocorre um *split*, ou seja, um novo registro $\hat{r}'$ é criado e a sub-árvore correspondente a $\hat{r}$ é então dividida entre os dois registros $\hat{r}$ e $\hat{r}'$. Esta operação de *split*, no pior caso, propaga-se até o registro que contem a raiz da árvore física.

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

3.3.2 O 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*, que veremos com mais detalhes no capítulo 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, por exemplo, porque a maior parte das consultas XQuery [37] fazem alguma referência à estrutura do documento, através de expressões XPath [10].

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 3.6 mostra a arquitetura do Módulo de Indexação, bem como a sua interação com o Módulo de Armazenamento, Servidor de Páginas e Processador de Consultas.

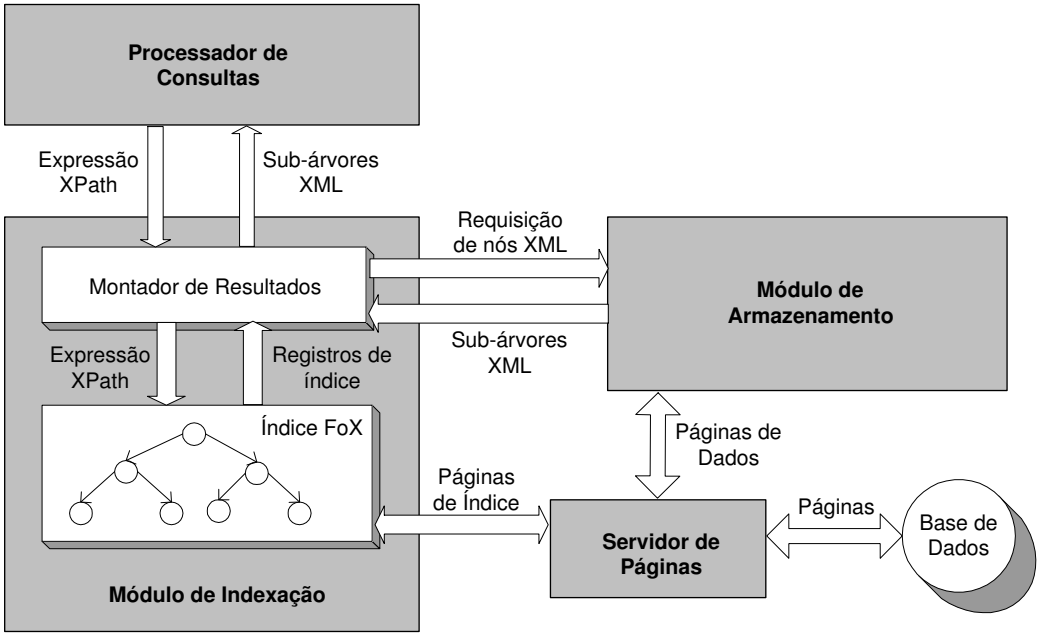


Figura 3.6: Arquitetura do Módulo de Indexação do FoX

Primeiramente, o Processador de Consultas envia uma expressão de caminho A 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 A. De posse destes nós de índice,

realizamos uma busca e chegamos a um conjunto R de *registros de índice*. Cada registro r em R corresponde ao endereço de um nó XML na Base de Dados, alcançável através de A. A figura 3.7 representa graficamente os nós e os registros do *Índice FoX*.

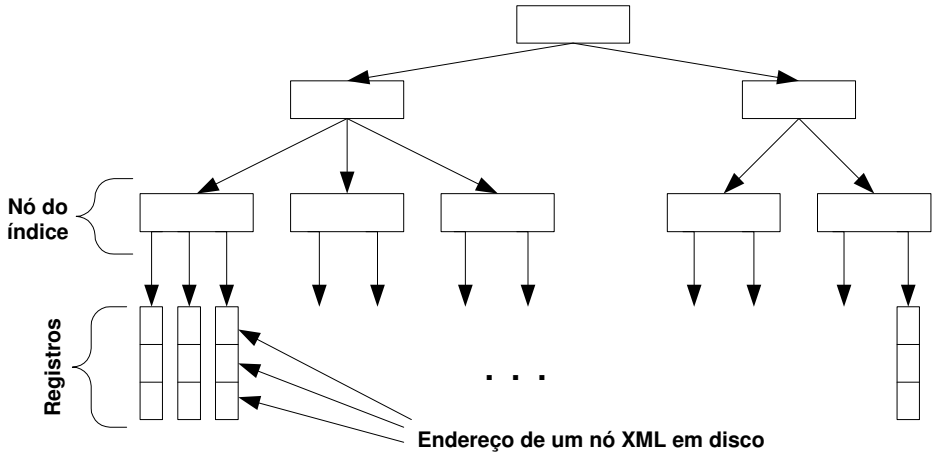


Figura 3.7: *Layout* do *Índice FoX*

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.

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 no capítulo 5.

Capítulo 4

Árvores de Indexação

Nesta seção, apresentamos as árvores de indexação utilizadas no *Índice FoX*. Estas árvores fazem parte de uma estrutura de dados denominada *String B-tree* [39], utilizada, em princípio, para indexar longas cadeias de caracteres.

Resumidamente, uma *String B-tree* é uma árvore balanceada que oferece as facilidades de busca encontradas nas árvore B+ [27] e, ao mesmo tempo, as vantagens de compressão de dados das árvores patrícias [31]. Com efeito, uma *String B-tree* é uma combinação destas duas estruturas de dados.

Antes de descrever os problemas que a *String B-tree* resolve eficientemente, vamos a algumas definições.

Com relação a uma *string* $\hat{s}$ qualquer, dizemos que:

- O *tamanho* em caracteres de $\hat{s}$ é representado por $|\hat{s}|$;
- Um *prefixo* de $\hat{s}$ é formado pelos seus primeiros i-ésimos caracteres, onde $1 \leq i \leq |\hat{s}|$;
- Um *sufixo* de $\hat{s}$ é formado pelos seus últimos j-ésimos caracteres, onde $1 \leq j \leq |\hat{s}|$.

Denominamos por $\mathcal{S}$ o conjunto de *strings* ao qual desejamos indexar. $\mathcal{S}'$ é definido como o conjunto de sufixos de $\mathcal{S}$, tal que: (i) para todo $s \in \mathcal{S}$, todos os sufixos de s pertencem a $\mathcal{S}'$ e (ii) todo elemento de $\mathcal{S}'$ é sufixo de alguma *string* em $\mathcal{S}$.

Uma *String B-tree*, correspondente a $\mathcal{S}$, é capaz de solucionar os seguintes problemas:

- *Problema* ***Busca-Prefixo***: Dada uma *string* k, recuperar todos os elementos de $\mathcal{S}$ que possuem k como prefixo;
- *Problema* ***Busca-Substring***: Dada uma *string* k, recuperar todos os elementos de $\mathcal{S}$ que possuem k como *substring*.

Note que se é possível identificar de qual *string* de $\mathcal{S}$ s' ($\in \mathcal{S}'$) é *substring*, podemos reescrever o problema *Busca-Substring* em função do problema *Busca-Prefixo* da seguinte

forma: "Dada uma *string* k, recuperar todos os elementos de $\mathcal{S}'$ que possuem k como prefixo". Neste caso, enquanto a *String B-tree*, correspondente ao problema *Busca-Prefixo*, é construída a partir das *strings* em $\mathcal{S}$, a *String B-tree*, correspondente ao problema *Busca-Substring*, é construída a partir das *strings* em $\mathcal{S}'$.

O restante do capítulo está organizado da seguinte forma: na seção 4.1, detalhamos a estrutura de dados da *String B-tree*. Em seguida, na seção 4.2 apresentamos os algoritmos de consulta e atualização feitos nesta estrutura. A seção 4.3 introduz e estrutura das árvores patrícias e a seção 4.4 detalha as operações feitas sobre tais árvores. Finalmente, a seção 4.5 traz um estudo de complexidade para a *String B-tree*.

4.1 Estrutura de Dados da *String B-tree*

Assumimos que as *strings* do conjunto $\mathcal{S}$ estão armazenadas em uma porção contígua do disco e que cada página de disco tem capacidade de B itens atômicos, os quais podem ser inteiros, caracteres ou ponteiros. A figura 4.1 mostra um conjunto $\mathcal{S}$ e o seu armazenamento em disco, onde $B = 8$. O caracter "$" é usado para delimitar o fim de cada *string*.

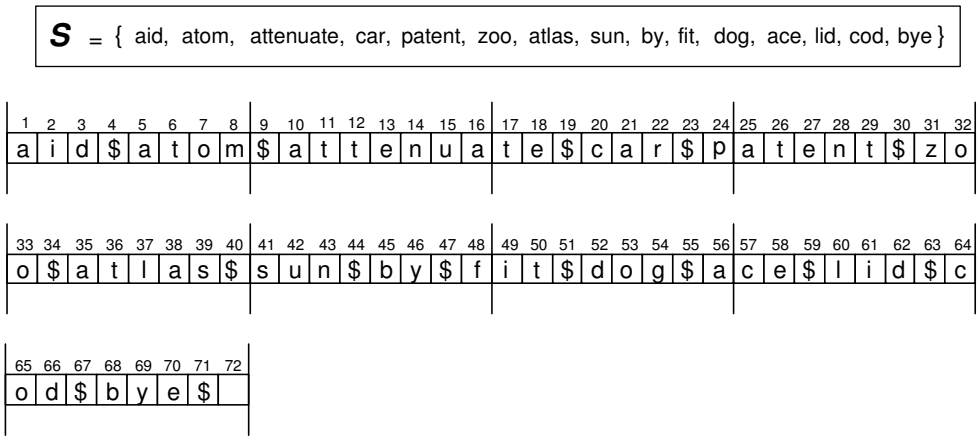


Figura 4.1: Conjunto $\mathcal{S}$ e seu armazenamento em disco

Cada *string* de $\mathcal{S}$ ou $\mathcal{S}'$ é sempre representada, em uma *String B-tree*, através do *deslocamento* do seu primeiro caracter, onde este deslocamento é obtido com relação ao ponto onde iniciou-se o armazenamento do conjunto $\mathcal{S}$. Por exemplo, a *string* "fit" é representada pelo inteiro 48, enquanto que a *string* "tenuate" é representada pelo inteiro 12. Chamamos este deslocamento de *ponteiro lógico*. Portanto, de agora em diante,

quando mencionarmos "*string*" no contexto de uma *String B-tree*, estaremos nos referindo, na verdade, ao seu respectivo ponteiro lógico.

Dado um conjunto de *strings* $\hat{\mathcal{S}}$ qualquer, definimos como $\hat{\mathcal{S}}_{ord}$ o conjunto onde todos os elementos de $\hat{\mathcal{S}}$ estão ordenados de acordo com a ordem lexicográfica $\leq_L$. Por exemplo, o conjunto ordenado $\mathcal{S}_{ord}$, referente às *strings* do conjunto $\mathcal{S}$ na figura 4.1, pode ser visto na figura 4.2.

$\mathcal{S}_{ord}$ = { ace, aid, atlas, atom, attenuate, by, bye, car, cod, dog, fit, lid, patent, sun, zoo }

Figura 4.2: Conjunto $\mathcal{S}_{ord}$

Dizemos que a *posição* de uma *string* k em um conjunto ordenado $\hat{\mathcal{S}}_{ord}$ qualquer é i, se $\hat{s}_{i-1} <_L k \leq_L \hat{s}_i$, onde $\{\hat{s}_{i-1}, \hat{s}_i\} \subseteq \hat{\mathcal{S}}_{ord}$. Por exemplo, em $\mathcal{S}_{ord}$, a posição de $k =$ "by" é 6 e a posição de $k =$ "flower" é 12.

Considere que $\hat{\mathcal{S}}$ é o conjunto de *strings* que desejamos indexar. (Como veremos adiante, dependendo do problema a ser resolvido, $\hat{\mathcal{S}}$ pode ser ou $\mathcal{S}$ ou $\mathcal{S}'$.) Com relação a um nó N qualquer da *String B-tree*, construída a partir de $\hat{\mathcal{S}}$, dizemos que:

- N está relacionado a um conjunto de *strings* $\hat{\mathcal{S}}_N$, através do qual o conjunto ordenado $\hat{\mathcal{S}}_{ord_N}$ é obtido e armazenado em N;
- N possui uma árvore patrícia PT_N, referente às *strings* em $\hat{\mathcal{S}}_N$;
- $b \leq |\hat{\mathcal{S}}_{ord_N}| \leq 2b$, onde $b = \Theta(B)$ é um inteiro par escolhido de maneira a fazer com que N caiba em uma única página de disco;
- $|\hat{\mathcal{S}}_{ord_N}| < b$ somente se N for o nó raiz;
- L_N é a *string* mais à esquerda de $\hat{\mathcal{S}}_{ord_N}$;
- R_N é a *string* mais à direita de $\hat{\mathcal{S}}_{ord_N}$.

Caso N seja um nó interno, N tem $n(N)$ filhos C_1, C_2, ..., $C_{n(N)}$ e seu conjunto ordenado $\hat{\mathcal{S}}_{ord_N} = \{L_{C_1}, R_{C_1}, ..., L_{C_{n(N)}}, R_{C_{n(N)}}\}$ é obtido copiando-se a *string* mais à esquerda e a mais à direita de cada um dos seus filhos. Logo, se $\hat{\mathcal{S}}_{ord_N}$ tem entre b e $2b$ elementos, então N tem entre $\frac{b}{2}$ e b filhos, onde $b > 2$ (figura 4.3).

Já as folhas de uma *String B-tree* formam uma lista duplamente encadeada.

Caso estejamos nos referindo à *String B-tree* do problema *Busca-Prefixo*, temos que $\hat{\mathcal{S}} = \mathcal{S}$ e as *strings* de $\mathcal{S}_{ord}$ são distribuídas entre as folhas do índice de tal sorte que, percorrendo-as da esquerda para a direita, obtemos $\mathcal{S}_{ord}$ novamente. Assim, se N é uma folha, $\mathcal{S}_{ord_N} \subset \mathcal{S}_{ord}$.

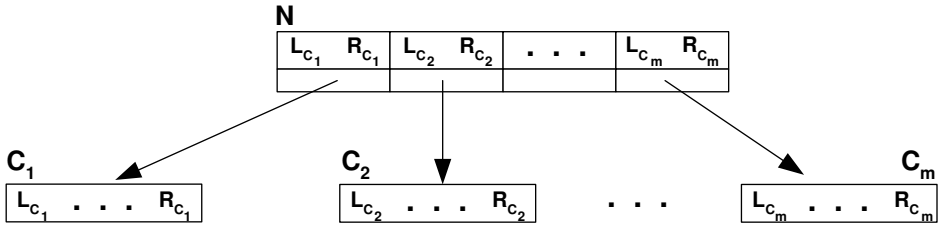


Figura 4.3: Layout de um nó interno N da *String B-tree*, onde $n(N) = m$

A *String B-tree* para o conjunto $\mathcal{S}$ da figura 4.1, pode ser vista na figura 4.4. Aqui, como já foi dito antes, as *strings* são sempre representadas por seus respectivos ponteiros lógicos. As árvores patrícias dos nós foram omitidas, por simplicidade.

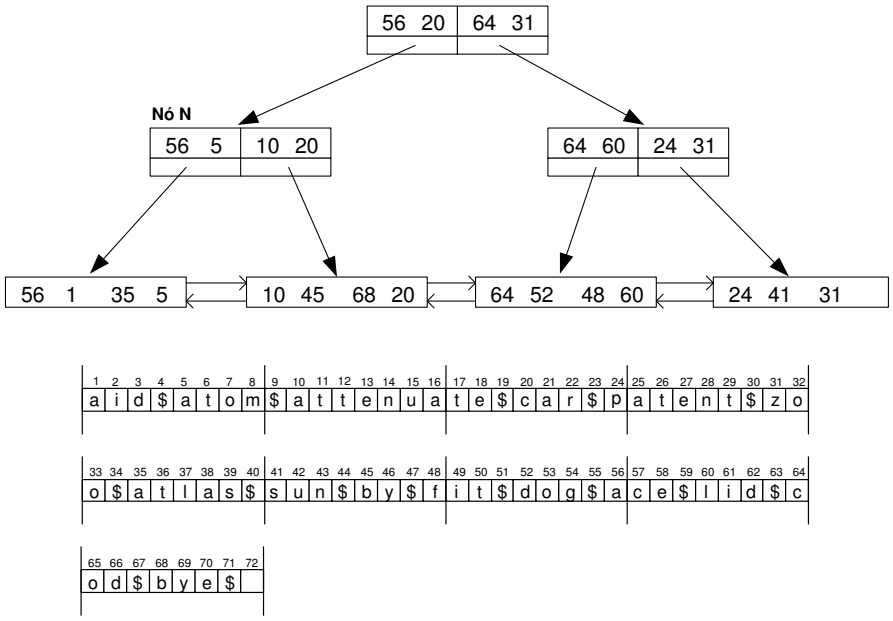


Figura 4.4: *String B-tree* para o conjunto $\mathcal{S}$

Caso o problema seja o de *Busca-Substring*, temos que $\hat{\mathcal{S}} = \mathcal{S}'$ e as *strings* de $\mathcal{S}'_{ord}$ é que são distribuídas entre as folhas da *String B-tree*. Percorrendo-as da esquerda para a direita, obtemos $\mathcal{S}'_{ord}$ novamente. Logo, se N é uma folha, $\mathcal{S}'_{ord_N} \subset \mathcal{S}'_{ord}$.

A *String B-tree* para o conjunto $\mathcal{S}'$, obtido através de $\mathcal{S} = \{$aid, atom, attenuate, car, patent, zoo, atlas$\}$, pode ser vista na figura 4.5. Aqui, mais uma vez, as *strings* são representadas por seus respectivos ponteiros lógicos. Os ponteiros entre folhas e as árvores patrícias também foram omitidos.

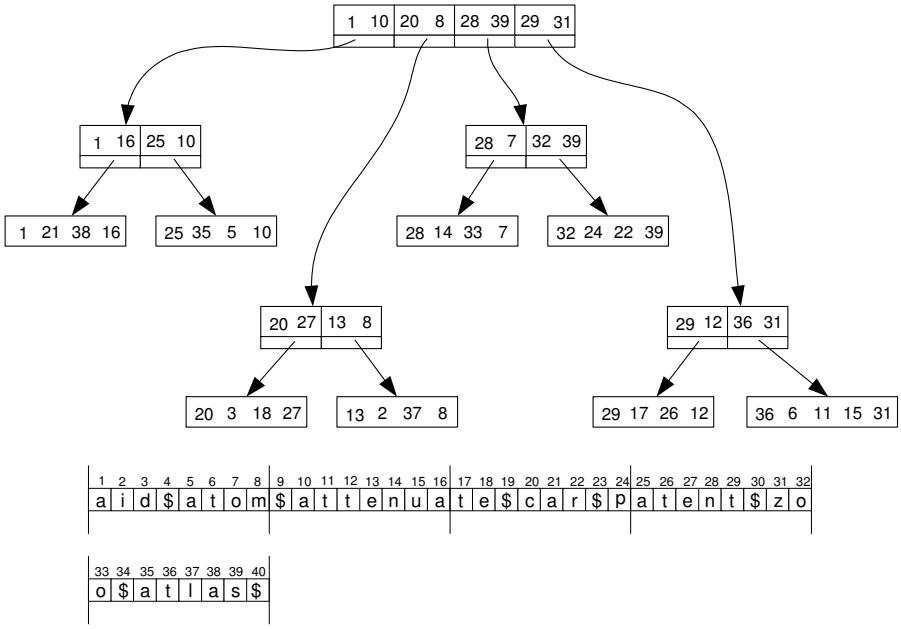


Figura 4.5: *String B-tree* para o conjunto $\mathcal{S}'$

4.2 Consultas e Atualizações em uma *String B-tree*

Seja $\hat{\mathcal{S}}$ o conjunto de chaves de busca da *String B-tree*, que pode ser ou $\mathcal{S}$ ou $\mathcal{S}'$, dependendo da situação. O algoritmo de busca ***SB-PrefixSearch(k)***, onde k é uma *string*, consiste em encontrar a posição i de k em $\hat{\mathcal{S}}_{ord}$ e em seguida retornar, como resultado da busca, toda *string* de $\hat{\mathcal{S}}_{ord}$ cuja posição é maior ou igual a i e que possui k como prefixo.

Para encontrar a posição de k em $\hat{\mathcal{S}}_{ord}$ nos utilizamos do procedimento ***SB-Search(k)***. Este procedimento retorna a folha L onde k está (ou onde k deveria estar, caso $k \notin \hat{\mathcal{S}}_{ord}$) e a sua posição no conjunto $\hat{\mathcal{S}}_{ord_L}$. Como as folhas da *String B-tree* formam uma lista duplamente encadeada, obtemos também a posição de k em $\hat{\mathcal{S}}_{ord}$.

A cada passo do algoritmo ***SB-Search(k)***, um nó da *String B-tree* é visitado, a começar pelo nó-raiz. A cada nó N visitado, determinamos a posição de k em $\hat{\mathcal{S}}_{ord_N}$. Caso o nó N seja uma folha, o algoritmo ***SB-Search()*** acaba aqui e N é a folha L retornada como resultado do algoritmo, juntamente com a posição de k em $\hat{\mathcal{S}}_{ord_L}$. Caso contrário, baseado na posição de k em $\hat{\mathcal{S}}_{ord_N}$, escolhemos o próximo nó, filho de N, a ser visitado. Seja $\hat{\mathcal{S}}_{ord_N} = \{\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n\}$. Temos três casos a considerar:

- Se $k <_L \hat{s}_1$, então N é o nó raiz. Retornamos L, a folha mais à esquerda da *String B-tree*, e a posição de k em $\hat{\mathcal{S}}_{ord_L}$ é 1;

- Se $k >_L \hat{s}_n$, então N é o nó raiz. Retornamos L, a folha mais à direita da *String B-tree*, e a posição de k em $\hat{\mathcal{S}}_{ord_L}$ é $|\hat{\mathcal{S}}_{ord_L}| + 1$;
- $\hat{s}_1 \leq_L k \leq_L \hat{s}_n$. Sejam $\hat{s}_{i-1}$ e $\hat{s}_i$ os dois elementos de $\hat{\mathcal{S}}_{ord_N}$ tal que $\hat{s}_{i-1} <_L k \leq_L \hat{s}_i$. Logo:
  - Se $\hat{s}_{i-1}$ e $\hat{s}_i$ são *strings* que pertencem a dois filhos distintos C_1 e C_2 de N, ou seja, se $\hat{s}_{i-1} = R_{C_1}$ e $\hat{s}_i = L_{C_2}$, então temos que $\hat{s}_{i-1}$ e $\hat{s}_i$ também são adjacentes em $\hat{\mathcal{S}}_{ord}$, devido à estrutura da *String B-tree*. Isto determina a posição de k em $\hat{\mathcal{S}}_{ord}$. Logo, retornamos a folha L, a mais à esquerda das folhas descendentes de C_2, e a posição de k em $\hat{\mathcal{S}}_{ord_L}$ é 1. A figura 4.6 ilustra esta situação, utilizando a *String B-tree* da figura 4.4. Os nós destacados são aqueles visitados pelo algoritmo.
  - Se ambos $\hat{s}_{i-1}$ e $\hat{s}_i$ pertencem ao mesmo filho C de N, então o próximo nó a ser visitado é C e o algoritmo ***SB-Search()*** continua. A figura 4.7 ilustra esta situação, utilizando a *String B-tree* da figura 4.4. Os nós destacados são aqueles visitados pelo algoritmo.

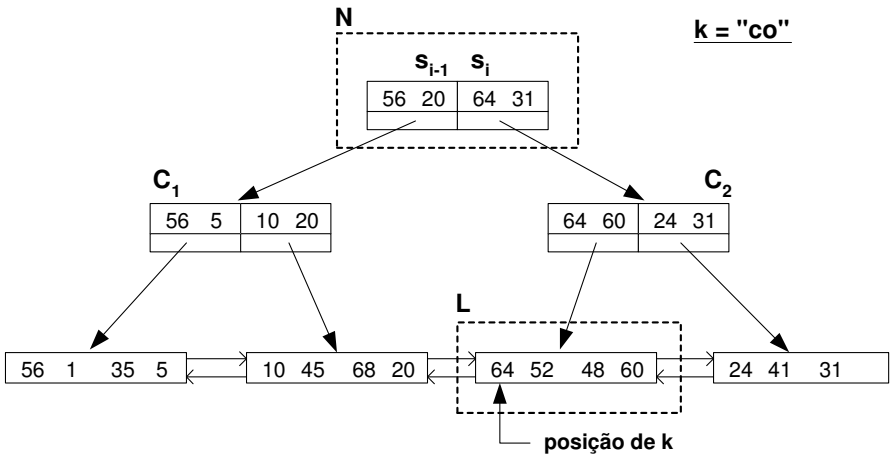


Figura 4.6: Busca em uma *String B-tree*

Note que, a cada nó N visitado pelo algoritmo ***SB-Search()***, devemos determinar a posição de k em $\hat{\mathcal{S}}_{ord_N}$. Caso utilizássemos um algoritmo de busca binária para tal tarefa, como ocorre nas árvores B+ convencionais, teríamos que examinar $O(\log_2 |\hat{\mathcal{S}}_{ord_N}|)$ $= O(\log_2 2b) = O(\log_2 B)$ *strings*. No entanto, as *strings* não estão armazenadas na *String B-tree* e, para recuperar cada *string* necessária à busca e compará-la com k, seriam necessários $O(\frac{|k|}{B})$ acessos a disco. Logo, para determinar a posição de k em $\hat{\mathcal{S}}_{ord_N}$

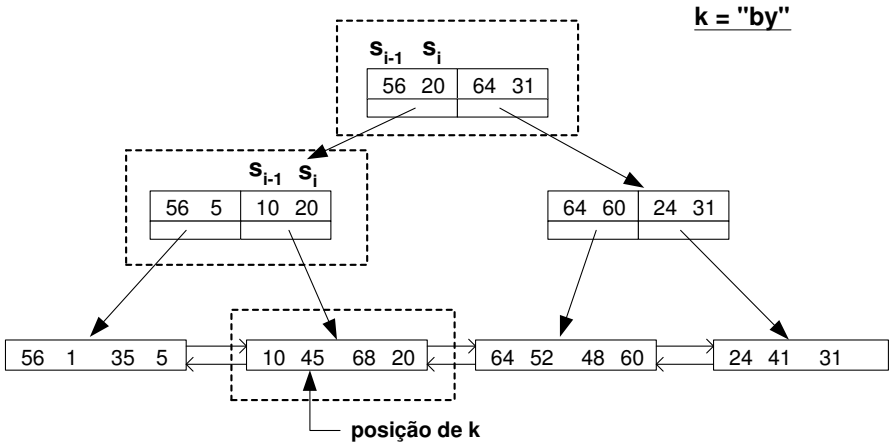


Figura 4.7: Busca em uma *String B-tree*

precisaríamos de um total de $O(\frac{|k|}{B}\log_2 B)$ acessos a disco, no pior caso. Com o objetivo de minimizar este número de acessos, uma estrutura de dados auxiliar é armazenada em cada nó da *String B-tree*. Esta estrutura é uma *árvore patrícia* [31].

Se N é um nó visitado pelo algoritmo ***SB-Search()***, a posição de k em $\hat{\mathcal{S}}_{ord_N}$ é determinada através do algoritmo ***PT-Search()***, que é uma busca na árvore patrícia PT_N. Esta busca requer apenas 1 acesso a disco, como veremos na seção 4.4.

Para inserir (remover) um elemento k na (da) *String B-tree*, primeiramente o algoritmo ***SB-Search(k)*** é executado, retornando uma folha L e posição i de k em $\hat{\mathcal{S}}_{ord_L}$. Seja P o nó pai de L na *String B-tree*.

O algoritmo de inserção de k na *String B-tree*, denominado ***SB-Insert(k,L,i)***, realiza a seguinte seqüência de passos:

- k é inserida na posição i em $\hat{\mathcal{S}}_{ord_L}$ (e conseqüentemente, em PT_L);
- Se $|\hat{\mathcal{S}}_{ord_L}| \leq 2b$, o algoritmo de inserção acaba aqui.
- Caso contrário, o nó L sofre um *split*:
  - Um novo nó L' (à direita de L) é criado e os últimos b elementos em $\hat{\mathcal{S}}_{ord_L}$ são transferidos para $\hat{\mathcal{S}}_{ord_{L'}}$;
  - A árvore patrícia PT_L é dividida em duas árvores menores PT_{L_1} e PT_{L_2}, tais que PT_L ($PT_{L'}$) é substituída por PT_{L_1} (PT_{L_2});
- O nó L' é inserido no conjunto de filhos de P;
- As *strings* R_L e $L_{L'}$ são inseridas em $\hat{\mathcal{S}}_{ord_P}$ e em PT_P;
- Caso $|\hat{\mathcal{S}}_{ord_P}| \leq 2b$, o algoritmo de inserção acaba aqui.

- Caso contrário, o nó P sofre um *split*.

Assim, continua recursivamente, até que seja encontrado um ancestral P' de L que possa acomodar inserções ou até que uma nova raiz seja criada.

O algoritmo de remoção de k da *String B-tree*, denominado ***SB-Remove(k,L,i)***, realiza a seguinte seqüência de passos:

- k é removida da posição i de $\hat{\mathcal{S}}_{ord_L}$ (e conseqüentemente, de PT_L);
- Se $|\hat{\mathcal{S}}_{ord_L}| \geq b$, o algoritmo de remoção acaba aqui.
- Caso contrário, seja L' um nó irmão de L. Sem perda de generalidade, assumimos que L' é o irmão à direita de L. Transferimos as duas *strings* mais à esquerda de $\hat{\mathcal{S}}_{ord_{L'}}$, $\hat{s}_i$ e $\hat{s}_{i+1}$, para $\hat{\mathcal{S}}_{ord_L}$;
- Inserimos as *strings* $\hat{s}_i$ e $\hat{s}_{i+1}$ em PT_L e as removemos de $PT_{L'}$;
- Se $|\hat{\mathcal{S}}_{ord_{L'}}| \geq b$, atualizamos as *strings* R_L e $L_{L'}$ em $\hat{\mathcal{S}}_{ord_P}$ e em PT_P. O algoritmo de remoção acaba aqui.
- Caso contrário, L e L' sofrem um *merge*:
  - Todos os elementos de $\hat{\mathcal{S}}_{ord_{L'}}$ são transferidos para $\hat{\mathcal{S}}_{ord_L}$.
  - Há uma fusão entre as árvores patrícias PT_L e $PT_{L'}$, de modo a gerar uma terceira árvore $PT_{LL'}$, tal que PT_L é substituída por $PT_{LL'}$;
- O nó L' é removido do conjunto de filhos de P;
- As *strings* R_L e $L_{L'}$ são removidas de $\hat{\mathcal{S}}_{ord_P}$ e de PT_P;
- Caso $|\hat{\mathcal{S}}_{ord_P}| \geq b$, o algoritmo de remoção acaba aqui.
- Caso contrário, esta operação de remoção se repete.

Assim, continua recursivamente até que seja encontrado um ancestral P' de L que possa sofrer remoções ou até que a raiz seja removida.

4.3 Árvores Patrícias

Árvores patrícias [31] são estruturas de dados utilizadas para buscas em um conjunto $\hat{\mathcal{S}}$, onde $\hat{\mathcal{S}}$ é um conjunto qualquer de *strings*. Denominamos por $PT_{\hat{\mathcal{S}}}$ a árvore patrícia correspondente a $\hat{\mathcal{S}}$. (A notação PT_N definida na seção 4.1 nada mais é que uma abreviação para $PT_{\mathcal{S}_N}$ ou $PT_{\mathcal{S}'_N}$.)

Cada folha de uma árvore patrícia corresponde a uma *string* de $\hat{\mathcal{S}}$, de tal sorte que, visitando-as da esquerda para a direita, obtemos $\hat{\mathcal{S}}_{ord}$. Porém, a principal característica

de uma árvore patrícia é que a mesma armazena as *strings* de $\hat{\mathcal{S}}$ de forma extremamente compacta em seus nós e arestas.

No contexto das árvores patrícias de uma *String B-tree*, as *strings* também são representadas por seus respectivos ponteiros lógicos. Portanto, mais uma vez, quando mencionarmos "*string*" estaremos nos referindo, na verdade, ao seu ponteiro lógico.

De uma forma simplificada, obtemos a árvore patrícia $PT_{\hat{\mathcal{S}}}$ em três passos:

- Construímos uma árvore compactada $CT_{\hat{\mathcal{S}}}$ [40], utilizando as *strings* de $\hat{\mathcal{S}}$;
- Rotulamos cada nó w de $CT_{\hat{\mathcal{S}}}$ com o inteiro resultante da soma entre (i) o rótulo do nó pai de w e (ii) o tamanho da *substring* presente na aresta cujo destino é w;
- Substituímos cada *substring* em alguma aresta de $CT_{\hat{\mathcal{S}}}$ pelo seu primeiro caracter.

Um exemplo de árvore patrícia para o conjunto $\hat{\mathcal{S}} = \{$ bcbcbba, abaaba, abac, bcbcabb, abaabb, bcbcbbba, bcbcaba $\}$ pode ser visto na figura 4.8. Embora nesta figura estejamos mostrando as *strings* correspondentes a cada uma das folhas da árvore patrícia (com o objetivo de deixar a figura mais clara), na verdade, o que está armazenado em cada folha é um ponteiros lógico, e não a *string* propriamente dita.

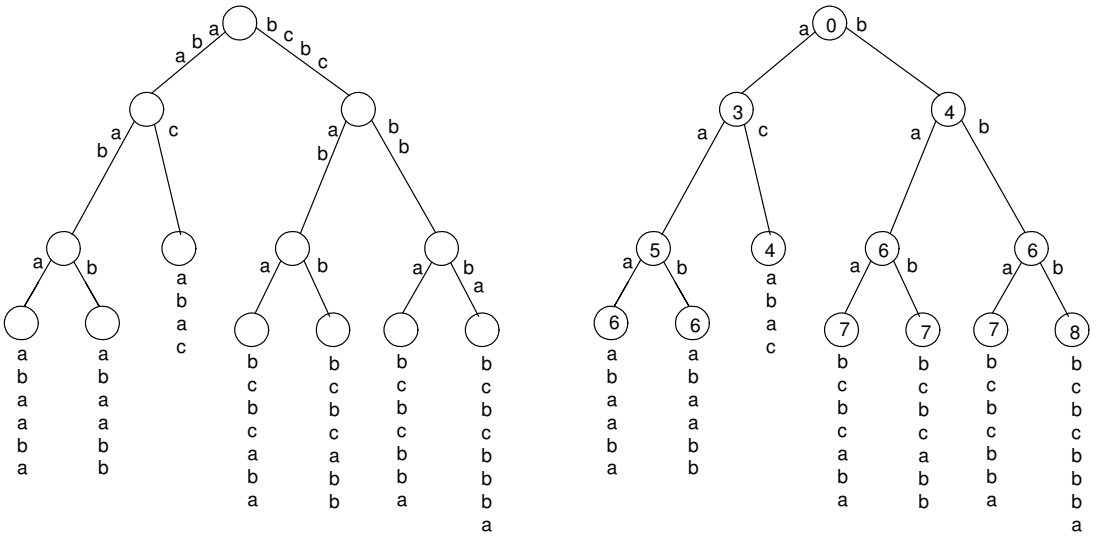


Figura 4.8: Árvore compactada (à esquerda) e árvore patrícia (à direita)

4.4 Buscas e Operações Dinâmicas em uma Árvore Patrícia

O algoritmo ***PT-Search(k,$\hat{S}_{ord}$)*** corresponde a uma busca pela *string* k na árvore patrícia $PT_{\hat{\mathcal{S}}}$. Este algoritmo tem duas etapas:

- Na *primeira etapa*, descemos na árvore $PT_{\hat{\mathcal{S}}}$ até encontrar a folha l, que não necessariamente identifica a posição de k em $\hat{\mathcal{S}}_{ord}$. A descida na árvore inicia-se no nó raiz de $PT_{\hat{\mathcal{S}}}$ e continua através da comparação de alguns dos caracteres de k com os caracteres presentes nas arestas de $PT_{\hat{\mathcal{S}}}$, até que encontremos a folha l ou até que não seja mais possível descer na árvore. Neste último caso, l é uma das folhas descendentes do último nó visitado.

- Na *segunda etapa*, fazemos um acesso a disco para recuperar a *string* correspondente à folha l e a comparamos com k para determinar o maior prefixo em comum $\hat{p}$ entre estas duas *strings*. Determinamos o nó ancestral, chamado de *nó alvo*, mais distante de l cujo rótulo é um inteiro maior ou igual a $|\hat{p}|$. Em seguida, encontramos a folha correspondente a k usando o $(|\hat{p}|+1)$-ésimo caracter de k e da *string* de l. Dizemos que a folha correspondente a k em $PT_{\hat{\mathcal{S}}}$ é aquela que armazena a *string* $\hat{s}_i$, tal que $\hat{s}_{i-1} <_L k \leq_L \hat{s}_i$ $(\{\hat{s}_{i-1}, \hat{s}_i\} \subseteq \hat{\mathcal{S}}_{ord})$.

A figura 4.9 ilustra a operação de busca pela *string* $k =$ "bcbabcba" na árvore patrícia da figura 4.8.

A primeira árvore da figura 4.9 mostra a *primeira etapa* da busca, onde os caracteres destacados em k são comparados com os caracteres das arestas em negrito. Note que a folha l não corresponde à posição correta de k.

A segunda árvore da figura 4.9 mostra a *segunda etapa* da busca, onde a *string* da folha l é carregada do disco e comparada a k. O maior prefixo em comum $\hat{p}$ entre ambas é "bcb", o que nos leva a escolher como *nó alvo*, o nó ancestral de l cujo rótulo é 4 $(4 \geq |\hat{p}|)$. Como 'a' $<_L$ 'c' – onde 'a' e 'c' são os $(|\hat{p}|+1)$-ésimos caracteres de k e da *string* de l respectivamente –, temos que a posição de k é determinada pela folha mais à esquerda, descendente do *nó alvo*.

Uma operação de *inserção* de uma *string* k em uma árvore patrícia $PT_{\hat{\mathcal{S}}}$ é feita através do algoritmo ***PT-Insert(k,$PT_{\hat{S}}$)***. Este procedimento consiste em inserir uma nova folha l em $PT_{\hat{\mathcal{S}}}$ cuja *string* é k. Para tanto, ocorre uma busca, utilizando-se o algoritmo ***PT-Search(k,$\hat{S}_{ord}$)***, para determinar qual é a folha l' que estaria imediatamente à direita de l. Em seguida, inserimos l à esquerda de l'.

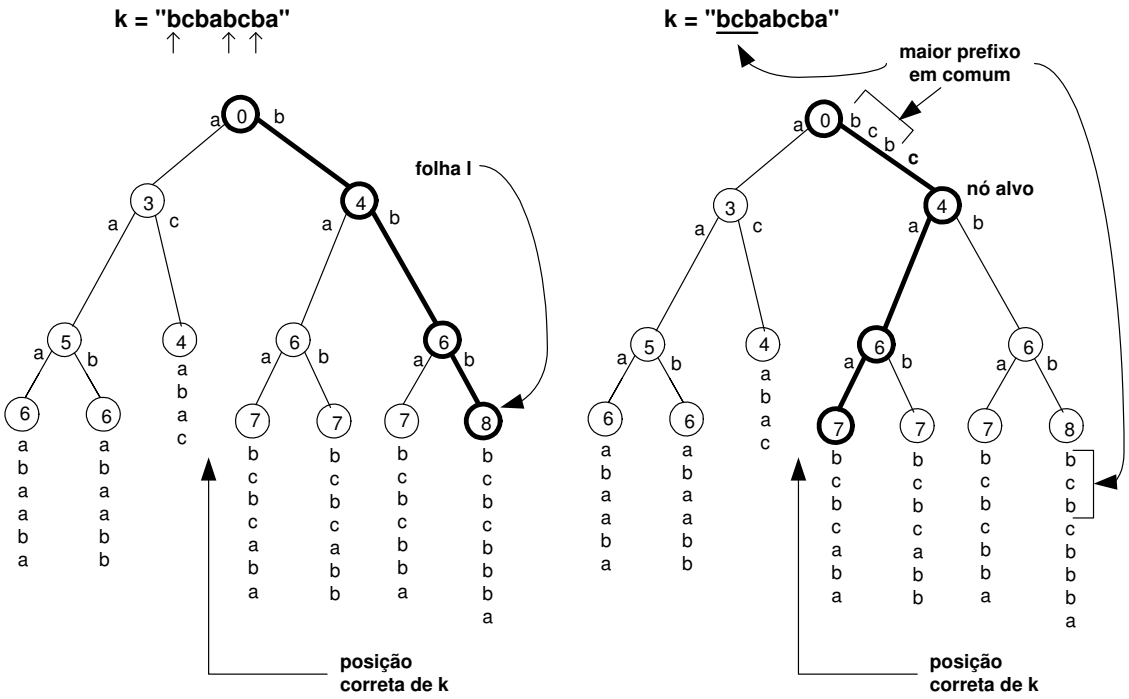


Figura 4.9: As duas etapas da busca em uma árvore patrícia

Uma operação de *remoção* de uma *string* k de uma árvore patrícia $PT_{\hat{\mathcal{S}}}$ é feita através do algoritmo ***PT-Remove(k,PT$_{\hat{S}}$)***. Este procedimento consiste em remover a folha l em $PT_{\hat{\mathcal{S}}}$ cuja *string* é k. Para tanto, ocorre uma busca, utilizando-se o algoritmo ***PT-Search(k,$\hat{S}_{ord}$)***, com o objetivo de determinar qual é a folha l. Em seguida, se a *string* de l é realmente k, removemos l e também o seu nó pai, caso este fique com apenas um filho após a remoção de l.

Uma operação de *split* em uma árvore patrícia $PT_{\hat{\mathcal{S}}}$, onde $\hat{\mathcal{S}}_{ord} = \{\hat{s}_1, \ldots, \hat{s}_j, \hat{s}_{j+1}, \ldots, \hat{s}_q\}$, consiste em dividir $PT_{\hat{\mathcal{S}}}$ em duas árvores menores $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$, tais que $\hat{\mathcal{S}}_{1_{ord}} = \{\hat{s}_1, \ldots, \hat{s}_j\}$ e $\hat{\mathcal{S}}_{2_{ord}} = \{\hat{s}_{j+1}, \ldots, \hat{s}_q\}$. O algoritmo ***PT-Split(k,PT$_{\hat{S}}$)*** é responsável por esta divisão, onde k é a *string* $\hat{s}_j$. Como já sabemos que $k \in \hat{\mathcal{S}}_{ord}$, para encontrar a folha que corresponde a k em $PT_{\hat{\mathcal{S}}}$, apenas a *primeira etapa* da busca ***PT-Search(k,$\hat{S}_{ord}$)*** é suficiente e, portanto, nenhum acesso a disco é feito.

Uma operação de *merge* entre duas árvores patrícias $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$ – tais que $\hat{\mathcal{S}}_{1_{ord}} = \{\hat{s}_1, \ldots, \hat{s}_j\}$, $\hat{\mathcal{S}}_{2_{ord}} = \{\hat{s}_m, \ldots, \hat{s}_q\}$ e $\hat{s}_j <_L \hat{s}_m$ – consiste na junção de $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$ de forma a gerar uma terceira árvore patrícia $PT_{\hat{\mathcal{S}}}$, onde $\hat{\mathcal{S}}_{ord} = \{\hat{s}_1, \ldots, \hat{s}_j, \hat{s}_m, \ldots, \hat{s}_q\}$. Obtemos $PT_{\hat{\mathcal{S}}}$ através do algoritmo ***PT-Merge(PT$_{\hat{S}_1}$,PT$_{\hat{S}_2}$)***. Neste algoritmo, primeiramente a *string* $\hat{s}_m$ é inserida em $PT_{\mathcal{S}_1}$ (necessitando, para tanto, de um acesso a disco) e, em

seguida, o caminho mais à direita de $PT_{\mathcal{S}_1}$ é fundido com o caminho mais à esquerda de $PT_{\mathcal{S}_2}$, transformando-os em um só. A árvore resultante desta operação é $PT_{\hat{\mathcal{S}}}$.

4.5 Análise de Complexidade

Uma vez que apresentamos os algoritmos de uma *String B-tree* em termos de número de acessos a disco, estamos aptos a fazer um estudo de complexidade sobre os mesmos.

Consideramos que $\mathcal{S}$ ($\mathcal{S}'$) é o conjunto de chaves de busca para a *String B-tree* correspondente ao problema *Busca-Prefixo* (*Busca-Substring*). Além disso, fazemos com que $M = |\mathcal{S}|$ ($M' = |\mathcal{S}'|$). Definimos como $\hat{H}$ a altura, ou número de níveis, de uma *String B-tree*.

4.5.1 Complexidades Relacionadas ao Problema *Busca-Prefixo*

A altura da *String B-tree* construída para este problema é determinada por $\hat{H} = O(\log_{b/2} M)$ [39]. Como $b = \Theta(B)$, temos que $\hat{H} = O(\log_B M)$.

Durante o algoritmo de busca pela *string* k, ocorrem dois tipos de acesso a disco: (i) aqueles com a finalidade de recuperar as páginas de índice e (ii) um acesso por nível da árvore, devido à busca em uma árvore patrícia. Logo, a complexidade de busca é $O(\log_B M + \frac{|k|}{B} \log_B M) = O((1 + \frac{|k|}{B}) \log_B M)$.

No algoritmo de inserção (remoção) de uma *string* k, em primeiro lugar, uma busca é feita para identificar qual é a posição de k em $\mathcal{S}_{ord}$. Em seguida, no pior caso, cada nível da árvore sofrerá um *split* (*merge*). Logo, a complexidade para atualizações na *String B-tree* é o número de acessos necessários à busca, juntamente com o número de acessos necessários para rebalancear a árvore.

Seja N um nó da árvore a sofrer um *split* ou um *merge*. O número de acessos a disco c para atualizar PT_N e $\mathcal{S}_{ord_N}$ é constante, ou seja, independe da quantidade de elementos em $\mathcal{S}_{ord_N}$. Logo, dizemos que a complexidade de rebalanceamento de uma árvore é $O(c \log_B M) = O(\log_B M)$, já que c é uma constante. Portanto, a complexidade de atualização – inserção ou remoção – é $O((\frac{|k|}{B} + 1) \log_B M) + O(\log_B M) = O((\frac{|k|}{B} + 1) \log_B M)$.

4.5.2 Complexidades Relacionadas ao Problema *Busca-Substring*

Analogamente ao problema anterior, a altura da *String B-tree* para este problema é dada por $\hat{H} = O(\log_B M')$. A complexidade da busca, dada uma *string* k, também é bastante semelhante: $O((1 + \frac{|k|}{B}) \log_B M')$.

Embora a complexidade da busca seja satisfatória, o mesmo não ocorre com as operações de atualização. Analisemos o caso da inserção como exemplo. Seja k uma *string* e seja k_i o sufixo de k formado pelos seus últimos i caracteres. Se desejamos indexar k, temos que inserir todos os k_i (tal que $1 \leq i \leq |k|$) nesta *String B-tree*. Para inserir cada sufixo k_i, precisamos de $g_i = O(\log_B M' + \frac{|k_i|}{B} \log_B M') = O(\frac{|k_i|}{B} \log_B M')$ acessos a disco. O custo total de inserção é, por fim, definido pela soma $\sum_{i=1}^{|k|} g_i$. Segundo [25], podemos deduzir que $\sum_{i=1}^{|k|} g_i = O((\frac{|k|}{B} + 1)|k| \log_B M')$, ou seja, o custo de inserção é quadrático em $|k|$. O mesmo cálculo se aplica às remoções.

O problema com as atualizações no problema *Busca-Substring* é que os sufixos são tratados como *strings* arbitrárias, o que faz com que tenhamos que percorrer k $|k|$ vezes. No entanto, o fato de estes sufixos fazerem parte da mesma *string* permite que algumas melhorias sejam feitas à estrutura desta *String B-tree*, de sorte que a complexidade de atualização passa a ser linear em $|k|$. Isto é obtido através da introdução de ponteiros auxiliares à estrutura de índice (vide [39], para maiores detalhes).

Capítulo 5

Indexação no FoX

Este capítulo apresenta todos os tópicos relacionados ao Módulo de Armazenamento de FoX. Primeiramente, apresentamos o *Índice FoX*, suas características e algoritmos. Em seguida, exibimos a estrutura das páginas de índice em disco. Na seção seguinte, o Montador de Resultados, um dos sub-módulos do Módulo de Indexação, é apresentado em termos de suas funcionalidades. Finalmente, exibimos os testes comparativos feitos com o *Índice FoX*.

5.1 O Índice FoX

O índice utilizado no FoX é uma *String B-tree* modificada. Nesta seção, explicaremos quais as diferenças entre ambos e definiremos mais formalmente a estrutura do *Índice FoX*, bem como seus algoritmos.

5.1.1 A *String B-tree* e o *Índice FoX*

O conjunto $\mathcal{S}$ de chaves de busca do *Índice FoX* é o conjunto de todas as expressões de caminho simples que iniciam-se na raiz de algum documento D_i, tal que $G_{D_i} \in BD$. Além disso, $\mathcal{S}$ não possui chaves repetidas. Na figura 5.1, temos uma Base de Dados XML – formada pela árvore correspondente a apenas um documento D – e seu respectivo conjunto $\mathcal{S}$, obtido através de uma busca em pré-ordem [25] em G_D. O armazenamento de $\mathcal{S}$ em disco está graficamente representado na figura 5.2.

A definição do conjunto $\mathcal{S}'$ é um pouco diferente da que tínhamos no capítulo 4. No contexto deste capítulo, $\mathcal{S}'$ é tal que: (i) para todo $s \in \mathcal{S}$, apenas os sufixos de s que também são expressões de caminho simples pertencem a $\mathcal{S}'$ e (ii) todo elemento de $\mathcal{S}'$

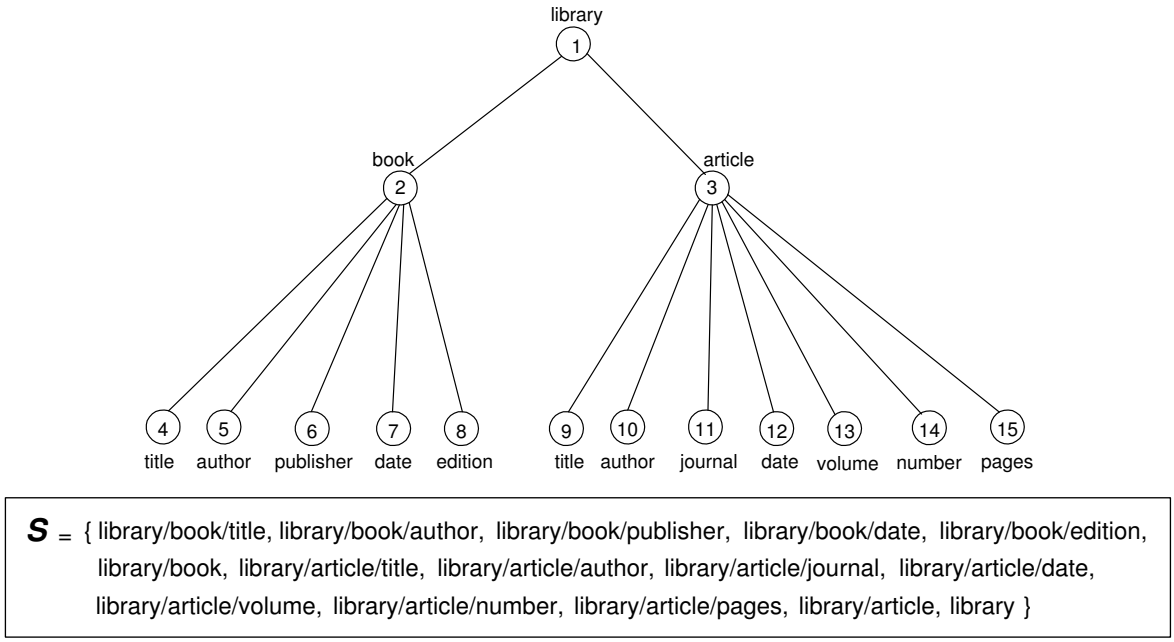


Figura 5.1: Base de Dados XML e seu conjunto $\mathcal{S}$ equivalente

é sufixo de alguma *string* de $\mathcal{S}$. Além disso, com o intuito de simplificar os algoritmos de busca e atualização, principalmente os das árvores patrícias, $\mathcal{S}'$ não possui chaves repetidas, a exemplo do que ocorre no conjunto $\mathcal{S}$. O conjunto $\mathcal{S}'$, correspondente ao conjunto $\mathcal{S}$ da figura 5.1, pode ser visto na figura 5.3. $\mathcal{S}'$ é o conjunto de chaves de busca do *Índice FoX*, ou seja, é a partir das *strings* do conjunto $\mathcal{S}'$, e não de $\mathcal{S}$, que o *Índice FoX* é construído.

Uma outra diferença entre o *Índice FoX* e a *String B-tree* é que, no *Índice FoX*, existe um novo componente que chamamos de *registro de índice*, ou simplesmente *registro*. Um *registro* guarda o identificador de um determinado nó em disco (entre outras informações adicionais), de maneira que, de posse de um registro, seja possível recuperar do disco seu nó correspondente.

Seja L uma folha qualquer do *Índice FoX*. Cada elemento s'_i em $\mathcal{S}'_{ord_L}$ possui um conjunto de registros R_{L_i} correspondente. Cada elemento em R_{L_i} é um registro que, por sua vez, corresponde a um nó resultante do casamento de padrões da expressão de caminho s'_i sobre a Base de Dados. O conjunto R_{L_i}, para toda folha L e todo elemento s'_i de $\mathcal{S}'_{ord_L}$, é armazenado contiguamente em disco.

O *Índice FoX*, correspondente às *strings* do conjunto $\mathcal{S}'$ da figura 5.3, pode ser visto na figura 5.4. Nesta figura, os ponteiros entre folhas, as árvores patrícias e os registros

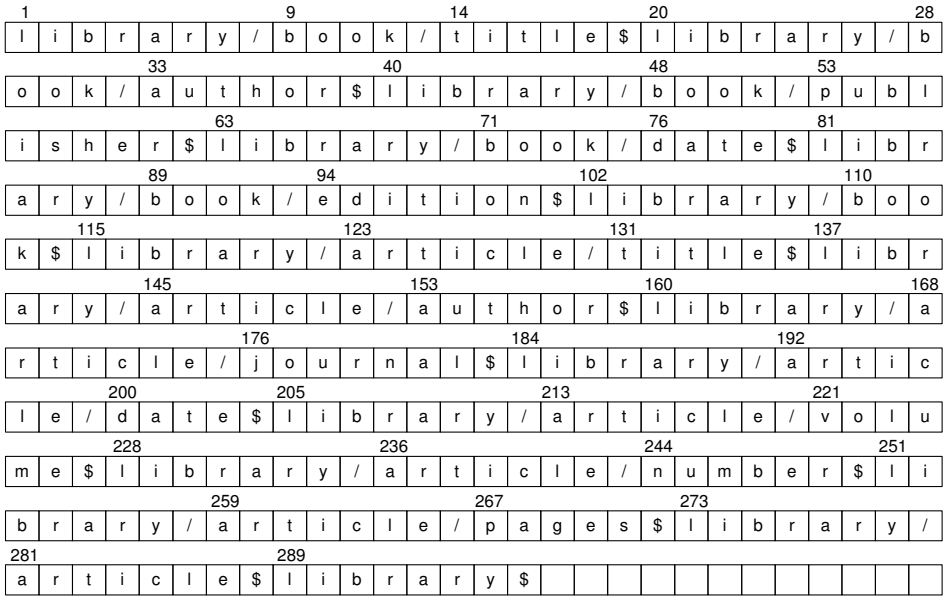


Figura 5.2: Armazenamento de $\mathcal{S}$ em disco

$\mathcal{S}'$ = { library/book/title, book/title, title, library/book/author, book/author, author, library/book/publisher, book/publisher, publisher, library/book/date, book/date, date, library/book/edition, book/edition, edition, library/book, book, library/article/title, article/title, library/article/author, article/author, library/article/journal, article/journal, journal, library/article/date, article/date, library/article/volume, article/volume, volume, library/article/number, article/number, number, library/article/pages, article/pages, pages, library/article, article, library }

Figura 5.3: Conjunto $\mathcal{S}'$

foram omitidos. Os conjuntos de registros correspondentes à folha L, assinalada na figura 5.4, estão graficamente representados na figura 5.5.

Uma busca no *Índice FoX* é um pouco diferente daquela vista na seção 4.1. Dada uma *string* k, procuramos pelos elementos de $\mathcal{S}'_{ord}$ que sejam exatamente iguais a k. Para tanto, descartamos a função ***SB-PrefixSearch()*** e utilizamos somente uma função equivalente à ***SB-Search()***. Como dito anteriormente, o índice não implementa chaves repetidas, logo, se $k \in \mathcal{S}'_{ord}$, então há exatamente uma *string* igual a k em $\mathcal{S}'_{ord}$.

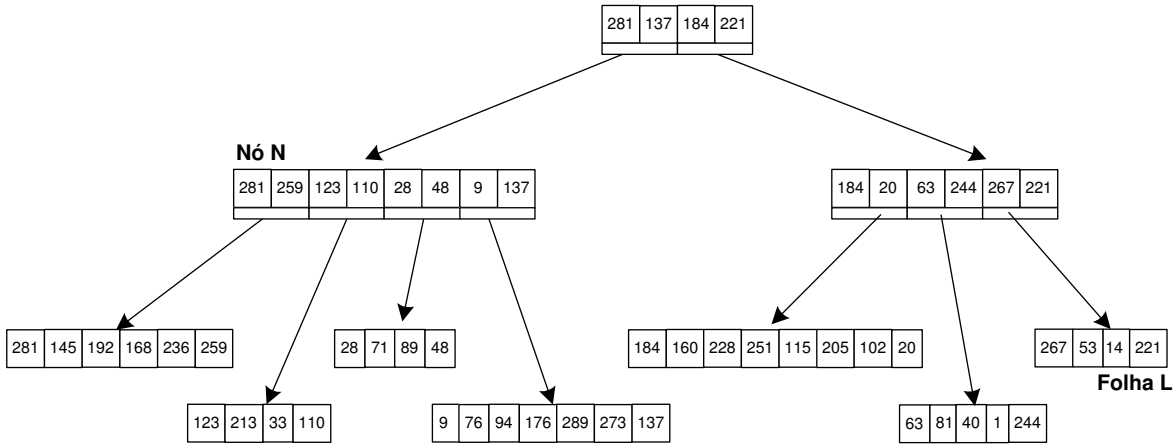


Figura 5.4: *Índice FoX*, onde $b = 4$

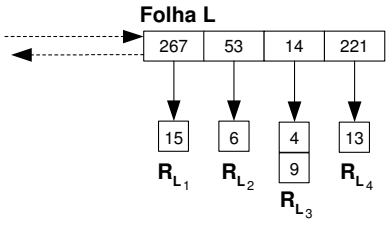


Figura 5.5: Folha L e seus conjuntos de registros

5.1.2 Análise de Complexidade

O *Índice FoX* pode ser entendido como uma variação da *String B-tree* utilizada para solucionar o problema *Busca-Substring*, visto no capítulo 4. As complexidades de atualização na *String B-tree* referente a este problema, tendo como parâmetro uma *string* k, são da ordem de $|k|^2$. No entanto, com a nova definição de $\mathcal{S}'$, algumas mudanças ocorrem no cálculo da complexidade, como veremos a seguir.

Dada uma expressão de caminho A qualquer, definimos como *profundidade* de A, ou d_A, o número de sufixos de A que também representam expressões de caminho. Por exemplo, se A = "library/article/date", temos que $d_A = 3$, pois os sufixos de A que também representam expressões de caminho são "library/article/date", "article/date" e "date". Note que d_A também corresponde ao número de nós que o caminho referente a A possui.

Seja G_D uma árvore, tal que $G_D \in BD$ corresponde a um documento D, gerenciado pelo FoX. Estendendo o conceito de *profundidade*, denotamos por *profundidade* de D,

ou d_D, o caminho mais longo em número de nós da raiz de G_D até alguma de suas folhas. Definimos também como d_{BD}, a *profundidade* de uma Base de Dados BD, onde $d_{BD} = \max\{d_{D_1}, d_{D_2}, \ldots, d_{D_n}\}$ ($G_{D_i} \in BD$, $1 \leq i \leq n$). Seja M' a cardinalidade de $\mathcal{S}'$ e M a cardinalidade de $\mathcal{S}$. Com a definição da constante d_{BD}, obtemos um limite superior para M', ou seja, M' é $O(d_{BD} \cdot M) = O(M)$.

A complexidade de busca do *Índice FoX*, dada uma *string* k, é a mesma do problema *Busca-Substring*, ou seja, é $O((1 + \frac{|k|}{B}) \log_B M')$, onde B é o tamanho da página de disco. Como $M' = O(M)$, temos $O((1 + \frac{|k|}{B}) \log_B M') = O((1 + \frac{|k|}{B}) \log_B M)$. Logo, concluímos que a complexidade de busca no *Índice FoX* é exatamente igual à complexidade de busca calculada para o problema *Busca-Prefixo* (seção 4.5.1).

Se desejamos indexar uma expressão de caminho k através do *Índice FoX*, não precisamos mais inserir no índice todos os seus sufixos. No máximo, ocorrerão d_k inserções. Seja k_i um sufixo da *string* k a ser inserido no índice. O custo de inserção de cada k_i é $g_i = O(\frac{|k_i|}{B} \log_B M')$, como visto na seção 4.5.2. No entanto, temos que $O(\frac{|k_i|}{B} \log_B M')$ $= O(\frac{|k|}{B} \log_B M') = O(\frac{|k|}{B} \log_B M)$ acessos a disco. O custo total de inserção é, portanto, $O(\frac{|k|}{B} d_k \log_B M)$. O mesmo cálculo se aplica, caso queiramos remover uma *string* $k \in \mathcal{S}$ do índice.

Note que as complexidades de atualização nesta variação do problema *Busca-Substring* não são mais quadráticas em $|k|$, como demonstrado na seção 4.5.2. Na verdade, as mesmas são muito melhores que as encontradas para o problema *Busca-Substring*, pois $M' = O(M)$, e apenas um pouco piores que as calculadas para o problema *Busca-Prefixo*, devido ao multiplicador d_k. Mesmo assim, na maioria dos casos, a razão entre d_k e $|k|$ é pequena, o que faz com que as complexidades de atualização do *Índice FoX* se aproximem ainda mais das relacionadas à *Busca-Prefixo*.

Observe ainda que, nos cálculos de complexidades apresentados nesta seção, não levamos em conta os *registros* do *Índice FoX*. Logo, estas complexidades não são definitivas. Voltaremos a examiná-las na seção 5.1.5.

5.1.3 Árvores Patrícias do FoX

5.1.3.1 Definição

Nesta seção, detalhamos a estrutura de dados de uma árvore patrícia. Para tornar suas características mais claras apresentamos, na figura 5.6, a árvore patrícia construída a partir das *strings* de $\hat{\mathcal{S}} = \mathcal{S}'_N$, onde o nó N está assinalado na figura 5.4.

Podemos definir formalmente a árvore patrícia $PT_{\hat{\mathcal{S}}}$ como uma árvore direcionada,

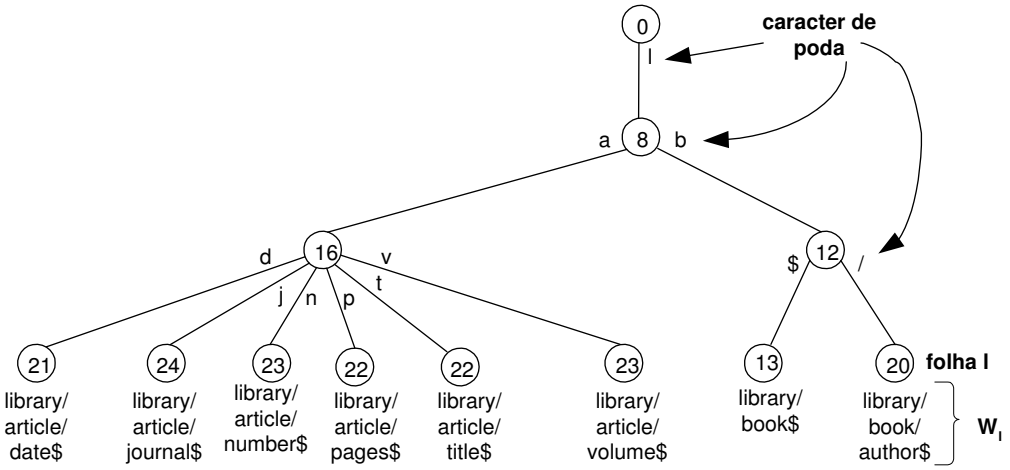


Figura 5.6: Árvore patrícia

onde suas folhas representam as *strings* em $\hat{\mathcal{S}}$ de tal sorte que, visitando-as da esquerda para a direita, obtemos $\hat{\mathcal{S}}_{ord}$. $PT_{\hat{\mathcal{S}}}$ é construída de forma a satisfazer as seguintes condições:

1. Cada aresta é rotulada com um caracter, denominado *caracter de poda* (figura 5.6).
2. Cada nó interno tem pelo menos dois filhos, ou seja, é origem de pelo menos duas arestas, cujos caracteres de poda são diferentes entre si. As arestas são ordenadas de acordo com esses caracteres. A raiz é o único nó que pode possuir apenas um filho.
3. Existe uma folha l distinta associada a cada *string* $\hat{s} \in \hat{\mathcal{S}}$. Denotamos a *string* $\hat{s}$, referente a l, por W_l. Na folha l, o ponteiro lógico correspondente a W_l é que é armazenado, no lugar da própria W_l. No entanto, para facilitar a visualização e o entendimento das operações em uma árvore patrícia, em todas as suas folhas, omitiremos os ponteiros lógicos e exibiremos as suas respectivas *strings*.
4. Toda folha l é rotulada com o inteiro len_l, que representa o tamanho de sua *string* W_l, ou seja, $len_l = |W_l|$.
5. Se o nó n é o mais próximo ancestral em comum de duas folhas l e l', então n é rotulado com o inteiro $len_n = |\hat{p}|$, onde $\hat{p}$ é o maior prefixo em comum entre W_l e $W_{l'}$. O rótulo da raiz é 0. Seja e_i (e_j) a aresta com origem em n que está no caminho entre n e l (l'). O caracter de poda de e_i (e_j) é o $(len_n + 1)$-ésimo caracter de W_l ($W_{l'}$). Por exemplo, na figura 5.6, seja l' a folha imediatamente à esquerda de l. O maior prefixo entre W_l e $W_{l'}$ é $\hat{p}$ = "library/book". Logo o nó pai de l, que é o mais próximo ancestral em comum de l e l', é rotulado com o inteiro $|\hat{p}| = 12$.

Além disso, a aresta e_j recebe o caracter de poda '\$' e a aresta e_i recebe o caracter de poda '/'.

Seja l uma folha qualquer de $PT_{\hat{\mathcal{S}}}$. A folha l armazena o ponteiro lógico correspondente a W_l, o inteiro len_l e um ponteiro $parent_l$ que aponta para seu nó pai. A exemplo do que ocorre numa *String B-tree*, sempre que mencionarmos "a *string* de uma folha l", estaremos nos referindo, na verdade, ao seu respectivo ponteiro lógico.

Representamos uma aresta e de $PT_{\hat{\mathcal{S}}}$ por $e(c, n, n')$, onde e tem origem em n, destino em n' e seu caracter de poda é c. Seja n agora um nó interno qualquer de $PT_{\hat{\mathcal{S}}}$. O nó n armazena um conjunto de arestas $\mathcal{E}_n = \{e_1, e_2, \ldots, e_m\}$ – ordenadas lexicograficamente pelos seus caracteres de poda –, o inteiro len_n e um ponteiro $parent_n$, que aponta para seu nó pai. Denotamos por $\hat{\mathcal{C}}_n$ o conjunto de filhos de n, onde $\hat{\mathcal{C}}_n = \{\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_m\}$ é tal que $\hat{c}_i$ é o nó destino da aresta $e_i \in \mathcal{E}_n$.

Nas seções seguintes, descreveremos os algoritmos de busca, inserção, remoção, *split* e *merge* em uma árvore patrícia.

5.1.3.2 Busca

Seja k uma *string* e $\hat{\mathcal{S}}_{ord}$ um conjunto de *strings* lexicograficamente ordenadas tal que $\hat{\mathcal{S}}_{ord} \subseteq \mathcal{S}'_{ord}$. O procedimento ***PT-Search(k,$\hat{\mathbf{S}}_{ord}$)*** retorna a posição de k em $\hat{\mathcal{S}}_{ord}$, através de uma busca na árvore patrícia $PT_{\hat{\mathcal{S}}}$. O objetivo desta busca é encontrar a folha l. Esta folha armazena uma *string* $\hat{s}_i \in \hat{\mathcal{S}}_{ord}$, tal que $\hat{s}_{i-1} <_L k \leq_L \hat{s}_i$. Uma vez que obtemos $\hat{s}_i$, obtemos também a posição de k em $\hat{\mathcal{S}}_{ord}$, que é i.

A busca em $PT_{\hat{\mathcal{S}}}$ é implementada em duas etapas:

1. Na primeira etapa, descemos em $PT_{\hat{\mathcal{S}}}$ até encontrar uma folha l': Partindo da raiz de $PT_{\hat{\mathcal{S}}}$, comparamos alguns dos caracteres de k com os caracteres de poda presentes nas arestas até que se chegue a uma folha l' ou até que não seja mais possível descer na árvore. No último caso, l' é uma das folhas descendentes do último nó visitado. $W_{l'}$ é, dentre as *strings* de $\hat{\mathcal{S}}_{ord}$, uma das que possuem o maior prefixo em comum com k [39].

2. Na segunda etapa, recuperamos do disco $W_{l'}$ e a comparamos com k para determinar o maior prefixo em comum $\hat{p}$ entre as mesmas. Seja c o $(|\hat{p}| + 1)$-ésimo caracter de k e c' o $(|\hat{p}| + 1)$-ésimo caracter de $W_{l'}$. O *nó alvo* é o nó ancestral n mais distante de l', tal que $len_n \geq |\hat{p}|$. Logo, temos duas situações a considerar:

   (a) $|\hat{p}| = len_n$. Sejam $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_m$ os caracteres de poda das arestas em $\mathcal{E}_n$. Nenhum deles casa com c. Logo:

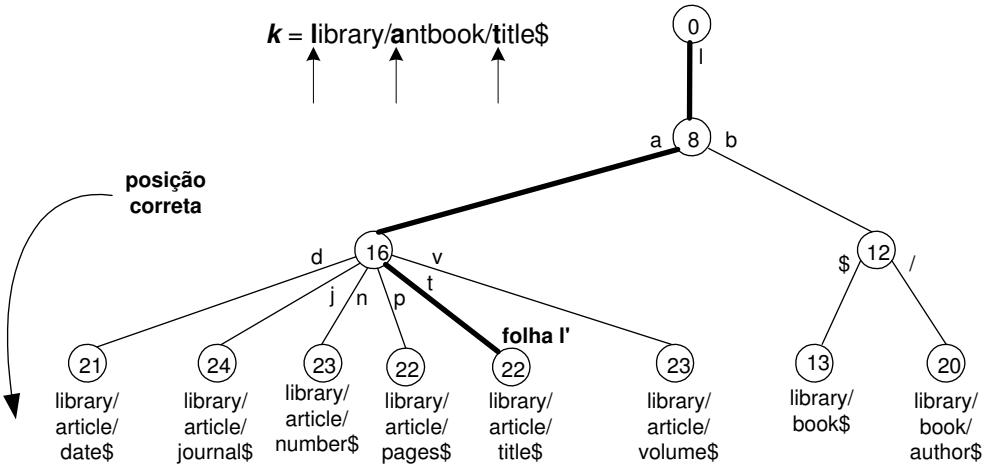


Figura 5.7: Primeira etapa da busca em uma árvore patrícia

i. Se $c <_L \hat{c}_1$, então l é a folha descendente de n mais à esquerda.

ii. Se $c >_L \hat{c}_m$, seja l'' a folha descendente de n mais à direita. l é a folha imediatamente à direita de l''.

iii. Se $\hat{c}_i <_L c <_L \hat{c}_{i+1}$, $0 \leq i < m$. Seja n' o nó destino da aresta cujo caracter de poda é $\hat{c}_{i+1}$. l é a folha descendente de n' mais à esquerda.

(b) $|\hat{p}| < len_n$. Logo:

i. Se $c <_L c'$, l é a folha descendente de n mais à esquerda.

ii. Se $c >_L c'$, seja l'' a folha descendente de n mais à direita. l é a folha imediatamente à direita de l''.

Para ilustrar esta operação, suponha que desejamos buscar a *string* k = "library/antbook/title$" na árvore patrícia da figura 5.6. Na primeira etapa da busca (figura 5.7), os caracteres assinalados em k são comparados aos caracteres de poda das arestas, de maneira a obtermos a folha l'. Claramente, l' não é a folha que procuramos.

Na segunda etapa da busca (figura 5.8), recuperamos $W_{l'}$ do disco e a comparamos com k, com o objetivo de identificar o maior prefixo em comum $\hat{p}$ entre as duas. Neste caso, $\hat{p}$ = "library/a". Com isso, encontramos o nó alvo, cujo rótulo é maior que $|\hat{p}|$. Em seguida, utilizamos os $(|\hat{p}| + 1)$-ésimos caracteres de k ('n') e de $W_{l'}$ ('r') para identificar a folha l que procurávamos no início, descendo nas arestas marcadas da árvore patrícia da figura 5.8.

A folha l corresponde à *string* "library/article/date$", que é a primeira *string* de $\hat{\mathcal{S}}_{ord} = \mathcal{S}'_{ord_N}$. Logo, a posição de k em $\mathcal{S}'_{ord_N}$ é 1.

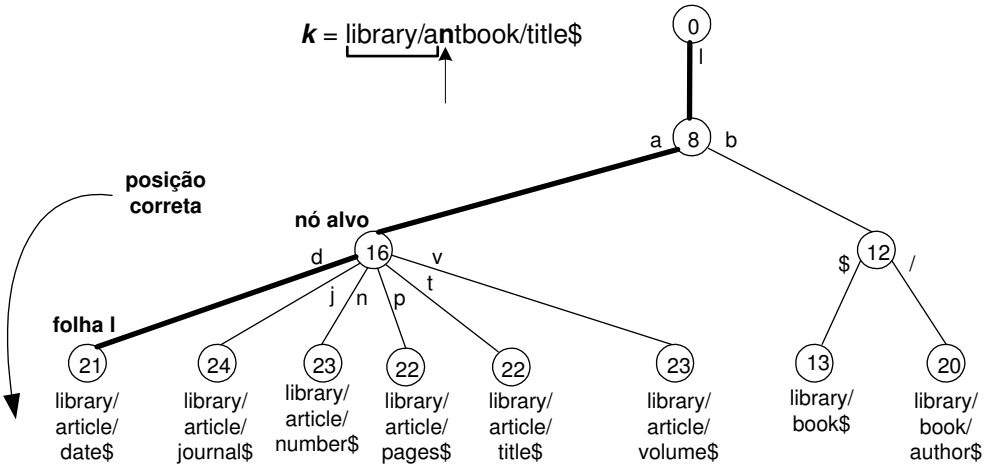


Figura 5.8: Segunda etapa da busca em uma árvore patrícia

5.1.3.3 Inserção

Seja ***PT-Insert(k,*** $PT_{\hat{S}}$***)*** o procedimento de inserção de uma *string* k, em uma árvore patrícia $PT_{\hat{S}}$. Seja l a nova folha a ser inserida em $PT_{\hat{S}}$, onde l corresponde a k.

Em primeiro lugar, o procedimento ***PT-Search(k,*** $\hat{S}_{ord}$***)*** é executado para recuperar a folha l', que estará imediatamente à direita de l, após a inserção. Recuperamos $W_{l'}$ e a comparamos com k para obter o maior prefixo em comum $\hat{p}$ entre as duas. Se $|\hat{p}| = |W_{l'}| = |k|$, então a inserção não pode ser feita pois k já é uma das *strings* de $\hat{\mathcal{S}}_{ord}$. Caso contrário, denotamos por c o $(|\hat{p}|+1)$-ésimo caracter de k e por c' o $(|\hat{p}|+1)$-ésimo caracter de $W_{l'}$.

Em seguida, obtemos o ancestral n mais próximo de l' tal que $len_n \leq |\hat{p}|$. Temos duas situações a considerar:

1. $|\hat{p}| = len_n$. Sejam $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_m$ os caracteres de poda das arestas de $\mathcal{E}_n$.
   (a) Se $c <_L \hat{c}_1$, inserimos $e(c, n, l)$ antes da primeira aresta de $\mathcal{E}_n$.
   (b) Se $c >_L \hat{c}_m$, inserimos $e(c, n, l)$ depois da última aresta de $\mathcal{E}_n$.
   (c) Se $\hat{c}_{i-1} <_L c <_L \hat{c}_i$ $(0 < i \leq m)$, inserimos $e(c, n, l)$ antes da aresta em $\mathcal{E}_n$ cujo caracter de poda é $\hat{c}_i$.
2. $|\hat{p}| > len_n$. Seja c'' o $(len_n + 1)$-ésimo caracter de $W_{l'}$ e seja n' o nó filho de n, correspondente à aresta $e(c'', n, n')$. Primeiro, criamos um novo nó interno n'' tal que $len_{n''} = |\hat{p}|$ e tal que passam a existir duas arestas $e(c, n'', l)$ e $e(c', n'', n')$, ordenadas em $\mathcal{E}_{n''}$, de acordo com a ordem lexicográfica de seus respectivos caracteres de poda. Em seguida, substituímos a aresta $e(c'', n, n')$ por uma nova aresta $e(c'', n, n'')$.

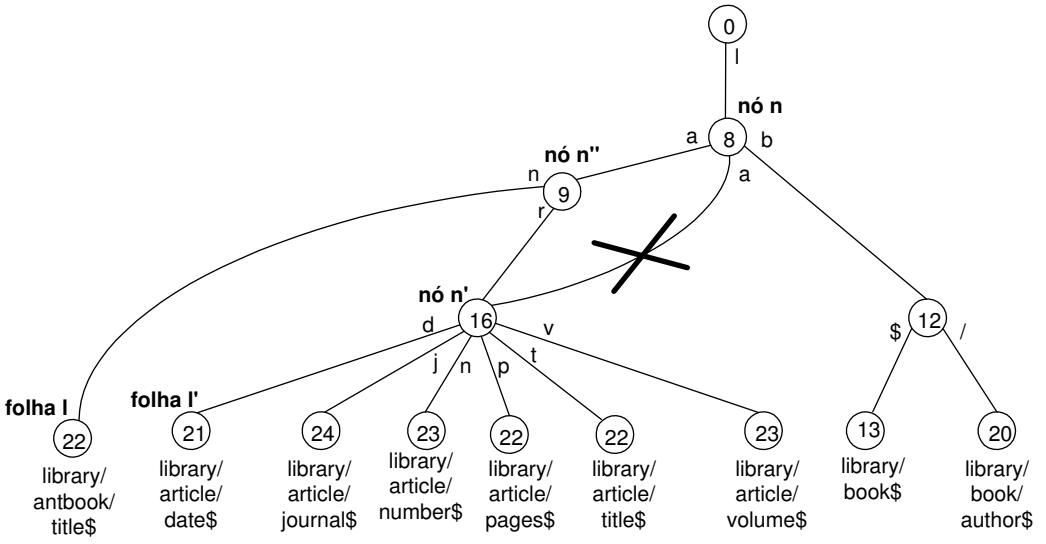


Figura 5.9: Inserção em uma árvore patrícia

Na figura 5.9, inserimos a *string* k = "library/antbook/title$" na árvore patrícia da figura 5.8. O maior prefixo em comum entre k e $W_{l'}$ é "library/a", logo c = 'n' e c' = 'r'. O caracter c'' é o nono caracter de $W_{l'}$, ou seja, c'' = 'a'. Como $|\hat{p}| < 16$, um novo nó n'' é criado e, juntamente com ele, duas arestas $e(\text{'n'}, n'', l)$ e $e(\text{'r'}, n'', n')$. A aresta $e(\text{'a'}, n, n')$ é substituída em $\mathcal{E}_n$ por $e(\text{'a'}, n, n'')$.

5.1.3.4 Remoção

Seja ***PT-Remove(k,$PT_{\hat{S}}$)*** o procedimento de remoção de uma folha l – correspondente a uma *string* $k \in \hat{\mathcal{S}}_{ord}$ – da árvore patrícia $PT_{\hat{\mathcal{S}}}$. Obtemos a folha l através do procedimento ***PT-Search(k,$\hat{S}_{ord}$)***.

Denotamos por n e n' o nó pai e o nó avô de l, respectivamente. Removemos o nó l (e, juntamente com ele, a aresta em $\mathcal{E}_n$ que liga n a l). Se $|\mathcal{E}_n| > 1$ após a remoção de l, o procedimento acaba aqui. Caso contrário, seja n'' o único nó em $\hat{\mathcal{C}}_n$ e seja c o caracter de poda da aresta em $\mathcal{E}_{n'}$ que liga n' a n. Caso n não seja a raiz de $PT_{\hat{\mathcal{S}}}$, duas operações ocorrem: (i) removemos n de $PT_{\hat{\mathcal{S}}}$ (e, juntamente com ele, a aresta em $\mathcal{E}_n$ que liga n a n'') e (ii) substituímos a aresta $e(c, n', n)$ por $e(c, n', n'')$ em $\mathcal{E}_{n'}$.

Na figura 5.10, vemos o procedimento de remoção da *string* k = "library/antbook/title$" da árvore patrícia da figura 5.9. A folha a ser removida é l, assinalada na figura. No entanto, o pai de l (n) deverá ser removido também, pois o mesmo não é raiz e só lhe restou um filho (n''). As arestas $e(\text{'n'}, n, l)$ e $e(\text{'r'}, n, n'')$ também são

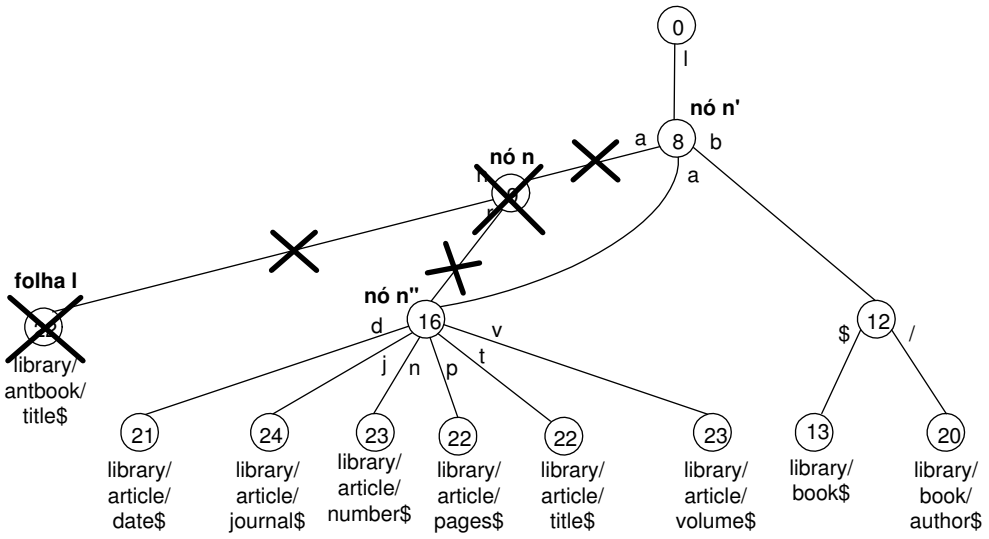


Figura 5.10: Remoção em uma árvore patrícia

removidas, juntamente com l e n. A aresta $e(\text{'a'}, n', n)$ é substituída por $e(\text{'a'}, n', n'')$ em $\mathcal{E}_{n'}$.

Note que após a remoção da folha l, correspondente a k, a árvore patrícia da figura 5.10 voltou a ser a mesma da figura 5.8.

5.1.3.5 *Split*

Seja ***PT-Split(k,PT$_{\hat{S}}$)*** o procedimento de divisão de uma árvore patrícia $PT_{\hat{\mathcal{S}}}$ em duas árvores menores $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$, de modo que em $\hat{\mathcal{S}}_{1_{ord}} \subset \mathcal{S}'_{ord}$ estejam todas as *strings* lexicograficamente menores ou iguais a k e em $\hat{\mathcal{S}}_{2_{ord}} \subset \mathcal{S}'_{ord}$ estejam todas as *strings* lexicograficamente maiores que k.

O primeiro passo deste procedimento é fazer uma cópia de $PT_{\hat{\mathcal{S}}}$, que chamaremos de $PT'_{\hat{\mathcal{S}}}$. Supomos que a *string* $k \in \hat{\mathcal{S}}$, logo apenas a primeira etapa do procedimento de busca é suficiente para encontrarmos a folha l (l') referente a k em $PT_{\hat{\mathcal{S}}}$ ($PT'_{\hat{\mathcal{S}}}$).

Em seguida, seja o caminho $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ ($\mathcal{P}' = \{p'_1, p'_2, \cdots, p'_n\}$), onde p_{i+1} (p'_{i+1}) é pai de p_i (p'_i), p_1 (p'_1) é l (l') e p_n (p'_n) é raiz de $PT_{\hat{\mathcal{S}}}$ ($PT'_{\hat{\mathcal{S}}}$).

Partindo de l (l') visitamos todos os nós do caminho $\mathcal{P}$ ($\mathcal{P}'$). Se p_i (p'_i) é o nó visitado no momento, removemos todos os seus irmãos à direita (esquerda). Se $\hat{\mathcal{C}}_{p_{i+1}} = \{p_i\}$ ($\hat{\mathcal{C}}_{p'_{i+1}} = \{p'_i\}$) e p_{i+1} (p'_{i+1}) não for a raiz de $PT_{\hat{\mathcal{S}}}$ ($PT'_{\hat{\mathcal{S}}}$), faremos a mesma seqüência de passos que ocorre nas remoções:

1. Removemos o nó p_{i+1} (p'_{i+1}) e a aresta em $\mathcal{E}_{p_{i+1}}$ ($\mathcal{E}_{p'_{i+1}}$) que liga p_{i+1} (p'_{i+1}) a p_i (p'_i).

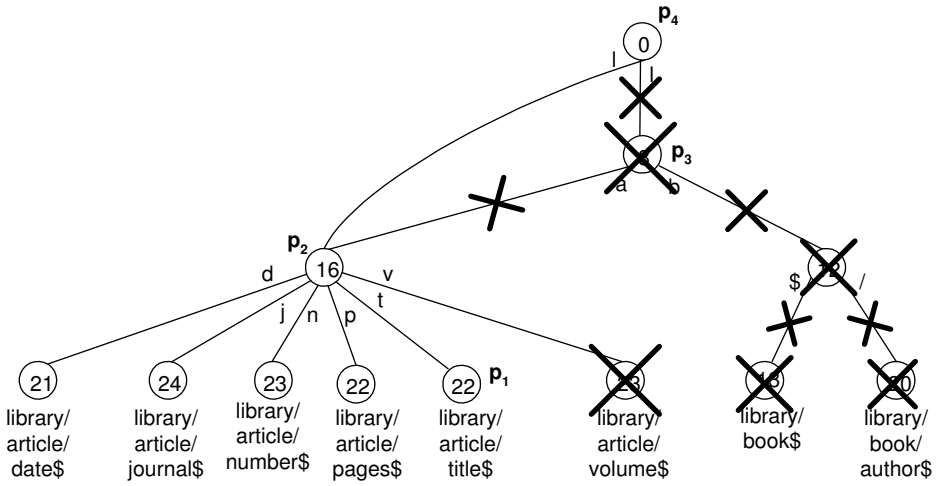


Figura 5.11: Árvore $PT_{\hat{\mathcal{S}}_1}$, resultante de um *split*

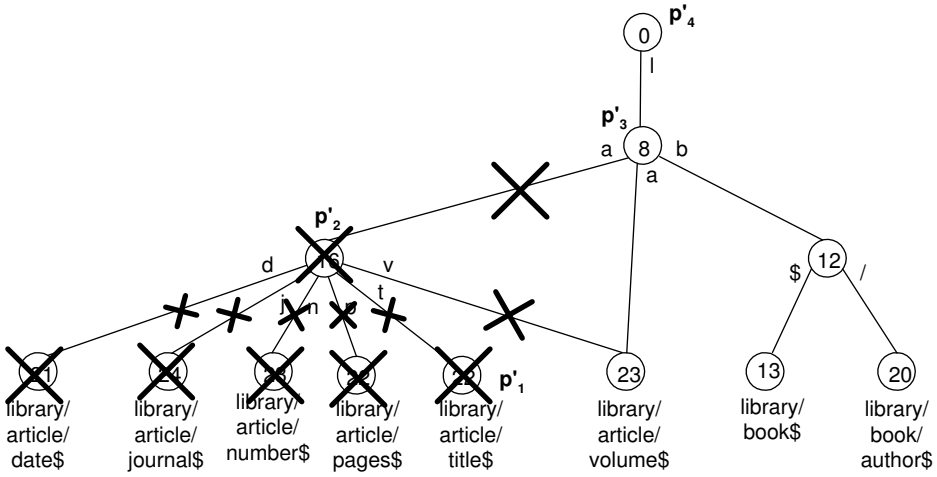


Figura 5.12: Árvore $PT_{\hat{\mathcal{S}}_2}$, resultante de um *split*

2. Seja c (c') o caracter de poda da aresta em $\mathcal{E}_{p_{i+2}}$ ($\mathcal{E}_{p'_{i+2}}$) que ligava p_{i+2} (p'_{i+2}) a p_{i+1} (p'_{i+1}). Substituiremos $e(c, p_{i+2}, p_{i+1}) \in \mathcal{E}_{p_{i+2}}$ ($e(c', p'_{i+2}, p'_{i+1}) \in \mathcal{E}_{p'_{i+2}}$) por uma nova aresta $e(c, p_{i+2}, p_i)$ ($e(c', p'_{i+2}, p'_i)$). Removemos $e(c, p_{i+2}, p_{i+1})$ ($e(c', p'_{i+2}, p'_{i+1})$).

3. O próximo nó de $\mathcal{P}$ ($\mathcal{P}'$) a ser visitado é p_{i+2} (p'_{i+2}).

Finalmente, removemos a folha l' de $PT'_{\hat{\mathcal{S}}}$ através de uma chamada ao algoritmo ***PT-Remove(k, PT'$_{\hat{\mathcal{S}}}$)***. A árvore $PT_{\hat{\mathcal{S}}}$ ($PT'_{\hat{\mathcal{S}}}$) resultante é $PT_{\hat{\mathcal{S}}_1}$ ($PT_{\hat{\mathcal{S}}_2}$).

Como exemplo, suponha que tenhamos a árvore da figura 5.8 e queiramos dividi-la

passando $k =$ "library/article/title$" como parâmetro. As árvores $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$ resultantes podem ser vistas nas figuras 5.11 e 5.12, respectivamente.

5.1.3.6 *Merge*

Seja ***PT-Merge*($PT_{\hat{S}_1}$,$PT_{\hat{S}_2}$)** o procedimento de fusão de duas árvores patrícias $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$. Sejam os conjuntos $\hat{\mathcal{S}}_{1_{ord}}, \hat{\mathcal{S}}_{2_{ord}} \subset \mathcal{S}'_{ord}$, tais que as *strings* de $\hat{\mathcal{S}}_{1_{ord}}$ são lexicograficamente menores que as *strings* de $\hat{\mathcal{S}}_{2_{ord}}$. Ao final do procedimento, devemos obter uma árvore patrícia $PT_{\hat{\mathcal{S}}}$ onde $\hat{\mathcal{S}}_{ord} = \hat{\mathcal{S}}_{1_{ord}} \cup \hat{\mathcal{S}}_{2_{ord}}$.

Seja $\hat{s}$ a *string* mais à esquerda de $\hat{\mathcal{S}}_{2_{ord}}$. Inserimos $\hat{s}$ em $PT_{\hat{\mathcal{S}}_1}$ através de uma chamada a ***PT-Insert*($\hat{s}$,$PT_{\hat{S}_1}$)**. Denominamos por l a folha mais à direita de $PT_{\hat{\mathcal{S}}_1}$ e por l' a folha mais à esquerda de $PT_{\hat{\mathcal{S}}_2}$. Seja o caminho $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ ($\mathcal{P}' = \{p'_1, p'_2, \ldots, p'_m\}$), onde p_{i+1} (p'_{j+1}) é pai de p_i (p'_j), p_1 (p'_1) é l (l') e p_n (p'_m) é a raiz de $PT_{\hat{\mathcal{S}}_1}$ ($PT_{\hat{\mathcal{S}}_2}$).

A cada passo do algoritmo, comparamos dois nós: um de $\mathcal{P}$ e outro de $\mathcal{P}'$, onde a primeira comparação envolve p_1 e p'_1. Dizemos que dois nós $p \in \mathcal{P}$ e $p' \in \mathcal{P}'$ são iguais se $len_p = len_{p'}$. Caso contrário, eles são ditos diferentes.

Sejam $p_i \in \mathcal{P}$ e $p'_j \in \mathcal{P}'$ os nós a serem comparados no momento. Temos três situações a considerar:

1. $p_i = p'_j$. Seja c o caracter de poda da aresta mais à direita de $\mathcal{E}_{p_i}$. Primeiro, substituímos o nó origem de toda aresta em $\mathcal{E}_{p'_j}$ por p_i e as inserimos em $\mathcal{E}_{p_i}$, com exceção talvez da aresta e mais à esquerda de $\mathcal{E}_{p'_j}$, se esta tiver c como caracter de poda. Neste caso, e deverá ser destruída. Em seguida, o nó p'_j é removido. Os próximos nós a serem comparados são p_{i+1} e p'_{j+1}.
2. $p_i \neq p'_j$ e $len_{p_i} > len_{p'_j}$. Seja c o caracter de poda da aresta que liga p'_j a p'_{j-1}. Neste caso, substituímos a aresta $e(c, p'_j, p'_{j-1})$ por uma nova aresta $e(c, p'_j, p_i)$ e, em seguida, $e(c, p'_j, p'_{j-1})$ é destruída. Os próximos nós a serem comparados são p_{i+1} e p'_j.
3. $p_i \neq p'_j$ e $len_{p_i} < len_{p'_j}$. Seja c o caracter de poda da aresta que liga p_i a p_{i-1}. Neste caso, substituímos a aresta $e(c, p_i, p_{i-1})$ por uma nova aresta $e(c, p_i, p'_j)$ e, em seguida, $e(c, p_i, p_{i-1})$ é destruída. Os próximos nós a serem comparados são p_i e p'_{j+1}.

Como exemplo, vamos fazer a fusão das árvores $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$ (figuras 5.11 e 5.12, respectivamente). Na figura 5.13, vemos as duas árvores após a inserção de $\hat{s} =$ "library/article/volume$" em $PT_{\hat{\mathcal{S}}_1}$. O resultado final, a árvore $PT_{\hat{\mathcal{S}}}$, pode ser visto na figura 5.14.

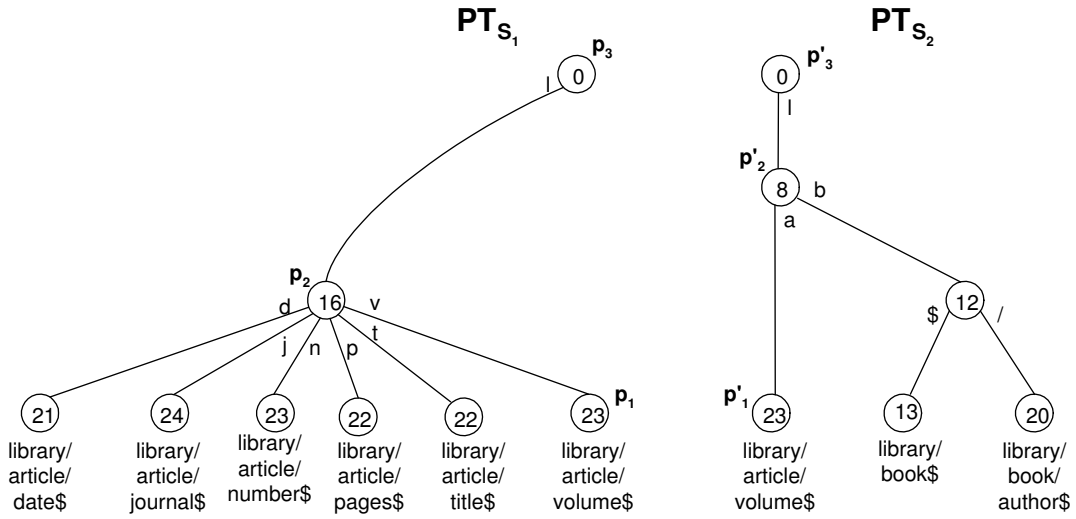


Figura 5.13: Árvores $PT_{\hat{\mathcal{S}}_1}$ e $PT_{\hat{\mathcal{S}}_2}$, antes do procedimento de fusão

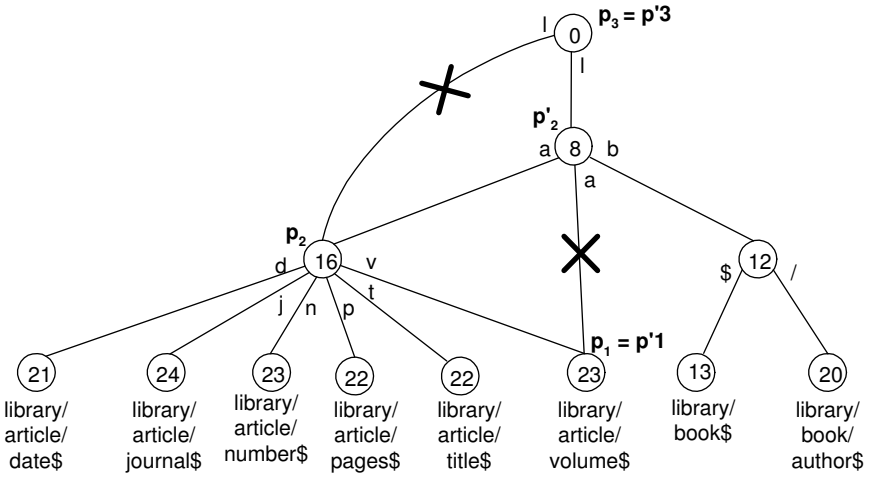


Figura 5.14: Árvore $PT_{\hat{\mathcal{S}}}$, resultante do procedimento de fusão

Note que a árvore da figura 5.14 voltou a ser a mesma da figura 5.10.

5.1.4 Índice FoX

5.1.4.1 Definição

Nesta seção, detalhamos a estrutura de dados do *Índice FoX*. As chaves de busca deste índice são as *strings* do conjunto $\mathcal{S}'$, definido na seção 5.1.1. Seja $\mathcal{S}'_{ord}$ o conjunto onde as *strings* em $\mathcal{S}'$ estão ordenadas lexicograficamente. Representamos estas *strings* no índice

pelos seus respectivos ponteiros lógicos, obtidos através do armazenamento de $\mathcal{S}$ em disco.

Para prevenir que alguma *string* em $\mathcal{S}'$ seja prefixo de outra, colocamos o caracter '$' ao final de cada uma delas. Isto é necessário para a construção correta das árvores patrícias. Com este mesmo intuito, não são permitidas chaves repetidas no índice.

Dado um nó N qualquer do índice, seu conjunto de *strings* lexicograficamente ordenadas é definido por $\mathcal{S}'_{ord_N}$. Denotamos por L_N a *string* mais à esquerda de $\mathcal{S}'_{ord_N}$ e por R_N a *string* mais à direita de $\mathcal{S}'_{ord_N}$.

Cada folha L do *Índice FoX* armazena um conjunto de *strings* ordenadas $\mathcal{S}'_{ord_L} \subseteq \mathcal{S}'_{ord}$ (onde $b \leq |\mathcal{S}'_{ord_L}| \leq 2b$) e a árvore patrícia PT_L construída a partir das *strings* de $\mathcal{S}'_{ord_L}$. Seja $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ o conjunto de folhas do *Índice FoX* tal que as *strings* de $\mathcal{S}'_{ord_{L_i}}$ são lexicograficamente menores que as *strings* de $\mathcal{S}'_{ord_{L_{i+1}}}$ (para todo $0 < i < n$). Lembramos que percorrendo os conjuntos $\mathcal{S}'_{ord_{L_1}}, \mathcal{S}'_{ord_{L_2}}, \ldots, \mathcal{S}'_{ord_{L_n}}$ obtemos novamente $\mathcal{S}'_{ord}$.

Uma folha L do *Índice FoX* também armazena três ponteiros: $prev_L$, que aponta para a folha vizinha à esquerda, $next_L$, que aponta para a folha vizinha à direita, e $parent_L$, que aponta para o nó pai de L. Além disso, em cada folha L existe um conjunto $\mathcal{R}_L$ de vetores de registros, onde cada vetor de registros $R_{L_i} \in \mathcal{R}_L$ corresponde aos nós obtidos através do casamento de padrões de $s'_i \in \mathcal{S}'_{ord_L}$ na Base de Dados (para todo $0 < i \leq |\mathcal{S}'_{ord_L}|$).

Seja $n(N)$ o número de filhos de um nó interno N do *Índice FoX*. Cada nó interno N tem um conjunto $\mathcal{C}_N = \{C_1, C_2, \ldots, C_{n(N)}\}$ de filhos e possui um conjunto ordenado de *strings* $\mathcal{S}'_{ord_N} = \{L_{C_1}, R_{C_1}, \ldots, L_{C_{n(N)}}, R_{C_{n(N)}}\}$, obtido através da cópia da *string* mais à esquerda e da *string* mais à direita de cada um de seus filhos. Como $n(N) = \frac{|\mathcal{S}'_{ord_N}|}{2}$, cada nó tem entre $\frac{b}{2}$ e b filhos, com exceção da raiz do índice, para a qual $2 \leq n(raiz) \leq b$. O nó N também possui uma árvore patrícia, PT_N construída a partir das *strings* de $\mathcal{S}'_{ord_N}$, e um ponteiro $parent_N$, que aponta para o nó pai de N.

Temos que assegurar que cada nó do *Índice FoX* não é maior que B, onde B é o tamanho de uma página de disco. Logo, escolhemos b de maneira a satisfazer esta condição.

Seja $R_{L_i} = r_1, r_2, \ldots, r_m$ um vetor de registros tal que $R_{L_i} \in \mathcal{R}_L$ e L é uma folha do *Índice FoX*. Cada registro $r_j \in R_{L_i}$ $(0 < j \leq m)$ armazena o endereço em disco de um nó da Base de Dados, tal que este nó é acessível através da expressão de caminho $s'_i \in \mathcal{S}'_{ord_L}$. Todos os registros do vetor R_{L_i} são armazenados contiguamente em disco. Logo, para recuperar R_{L_i} precisaremos de m/B acessos, onde B é o tamanho da página de disco.

O algoritmo de busca ***FI-Search(k)*** recupera o vetor de registros referente a uma expressão de caminho k. Para tanto, chamamos o procedimento de busca em uma *String B-tree*, denominado ***SB-Search(k)***, para identificar a posição de k em $\mathcal{S}'_{ord}$. Esta posição

é determinada por um par (L, i), onde L é uma folha e i é a posição de k no conjunto ${\mathcal{S}'}_{ord_L}$. Caso $k \in {\mathcal{S}'}_{ord}$, o vetor de registros $R_{L_i} \in \mathcal{R}_L$ é retornado.

O algoritmo de inserção ***FI-Insert(k,r)*** insere o registro r no vetor de registros correspondente a k. Para tanto, primeiro chamamos o procedimento ***SB-Search(k)*** para obter o par (L, i). Caso $k \in {\mathcal{S}'}_{ord}$, o registro r é adicionado ao vetor de registros $R_{L_i} \in \mathcal{R}_L$. Caso contrário, inserimos k na i-ésima posição de ${\mathcal{S}'}_{ord_L}$ chamando o procedimento de inserção em uma *String B-tree*, denominado ***SB-Insert(k,L,i)***. Em seguida, inserimos r no novo vetor de registros R_{L_i} ($R_{L_i} \in \mathcal{R}_L$) correspondente a k.

O algoritmo de remoção ***FI-Remove(k,r)*** remove o registro r do vetor de registros correspondente a k. Para tanto, primeiro chamamos o procedimento ***SB-Search(k)*** para obter o par (L, i). Caso $k \in {\mathcal{S}'}_{ord}$, o registro r é removido do vetor de registros $R_{L_i} \in \mathcal{R}_L$. Caso $|R_{L_i}| = 0$, removemos k da i-ésima posição de ${\mathcal{S}'}_{ord_L}$ chamando o procedimento de remoção em uma *String B-tree*, denominado ***SB-Remove(k,L,i)***.

Nas seções seguintes, descrevemos os algoritmos ***SB-Search(k)***, ***SB-Insert(k,L,i)*** e ***SB-Remove(k,L,i)***, mencionados acima.

Definiremos aqui algumas expressões comuns aos algoritmos seguintes. Seja N um nó qualquer do índice e seja N' seu vizinho:

1. Dizemos que uma *string* k é ***incluída*** em ${\mathcal{S}'}_{ord_N}$, quando a mesma é inserida lexicograficamente na j-ésima posição de ${\mathcal{S}'}_{ord_N}$. A *string* k também é inserida em PT_N, através do algoritmo ***PT-Insert(k,PT$_N$)***. Um novo vetor de registros vazio R é inserido na j-ésima posição de $\mathcal{R}_N$, caso N seja uma folha.

2. Quando ***removemos*** uma *string* $s'_i \in {\mathcal{S}'}_{ord_N}$, estamos também removendo s'_i de PT_N, através do algoritmo ***PT-Remove(s'_i,PT$_N$)***. O vetor de registros R_{N_i} é removido de $\mathcal{R}_N$, caso N seja uma folha.

3. Quando ***atualizamos*** uma *string* $s'_i \in {\mathcal{S}'}_{ord_N}$ estamos trocando-a por uma outra *string* k. Neste caso, primeiro chamamos ***PT-Remove(s'_i,PT$_N$)*** para remover s'_i de PT_N e, em seguida, chamamos ***PT-Insert(k,PT$_N$)*** para inserir k em PT_N.

4. Dizemos que estamos ***incluindo*** um nó C em $\mathcal{C}_N$, quando inserimos lexicograficamente as *strings* L_C e R_C nas respectivas posições j e $j+1$ de ${\mathcal{S}'}_{ord_N}$ e, analogamente, inserimos C na $(\frac{j+1}{2})$-ésima posição de $\mathcal{C}_N$. Além disso, duas chamadas ***PT-Insert(L$_C$,PT$_N$)*** e ***PT-Insert(R$_C$,PT$_N$)*** são feitas para incluir L_C e R_C em PT_N. O ponteiro $parent_C$ agora aponta para N.

5. Quando uma *string* $s'_i \in {\mathcal{S}'}_{ord_{N'}}$ é ***copiada*** para ${\mathcal{S}'}_{ord_N}$, s'_i é inserida lexicograficamente numa posição j de ${\mathcal{S}'}_{ord_N}$ e, analogamente, $R_{N'_i} \in \mathcal{R}_{N'}$ é inserido na j-ésima

posição de $\mathcal{R}_N$. No entanto, a árvore patrícia PT_N não é alterada.

6. Dizemos que um nó $C \in \mathcal{C}_{N'}$ é ***copiado*** para $\mathcal{C}_N$, quando inserimos lexicograficamente as *strings* L_C e R_C nas respectivas posições j e $j+1$ de ${\mathcal{S}'}_{ord_N}$ e, analogamente, inserimos C na $(\frac{j+1}{2})$-ésima posição de $\mathcal{C}_N$. No entanto, a árvore patrícia PT_N não é alterada.

7. Quando uma *string* $s'_i \in {\mathcal{S}'}_{ord_{N'}}$ é ***transferida*** para ${\mathcal{S}'}_{ord_N}$, s'_i é inserida lexicograficamente numa posição j de ${\mathcal{S}'}_{ord_N}$ e é removida de ${\mathcal{S}'}_{ord_{N'}}$. Analogamente, $R_{N'_i} \in \mathcal{R}_{N'}$ é inserido na j-ésima posição de $\mathcal{R}_N$ e é removido de $\mathcal{R}_{N'}$. Além disso, chamamos o procedimento ***PT-Insert(s'_i,PT_N)*** para inserir s'_i em PT_N e o procedimento ***PT-Remove(s'_i,$PT_{N'}$)*** para remover s'_i de $PT_{N'}$.

8. Dizemos que um nó $C \in \mathcal{C}_{N'}$ é ***transferido*** para $\mathcal{C}_N$, quando inserimos lexicograficamente as *strings* L_C e R_C nas respectivas posições j e $j+1$ de ${\mathcal{S}'}_{ord_N}$ e removemos L_C e R_C de ${\mathcal{S}'}_{ord_{N'}}$. Analogamente, inserimos C na $(\frac{j+1}{2})$-ésima posição de $\mathcal{C}_N$ e o removemos de $\mathcal{C}_{N'}$. Além disso, duas chamadas ***PT-Insert(L_C,PT_N)*** e ***PT-Insert(R_C,PT_N)*** são feitas para incluir L_C e R_C em PT_N e outras duas chamadas ***PT-Remove(L_C,$PT_{N'}$)*** e ***PT-Remove(R_C,$PT_{N'}$)*** são feitas para remover L_C e R_C de $PT_{N'}$. O ponteiro $parent_C$ agora aponta para N.

5.1.4.2 Busca

O procedimento de busca ***SB-Search(k)*** procura a posição de uma *string* k em ${\mathcal{S}'}_{ord}$. O retorno da função é o par (L, i), onde L é uma folha da *String B-tree* e i é a posição de k em ${\mathcal{S}'}_{ord_L}$.

Seja ${\mathcal{S}'}_{ord} = \{s'_1, s'_2, \ldots, s'_n\}$ e $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$. Dois testes são feitos:

1. Se $k \leq_L s'_1$, o resultado da busca é $(L_1, 1)$.
2. Se $k >_L s'_n$, o resultado da busca é $(L_m, n+1)$.

Caso a posição de k não seja nenhuma das duas acima, começamos o procedimento de descida na árvore, partindo da raiz. Seja N o próximo nó do índice a ser visitado. Carregamos do disco a página referente a N e fazemos ${\mathcal{S}'}_{ord_N} = \{s''_1, s''_2, \ldots, s''_q\}$. A seguinte seqüência de passos ocorre:

1. Obtemos a posição de k em ${\mathcal{S}'}_{ord_N}$ através do algoritmo ***PT-Search(k,$\mathcal{S}'_{ord_N}$)***, de maneira a obtermos duas *strings* s''_{j-1} e s''_j tais que $s''_{j-1} <_L k \leq_L s''_j$.
2. Se N é uma folha, retornamos como resultado da busca o par (N, j). O algoritmo acaba aqui.

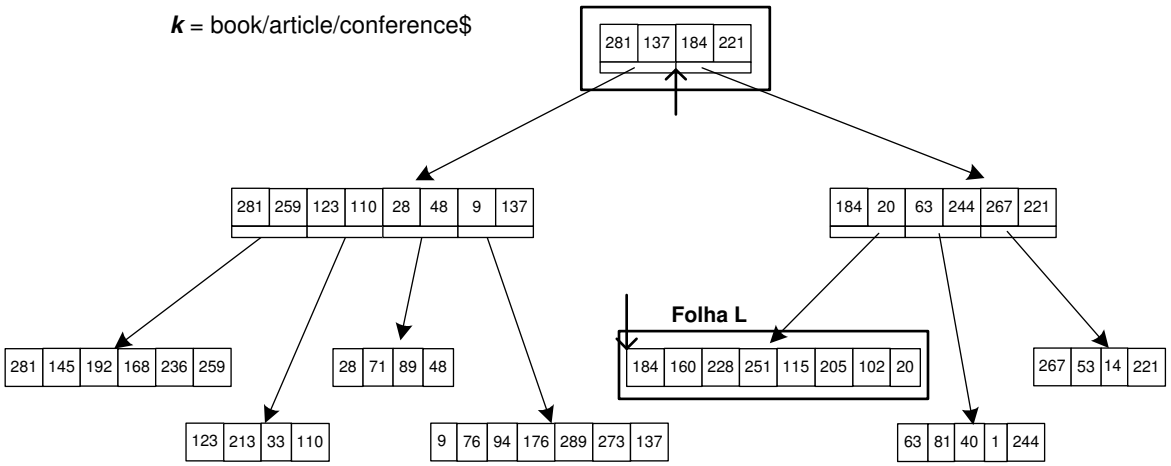


Figura 5.15: Busca em uma *String B-tree* quando k não está em ${\mathcal{S}'}_{ord}$

3. Se N é um nó interno, então:

   (a) Se s''_{j-1} e s''_j pertencem a dois filhos distintos de N, ou seja $s''_{j-1} = R_{C'}$ e $s''_j = L_C$, onde $C', C \in \mathcal{C}_N$. Então as s''_{j-1} e s''_j são adjacentes também em ${\mathcal{S}'}_{ord}$ [39]. Logo, a folha N' que procuramos é a folha mais à esquerda cujo ancestral é C e $i = 1$, pois $L_{N'} = L_C = s''_j$. Retornamos o par $(N', 1)$ e o algoritmo acaba aqui.

   (b) Se ambos s''_{j-1} e s''_j pertencem ao mesmo filho C de N, ou seja $s''_{j-1} = L_C$ e $s''_j = R_C$, então o próximo nó a ser visitado é C.

Exemplo de buscas na árvore da figura 5.4 podem ser vistos nas figuras 5.15 e 5.16. Em ambas, os nós visitados são assinalados, bem como a posição de k em cada nó.

5.1.4.3 Inserção

O procedimento ***SB-Insert(k,L,i)*** inclui a *string* k em ${\mathcal{S}'}_{ord}$, mais precisamente na i-ésima posição de ${\mathcal{S}'}_{ord_L}$.

Denotamos por k' a i-ésima *string* de ${\mathcal{S}'}_{ord_L}$. Caso $i > |{\mathcal{S}'}_{ord_L}|$, k' é a última *string* de ${\mathcal{S}'}_{ord_L}$. Seja $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ um caminho de nós do *Índice FoX* onde P_{j+1} é pai de P_j (para todo $0 < j < n$), $P_1 = L$ e $P_n = raiz$.

O algoritmo recursivo ***Adjust-Ins(N,k',k)*** atualiza os nós $N \in \mathcal{P}$, caso necessário. Na primeira chamada, $N = P_1$. Seja P_j um nó de $\mathcal{P}$ visitado pelo algoritmo. Incluímos k – em ordem lexicográfica – em ${\mathcal{S}'}_{ord_{P_j}}$ e, caso P_j seja um nó interno, removemos k' de ${\mathcal{S}'}_{ord_{P_j}}$. Seja i_j a posição de k após a sua inserção em ${\mathcal{S}'}_{ord_{P_j}}$. Se $i_j = 1$ ou $i_j = |{\mathcal{S}'}_{ord_{P_j}}|$,

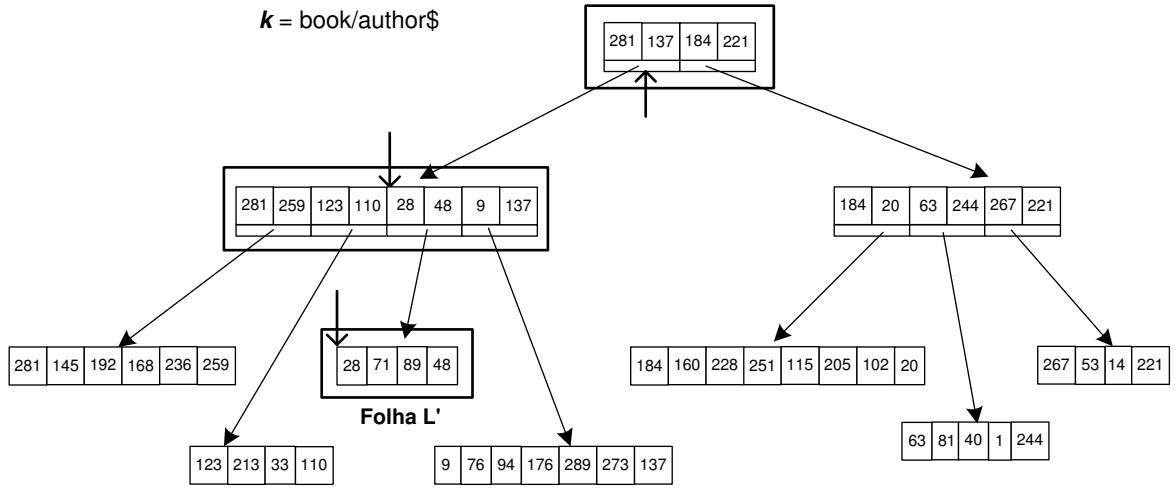


Figura 5.16: Busca em uma *String B-tree* quando k está em ${\mathcal{S}'}_{ord}$

então uma nova chamada ***Adjust-Ins(P_{j+1},k',k)*** é feita. Se $|{\mathcal{S}'}_{ord_L}| \leq 2b$, o algoritmo de inserção termina aqui. Caso contrário, L sofre um *split*.

Seja P o nó pai de L. Caso L seja a raiz do *Índice FoX*, um novo nó interno P é criado e, em seguida, incluímos L em $\mathcal{C}_P$. O procedimento de *split* de uma folha L é descrito pela seguinte seqüência de passos:

1. Um novo nó folha L' (à direita de L) é criado. Copiamos as $b+1$ últimas *strings* de ${\mathcal{S}'}_{ord_L}$ para ${\mathcal{S}'}_{ord_{L'}}$.
2. O algoritmo ***PT-Split(s',PT_L)*** é chamado, onde s' é a b-ésima *string* de ${\mathcal{S}'}_{ord_L}$. Como resultado deste procedimento, temos duas árvores patrícias menores PT_{L_1} e PT_{L_2} tal que PT_{L_1} (PT_{L_2}) corresponde às *strings* de ${\mathcal{S}'}_{ord_L}$ (${\mathcal{S}'}_{ord_{L'}}$). Logo, PT_{L_1} (PT_{L_2}) passa a ser a nova árvore patrícia de ${\mathcal{S}'}_{ord_L}$ (${\mathcal{S}'}_{ord_{L'}}$).
3. Atualizamos o valor de R_L em ${\mathcal{S}'}_{ord_P}$ e incluímos L' em $\mathcal{C}_P$.

Caso $|{\mathcal{S}'}_{ord_P}| \leq 2b$, o algoritmo de inserção acaba aqui. Caso contrário, o nó interno P sofre um *split*.

Seja N o nó interno tal que $|{\mathcal{S}'}_{ord_N}| > 2b$ e seja P' o seu nó pai. Caso N seja a raiz do *Índice FoX*, um novo nó interno P' é criado e, em seguida, incluímos N em $\mathcal{C}_{P'}$. O *split* de nó interno é ligeiramente diferente do *split* de nó folha e é descrito pela seguinte seqüência de passos:

1. Um novo nó interno N' (à direita de N) é criado. Copiamos os $\frac{b}{2}+1$ últimos elementos de $\mathcal{C}_N$ para $\mathcal{C}_{N'}$.

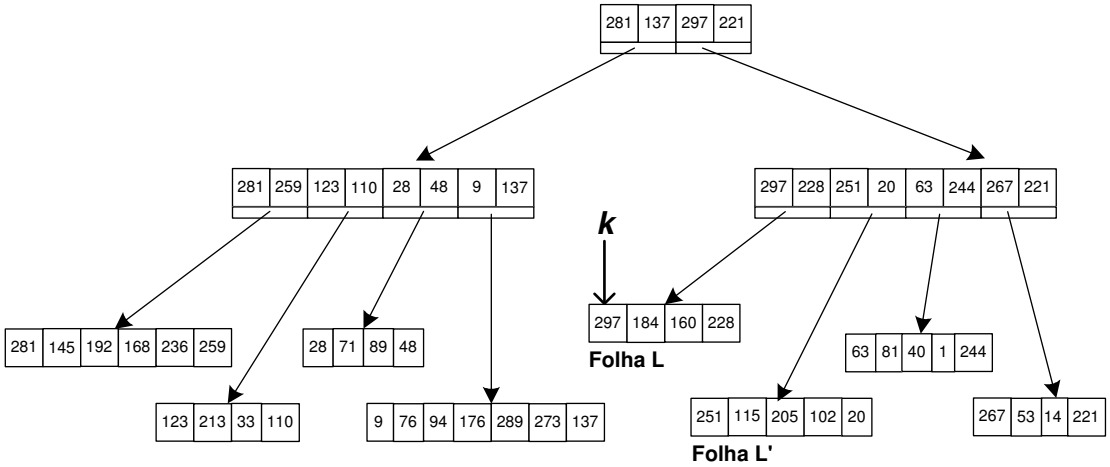


Figura 5.17: Inserção em uma *String B-tree*

2. O algoritmo ***PT-Split(s′,PT$_N$)*** é chamado, onde s' é a b-ésima *string* de ${\mathcal{S}'}_{ord_N}$. Como resultado deste procedimento, temos duas árvores patrícias menores PT_{N_1} e PT_{N_2} tal que PT_{N_1} (PT_{N_2}) corresponde às *strings* de ${\mathcal{S}'}_{ord_N}$ (${\mathcal{S}'}_{ord_{N'}}$). Logo, PT_{N_1} (PT_{N_2}) passa a ser a nova árvore patrícia de ${\mathcal{S}'}_{ord_N}$ (${\mathcal{S}'}_{ord_{N'}}$).
3. Atualizamos o valor de R_N em ${\mathcal{S}'}_{ord_{P'}}$ e incluímos N' em $\mathcal{C}_{P'}$. Caso $|{\mathcal{S}'}_{ord_{P'}}| \leq 2b$, o algoritmo acaba aqui. Caso contrário, fazemos $N = P'$.

A figura 5.17 mostra o índice da figura 5.4 após a inserção de k = "book/article/conference$". Suponha que k já tenha sido gravada em disco e que seu ponteiro lógico é 297. A busca pela *string* k já foi feita na figura 5.15 e com ela descobrimos que a posição de k é definida pelo par $(L, 1)$, onde L está assinalada na figura 5.15. Em seguida, k é inserida em ${\mathcal{S}'}_{ord_L}$ e os devidos ajustes são feitos em seus ancestrais. Além disso, como $|{\mathcal{S}'}_{ord_L}| > 2b$, um *split* entre folhas ocorre, originando uma nova folha L'.

5.1.4.4 Remoção

O procedimento ***SB-Remove(k,L,i)*** remove a *string* k de ${\mathcal{S}'}_{ord}$, mais precisamente da i-ésima posição de ${\mathcal{S}'}_{ord_L}$.

Denotamos por k' a $(i+1)$-ésima *string* de ${\mathcal{S}'}_{ord_L}$. Caso $i = |{\mathcal{S}'}_{ord_L}|$, k' é $(i-1)$-ésima *string* de ${\mathcal{S}'}_{ord_L}$. Seja $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ um caminho de nós do índice onde P_{j+1} é pai de P_j $(0 < j < n)$, $P_1 = L$ e $P_n = raiz$.

O algoritmo recursivo ***Adjust-Rem(N,k′,k)*** atualiza os nós $N \in \mathcal{P}$, caso necessário.

Na primeira chamada, $N = P_1$. Seja P_j $(0 < j < n)$ um nó de $\mathcal{P}$ visitado pelo algoritmo. Removemos k de ${\mathcal{S}'}_{ord_{P_j}}$ e, caso P_j seja um nó interno, incluímos k' (em ordem lexicográfica) em ${\mathcal{S}'}_{ord_{P_j}}$. Seja i_j a posição de k' em ${\mathcal{S}'}_{ord_{P_j}}$. Se $i_j = 1$ ou $i_j = |{\mathcal{S}'}_{ord_{P_j}}|$, então uma nova chamada ***Adjust-Rem(P_{j+1},k',k)*** é feita. Se $|{\mathcal{S}'}_{ord_L}| \geq b$ ou L é a raiz do índice, o algoritmo de remoção termina aqui.

Caso contrário, seja L' um nó irmão de L e P o nó pai de ambos. Uma transferência entre as folhas L e L' ocorre, descrita pela seguinte seqüência de passos:

1. L' é o irmão à direita de L:
   (a) O primeiro elemento de ${\mathcal{S}'}_{ord_{L'}}$ é transferido para ${\mathcal{S}'}_{ord_L}$ e atualizamos as *strings* R_L e $L_{L'}$ em ${\mathcal{S}'}_{ord_P}$. Se $|{\mathcal{S}'}_{ord_{L'}}| \geq b$, o algoritmo termina aqui.
   (b) Se $|{\mathcal{S}'}_{ord_{L'}}| < b$, um *merge* entre L e L' ocorre: Todas as *strings* de ${\mathcal{S}'}_{ord_{L'}}$ são copiadas para ${\mathcal{S}'}_{ord_L}$ e o procedimento ***PT-Merge(PT_L,$PT_{L'}$)*** é chamado para fundir as duas árvores patrícias PT_L e $PT_{L'}$. PT_L é substituída pela árvore patrícia resultante deste procedimento.
   (c) L' é removido de $\mathcal{C}_P$ e em seguida, destruído. A *string* R_L é atualizada em ${\mathcal{S}'}_{ord_P}$.
2. L não possui irmão à direita, logo L' é irmão à esquerda de L:
   (a) O último elemento de ${\mathcal{S}'}_{ord_{L'}}$ é transferido para ${\mathcal{S}'}_{ord_L}$ e atualizamos as *strings* $R_{L'}$ e L_L em ${\mathcal{S}'}_{ord_P}$. Se $|{\mathcal{S}'}_{ord_{L'}}| \geq b$, o algoritmo termina aqui.
   (b) Se $|{\mathcal{S}'}_{ord_{L'}}| < b$, um *merge* entre L' e L ocorre: Todas as *strings* de ${\mathcal{S}'}_{ord_{L'}}$ são copiadas para ${\mathcal{S}'}_{ord_L}$ e o procedimento ***PT-Merge($PT_{L'}$,PT_L)*** é chamado para fundir as duas árvores patrícias $PT_{L'}$ e PT_L. PT_L é substituída pela árvore patrícia resultante deste procedimento.
   (c) L' é removido de $\mathcal{C}_P$ e em seguida, destruído. A *string* L_L é atualizada em ${\mathcal{S}'}_{ord_P}$.

   Caso $|{\mathcal{S}'}_{ord_P}| \geq b$, o algoritmo termina aqui.

Caso contrário, seja N o nó interno tal que $|{\mathcal{S}'}_{ord_N}| < b$. Seja N a raiz do índice. Se $|\mathcal{C}_N| = 1$, então fazemos com que o único filho C de N seja a nova raiz e N é destruído. O algoritmo termina aqui.

Se N não é a raiz do índice, seja N' um nó irmão de N e P' o nó pai de ambos. Uma transferência entre os nós internos N e N' ocorre. A transferência entre nós internos é ligeiramente diferente da transferência entre nós folhas e é descrita pela seguinte seqüência de passos:

1. N' é o irmão à direita de N:

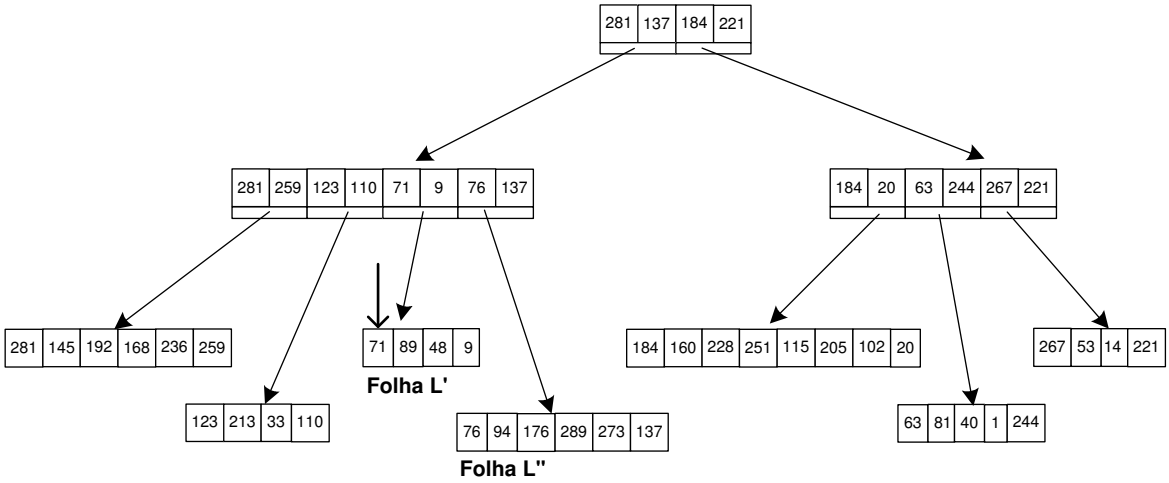


Figura 5.18: Remoção em uma *String B-tree*

(a) O primeiro filho de N' é transferido para N e atualizamos as *strings* R_N e $L_{N'}$ em ${\mathcal{S}'}_{ord_{P'}}$. Se $|{\mathcal{S}'}_{ord_{N'}}| \geq b$, o algoritmo termina aqui.

(b) Se $|{\mathcal{S}'}_{ord_{N'}}| < b$, um *merge* entre N e N' ocorre: Todos os filhos de N' são copiados para N e o procedimento ***PT-Merge(PT_N,$PT_{N'}$)*** é chamado para fundir as duas árvores patrícias PT_N e $PT_{N'}$. PT_N é substituída pela árvore patrícia resultante deste procedimento.

(c) N' é removido de $C_{P'}$ e em seguida, destruído. A *string* R_N é atualizada em ${\mathcal{S}'}_{ord_{P'}}$. Caso ${\mathcal{S}'}_{ord_{P'}} \geq b$, o algoritmo acaba aqui. Caso contrário fazemos $N = P'$.

2. N não possui irmãos à direita, logo N' é irmão à esquerda de N:

(a) O último filho de N' é transferido para N e atualizamos as *strings* $R_{N'}$ e L_N em ${\mathcal{S}'}_{ord_{P'}}$. Se $|{\mathcal{S}'}_{ord_{N'}}| \geq b$, o algoritmo termina aqui.

(b) Se $|{\mathcal{S}'}_{ord_{N'}}| < b$, um *merge* entre N e N' ocorre: Todos os filhos de N' são copiados para N e o procedimento ***PT-Merge($PT_{N'}$,PT_N)*** é chamado para fundir as duas árvores patrícias $PT_{N'}$ e PT_N. PT_N é substituída pela árvore patrícia resultante deste procedimento.

(c) N' é removido de $C_{P'}$ e em seguida, destruído. A *string* L_N é atualizada em ${\mathcal{S}'}_{ord_{P'}}$. Caso ${\mathcal{S}'}_{ord_{P'}} \geq b$, o algoritmo acaba aqui. Caso contrário fazemos $N = P'$.

A figura 5.18 mostra o índice da figura 5.4 após a remoção de k = "book/author$". A busca pela *string* k já foi feita na figura 5.16 e com ela descobrimos que a posição de

k é definida pelo par $(L', 1)$, onde L' está assinalada na figura 5.16. Em seguida, k é removida de ${\mathcal{S}'}_{ord_{L'}}$ e os devidos ajustes são feitos em seus ancestrais. Além disso, como $|{\mathcal{S}'}_{ord_{L'}}| < b$, uma transferência entre folhas de L'' para L' ocorre.

5.1.5 Análise de Complexidade Revisitada

Com a introdução dos *registros* na estrutura do *Índice FoX*, as complexidades devem ser revistas uma vez que, em qualquer operação de busca ou atualização, é necessário recuperar um conjunto de registros. Como já foi dito antes, o conjunto R_{L_i} de registros, referente a uma folha L qualquer e a um elemento $s'_i \in {\mathcal{S}'}_{ord_L}$, é armazenado contiguamente em disco.

No caso da operação de busca ***FI-Search(k)***, o parâmetro da busca é uma *string* k e o retorno da função é o conjunto de registros R_k, correspondente a k. Logo, a complexidade total da busca de uma *string* k no *Índice FoX* é $O((1 + \frac{|k|}{B}) \log_B M) + O(|R_k| \cdot \frac{|r|}{B}) = O(\frac{|k|}{B} \log_B M + |R_k|)$ acessos a disco, onde $|r|$ é o tamanho de um registro r qualquer, $|R_k|$ é a cardinalidade do conjunto de registros R_k, B é o tamanho da página de disco e $M = |\mathcal{S}|$.

Uma operação de inserção ***FI-Insert(k,r)*** possui dois parâmetros: o registro r que desejamos inserir no índice e a expressão de caminho k referente a ele. Seja k_i um sufixo qualquer de k que também é uma expressão de caminho simples. Se $k \in \mathcal{S}$, devemos inserir r em todos os R_{k_i}. Se $k \notin \mathcal{S}$, através do procedimento ***SB-Insert()*** inserimos cada k_i no *Índice FoX* e, neste caso, o único registro de qualquer conjunto R_{k_i} será r.

O pior caso para a inserção de um novo registro r no *Índice FoX* ocorre quando sua expressão de caminho k equivalente já pertence a $\mathcal{S}$. Neste caso, temos que recuperar o conjunto de registros referente a cada sufixo k_i de k, onde $k_i \in \mathcal{S}'$, e inserir r ao final de cada um deles.

Para inserir o registro r em um conjunto de registro R_{k_i} qualquer, primeiramente uma busca no *Índice FoX* é feita, para que seja possível encontrar a folha e a posição de k_i, dentro do conjunto de *strings* ordenadas desta folha. Em seguida, devemos recuperar o conjunto de registros R_{k_i} do disco para que seja possível inserir r ao final do mesmo. Sabemos que esta operação de inserção custa $O((1 + \frac{|k_i|}{B}) \log_B M) + O(|R_{k_i}|) = O(\frac{|k_i|}{B} \log_B M + |R_{k_i}|) = O(\frac{|k|}{B} \log_B M + |R_{k_i}|)$ acessos a disco. Seja $\hat{R}$ um inteiro tal que $\hat{R} = \max\{|R_{k_i}|\}$, para todo $k_i \in \mathcal{S}'$, sufixo de k. Logo, podemos dizer que $O(\frac{|k|}{B} \log_B M + |R_{k_i}|) = O(\frac{|k|}{B} \log_B M + \hat{R})$. Finalmente, deduzimos o custo total de inserção no *Índice FoX* de um registro r, cuja *string* correspondente é k, como $O(d_k(\frac{|k|}{B} \log_B M + \hat{R}))$ acessos a disco, onde d_k é a profundidade de k.

Uma operação de remoção ***FI-Remove(k,r)*** também possui dois parâmetros: o registro r que desejamos remover do índice e a expressão de caminho k referente à ele. Seja k_i um sufixo qualquer de k que também é uma expressão de caminho simples. Se $k \in \mathcal{S}$, devemos remover r de todos os R_{k_i}, onde $k_i \in \mathcal{S}'$. Se $R_{k_i} = \emptyset$ após a remoção de r, removemos também k_i do *Índice FoX* através do procedimento ***SB-Remove()***.

O pior caso para a remoção de um novo registro r no *Índice FoX* ocorre quando sua expressão de caminho k equivalente pertence a $\mathcal{S}$. Neste caso, temos que recuperar o conjunto de registros referente a cada sufixo k_i de k, onde $k_i \in \mathcal{S}'$, e remover r de cada um deles.

Para remover o registro r de um conjunto de registro R_{k_i} qualquer, primeiramente uma busca no *Índice FoX* é feita, para que seja possível encontrar a folha e a posição de k_i, dentro do conjunto de *strings* ordenadas desta folha. Em seguida, devemos recuperar o conjunto de registros R_{k_i} do disco e, em seguida, fazer uma busca por r dentro deste conjunto. Finalmente, removemos r do mesmo. Logo, o número de acessos a disco necessário à remoção é o mesmo daquele necessário à inserção. Ou seja, é $O(d_k(\frac{|k|}{B}\log_B M + \hat{R}))$ acessos a disco, onde d_k é a profundidade de k e $\hat{R} = \max\{|R_{k_i}|\}$.

5.2 Páginas do *Índice FoX*

As páginas em disco destinadas ao *Índice FoX* podem armazenar: (i) o conjunto $\mathcal{S}$, (ii) um conjunto de registros ou (iii) nós do índice. No primeiro caso, armazenamos o conjunto $\mathcal{S}$ em páginas contíguas de disco. As *strings* em $\mathcal{S}$ são armazenadas seqüencialmente e delimitadas pelo caracter especial '$'.

No segundo caso, a página em disco tem o *layout* mostrado na figura 5.19. Seja $R = \{r_1, r_2, \ldots, r_n\}$ um vetor de registros do índice. Um registro r qualquer tem tamanho fixo e, no atual estágio do FoX, armazena apenas o endereço de um determinado nó XML, em disco. Seja B o tamanho em *bytes* de uma página de disco e $|r|$ o tamanho, também em *bytes*, de um registro. Os registros de R são distribuídos em páginas de disco e estas páginas formam uma lista duplamente encadeada. Logo, temos que uma página pode comportar até $\frac{B}{|r|} - 2$ registros e reserva espaço para mais dois endereços de disco, que serve à manutenção da lista encadeada.

Supondo que estamos utilizando 32 *bits* (4 *bytes*) para endereçamento de disco e que o tamanho da página é de 2KB (2048 *bytes*), temos que cada página pode comportar até 512 endereços de disco, sendo que destes, 510 referem-se a endereços de nós XML e 2 são destinados a manter a lista encadeada.

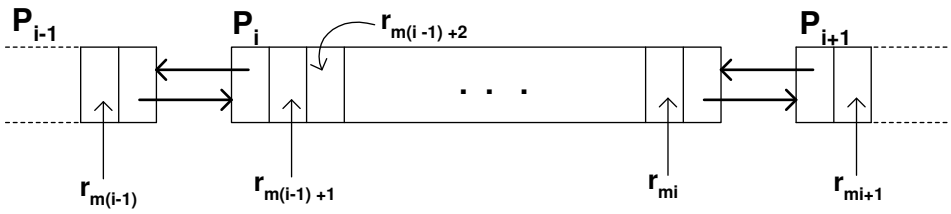


Figura 5.19: Página de Registros

A página em disco que representa um nó N qualquer do *Índice FoX* deve armazenar o conjunto $\mathcal{S}'_{ord_N}$, a árvore patrícia PT_N e o ponteiro $parent_N$. Caso N seja um nó interno, devemos também armazenar nesta página o conjunto $\mathcal{C}_N$ de ponteiros para os filhos de N. Caso N seja um nó folha, o que devemos armazenar na página é o conjunto $\mathcal{R}_N$ de ponteiros para vetores de registros e os ponteiros $prev_N$ e $next_N$. Lembramos que o inteiro b deve ser escolhido de modo que o tamanho de um nó em disco não seja maior que B.

Seja $y = |\mathcal{S}'_{ord_N}|$ o número de folhas da árvore patrícia PT_N onde, no pior caso, $y = 2b$. De acordo com [39], temos que PT_N não tem mais do que y nós internos. Portanto, se o tamanho de um nó n qualquer de uma árvore patrícia é fixo e ocupa $|n|$ *bytes*, então o espaço total necessário para armazenar a árvore patrícia PT_N é de, no máximo, $2 \cdot y \cdot |n| = 2 \cdot 2b \cdot |n| = 4b \cdot |n|$ *bytes*.

Cada nó n de PT_N armazena o ponteiro $parent_n$, o caracter c da aresta $e(c, parent_n, n)$, o rótulo len_n e, caso n seja uma folha, um ponteiro lógico referente à *string* de n. Como toda a árvore PT_N é armazenada em uma única página de disco, o ponteiro $parent_n$ é o deslocamento do armazenamento do nó pai de n, na página onde PT_N está. Logo, $parent_n$ é um inteiro, assim como o rótulo len_n e o ponteiro lógico referente à *string* de n, como já foi visto anteriormente. Portanto, se representamos um inteiro por 4 *bytes* e um caracter por 1 *byte*, temos que cada nó da árvore patrícia ocupa 13 *bytes*. Com isso, temos que uma árvore patrícia ocupa um total de $4b \cdot 32 = 52b$ *bytes*.

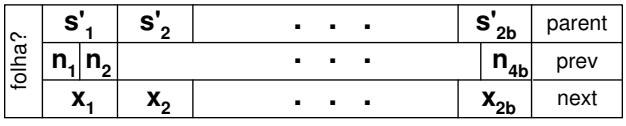


Figura 5.20: Página correspondente a um nó do *Índice FoX*

Na figura 5.20 vemos a estrutura de uma página em disco correspondente a um nó N do *Índice FoX*. Aqui, $s'_i \in \mathcal{S}'_{ord_N}$ $(0 < i \leq 2b)$ e n_j é um nó de PT_N $(0 < j \leq 4b)$. Caso

N seja um nó interno, $x_i \in \mathcal{C}_N$ $(0 < i \leq b)$. Caso contrário, $x_i \in \mathcal{R}_N$ $(0 < i \leq 2b)$. Para identificar se um nó é folha ou não, um *byte* é utilizado. Como representamos endereços de disco e inteiros com 4 *bytes* e caracteres com 1 *byte*, ocuparemos o seguinte espaço, no pior caso:

- O conjunto $\mathcal{S}'_{ord_N}$ ocupa no máximo $2b \cdot 4 = 8b$ *bytes*;
- A árvore patrícia PT_N ocupa no máximo $52b$ *bytes*, como vimos anteriormente, e
- O conjunto $\mathcal{X} = \{x_1, x_2, \ldots, x_{2b}\}$ ocupa $2b \cdot 4 = 8b$ *bytes*, no máximo.

Logo, se assumimos que o tamanho da página em disco é de 2048 *bytes*, determinamos o inteiro b através da seguinte equação: $1 + 8b + 52b + 8b + 4 + 4 + 4 = 2048$. Neste caso, temos que $b = 29$ e cada nó do *Índice FoX* tem entre 29 e 58 elementos.

5.3 Montador de Resultados

O Montador de Resultados funciona como uma *interface* entre o índice e os demais componentes do FoX. Ele fornece um conjunto de funções que: (i) transforma o resultado de uma consulta ao índice em uma floresta de sub-árvores XML e a envia às camadas superiores do FoX e (ii) permite que o Gerenciador de Armazenamento atualize o índice toda vez que a Base de Dados for alterada. Nas sub-seções seguintes, veremos como isto acontece, apresentando os cenários de consulta e atualização do Módulo de Indexação do FoX.

5.3.1 Cenário de Consulta ao Módulo de Indexação

Na figura 5.21 vemos um cenário de consulta ao Módulo de Indexação. Primeiramente, uma expressão de caminho simples A é enviada ao Montador de Resultados, que, por sua vez, a envia ao *Índice FoX* através da função ***RA-GetRegisters(A)***. Esta função deve retornar um vetor de registros R, cujos registros correspondem aos nós resultantes do casamento de padrões de A na Base de Dados. Para que A seja avaliada no índice, algumas páginas P são requisitadas ao Servidor de Páginas, através do procedimento ***FI-GetPage(P)***.

Seja r_i um registro de R e id_i o o endereço de disco armazenado em r_i. Uma chamada à função ***RA-GetSubTree(id$_i$)*** é feita para cada id_i, $0 < i \leq |R|$. Esta função retorna uma sub-árvore de G_{BD}, cuja raiz é o nó armazenado no endereço id_i. Em seguida, as sub-árvores obtidas são repassadas às camadas superiores do FoX.

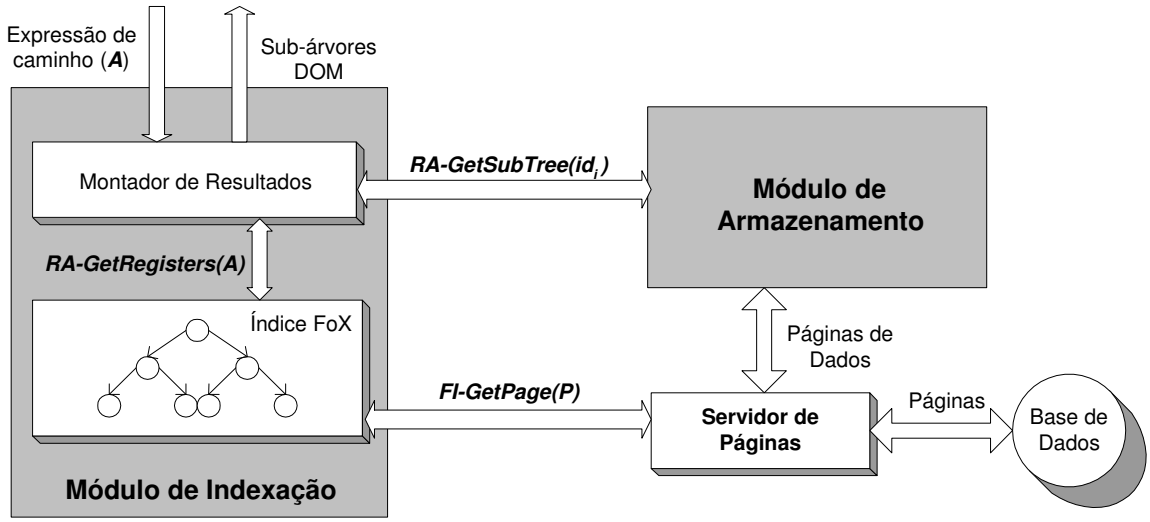


Figura 5.21: Cenário de Consulta ao Módulo de Indexação

5.3.2 Cenário de Atualização no Módulo de Indexação

Antes de explicar este cenário, vamos a algumas definições. Seja BD a Base de Dados XML, D um documento XML qualquer tal que $G_D = (V_{G_D}, E_{G_D}) \in BD$, e $root_{G_D} \in V_{G_D}$ o nó raiz de G_D. Considere $T_D = (V_{T_D}, E_{T_D})$ uma sub-árvore de G_D, tal que $root_{T_D} \in V_{T_D}$ é a sua raiz. Denotamos por $A_{G_D,v}$ a expressão de caminho simples, com origem em $root_{G_D}$, que casa com um nó $v \in V_{G_D}$. Com isso, podemos definir o conjunto $\mathcal{A}_{G_D,T_D}$ $= \{(A_{G_D,v_1}, id_{v_1}), (A_{G_D,v_2}, id_{v_2}), \ldots, (A_{G_D,v_n}, id_{v_n})\}$ como o conjunto de expressões de caminho simples e identificadores, com origem em $root_{G_D}$ e término em v_i, para todo $v_i \in V_{T_D}$. Além disso, fazemos com que o endereço em disco (ou identificador) de $v_i \in V_{T_D}$ seja representado por id_{v_i}.

Uma atualização na Base de Dados do FoX envolve sempre a remoção de uma sub-árvore T_R e a inserção de uma outra sub-árvore T_I, mesmo que $T_R = \emptyset$ ou $T_I = \emptyset$. Na figura 5.22, mostramos a atualização a ser feita na Base de Dados da figura 3.4, para que possamos inserir o nó v'. A árvore a ser removida (T_R) e a árvore a ser inserida (T_I) estão assinaladas na figura. Para refletir esta atualização no *Índice FoX*, devemos primeiramente remover do índice todos os registros referentes a T_R e em seguida, inserir no índice todos os registros referentes a T_I.

Considere que D é o documento em BD que sofre atualizações. Como já foi dito anteriormente, um registro r_v armazena o endereço de um dado nó v em disco. Logo, se $v \in V_{T_R}$ é um nó que desejamos remover da Base de Dados e se sabemos a expressão

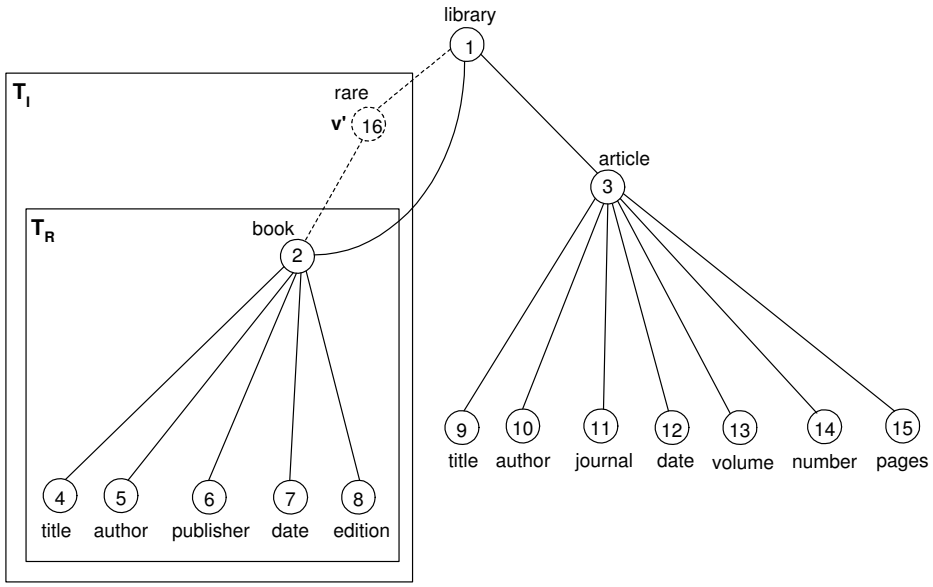


Figura 5.22: Atualização em uma Base de Dados XML

de caminho simples $A_{G_D,v}$ – com origem em $root_{G_D}$ que casa com v – e o endereço de v em disco, ou seja id_v, podemos então remover os registros r_v, correspondentes a v, do *Índice FoX* da seguinte forma: (i) primeiramente, através de $A_{G_D,v}$, descobrimos qual folha do *Índice FoX* aponta para o vetor de registros R, tal que $r_v \in R$, e (ii) em seguida, percorremos R até encontrar o registro r_v e o removemos.

Os registros a serem removidos no índice podem ser encontrados a partir do conjunto inicial $\mathcal{A}_{G_D,T_R}$. Este conjunto está representado na figura 5.23.

$\boldsymbol{A}_{\boldsymbol{G_D},\boldsymbol{T_R}}$ = { (library/book/title, 4), (library/book/author, 5), (library/book/publisher, 6), (library/book/date, 7), (library/book/edition, 8), (library/book, 2) }

Figura 5.23: Conjunto $\mathcal{A}_{G_D,T_R}$ da árvore T_R

Uma vez que no *Índice FoX* são inseridos todos os sufixos, que também são expressões de caminho simples, de uma dada expressão $A_{G_D,v}$, os registros a serem removidos do índice são definidos através do conjunto $\mathcal{A}'_{G_D,T_R}$, exibido na figura 5.24. Note que a diferença entre os conjuntos $\mathcal{A}_{G_D,T_R}$ e $\mathcal{A}'_{G_D,T_R}$ é que em $\mathcal{A}'_{G_D,T_R}$, temos um elemento para cada um dos sufixos – que também são expressões de caminho simples – de todo $A_{G_D,v}$, tal que $(A_{G_D,v}, id_v) \in \mathcal{A}_{G_D,T_R}$.

$\boldsymbol{A'_{G_D,T_R}}$ = { (library/book/title, 4), (book/title, 4), (title, 4), (library/book/author, 5), (book/author, 5), (author, 5), (library/book/publisher, 6), (book/publisher, 6), (publisher, 6), (library/book/date, 7), (book/date, 7), (date, 7), (library/book/edition, 8), (book/edition, 8), (edition, 8), (library/book, 2), (book, 2) }

Figura 5.24: Conjunto $\mathcal{A}'_{G_D,T_R}$ da árvore T_R

O procedimento de remoção ***RA-Remove***$(\boldsymbol{A'_{G_D,T_R}})$ possui como parâmetro o conjunto $\mathcal{A}'_{G_D,T_R}$. O procedimento de inserção ***RA-Insert***$(\boldsymbol{A'_{G_D,T_I}})$ é análogo.

Se $v \in V_{T_I}$ é um nó recém-inserido na Base de Dados e se sabemos a expressão de caminho simples $A_{G_D,v}$ – com origem em $root_{G_D}$ que casa com v – e o endereço de v em disco, ou seja id_v, podemos então inserir os registros r_v, correspondentes a v, no *Índice FoX* da seguinte forma: (i) primeiramente descobrimos em qual folha L do *Índice FoX* a *string* $A_{G_D,v}$ está armazenada (caso $A_{G_D,v} \notin \mathcal{S}'_{ord_L}$, a inserimos em $\mathcal{S}'_{ord_L}$) e (ii) em seguida, incluímos r_v no conjunto R de registros correspondente a $A_{G_D,v}$.

Os locais de inserção dos registros, correspondentes aos nós da árvore T_I, podem ser determinados no índice a partir do conjunto inicial $\mathcal{A}_{G_D,T_I}$. Este conjunto está representado na figura 5.25.

$\boldsymbol{A_{G_D,T_I}}$ = { (library/rare/book/title, 4), (library/rare/book/author, 5), (library/rare/book/publisher, 6), (library/rare/book/date, 7), (library/rare/book/edition, 8), (library/rare/book, 2) (library/rare, 16) }

Figura 5.25: Conjunto $\mathcal{A}_{G_D,T_I}$ da árvore T_I

Uma vez que no *Índice FoX* são inseridos todos os sufixos, que também são expressões de caminho simples, de uma dada expressão $A_{G_D,v} \in \mathcal{A}_{G_D,T_I}$, a localização dos registros a serem inseridos no índice é definida através do conjunto $\mathcal{A}'_{G_D,T_I}$, exibido na figura 5.26. Note que a diferença entre os conjuntos $\mathcal{A}_{G_D,T_I}$ e $\mathcal{A}'_{G_D,T_R}$ é que em $\mathcal{A}'_{G_D,T_I}$, temos um elemento para cada um dos sufixos – que também são expressões de caminho simples – de todo $A_{G_D,v}$, tal que $(A_{G_D,v}, id_v) \in \mathcal{A}_{G_D,T_I}$.

Na figura 5.27 vemos um cenário de atualização do *Índice FoX*. Durante uma atualização, o Gerenciador de Armazenamento chama a funções ***RA-Remove***$(\boldsymbol{A'_{G_D,T_R}})$, para remover os registros referentes a uma sub-árvore T_R, e em seguida, ***RA-Insert***$(\boldsymbol{A'_{G_D,T_I}})$, para inserir os registros referentes a uma nova sub-árvore T_I. Os registros são removidos e

A'_{G_D,T_I} = { (library/rare/book/title, 4), (rare/book/title, 4), (book/title, 4), (title, 4), (library/rare/book/author, 5), (rare/book/author, 5), (book/author, 5), (author, 5), (library/rare/book/publisher, 6), (rare/book/publisher, 6), (book/publisher, 6), (publisher, 6), (library/rare/book/date, 7), (rare/book/date, 7), (book/date, 7), (date, 7), (library/rare/book/edition, 8), (rare/book/edition, 8), (book/edition, 8), (edition, 8), (library/rare/book, 2), (rare/book, 2), (book, 2), (library/rare, 16), (rare, 16) }

Figura 5.26: Conjunto $\mathcal{A}'_{G_D,T_I}$ da árvore T_I

inseridos no índice através dos procedimentos ***FI-Remove(A,r_A)*** e ***FI-Insert(A,r_A)***, onde A é uma expressão de caminho e r_A é o registro correspondente a A. Para tanto, páginas P do índice são requisitadas ao Servidor de Páginas através do procedimento ***FI-GetPage(P)***.

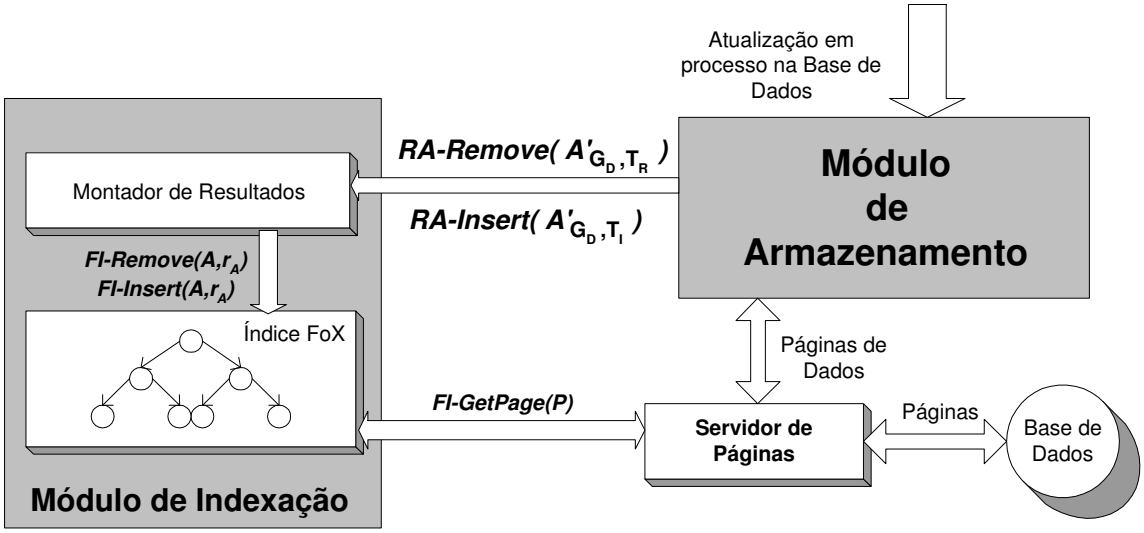


Figura 5.27: Cenário de Atualização do Módulo de Indexação

5.4 Estudo de Caso

Implementamos o *Índice FoX* em C++ utilizando o Microsoft Visual C++ 6.0 em uma estação Pentium II rodando Windows 98. A Base de Dados escolhida para testes foi o *benchmark Shakespeare Plays* [41], detalhado na tabela 5.1. O *parser* Xerces 2.3.0 [42] foi utilizado para a obtenção das árvores DOM, correspondentes aos documentos XML na

Base de Dados. Através do percurso destas árvores, geramos as expressões de caminho utilizadas como chaves de busca no *Índice FoX*. O tamanho da unidade de transferência (página de disco) é de 2KB, que é o tamanho padrão das páginas no Servidor de Páginas.

Para validar o *Índice FoX*, simulamos o caso em que a estratégia de armazenamento é a de "text file" [8], estratégia utilizada pelo Oracle [12] quando os documentos armazenados não possuem esquema. Nesta estratégia, a cada consulta feita à Base de Dados, os documentos XML são inteiramente recuperados do disco e submetidos a um *parser* XML, para que árvores XML correspondentes aos mesmos sejam, por fim, geradas. A resolução da consulta é feita através do percurso destas árvores XML. No caso do Oracle, e no nosso também, estamos supondo que a árvore XML está no formato DOM [35].

Como visto na seção 5.2, uma vez que o tamanho de página é 2KB, o inteiro $b = 29$. Ou seja, cada nó N do *Índice FoX* tem entre 29 e 58 elementos e cabe em uma página de disco. Com isso, temos que a altura do *Índice FoX* é 2. Isto significa que, para recuperar a localização em disco do conjunto de registros correspondente a qualquer expressão de caminho na Base de Dados, precisaremos de apenas 4 acessos: 2 para recuperar os nós do índice e mais 2 acessos correspondentes à busca na árvore patrícia de cada nó visitado.

Na seção 5.2, estávamos considerando que um registro armazenava somente um endereço de disco. Neste caso, como o armazenamento não é XML nativo, o que guardamos em cada registro é: (i) um identificador de documento XML, (ii) os pontos – com relação ao início do documento – onde começa e termina cada elemento XML, ou seja, o deslocamento do seu primeiro e último caracter. Com isso, cada registro agora armazena três inteiros e ocupa 12 *bytes*, já que estipulamos, na seção 5.2 que um inteiro é representado por 4 *bytes*.

Logo, temos que uma página em disco pode comportar até 170 registros. Os 8 *bytes* que restam, por página, correspondem a dois endereços de disco, necessários à manutenção da lista encadeada de registros.

O número de acessos necessários para recuperar o conjunto de registros, referente à expressão de caminho A, depende da *seletividade* de A – ou seja, depende de quantos nós nas árvores DOM são alcançáveis através de A – dentre outros fatores. Para comparar a performance do índice, escolhemos algumas expressões de caminho XPath [10], elencadas na tabela 5.2, com algumas características como alta, média ou baixa seletividade, tipo de nó alvo (nó folha ou nó interno) e proximidade do nó raiz. Ao lado de cada expressão de caminho A, colocamos o número de acessos a disco feitos para recuperar as páginas de índice, necessárias à resolução de A.

Arquivo	Tamanho (bytes)	# de nós
a_and_c.xml	251.898	6.347
all_well.xml	209.730	5.031
as_you.xml	192.140	4.522
com_err.xml	136.841	3.153
coriolan.xml	260.100	6.285
cymbelin.xml	247.365	5.782
dream.xml	145.040	3.361
hamlet.xml	279.663	6.636
hen_iv_1.xml	214.973	4.825
hen_iv_2.xml	234.182	5.255
hen_v.xml	226.266	4.971
hen_vi_1.xml	196.460	4.334
hen_vi_2.xml	226.008	5.046
hen_vi_3.xml	220.202	4.907
hen_viii.xml	217.820	4.905
j_caesar.xml	183.536	4.455
john.xml	178.088	3.926
lear.xml	245.849	5.984
lll.xml	206.945	5.056
m_for_m.xml	202.730	4.877
m_wives.xml	207.174	4.958
macbeth.xml	163.077	3.975
merchant.xml	182.039	4.145
much_ado.xml	195.214	4.727
othello.xml	248.777	6.194
pericles.xml	169.289	4.000
r_and_j.xml	218.508	5.081
rich_ii.xml	192.848	4.116
rich_iii.xml	271.414	6.224
t_night.xml	186.075	4.568
taming.xml	194.295	4.675
tempest.xml	154.645	3.757
timon.xml	177.463	4.339
titus.xml	180.543	3.932
troilus.xml	249.421	6.078
two_gent.xml	164.619	4.141
win_tale.xml	217.135	5.050
TOTAL	**7.648.372 = 7.3 MB**	**179.681**

Tabela 5.1: Descrição do *benchmark Shakespeare Plays*

Expressão de caminho XPath		Características	I[a]+R[b]
Q_1	//SPEECH/SPEAKER	Nó folha; Alta Seletividade	4+180
Q_2	//STAGEDIR	Nó folha; Média Seletividade	4+37
Q_3	//TITLE	Nó folha; Média Seletividade	4+7
Q_4	//SCENE/SPEECH/SUBHEAD	Nó folha; Baixa Seletividade	4+1
Q_5	/PLAY/ACT/SCENE/SPEECH	Nó interno; Alta Seletividade; Distante da Raiz	4+182
Q_6	/PLAY/ACT/SCENE	Nó interno; Média Seletividade; A meio caminho da raiz	4+3
Q_7	/PLAY/ACT	Nó interno; Média Seletividade; Próximo à raiz	4+2
Q_8	//INDUCT/SCENE/SPEECH	Nó interno; Baixa Seletividade; Distante da raiz	4+1
Q_9	//INDUCT/SCENE	Nó interno; Baixa Seletividade; A meio caminho da raiz	4+1

[a]Número de acessos a disco para ir da raiz até alguma folha do *Índice FoX*

[b]Número de acessos a disco para recuperar o conjunto de registros referente à consulta

Tabela 5.2: Consultas XPath

O tipo de armazenamento utilizado, neste caso, favorece as buscas sem índice, principalmente nas consultas que recuperam nós folha, pois estes são mais numerosos e também mais espalhados nos documentos XML. A figura 5.28 mostra a quantidade de acessos a disco necessária para a resolução de cada uma das expressões de caminho da tabela 5.2.

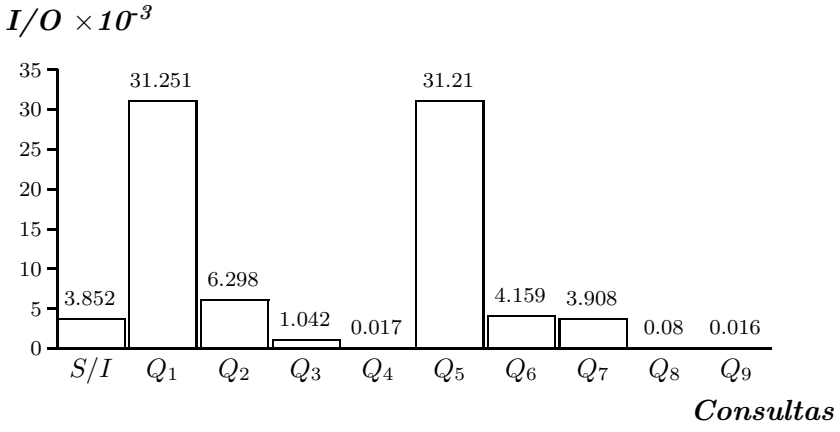


Figura 5.28: Número de acessos a disco

A primeira barra da figura 5.28 mostra o número de acessos necessários à recuperação de toda a Base de Dados, que é o que ocorre quando não utilizamos o *Índice FoX*.

As consultas Q_1, Q_2, Q_3 e Q_4 recuperam nós folha e logo, o número de acessos a disco

necessários para resolver Q_i ($1 \leq i \leq 4$) é exatamente igual à seletividade de Q_i na Base de Dados. Note que, neste caso, o desempenho do *Índice FoX* melhora à medida que a seletividade de Q_i diminui.

As consultas seguintes recuperam nós-internos, o que significa que temos que trazer do disco fragmentos XML correspondentes às sub-árvores cujas raízes são os nós alcançáveis através de Q_j ($5 \leq j \leq 9$). Por isso, influi muito se um nó, alcançável através de Q_j, está próximo ou distante da raiz. Quanto mais distante, menor é o fragmento XML a ser recuperado pelo índice. No entanto, como estes fragmentos estão armazenados como texto ou seja, de forma bastante clusterizada, temos que o número de acessos para recuperar os nós distantes da raiz é maior que o número de acessos necessários para recuperar os nós próximos à raiz. Isto ocorre porque, quanto mais distante da raiz são os nós a serem recuperados, mais espalhados no documento eles estão. Além disso, os nós distantes da raiz geralmente têm seletividade maior (ou seja, são mais numerosos) que os próximos à raiz.

As consultas Q_5, Q_6 e Q_7 referem-se a nós internos com seletividade entre média e alta e em três estágios distintos de proximidade da raiz. Note que quanto mais distante da raiz são os nós a serem recuperados, maior é o número de acessos a disco, como havíamos dito anteriormente. Aqui, o fator decisivo para o número de acessos é a seletividade dos nós. Ou seja, como a seletividade é consideravelmente alta em Q_5, Q_6 e Q_7, não compensa utilizarmos o *Índice FoX*.

Já as consultas Q_8, Q_9 também referem-se a nós internos, porém apresentam baixa seletividade em dois estágios distintos de proximidade da raiz. Mais uma vez, quanto mais distante da raiz são os nós a serem recuperados pelo índice, maior é o número de acessos a disco. No entanto, como a seletividade das consultas Q_8 e Q_9 é baixa, compensa utilizarmos o *Índice FoX*.

Logo, podemos concluir que, no caso em que os nós estiverem armazenados de acordo com a estratégia "text file", o *Índice FoX* apresenta vantagens quando a seletividade das consultas for baixa. Resultados ainda melhores são obtidos quando, além de baixa seletividade, os nós alcançáveis pelas consultas são distantes da raiz ou quando os mesmos têm um número reduzido de filhos.

Capítulo 6

Conclusão

Neste capítulo, concluímos a dissertação apresentando, primeiramente, as principais contribuições deste trabalho e fazendo algumas comparações entre o mesmo e outros trabalhos já existentes. Em seguida, apresentamos os trabalhos futuros a serem realizados, baseados nas idéias desta dissertação.

6.1 Contribuições e Trabalhos Relacionados

A maior contribuição deste trabalho é um índice eficiente para as expressões de caminho simples presentes em uma Base de Dados XML. Tal trabalho é importante, pois a maioria das consultas sobre dados XML é feita utilizando expressões de caminho, como chaves de busca.

Algumas estratégias de indexação vistas no capítulo 2 utilizam estruturas de dados otimizadas para busca como um meio de indexar expressões de caminho simples, como é o caso do *Index Fabric* [32], do *RIST / ViST* [33] e do *PathGuide* [34].

O *Index Fabric* utiliza uma versão balanceada de árvores patrícias [31] para indexar expressões de caminho. Porém, apenas expressões de caminho simples e que se iniciam na raiz do documento são consideradas como chaves de busca, o que impõe uma séria limitação ao índice.

O *RIST / ViST* utiliza árvores B+ [27] para indexar expressões de caminho. Com relação às árvores B+, o *Índice FoX* apresenta vantagens na operação de busca, quando as expressões de caminho são extensas. Caso contrário, o desempenho de ambas é equivalente, como pode ser percebido pelas seções de análise de complexidade, no capítulo 5.

Além disso, estratégias como o *RIST / ViST* e o *Index Fabric* procuram reduzir o

tamanho das expressões de caminho substituindo os nomes de elementos e atributos por *designators*, ou abreviações. A relação entre um *designator* e o nome de um elemento é mantida através de uma tabela. Portanto, além da estrutura de indexação, é necessária a implementação de uma segunda estrutura de dados, que é a tabela de *designators*. À medida que a Base de Dados aumenta, com a inclusão de novos elementos e atributos de nomes distintos, surge a necessidade de buscar mais rapidamente também elementos desta tabela. Isto implica em estratégias de indexação auxiliares para a própria tabela de *designators*, o que leva a um aumento de complexidade do índice e uma demora maior na resolução de expressões de caminho.

O *PathGuide* utiliza árvores sufixo [30] para indexar expressões de caminho simples. No entanto, árvores sufixo são desbalanceadas e, portanto, inadequadas ao armazenamento em memória secundária, ao contrário da estratégia de indexação proposta neste trabalho.

Uma segunda contribuição deste trabalho é a arquitetura de um Módulo de Indexação para o FoX, cujo componente principal é o *Índice FoX*. Além do índice propriamente dito, este módulo apresenta mais um componente chamado Montador de Resultados, que é uma interface entre as demais camadas do FoX e o próprio índice.

Ainda como contribuição deste trabalho, podemos citar a implementação da estrutura de Dados *String B-Tree* [39], dentro do contexto de Bancos de Dados XML Nativos.

6.2 Trabalhos Futuros

Como trabalhos futuros pretendemos, primeiramente, integrar o Módulo de Armazenamento, o Módulo de Indexação e o Servidor de Páginas do FoX. Com isso, oferecemos uma boa base para futuros trabalhos de pós-graduação, dentro do contexto do FoX.

Em seguida, pretendemos desenvolver uma estratégia de numeração que permita que o *Índice FoX* resolva eficientemente não apenas expressões de caminho simples, mas também expressões regulares de caminho. Para tanto, estenderemos a funcionalidade do Montador de Resultados com adaptações de alguns dos algoritmos de junção vistos na seção 2. A estratégia que desejamos adotar é semelhante à do XISS [28].

No XISS, árvores B+ são utilizadas para indexar nomes de elementos e atributos e algoritmos de junção são implementados para resolver expressões regulares de caminho, inclusive as expressões de caminho simples, tomando como parâmetros os números dados a cada elemento ou atributo. Em nosso caso, estaremos indexando, através do *Índice FoX*, expressões de caminho simples, eliminando assim a necessidade de aplicar algoritmos de junção específicos para expressões de caminho simples. A diminuição de tempo de

processamento utilizando a estratégia do *Índice FoX* torna-se significativa à medida que as expressões de caminho simples tornam-se mais extensas.

Por fim, em um projeto futuro, pretendemos adaptar o FoX para dispositivos móveis, como *palms* e celulares.

Capítulo 7

Referências Bibliográficas

[1] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, pages 25–30, Philadelphia, Pennsylvania, USA, June 1999.

[2] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference On Data Engineering (ICDE)*, page 198, San Diego, California, USA, February 2000.

[3] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[4] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 431–442, Philadelphia, Pennsylvania, USA, May 1999.

[5] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David De Witt, and Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, UK, September 1999.

[6] Iraklis Varlamis and Michalis Vazirgiannis. Bridging XML-Schema and Relational Databases. A System for Generating and Manipulating Relational Databases Using Valid XML Documents. In *Proceedings of the ACM Symposium on Document Engineering*, pages 105–114, Atlanta, Georgia, USA, November 2001.

[7] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, pages 47–52, Dallas, Texas, USA, May 2000.

[8] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and Updating the File. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 73–84, Dublin, Ireland, August 1993.

[9] Airi Salminen and Frank Wm. Tompa. Requirements for XML Document Database Systems. In *ACM Symposium on Document Engineering*, pages 85–94, Atlanta, Georgia, USA, November 2001.

[10] World Wide Web Consortium, `http://www.w3.org/TR/xpath`. *XPath 1.0*, 1999.

[11] World Wide Web Consortium, `http://www.w3.org/XML/Schema`. *XML Schema Specification*, 2001.

[12] Oracle, `http://otn.oracle.com/tech/xml/xmldb/`. *Oracle XML DB, Technical Whitepaper*, 2003.

[13] IBM, `http://www-3.ibm.com/software/data/db2/extenders/`. *DB2 Extenders.*

[14] Deirdre Kong. *DB2 XML Extender.* `http://wwwcsif.cs.ucdavis.edu/~kong/DB2_XML_Extender.pdf`.

[15] Y. Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the 11th IEEE International Conference on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, March 1995.

[16] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[17] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo, and Anand Rajaraman. Indexing Semistructured Data. Technical report, Stanford University, 1998.

[18] Flavio Rizzolo and Alberto Mendelzon. Indexing XML Data with ToXin. In *Proceedings of 4th International Workshop on the Web and Databases (WebDb)*, pages 49–54, Santa Barbara, California, USA, May 2001.

[19] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 277–295, Jerusalem, Israel, January 1999.

[20] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engeneering (ICDE)*, pages 129–140, San Jose, California, USA, February 2002.

[21] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: an Adaptive Path Index for XML Data. In *Proceedings of the ACM International Conference on Management of Data*, pages 121–132, Madison, Wisconsin, USA, June 2002.

[22] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engeneering (ICDE)*, pages 141–152, San Jose, California, USA, February 2002.

[23] Dao Dihn Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An XML Indexing Structure with Relative Region Coordinates. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 313–320, Heidelberg, Germany, April 2001.

[24] Paul F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 122–127, San Francisco, California, USA, May 1982.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[26] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274, Hong Kong, China, August 2002.

[27] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[28] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370, Rome, Italy, September 2001.

[29] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engeneering (ICDE)*, pages 253–264, Bangalore, India, March 2003.

[30] Edward M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of ACM*, 23(2):262–272, April 1976.

[31] Donald R. Morrison. PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of ACM*, 15(4):514–534, January 1968.

[32] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 341–350, Roma, Italy, August 2001.

[33] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the ACM International Conference on Management of Data*, pages 110–121, San Diego, California, USA, June 2003.

[34] Jiefeng Cheng, Ge Yu, Guoren Wang, and Jeffrey Xu Yu. PathGuide: An Efficient Clustering Based Indexing Method for XML Path Expressions. In *Proceedings of the 18th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 257–264, Kyoto, Japan, March 2003.

[35] World Wide Web Consortium, `http://www.w3.org/TR/REC-DOM-Level-1/`. *Document Object Model (DOM) Level 1 Specification*, 1998.

[36] World Wide Web Consortium, `http://www.w3.org/TR/2000/WDquery-algebra-20001204/`. *The XML Query Algebra*, 2000.

[37] World Wide Web Consortium, `http://www.w3.org/TR/xquery`. *XQuery 1.0: An XML Query Language*, 2001.

[38] R. C. Mauro. Aspectos de Gerência de Objetos Persistentes: A Implementação do GOA++. Master's thesis, COPPE, UFRJ, 1998.

[39] Paolo Ferragina and Roberto Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of ACM*, 46(2):236–280, 1999.

[40] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[41] Jon Bosak. Complete Plays of Shakespeare in XML. `http://www.ibiblio.org/xml/examples/shakespeare`.

[42] The Apache XML Project. Xerces C++ Parser. `http://xml.apache.org/xerces-c`.

Apêndice A

Código fonte do *Índice FoX*

A.1 FoXSBTree.h

```
#ifndef FoXSBTreeH
#define FoXSBTreeH

#include "FoXSBTreeNode.h"
#include <math.h>
#include <iostream>
#include <string>

using namespace std;

class FoXSBTree{

private:
  int order;
  FoXSBTreeNode* root;

  inline void SetOrder(int order) { this->order = order; }
  inline void SetRoot(FoXSBTreeNode* node) { root = node; }
  void Free(FoXSBTreeNode* node);
  FoXSBTreeNode* LeftMostLeaf(FoXSBTreeNode* node);
  FoXSBTreeNode* RightMostLeaf(FoXSBTreeNode* node);
  bool SearchNode(const string &word, FoXSBTreeNode* &node, int &pos);
  void AdjustSides(FoXSBTreeNode* node, int posWord, const string &word,
                   int logicalPointer, bool isInsertion);
  void Split(FoXSBTreeNode* node, FoXSBTreeNode* sibling);
  void InsertLeaf(FoXSBTreeNode* node, const string &word, int logicalPointer,
                  int pos, const FoXRegisterItem &registerItem);
  void InsertInternalNode(FoXSBTreeNode* parent, FoXSBTreeNode* node,
                          FoXSBTreeNode* sibling);
  void DeleteLeaf(FoXSBTreeNode* node, const string &word, int pos);
  void DeleteInternalNode(FoXSBTreeNode* parent, FoXSBTreeNode* node,
                          FoXSBTreeNode* sibling, bool isRightSibling);

public:
  FoXSBTree(int order);
  ~FoXSBTree();
  inline int Order() { return order; }
```

```
  inline FoXSBTreeNode* Root() { return root; }
  bool Search(const string &word);
  bool Search(const string &word, vector<FoXRegisterItem> &items);
  bool Insert(const string &word, int logicalPointer,
              const FoXRegisterItem &registerItem);
  bool Delete(const string &word, const FoXRegisterItem &registerItem);
};
#endif
```

A.2 FoXSBTreeNode.h

```
#ifndef FoXSBTreeNodeH
#define FoXSBTreeNodeH

#include "FoXPTrie.h"
#include "FoXFileAccess.h"
#include "FoXRegisterItem.h"
#include <math.h>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class FoXSBTreeNode{

private:
  int order;
  int* logicalPointer;
  vector<FoXRegisterItem>** registers;
  FoXPTrie* pTrie;
  FoXSBTreeNode** child;
  FoXSBTreeNode* parent;
  int nElements;
  bool isLeaf;
  FoXSBTreeNode* leftSibling;
  FoXSBTreeNode* rightSibling;
  FoXFileAccess* file;

  int SearchPos(int logicalPointer);

public:
  FoXSBTreeNode(int order, FoXSBTreeNode* parent, FoXSBTreeNode* leftSibling,
                FoXSBTreeNode* rightSibling, bool isLeaf);
  ~FoXSBTreeNode();
  inline int Order() { return order; }
  inline int LogicalPointer(int pos) { return logicalPointer[pos]; }
  inline string Word(int pos) { return file->Path(logicalPointer[pos]); }
  inline vector<FoXRegisterItem>* Registers(int pos) {return registers[pos]; }
  inline FoXPTrie* PTrie() { return pTrie; }
  inline FoXSBTreeNode* Child(int pos) { return child[pos]; }
  inline FoXSBTreeNode* Parent() { return parent; }
  inline int NElements() { return nElements; }
  inline bool IsLeaf() { return isLeaf; }
```

```
  inline void SetOrder(int order) { this->order = order; }
  inline void SetLogicalPointer(int pos, int logicalPointer)
              { this->logicalPointer[pos] = logicalPointer; }
  inline void SetRegister(int pos, vector<FoXRegisterItem>* oneRegister)
              { registers[pos] = oneRegister; }
  inline void SetPTrie(FoXPTrie* pTrie) { this->pTrie = pTrie; }
  inline void SetChild(int pos, FoXSBTreeNode* child) { this->child[pos] = child; }
  inline void SetParent(FoXSBTreeNode* parent) { this->parent = parent; }
  inline void SetNElements(int nElements) { this->nElements = nElements; }
  inline void SetIsLeaf(bool isLeaf) { this->isLeaf = isLeaf; }
  inline void SetLeftSibling(FoXSBTreeNode* sibling) { leftSibling = sibling; }
  inline void SetRightSibling(FoXSBTreeNode* sibling) { rightSibling = sibling; }
  inline int LeftLogicalPointer() { return logicalPointer[0]; }
  inline int RightLogicalPointer() { return logicalPointer[nElements-1]; }
  inline string LeftWord() { return file->Path(logicalPointer[0]); }
  inline string RightWord() { return file->Path(logicalPointer[nElements-1]); }
  inline vector<FoXRegisterItem>* LeftRegister() {return registers[0]; }
  inline vector<FoXRegisterItem>* RightRegister() {return registers[nElements-1]; }
  inline FoXSBTreeNode* LeftChild() { return child[0]; }
  inline FoXSBTreeNode* RightChild() { return child[NChildren()-1]; }
  FoXSBTreeNode* LeftSibling();
  FoXSBTreeNode* RightSibling();
  inline int NChildren() { return int(floor(nElements/2.0)); }
  inline bool IsFull() { return (nElements > (2*order)); }
  inline bool IsHalfFull() { return (nElements < order); }
  int Search(const string &word, bool &hasWord);
  int Search(FoXSBTreeNode* node);
  void Insert(const string &word, int logicalPointer, int pos);
  void InsertRoot(FoXSBTreeNode* child);
  void Insert(int pos, const FoXRegisterItem &registerItem);
  void Insert(const string &word, int logicalPointer, int pos,
              const FoXRegisterItem &registerItem);
  void Insert(FoXSBTreeNode* child, int pos);
  void Split(FoXSBTreeNode* neighbour, int median);
  void Delete(int pos);
  int Delete(int pos, const FoXRegisterItem &registerItem, bool &result);
  void Delete(int pos, const string &word);
  void Delete(int pos, bool isRightNeighbour);
  void Transfer(FoXSBTreeNode* sibling, bool isRightSibling);
  void Merge(FoXSBTreeNode* neighbour, bool isRightNeighbour);
};
#endif
```

A.3 FoXPTrie.h

```
#ifndef FoXPTrieH
#define FoXPTrieH

#include "FoXPTrieNode.h"
#include <iostream>
#include <string>

using namespace std;
```

```
class FoXPTrie{

private:
  FoXPTrieNode* root;

  void Free(FoXPTrieNode* node);
  FoXPTrieNode* LeftMostLeaf(FoXPTrieNode* node);
  FoXPTrieNode* RightMostLeaf(FoXPTrieNode* node);
  void BlindSearch(FoXPTrieNode* &node, const string &word);
  int CommonPrefix(const string &word, const string &nodeWord);
  FoXPTrieNode* HitNode(FoXPTrieNode* node, const string &word,
                        int commonPrefixSize);
  FoXPTrieNode* RightSibling(FoXPTrieNode* node);
  bool SearchNode(FoXPTrieNode* &node, const string &word);
  void Delete(FoXPTrieNode* node);
  void Duplicate(FoXPTrieNode* thisNode, FoXPTrieNode* neighbourNode);
  void Branch(const string &word, bool rightSide);

public:
  FoXPTrie();
  ~FoXPTrie();
  inline FoXPTrieNode* Root() { return root; }
  inline void SetRoot(FoXPTrieNode* node) { root = node; }
  string LeftMostWord();
  string RightMostWord();
  bool Search(const string &word, int &logicalPointer);
  void Insert(const string &nodeWord, FoXPTrieNode* &node, const string &word,
              int logicalPointer);
  bool Insert(const string &word, int logicalPointer);
  void Split(FoXPTrie* neighbour, const string &word);
  void Delete(int pos);
  bool Delete(const string &word);
  void Merge(FoXPTrie* neighbour);

};
#endif
```

A.4 FoXPTrieNode.h

```
#ifndef FoXPTrieNodeH
#define FoXPTrieNodeH

#include "FoXFileAccess.h"
#include <string>

using namespace std;

class FoXPTrieNode{

private:
  int nChildren;
  bool isLeaf;
  FoXPTrieNode** child;
  char* arcChar;
```

```
  int sizeWord;
  FoXPTrieNode* parent;
  int logicalPointer;
  FoXFileAccess* file;

  void SetChild(int pos, FoXPTrieNode* node);
  void RemoveChild(int pos);
  void SetArcChar(int pos, char arcChar);
  void RemoveArcChar(int pos);

public:
  FoXPTrieNode(FoXPTrieNode* parent, int sizeWord);
  FoXPTrieNode(FoXPTrieNode* parent, int sizeWord, int logicalPointer);
  FoXPTrieNode(FoXPTrieNode* parent, int sizeWord, int logicalPointer,
               bool isLeaf);
  ~FoXPTrieNode();
  inline int NChildren() { return nChildren; }
  inline bool IsLeaf() { return isLeaf; }
  inline FoXPTrieNode* Child(int pos) { return child[pos]; }
  inline char ArcChar(int pos) { return arcChar[pos]; }
  inline int SizeWord() { return sizeWord; }
  inline FoXPTrieNode* Parent() { return parent; }
  inline int LogicalPointer() { return logicalPointer; }
  inline string Word() { return file->Path(logicalPointer); }
  inline void SetNChildren(int nChildren) { this->nChildren = nChildren; }
  inline void SetIsLeaf(bool isLeaf) { this->isLeaf = isLeaf; }
  inline void SetSizeWord(int sizeWord) { this->sizeWord = sizeWord; }
  inline void SetParent(FoXPTrieNode* node) { parent = node; }
  inline void SetLogicalPointer(int logicalPointer)
              { this->logicalPointer = logicalPointer; }
  int SearchArcChar(char wordChar);
  int SearchPosArcChar(char wordChar);
  int SearchChild(FoXPTrieNode* child);
  void InsertChild(int pos, FoXPTrieNode* child, char arcChar);
  void InsertChild(FoXPTrieNode* child, char arcChar);
  void ReplaceChild(FoXPTrieNode* oldChild, FoXPTrieNode* newChild);
  void DeleteChild(FoXPTrieNode* child);
  void DeleteChild(int pos, bool rightSide);
  void Merge(FoXPTrieNode* node);
};
#endif
```