UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMOS DE REMOÇÃO PARA A ESTRUTURA DE INDEXAÇÃO ONION-TREE

DEBORA GONÇALVES RODRIGUES MARRACH

ORIENTADOR: PROF. DR. RICARDO RODRIGUES CIFERRI

SÃO CARLOS - SP
JULHO/2013

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMOS DE REMOÇÃO PARA A ESTRUTURA DE INDEXAÇÃO ONION-TREE

DEBORA GONÇALVES RODRIGUES MARRACH

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software, Banco de Dados e Interação Humano Computador.

Orientador: Prof. Dr. Ricardo Rodrigues Ciferri.

SÃO CARLOS - SP

JULHO/2013

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

M358ar

Marrach, Debora Gonçalves Rodrigues.
Algoritmos de remoção para a estrutura de indexação Onion-tree / Debora Gonçalves Rodrigues Marrach. -- São Carlos : UFSCar, 2013.
125 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2013.

1. Algoritmos de computador. 2. Método *onion-tree*. 3. Remoção de dados. 4. Método de acesso métrico. 5. Indexação em memória primária. 6. Consultas por abrangência. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Algoritmos de Remoção para a Estrutura de Indexação Onion-Tree"

Débora Gonçalves Rodrigues Marrach

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Ricardo Rodrigues Ciferri
(Orientador - DC/UFSCar)

Prof. Dr. Renato Bueno
(DC/UFSCar)

Prof. Dr. Joaquim Cezar Felipe
(USP/RP)

São Carlos
Agosto/2013

Dedico esta dissertação a meu marido Beto e a meu filho Heron

"A grandeza de Napoleão consistia sobretudo em ser sempre o mesmo homem.
Antes e durante uma batalha, após uma vitória, após uma derrota,
sempre se mantinha de pé firme e tinha a noção clara do que se devia fazer."
Goethe

AGRADECIMENTO

Agradeço a Deus e a seu filho Jesus, por tudo.

A meu marido Beto, por todo amor e amizade incondicionais durante todo o desenvolvimento deste trabalho.

A meu filho Heron, principal razão para que eu me mantenha produtiva e feliz.

Às melhores amigas Milene e Vera, por todos os conselhos sinceros.

À minha mãe Naidete e irmãos Marcello e Fabrício, por todo apoio em forma de amor e paz de uma família.

Aos amigos Luana, Vanessa, Walter, Thiago, Renata e Vinicius por todo incentivo.

Especialmente ao amigo Cauê Cury, pela ajuda desinteressada, no momento em que eu mais precisei.

Agradeço especialmente ao Prof. Ricardo Rodrigues Ciferri, por toda orientação, ensinamento, e principalmente pela confiança em ter me selecionado para este mestrado.

Agradeço aos professores Marilde Terezinha Prado Santos, Renato Bueno e Cristina Dutra de Aguiar Ciferri por toda a ajuda prestada, e também por suas amizades.

RESUMO

A Onion-tree é um método de acesso métrico eficiente baseado em memória primária para pesquisa por similaridade. Esta estrutura de indexação já provê algoritmos para a inserção de elementos e o processamento de consultas por similaridade dos tipos *Range Query* (consulta por abrangência) e *KNN* (consulta aos *k-vizinhos* mais próximos). Entretanto, ainda não foi proposto na literatura um algoritmo para a remoção de elementos na Onion-tree. Para que a Onion-tree possa ser efetivamente incorporada a um Sistema Gerenciador de Banco de Dados, portanto, é necessário a proposta e a implementação de, pelo menos, um algoritmo de remoção. Esta pesquisa de mestrado se concentrou primeiramente na implementação e na avaliação de desempenho do algoritmo de remoção lógica proposto em (CARÉLO et al., 2011). A proposta feita em (CARÉLO et al., 2011) deu origem à implementação de três algoritmos de remoção lógica, denominados *LogicalDelete, ReplaceReducing* e *ReplaceGrowing.* O algoritmo *LogicalDelete* aplica a remoção lógica, enquanto os algoritmos *ReplaceReducing* e *ReplaceGrowing* são especializações da remoção lógica, adicionando tratamento especial para a remoção de elementos em nós internos com filhos exclusivamente folha. O algoritmo *ReplaceReducing* permite a diminuição do raio do nó que sofreu a remoção. De forma antagônica, o algoritmo *ReplaceGrowing* permite o aumento deste raio. Adicionalmente, foram propostos e avaliados algoritmos de remoção física que podem ser aplicados em qualquer nível da estrutura da Onion-tree: O algoritmo *ReorgAll* reorganiza todos os elementos da hierarquia do nó que sofreu a remoção, removendo-os fisicamente e reinserindo-os no índice usando o algoritmo de inserção de elementos; e o algoritmo *PromoteNode*, o qual estende o algoritmo *ReorgAll*, promovendo, quando houver condições para tal, outro nó em substituição àquele que sofreu a remoção. Os testes experimentais dos algoritmos de remoção *LogicalDelete*, *ReplaceReducing* e *ReplaceGrowing* mostraram que o algoritmo *LogicalDelete* tem melhor relação custo/benefício que os algoritmos *ReplaceReducing* e *ReplaceGrowing* no processamento de consultas por abrangência após a remoção de elementos. Os testes experimentais dos algoritmos de remoção física mostraram que a promoção de um nó, em substituição ao nó removido, efetuada pelo algoritmo *PromoteNode* apresenta vantagens em relação a simples reorganização da hierarquia que sofreu a remoção. Além de apresentar menor custo de remoção dos elementos no índice, o algoritmo *PromoteNode* também apresenta desempenho superior no processamento de consultas por abrangência após a remoção de elementos. Quando comparados com o algoritmo de remoção lógica, para uma grande quantidade de operações de remoção, os algoritmos *ReorgAll* e *PromoteNode* produziram melhora de 21,6% no desempenho do processamento de consultas por abrangência. Porém, na mesma comparação, estes algoritmos apresentaram custo de remoção muito maior.

Palavras-chave: Onion-tree, Método de Acesso Métrico, Remoção de Dados, Consulta por Similaridade, Consultas por Abrangência, Indexação em Memória Primária

ABSTRACT

The Onion-tree is an efficient metric access method based on main memory for similarity search. The Onion-tree has already provided algorithms for insertion and processing of similarity queries (range query and k-nearest neighbors query). However, in the literature no algorithm has been proposed for removing elements in Onion-tree. For this index be incorporated into a database management system, it is necessary the proposal and implementation of at least one algorithm of deletion. This master's research focused primarily on the implementation and performance evaluation of the algorithms proposed for logical deletion in (CARÉLO et al., 2011). The proposal presented in (CARÉLO et al., 2011) led to the implementation of three algorithms, called *LogicalDelete*, *ReplaceReducing* and *ReplaceGrowing*. The first algorithm applies the logic deletion, while the other two algorithms are specializations adding special treatment for the deletion of elements in internal nodes with children exclusively leaf. The *ReplaceReducing* algorithm allows the reduction of the radius of the node that contains de deleted element. On the other hand, the *ReplaceGrowing* algorithm allows increasing this radius. In addition, algorithms have been proposed and evaluated for physical deletion that can be applied at any level of the Onion-tree. The algorithm *ReorgAll* rearranges all the elements in the hierarchy of the node that contains de deleted element, by physically removing the elements and reinserting them using the insertion algorithm, and algorithm *PromoteNode*, which extends the algorithm *ReorgAll*, promotes, when exists conditions for such operation, other node to replace the one that contains the deleted element. Experimental evaluation of the algorithms *LogicalDelete*, *ReplaceReducing* and *ReplaceGrowing* showed that the algorithm *LogicalDelete* is more cost effective than the algorithms *ReplaceReducing* and *ReplaceGrowing* in query processing after the deletion of elements. Experimental evaluation of physical removal algorithms showed that the promotion of a node to replace the removed node has advantages over the simple reorganization of the hierarchy of the node that contains the deleted element. Besides presenting lower cost of deletion of elements, the algorithm *PromoteNode* also outperformed the algorithm *ReorgAll* in query processing after removing elements. When compared with the logic deletion algorithm, for a large amount of deletion operations, the algorithms *ReorgAll* and *PromoteNode* produced performance gain of 21.6% in range query processing. However, in the same comparison, these algorithms have a much higher cost of deletion.

Keywords: Onion-tree, Metric Access Methods, Data Remove, Query by Similarity, Primary Memory Data Indexing

LISTA DE FIGURAS

LISTA DE TABELAS

ABREVIATURAS E SIGLAS

CBIR: Content Based Image Retrieval, 23, 27
CBR: *Content Based Retrieval*, 20, 23

KNN Query: Consulta pelos K-Vizinhos mais próximos, 17
KNNQ. *Consulte* KNN

MA: Método de acesso, 19
MAM: Método de acesso métrico, 23, 33, 34, 35, 38, 40, 42, 49, 53, 54

Range Query: Consultas por abragência, 17
ROT: Relação de ordem total, 19

SGBD: Sistema Gerenciador de Banco de Dados, 17, 19
SGBDs. *Consulte* SGBD

TOM: Taxa de ocupação mínima, 54, 55, 62

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 **Contexto**

Uma das consequências do aumento da capacidade de armazenamento de dados e da queda do custo dos recursos computacionais é a maior utilização de sistemas de informação para a manipulação de dados em formatos mais complexos, além dos dados convencionais numéricos, data e textuais já amplamente utilizados. Assim por exemplo, dados multimídia, dados espaciais e sequências genômicas passaram a ser armazenados e utilizados amplamente em SGBDs (*Sistemas Gerenciadores de Banco de Dados)* para que sejam indexados e recuperados com eficiência. Portanto, o acesso a recursos computacionais mais poderosos e mais baratos possibilitou a exploração destes dados complexos que são extensivos em armazenamento e intensivos em processamento. Atualmente, a manipulação destes dados tem se demonstrado cada vez mais importante, nas mais variadas áreas: desde a criminalística, incluindo as agências de notícia até a medicina.

As características intrínsecas dos dados complexos criam novas questões relacionadas ao armazenamento destes grandes conjuntos de dados como também questões relacionadas às peculiaridades inerentes a sua recuperação. Em relação ao armazenamento, suas estruturas são consideravelmente mais complexas que a dos dados convencionais. Dados complexos possuem um tamanho muito maior, e também muito mais variável que os dados convencionais. Ademais, dados complexos não possuem relação de ordem total (ROT). Para responder a uma consulta, ou seja, em sua recuperação, uma varredura sequencial em uma base tão extensa não é eficiente. Destas questões, surgiram novos requisitos para a indexação, como comparações baseadas na similaridade, por exemplo, em graus de cores, texturas e formas, ou seja, a busca por características intrínsecas no conteúdo da do dado complexo.

A eficiência e a precisão na recuperação de dados complexos dependem do correto tratamento das características intrínsecas destes dados pelo SGBD, e esse cenário tem motivado o interesse dos pesquisadores por métodos capazes de indexar apropriadamente estes dados.

1.2 **Motivação e Objetivos**

A recuperação baseada no conteúdo intrínseco de dados complexos utiliza funções métricas específicas a cada característica do dado complexo que se deseja analisar para determinar o grau de similaridade entre dois elementos. Expressa como uma medida numérica, quanto maior for este valor mais diferenças há entre os elementos. Quanto menor for o valor desta medida, mais similares são os elementos analisados. Uma vez que as buscas são baseadas no grau de similaridade entre os elementos, os índices empregam estruturas também baseadas em similaridade.

A Onion-tree (CARÉLO et al., 2009) é uma árvore métrica de indexação baseada em memória primária. Novamente, em função do aumento da capacidade de armazenamento dos dados e da diminuição dos custos dos recursos computacionais, especialmente dos custos de memória primária e processadores, uma estrutura de indexação baseada em memória primária pode ser a resposta para aplicações que possuam um conjunto de dados cujo tamanho pode ser armazenado em memória primária e que necessitem muitas vezes reconstruir um índice rapidamente. Além disso, uma estrutura de indexação em memória primária também pode ser utilizada como ferramenta de otimização de subconsultas.

Um método de acesso (ou índice) deve prover um conjunto de operações de manutenção dos dados, ou seja, deve oferecer suporte às operações de inserção, remoção, atualização, bulk-loading (carga intensiva de dados) e consulta dos elementos indexados. A Onion-tree já provê algoritmos para a inserção de elementos e para algumas consultas por similaridade como Range Query (consulta por abrangência) e KNN-query (consulta aos k-vizinhos mais próximos). Porém, ainda não foi proposto na literatura o algoritmo de remoção de elementos da Onion-tree (CARÉLO et al., 2011). Portanto, para que a Onion-tree possa ser efetivamente incorporada a um SGBD, é necessária a proposta e a implementação de, pelo menos, um algoritmo de remoção de elementos.

Esta pesquisa de mestrado se concentrou primeiramente na implementação e na avaliação do algoritmo de remoção lógica proposto teoricamente em (CARÉLO et al., 2011), bem como na proposta e avaliação de dois algoritmos de remoção física para a Onion-tree.

A proposta feita em (CARÉLO et al., 2011) deu origem à implementação de três algoritmos de remoção lógica com as características abaixo:

- Algoritmo *LogicalDelete:* Consiste em apenas marcar, pelo uso de um campo extra, que o elemento encontra-se removido e, portanto, deve ser desconsiderado no processamento de consultas;
- Algoritmo *ReplaceReducing*: Estende o algoritmo *LogicalDelete* adicionando tratamento especial na remoção em nós internos com filhos exclusivamente folha. O objetivo foi avaliar o efeito da diminuição do raio do nó, causado pela substituição do elemento removido;
- Algoritmo *ReplaceGrowing*: Também estende o algoritmo *LogicalDelete* adicionando tratamento especial na remoção em nós internos com filhos exclusivamente folha. Porém analisando o efeito do aumento no raio do nó causado pela substituição do elemento removido.

Além da implementação e avaliação dos algoritmos *LogicalDelete*, *ReplaceReducing* e *ReplaceGrowing*, também foram propostos e avaliados dois algoritmos de remoção física em qualquer nível da estrutura da Onion-tree:

- Algoritmo *ReorgAll*: Reorganiza todos os elementos da hierarquia do nó que sofreu a remoção, removendo-os fisicamente e reinserindo-os usando o algoritmo de inserção de elementos da Onion-tree;
- Algoritmo *PromoteNode*: Estende o algoritmo *ReorgAll*, promovendo, quando houver condições para tal, outro nó em substituição ao nó que sofreu a remoção.

1.3 **Organização do Trabalho**

Esta monografia está organizada da seguinte forma:

- O Capítulo 2 descreve os fundamentos teóricos necessários para a compreensão deste trabalho;

- O Capítulo 3 é dedicado a um estudo específico dos índices métricos que já possuem algoritmos de remoção implementados;
- O Capítulo 4 apresenta os algoritmos de remoção lógica *LogicalDelete*, *ReplaceReducing* e *ReplaceGrowing*, bem como os testes de desempenho e uma análise dos resultados;
- O Capítulo 5 apresenta os algoritmos de remoção física *ReorgAll* e *PromoteNode* propostos nesta pesquisa de mestrado, bem como os testes de desempenho e uma análise dos resultados;
- O Capítulo 6 encerra esta dissertação com as conclusões e apresenta propostas para trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Dados armazenados e indexados por SGBDs podem ser de vários tipos. Em domínios numéricos e lexicográficos, os métodos de acesso utilizam a relação de ordem total (ROT) para construir hierarquias baseadas nas relações de precedência entre os dados. Os MAs mais importantes nestes domínios são as B-trees (BAYER, R. AND MCCREIGHT, 1972) e suas variantes, como a B*-tree (BAYER, R. AND MCCREIGHT, 1972) e a B^+-tree (BAYER, R. AND MCCREIGHT, 1972).

Contudo, existem domínios que manipulam dados complexos, também chamados de dados não convencionais, tais como imagens, sons, vídeos e sequências de nucleotídeos e de aminoácidos. As operações relacionadas aos dados complexos diferem daquelas aplicadas sobre dados convencionais. O domínio de dados complexos não está sujeito à ROT. Para estes dados não é possível aplicar uma relação de ordem sem a utilização de um atributo extra não complexo. Nestes domínios, os predicados de recuperação frequentemente utilizam o grau de similaridade entre um elemento (que é a base da consulta) e os elementos armazenados no banco de dados para determinar o conjunto resposta de uma consulta. A similaridade entre imagens, por exemplo, pode ser calculada a partir de características intrínsecas das imagens tais como cor, forma e textura (FELIPE, 2005). Expressa como uma medida numérica, quanto maior for este valor, maiores são as diferenças entre os elementos analisados, e quanto menor for o valor desta medida, mais similares são estes elementos. Desta forma as consultas mais frequentes em dados complexos têm como objetivo recuperar uma determinada quantidade de elementos mais similares (*K-NN Query)*, ou todos aqueles elementos que sejam similares ao elemento base da consulta em até certo grau (*Range Query).* Portanto, a operação de busca executada pelo SGBD depende da determinação da similaridade entre os elementos do banco de dados.

As características intrínsecas de um dado complexo são extraídas por meio de extratores e representadas em vetores numéricos denominados vetores de

características, e a comparação destes vetores torna possível determinar o grau de similaridade entre os elementos. Este processo é conhecido como "recuperação de dados por conteúdo" (*CBR - Content Based Retrieval)*. No caso de imagens, estes vetores podem representar, por exemplo, o histograma de cores das imagens.

Quando os vetores de características possuem dimensão fixa, por exemplo, dados com 10 dimensões, e podem ser representados em R, então os dados podem ser indexados em um espaço multidimensional. Um sistema de coordenadas geo-referenciadas é um exemplo de dado que pode ser indexado em um espaço multidimensional (CIFERRI, 2002).

Porém, existem dados complexos em que na composição dos vetores a quantidade de dimensões pode variar conforme as propriedades extraídas do dado. Nestes casos, os elementos não terão dimensões fixas, ou seja, seus elementos serão adimensionais.

Estes conjuntos de dados adimensionais podem ser representados e indexados em um espaço métrico, desde que para isso seja definida uma função de distância métrica. Para este propósito, deve-se definir uma função de distância que determine a medida da dissimilaridade entre os elementos do conjunto de dados.

Um espaço métrico M é definido pelo par $<\mathbb{S},d()>$, onde $\mathbb{S}$ é o conjunto de todos os elementos que satisfazem as propriedades do domínio, e *d()* é uma função de distância entre estes elementos (Expressão 2.1). Assim, seja uma função de distância, definida sobre os elementos do domínio:

$$d: \mathbb{S} \times \mathbb{S} \rightarrow R^{+} \quad (2.1)$$

Sejam s_1, s_2 *e* s_3 elementos pertencentes à $\mathbb{S}$, se a função de distância *d* satisfaz as seguintes propriedades ela é considerada uma função de distância métrica:

1. Simetria: $d(s_1,s_2) = d(s_2,s_1)$;
2. Não negatividade: $0 < d(s_1,s_2) < \infty$, $s_1 \neq s_2$, $d(s_1,s_1) = 0$**;**
3. Desigualdade triangular: $d(s_1,s_3) \leq d(s_1 , s_2) + d(s_2,s_3)$.

A recuperação eficiente baseada em conteúdo envolve o armazenamento das características de interesse em vetores de características e o emprego de estruturas

de indexação apropriadas, sendo que a indexação em espaços métricos facilita as operações em que a similaridade é o fator primordial.

A ideia geral de um método de acesso métrico é selecionar para cada subconjunto dos dados, um ou mais elementos e utilizá-los como seus representantes. Nestes métodos de acesso, o conjunto de dados tem sua estrutura e operações de busca guiadas unicamente com base nas distâncias (dissimilaridade) entre os elementos e os seus representantes, sendo que estas distâncias podem ser calculadas ou ainda previamente armazenadas.

Porém, uma situação particular nesta estrutura é a possibilidade de haver sobreposições na cobertura do espaço, ou seja, que um determinado elemento esteja contido em mais de um subconjunto de dados e neste caso, em uma consulta não há garantias de que a busca fique restrita a um único caminho. Todavia, quando não houver sobreposição de regiões e a distância for gerada por meio de uma função de distância métrica, através da propriedade de desigualdade triangular é possível descartar *("podar")* conjuntos de dados não relevantes para a pesquisa.

A propriedade de desigualdade triangular utiliza o valor armazenado das distâncias entre os elementos do nó e seus representantes para inferir os limites inferior e superior de descarte de elementos para a busca conhecendo-se apenas duas distâncias. Desta forma, uma *"poda"* pode ser feita sem que seja necessário calcular todas as distâncias entre o elemento de consulta e os elementos armazenados, reduzindo a quantidade de cálculos de distância necessários à busca, e evitando que sejam desnecessariamente percorridos alguns ramos da estrutura de indexação (BUENO, 2009).

Dado o espaço métrico $<\mathbb{S},d()>$ e o conjunto de elementos $S \subseteq \mathbb{S}$, o elemento de consulta $s_q \in \mathbb{S}$, o raio de consulta r_q e um elemento representante $s_{rep} \in S$, em uma busca, um elemento $s_i \in S$ poderá ser descartado pela propriedade de desigualdade triangular se uma das duas condições a seguir for satisfeita (BUENO, 2009).

$$d(s_{rep}, s_i) < d(s_{rep}, s_q) - r_q \quad (2.2)$$

$$d(s_{rep}, s_i) > d(s_{rep}, s_q) + r_q \quad (2.3)$$

Uma ilustração da aplicação da propriedade de desigualdade triangular para *"podar"* cálculos de distância é demonstrada na Figura 2.1. Na figura, o elemento f_0 é

o elemento representante do nó da árvore; o elemento de consulta é representado por s_q; e r_q é o raio da consulta. O conjunto de dados do domínio métrico $S \subset S$ é representado pelos elementos s_1, s_2, s_3 e s_4.

Com base na desigualdade triangular, analisando a Figura 2.1 pode-se verificar que: (i) os elementos s_2 e s_3 podem ser descartados; (ii) o elemento s_1 deve ser verificado, porém não fará parte do conjunto resposta; (iii) o elemento s_4 deve ser verificado e fará parte do conjunto resposta (BUENO, 2009). Ainda, a região mais interna **A** equivale à condição da inequação 2.2 e a região mais externa **C** a condição da inequação 2.3.

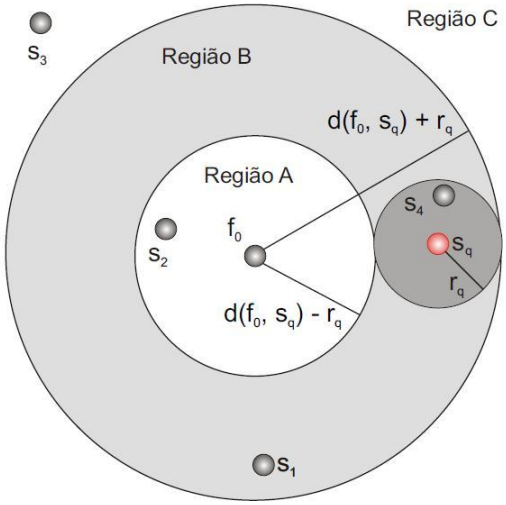


Figura 2.1: Poda de elementos pela desigualdade triangular (BUENO, 2009)

Como a distância entre o elemento do nó ao seu representante é armazenada na árvore, qualquer elemento que estiver nas regiões **A** ou **C** pode ser descartado através da propriedade da desigualdade triangular sem mais cálculos. Já na região **B**, há a necessidade de se calcular as distâncias entre o elemento de consulta e os elementos pertencentes a esta região para verificar se esses elementos devem fazer parte do conjunto resposta (BUENO, 2009).

Vários métodos de acesso métrico (MAM) têm sido desenvolvidos com o objetivo de aumentar a eficiência no processamento de consultas por similaridade (POLA, 2010). Estes métodos diferem tanto na forma de armazenamento dos dados indexados, os quais podem utilizar memória primária ou memória secundária (disco magnético), quanto na forma como os dados são organizados.

Como exemplo dos principais MAM propostos na literatura podemos citar a Metric-tree (UHLMANN, 1991), VP-tree *(Vantage point tree)* (YIANILOS, 1993) e sua extensão MVP-tree *(Multi-Vantage Point tree)* (BOZKAYA; OZSOYOGLU, 1997), a GH-tree *(Generalized Hyperplane Decomposition tree)* (UHLMANN, 1991), a M-tree (CIACCIA et al., 1997) e a sua extensão Slim-tree (TRAINA JR. et al., 2000) (TRAINA JR. et al., 2002), a MM-tree (POLA et al., 2007) e sua extensão a Onion-tree (CARÉLO et al., 2009) (CARÉLO et al., 2011). Os MAM mais relevantes a esta dissertação, a MM-tree e sua extensão a Onion-tree serão tratados em detalhes na seção 2.6 e 2.7 deste capítulo.

2.1 **Recuperação de Dados por Conteúdo**

A maneira mais simples de busca em dados complexos é por meio do uso de rótulos ou descrições atribuídas pelo usuário. Associa-se ao dado complexo um texto descritivo e a busca é feita por intermédio da localização de palavras chaves neste texto descritivo (POLA, 2010).

Neste sistema, a descrição, por exemplo, de uma imagem por meio de texto restringe a busca e também está sujeita a alta subjetividade. Uma forma mais eficiente é fazer com que a busca seja feita com base no conteúdo, ou seja, com base no conjunto de características intrínsecas de uma imagem (FALOUTSOS, 1996) (SMEULDERS; WORRING, M.; et al., 2000). Para imagens, características como cor, textura e forma podem identificar a imagem de forma mais objetiva, possibilitando uma maior gama de consultas.

As características intrínsecas de um dado complexo podem ser extraídas por extratores com base em uma função de distância (por exemplo, métrica) que as expressam por meio de uma representação numérica, tornando assim possível a comparação destas características entre os elementos do conjunto de dados. Assim, o valor resultante da comparação entre dois elementos expressa o quanto um elemento é semelhante ou não ao outro, ou pode-se dizer que expressa a distância entre estes elementos. Esta distância passa então a guiar as operações de busca baseada na similaridade entre os elementos.

Assim, a recuperação de dados por conteúdo se aplica a vários tipos de dados complexos, como imagem e áudio. Embora envolvendo maior processamento, este processo pode possibilitar a descrição de uma quantidade

maior de características dos dados. Ademais, este processo ainda permite que dados complexos possam ser indexados através de estruturas de indexação (PETRAKIS; FALOUTSOS, 1997) (PETRAKIS et al., 2002). Um sistema de recuperação de dados baseado em conteúdo basicamente possui quatro componentes principais (SMEULDERS; WORRING, M; et al., 2000):

- Um módulo responsável pela extração automática de características intrínsecas que representem cada elemento do conjunto de dados;
- Um conjunto definido de métricas capazes de avaliar a similaridade entre os elementos do domínio;
- Uma interface de usuário que permita tanto a definição dos parâmetros para a consulta, quanto a visualização dos resultados obtidos;
- Um mecanismo de busca, que realiza as operações de busca sobre o conjunto de dados armazenados. Os mecanismos de busca mais relevantes a esta pesquisa serão discutidos nas seções 2.6 e 2.7.

2.2 **Dimensionalidade**

A representação de dados complexos pode envolver a formação de um vetor com muitas características (ou dimensões). No caso de imagens, por exemplo, a quantidade de características necessárias para representá-las pode ultrapassar a casa das centenas. Esta alta dimensionalidade, comum para imagens, aumenta a complexidade na manipulação de dados complexos.

Para um grande número de dimensões (ou para uma alta dimensionalidade), as distâncias entre os elementos tendem a se homogeneizar. A grande quantidade de dimensões aumenta o espaço de busca e as distâncias entre pares de elementos tendem a ser muito próximas. Assim, conforme a dimensionalidade aumenta, os espaços multidimensionais tendem a ser mais esparsos, produzindo um fenômeno conhecido como *"maldição da dimensionalidade"* (TALAVERA, 1999). A indexação destes conjuntos de dados de alta dimensionalidade em um método de acesso multidimensional apresenta desempenho degradado e uma busca sequencial pode ser mais vantajosa, desde que grande parte da estrutura de indexação tem que ser percorrida para a obtenção da resposta de uma consulta. Métodos de acesso multidimensionais como a R-tree (GUTTMAN, 1984), e suas derivadas R^+-tree (SELLIS et al., 1987) e R^*-tree (BECKMANN et al., 1990) têm degradados seus

desempenhos tornando-se inviáveis para dimensionalidades maiores que uma dezena. Em função disso, muitos trabalhos foram dedicados a reduzir a dimensionalidade desses conjuntos de dados (SOUSA, 2006). Os métodos de acesso métricos são mais apropriados em domínios com um grande número de dimensões, sobretudo porque são construídos com base na dissimilaridade entre os elementos.

2.3 **Funções de Distância**

As funções de distância, quando atendem as propriedades descritas de simetria, não negatividade e desigualdade triangular são também chamadas funções de distância métrica. Os vetores de valores numéricos (vetores de características) gerados pelos extratores de características podem ser comparados utilizando-se diversas funções de distância métrica. As funções mais utilizadas são as funções da família *Minkowski* (L_p) (WILSON; MARTINEZ, 1996). Estas métricas são usualmente utilizadas em domínios de espaços multidimensionais e em árvores métricas quando uma função de distância métrica estiver associada ao domínio (POLA, 2005).

Assim, sendo dois vetores de características de dimensão *E*, $a = \{a_1, a_2, a_3 ..., a_e\}$ e $b = \{b_1, b_2, b_3 ..., b_e\}$, a família de distâncias (L_p) é definida como:

$$L_p(a,b) = \left(\sum_{i=1}^{E} |a_i - b_i|^p \right)^{1/p} \qquad (2.4)$$

Nesta família de funções, existem três métricas muito utilizadas: a L_1, a L_2 e a L_∞. A métrica L_1 mais conhecida como *City-Block* ou *Manhattan* corresponde ao somatório dos módulos das diferenças entre as coordenadas dos vetores de características. A métrica L_2 conhecida *distância Euclidiana* é mais usual para cálculo de distância entre vetores de características. Esta métrica corresponde à raiz quadrada do somatório dos módulos das diferenças ao quadrado entre as coordenadas dos vetores de características. A métrica L_∞ conhecida como *Chebyshev* é obtida com o cálculo do limite da equação 2.4 quando *p* tende ao infinito. A Figura 2.2 ilustra as diferentes regiões de cobertura para cada função de distância Minkowski.

- $L_1(a,b)$ - *City-Block ou Manhattan:* $\sum_{i=1}^{k} |a_i - b_i|$

- *L_2(a,b) - Distância Euclidiana:* $(\sum_{i=1}^{v} |a_i - b_i|^2)^{1/2}$
- *L_∞ (a,b) - Chebyshev:* $max_{i=1}^{v} |a_i - b_i|$

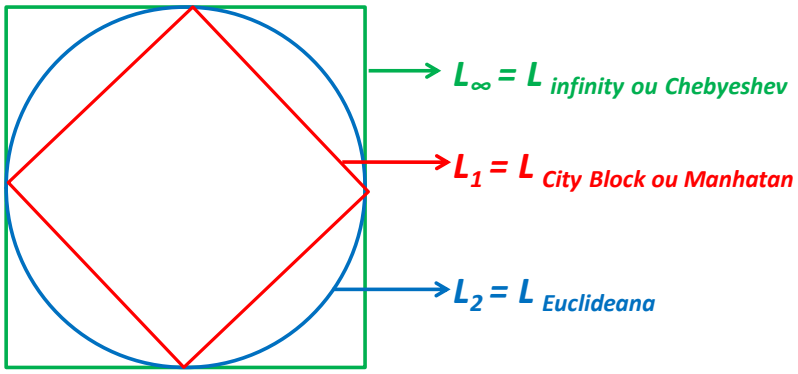


Figura 2.2: Métricas Minkowski

Conjuntos de palavras de uma linguagem - um domínio adimensional - são dados que não podem ser representados em espaços multidimensionais. Em conjuntos de sequências de caracteres, por exemplo, uma das funções de distância que pode ser usada para medir a similaridade entre duas palavras é a função de distância *Levenshtein*, conhecida como $L_{edit}(x,y)$ (BUENO, 2009). A $L_{edit}(x,y)$ indica a quantidade mínima de edições necessárias para transformar uma cadeia de caracteres em outra, ou seja, retorna a quantidade de caracteres que devem ser inseridos ou removidos para transformar uma palavra *x* em outra palavra *y*. Por exemplo, L_{edit}*("casa","assa") = 2*, é resultado de uma operação de remoção da letra *"c"* e outra operação de inserção da letra *"s"*.

2.4 **Consultas por Similaridade**

Em um espaço métrico, os dados são geralmente recuperados utilizando-se consultas por similaridade, pois a função de distância associada representa a dissimilaridade entre os elementos do espaço. Estas consultas envolvem um espaço, um elemento de consulta e um conjunto de parâmetros que é definido pelo tipo de consulta por similaridade em questão.

As consultas por similaridade mais usadas são as consultas por abrangência também conhecidas por *Range Query*, onde o objetivo é recuperar todos os elementos que sejam dissimilares do elemento de consulta até no máximo certo limite; e as consultas aos *K-vizinhos* mais próximos, cujo objetivo é recuperar os *k*

elementos mais similares ao elemento de consulta. Assim, dado um espaço métrico <S,*d*()> e um subconjunto de S = $t_1,t_2,...,t_n \subseteq$ M, uma definição formal para cada uma destas consultas é apresentada nas seções a seguir (BÖHM et al., 2001)(CHÁVEZ et al., 2001).

2.4.1 Range Query

Fornecido um elemento de consulta $s_q \in \mathbb{S}$, e um conjunto de elementos $S \subseteq \mathbb{S}$, uma função de distância *d()* e uma distância máxima (ou raio) de busca r_q, a consulta $RQ(s_q,r_q)$ recupera todos os elementos de *S* que estejam distantes do elemento s_q no máximo r_q. O subconjunto resposta resultante $S' \subseteq S$ é $\{s_i \in S \mid d(s_q,s_i) \leq r_q\}$.

Consultas por abrangência têm utilidade em vários tipos de aplicação. Um exemplo de consulta por abrangência pode ser *"Selecione todas as palavras que sejam diferentes da palavra p em até três caracteres, onde o universo S é o conjunto de palavras"*. Neste exemplo, o elemento de consulta s_q é a palavra *p*, o raio de busca r_q é igual a três caracteres, a métrica *d()* é a função L_{edit}, e S é o conjunto de dados contendo as palavras conhecidas. Assim, o conjunto de resposta para a consulta *RQ(p,3)* será o subconjunto $S' \subseteq S$ tal que $\{a \in S \mid d(a,p) <= 3\}$. A Figura 2.3 ilustra uma *RQ.*

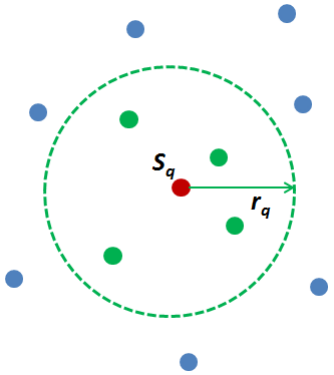


Figura 2.3: Consulta por abrangência

2.4.2 K-NN Query

Fornecido um elemento de consulta $s_q \in \mathbb{S}$, e um conjunto de elementos $S \subseteq \mathbb{S}$, uma função de distância *d()* e um número inteiro *k*, a consulta *K-NNQ(s_q,k)* recupera todos os *k* elementos mais próximos de s_q em *S*. Formalmente, o subconjunto resposta resultante $S' \subseteq S$ é:

$$\{s_i \mid s_i \in S \| S'\| = k \; e \; \forall s_i \in S', \forall s_j \in [S - S'], d(s_q,s_i) \leq d(s_q,s_j)\}$$

Utilizando o mesmo exemplo de palavras, uma consulta para *k-vizinhos* mais próximos é: *"Selecione as cinco palavras que sejam mais semelhantes à palavra p"*. É importante observar que não se utiliza mais o raio de busca r_q e sim *k* que é a quantidade limite de elementos para a resposta *K-NNQ(p,5)*. Também é importante ressaltar que ocorrerá um empate no caso de dois elementos possuírem a mesma distância do elemento de consulta. Quando ocorrer empate no elemento que completa o conjunto, ou seja, no *k-ésimo elemento*, a resposta pode conter apenas um dos elementos. Sendo assim, deve-se adotar uma política de escolha determinando se o conjunto resposta a ser retornado deve ou não ser maior que o solicitado. A Figura 2.4 ilustra uma *K-NNQ*.

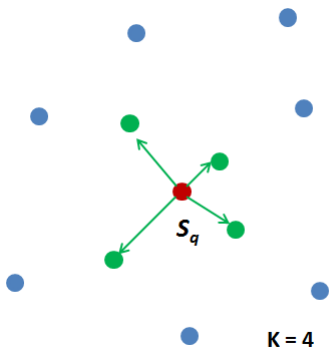


Figura 2.4: Consulta *K-vizinhos* mais próximos (KNNQ)

2.5 Índices Métricos

Vários métodos de acesso métrico (MAM) têm sido propostos com o objetivo de aumentar a eficiência, tanto na construção, quanto na forma como executam as operações de atualização e consultas de dados complexos.

Esses trabalhos começaram quando Burkhard e Keller (BURKHARD; KELLER, 1973) pela primeira vez apresentaram formas de particionamento recursivo

do espaço métrico por meio de representantes para guiar a busca. Neste trabalho, a primeira técnica proposta determina um elemento como representante de um conjunto e agrupa os demais elementos de acordo com suas distâncias para o representante.

Baseadas na primeira técnica surgiram vários trabalhos. Uhlman (UHLMANN, 1991) definiu duas formas de estruturar o domínio métrico através da decomposição por *"bolas" (ball decomposition)*. A primeira forma considera que uma *"bola"* métrica é uma região determinada por um *centroide* e um raio de cobertura. A segunda considera a divisão por *"hiperplanos generalizados" (generalized hyperplane decomposition)* produzindo uma divisão do espaço de busca sem sobreposições.

Como exemplo de árvores baseadas na decomposição por *"bolas"* métricas pode-se citar a Metric-tree de Uhlman (UHLMANN, 1991), a VP-tree *(Vantage point tree)* de Yanilos (YIANILOS, 1993) e a sua evolução MVP-tree *(Multi-Vantage Point tree)* (BOZKAYA; OZSOYOGLU, 1997), e também a FQ-tree *(Fixed Queries tree)*. Enquanto a VP-tree utiliza para divisão, um único elemento como *vantage-point* em cada nó, a MVP-tree utiliza mais de um *vantage-point, e a* FQ-tree utiliza um mesmo vantage-point para todos os elementos de um nível da árvore.

Utilizando a divisão de nós baseada em *hiperplanos generalizados* pode-se citar a GH-tree *(Generalized Hyperplane Decomposition tree)* (UHLMANN, 1991). Esta estrutura particiona recursivamente o conjunto de dados em dois, selecionando dois elementos como representantes e associando os demais ao seu representante mais próximo. Assim, os elementos mais à esquerda estarão associados ao primeiro representante e aqueles mais à direita ao segundo. A GNAT *(Geometric Near-neighbor Acess tree)* (BRIN, 1995) é uma extensão da GH-tree, permitindo a escolha de mais do que dois representantes por nó e armazenando as distâncias mínimas e máximas para elementos de todos os outros representantes, aumentando assim a capacidade de poda pela desigualdade triangular.

Os MAMs ainda podem ser classificados como estáticos ou dinâmicos de acordo com o suporte à operação de inserção após a criação da estrutura, como também ainda podem ser classificados conforme o armazenamento da estrutura em memória primária ou em memória secundária (tipicamente em disco magnético) (POLA, 2005). Os MAMs dinâmicos permitem operações de inserção ou remoção de elementos após a construção da estrutura. Todos os MAM citados até aqui são estáticos, e por não permitirem inserções ou remoções após sua construção, todo o

conjunto de dados deve estar disponível no momento da criação da estrutura de indexação.

A M-tree (CIACCIA et al., 1997) foi o primeiro MAM dinâmico apresentado na literatura. A Slim-tree (TRAINA JR. et al., 2000) (TRAINA JR. et al., 2002) aperfeiçoou a M-tree, propondo a primeira técnica de medição e redução de sobreposição de ramos da árvore. Esta estrutura ainda possui o algoritmo *Slim-Down*, que permite reorganizar a estrutura em qualquer momento, com o objetivo de reduzir a sobreposição entre as subárvores. Após sua construção, a M-tree e a Slim-tree não possuem algoritmo para a remoção efetiva dos elementos, apenas sinalizam que os elementos estão logicamente removidos (BUENO, 2009).

A MM-tree (POLA et al., 2007) indexa dados métricos em memória primária. A sua estrutura divide o espaço em bolas que se intersectam no espaço métrico e possui técnicas que controlam o balanceamento da estrutura.

A Onion-tree (CARÉLO et al., 2009) (CARÉLO et al., 2011) é uma extensão da MM-tree, porém segue novos padrões de construção e particionamento do espaço. Esta estrutura leva em consideração o tamanho da região externa ao nó, e quando necessário aplica uma quantidade maior de particionamentos, produzindo uma melhor cobertura do espaço pelo nó em questão conseguindo com isso um melhor desempenho no processamento de consultas por similaridade (i.e. tanto para *range query* quanto para *KNN query*).

2.6 **MM-tree**

A MM-tree *(Main Memory Metric-tree)* (POLA et al., 2007) é um MAM dinâmico e foi desenvolvido visando atender a necessidade de responder rapidamente as consultas por similaridade sobre conjuntos de dados em espaços métricos que possam ser mantidos em memória primária.

Estrutura de Dados

A base de construção da estrutura MM-tree é recursivamente dividir o espaço em quatro regiões distintas utilizando dois elementos s_1 e s_2 (chamados de *pivôs*) para representá-las. Cada elemento inserido s_i é associado a uma única região. A determinação da região é feita com base na distância entre o elemento em inserção

e os pivôs representantes do nó como indicado na Tabela 2.1 onde *r* é a distância entre os pivôs, ou seja, $r = d(s_1, s_2)$.

Tabela 2.1: MM-tree - Regiões do nó

$d(s_i, s_1)$	$d(s_i, s_2)$	
		Região
r	***r***	
<	<	*I*
<	≥	*II*
≥	<	*III*
≥	≥	*IV*

Todos os nós da MM-tree têm a mesma estrutura, que é definida como segue:

$$Nó\ MM\text{-}Tree = [s_1, s_2, d(s_1, s_2), Ptr_1, Ptr_2, Ptr_3, Ptr_4]$$

Onde s_1 e s_2 são os pivôs, $d(s_1, s_2)$ é a distância entre os pivôs e Ptr_1, Ptr_2, Ptr_3, Ptr_4 são os ponteiros para as quatro regiões que armazenam os elementos filhos no nó, determinadas pela Tabela 2.1. Cada nó possui apenas dois elementos, os mesmos que são utilizados como representantes de suas regiões.

Vale ainda observar que s_1 e s_2 não são os elementos e sim os vetores de característica que os representam. Por exemplo, em imagens não serão indexados os pixels e sim o vetor da característica de interesse, por exemplo, o vetor contendo o histograma de cores.

Uma ilustração desta estrutura pode ser observada na Figura 2.5.

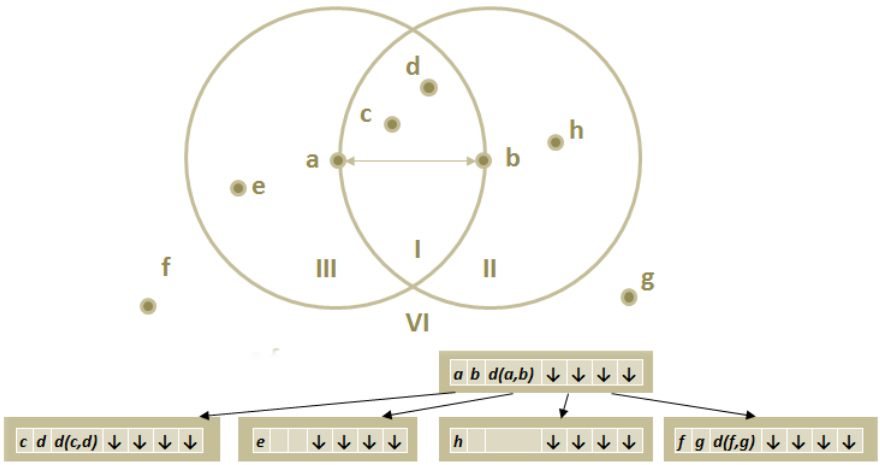


Figura 2.5: MM-Tree com oito elementos (Figura adaptada de (POLA, 2010)**)**

Na Figura 2.5, no plano superior pode-se observar a representação gráfica dos elementos no espaço, e no plano inferior pode-se observar a estrutura dos nós da MM-tree. Na representação gráfica do espaço verificamos os pivôs *a* e *b* e o raio *r* entre eles, ou seja, a distância *d(a,b)*. O raio centrado em cada pivô determina a área circular de sua abrangência. A intersecção das áreas cobertas por cada pivô configura as regiões hierarquicamente subordinadas a estes pivôs conforme descrito na Tabela 2.1.

Na representação da estrutura de um nó MM-tree pode-se observar a hierarquia gerada, contendo no primeiro nível os pivôs *a* e *b*, o valor do raio *r* entre eles, ou seja, a distância *d(a,b)*, e os ponteiros para as quatro regiões subordinadas. No segundo nível verificamos os nós correspondentes aos elementos filhos *c*, *d*, *e*, *f*, *g*, e *h*.

Inserção

É importante observar que a MM-tree é uma estrutura dinâmica, e inserções podem ocorrer a qualquer momento.

A inserção de elementos na MM-tree sempre é feita apenas em nós folhas. A partir da raiz e utilizando a Tabela 2.1 de determinação de regiões, o algoritmo de inserção percorre a estrutura hierárquica procurando pelo nó apropriado para armazenar o novo elemento. Assim, a cada nó visitado, o algoritmo utiliza as informações armazenadas do nó para consultar a Tabela 2.1 e determinar qual caminho percorrer (i.e., qual região o elemento deve ser inserido e, portanto, qual sub-árvore percorrer). Este procedimento é executado até que se encontre um nó folha para a inserção. Caso o elemento já se encontre na estrutura o algoritmo de inserção é interrompido de maneira a evitar duplicidade de elementos.

Um fator a ser considerado em operações de inserção, principalmente em estruturas dinâmicas em que a inserção pode ser feita após a construção da hierarquia, é o controle do balanceamento da estrutura. As buscas em árvores desbalanceadas na altura produzem um comportamento imprevisível. Para isso, a MM-tree possui um algoritmo apropriado para manter a estrutura balanceada tanto quanto possível.

O algoritmo de semi-balanceamento da MM-tree atua somente nos nós em que a inserção irá ocorrer, ou seja, nos nós folha. A Figura 2.6 e a Figura 2.7 ilustram o funcionamento deste algoritmo.

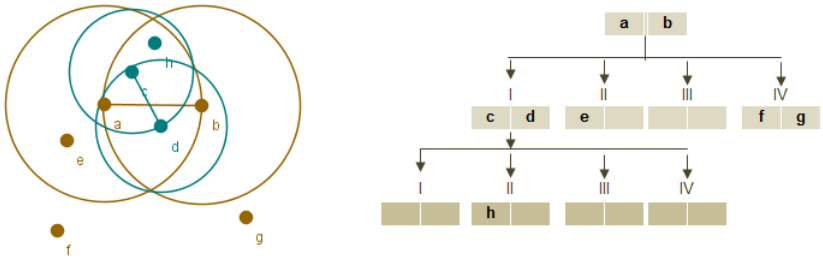


Figura 2.6: MM-tree - Algoritmo de semi-balanceamento: Situação inicial

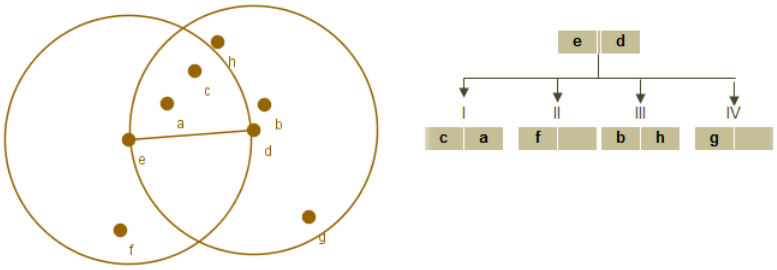


Figura 2.7: MM-tree: Algoritmo de semi-balanceamento: Situação final

O plano esquerdo da Figura 2.6 ilustra a inserção de um novo elemento – o elemento *h* - sem a aplicação do algoritmo de semi-balanceamento. Embora havendo espaço nos nós, o raio dos atuais pivôs determina (através da Tabela 2.1) que esta inserção crie um novo nível hierárquico na estrutura.

Na Figura 2.7, pode-se visualizar o resultado da aplicação do algoritmo de semi-balanceamento. Os pivôs *a* e *b* foram trocados pelos pivôs *e* e *d*. Os demais elementos foram realocados conforme a Tabela 2.1 de determinação de regiões e não foi mais necessária a criação de um novo nível hierárquico.

Quando houver espaço em nós irmãos, e for identificada a necessidade de criação de um novo nível hierárquico, o algoritmo executa uma análise combinatória de todos os elementos da hierarquia, determinando novos pivôs. Os pivôs são escolhidos de maneira a se evitar a criação de um novo nível hierárquico, determinando um melhor arranjo dos elementos. Se a análise combinatória não puder determinar um melhor arranjo, cria-se um novo nível hierárquico.

Busca

Além das consultas pontuais, a MM-tree responde às consultas *Range Query (RQ)* e também às consultas *K-vizinhos mais próximos (KNNQ)*. Esta seção abordará estas consultas.

Os dois algoritmos de consulta se utilizam da mesma informação para determinar quais regiões intersectam a região de consulta. Esta intersecção é determinada através da Tabela 2.2.

Os parâmetros da consulta *RQ* são o elemento base para a consulta s_q e o raio r_q. Em cada nó visitado, se a distância de s_q ao pivô for menor que r_q então o pivô é adicionado ao conjunto da resposta. A seguir, o algoritmo visita as regiões, subordinadas ao pivô, em que há interseção do raio r_q seguindo a Tabela 2.2 de regiões do nó MM-tree.

Tabela 2.2: MM-tree - Determinação do caminho de busca

Região I	$(d(s_q, s_2) < r_q + r) \wedge (d(s_q, s_1) < r_q + r)$
Região II	$(d(s_q, s_2) + r_q \geq r) \wedge (d(s_q, s_1) < r_q + r)$
Região III	$(d(s_q, s_2) < r_q + r) \wedge (d(s_q, s_1) + r_q \geq r)$
Região IV	$(d(s_q, s_2) < r_q \geq r) \wedge (d(s_q, s_1) + r_q \geq r)$

Na consulta *KNNQ* o raio de consulta é dinâmico. Os parâmetros desta consulta são o elemento base para a consulta s_q, e a quantidade desejada k de elementos mais próximos ao elemento base. O algoritmo trabalha com duas informações dinâmicas: o raio r_q e uma lista de elementos que farão parte da resposta. Inicia a partir da raiz da hierarquia com o valor de raio $r_q = \infty$, e com lista vazia. Cada vez que é encontrado um elemento mais próximo de s_q, ou seja, encontrado um elemento cuja distância esteja entre os K elementos mais próximos, o raio dinâmico é reduzido. Isso ocorre pela eliminação do elemento mais distante na lista (k-ésimo elemento) e a recomputação do raio com base no novo elemento

inserido na lista. O raio passa a ser a distância do elemento mais distante na lista ao elemento base. Assim como o algoritmo da consulta *RQ*, o algoritmo *KNNQ* também consulta a tabela de sobreposições em busca das intersecções que deve continuar percorrendo.

2.7 **Onion-tree**

O particionamento do espaço métrico é fator determinante no desempenho de um MAM no processamento de consultas por similaridade. Observando-se o particionamento gerado pela MM-tree (Figura 2.5) verifica-se uma discrepância entre as áreas de cobertura das regiões geradas. A região IV, ou seja, a região mais externa do nó possui uma área de cobertura muito extensa. A Onion-tree surgiu como uma evolução da MM-tree, em busca de um melhor particionamento do espaço métrico (CARÉLO et al., 2009) (CARÉLO et al., 2011).

Usando somente a multiplicidade do raio, portanto sem aumentar os cálculos de distância para a sua construção, a Onion-tree gera um melhor particionamento da região externa do nó. Outra característica é que esta estrutura de indexação é capaz de flexibilizar a política de particionamento aplicada em sua construção. Ao gerar cada nó, a Onion-tree pode analisar a necessidade ou não de gerar mais regiões para uma melhor cobertura do espaço, ou seja, é possível fazer com que os nós tenham diferentes quantidades de regiões, possibilitando maior eficiência nas consultas por meio da redução da quantidade de cálculos de distância (CARÉLO et al., 2009). A possibilidade de alterar a política de particionamento das regiões amplia as possibilidades de análise do comportamento e ajuste da estrutura em relação a vários domínios de dados.

Estrutura de Dados

A Onion-tree pode dividir o espaço métrico de cada nó em mais que quatro regiões disjuntas. Tendo como base a estrutura da MM-tree, inicialmente a Onion-tree gera quatro regiões. A Figura 2.8 ilustra a dinâmica do particionamento gerado pela Onion-tree.

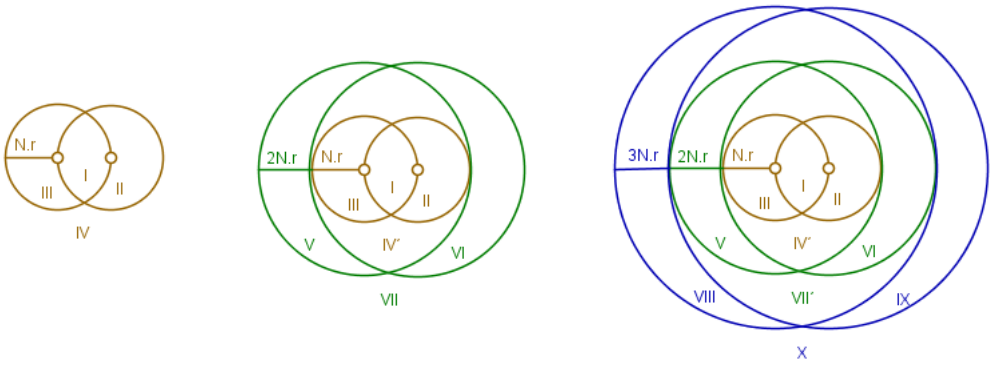


Figura 2.8: Onion-tree - Particionamento do nó

No plano esquerdo da Figura 2.8 visualiza-se a estrutura inicial do nó, gerado a partir do valor do raio entre os dois pivôs, determinando as quatro regiões *I*, *II*, *III* e *IV*. No centro da Figura 2.8, visualiza-se a aplicação da primeira expansão do nó. *A* Onion-tree duplica o valor inicial do raio entre os pivôs, e com isso, a região IV, a região mais externa, é particionada em quatro regiões – IV´, *V, VI* e *VII*. Após a terceira expansão, que triplica o valor inicial do raio entre os pivôs, o plano direito da Figura 2.8 apresenta a situação final do nó, com a região *VII* particionada em quatro regiões – VII´, *VIII, IX* e *X*. Esta dinâmica pode ser repetida até que o particionamento da região mais externa do nó seja considerado satisfatório.

Esta política de particionamento para os nós da Onion-tree é denominada *keep-small strategy,* pois o objetivo é manter a região externa o menor possível, sendo aplicada quando o raio entre os pivôs do nó é menor que a metade do raio de seu nó pai. Como se pôde observar, a política de expansão do nó exerce papel principal na construção da estrutura.

Para que se possa estudar o comportamento da estrutura em vários domínios de dados, e também em face às várias dimensionalidades, a Onion-tree oferece duas opções para a política de expansões a ser aplicada ao nó: A *F-Onion-tree* com quantidade fixa de expansões (ou particionamento) para todos os seus nós, e a *V-Onion-tree* com quantidade variável de expansões para cada nó da Onion-tree. Portanto, a *F-Onion-tree* aplica a mesma quantidade de expansões em todos os nós. Já a *V-Onion-tree* - com base na política *keep-small strategy* - analisa a cada nó a necessidade ou não de mais expansões para uma melhor cobertura do espaço. Na opção *F-Onion-tree* ainda é possível informar a quantidade de expansões que se deseja aplicar à construção da árvore, ampliando as possibilidades de análise do

comportamento da estrutura em relação a vários tipos de dados e dimensionalidades.

Os nós da Onion-tree possuem a estrutura definida como segue:

$$Nó\ Onion\text{-}tree = [s_1, s_2, d(s_1, s_2), Q_e, Q_r, [Ptr_1, Ptr_2, Ptr_3, Ptr_4 ....]]$$

Onde s_1 e s_2 são os pivôs representados por seus vetores de características, $d(s_1, s_2)$ é a distância entre os pivôs, Q_e é a quantidade de expansões aplicadas ao nó, Q_r é a quantidade de regiões do nó e $[Ptr_1, Ptr_2, Ptr_3, Ptr_4 ....]$ é um vetor de ponteiros para as regiões que armazenam os elementos filhos no nó.

Inserção

A Onion-tree é um MAM dinâmico. E por aceitar inserções após a construção da estrutura, possui uma política de substituição de pivôs em nós pai de nós folha *(Replacement Technique)* que é aplicada na inserção de um novo elemento e identifica se o subespaço do nó pode ser melhor particionado a partir de outros representantes. O algoritmo realiza uma análise combinatória entre o elemento em inserção e os dois pivôs do nó folha, e aplica uma política de substituição de pivôs. A Onion-tree possui três opções para a política de substituição de pivôs em busca de um melhor particionamento do espaço: *Keep-small*, *Maximize-expansions* e *Minimize-expansions*. A Figura 2.9 ilustra a substituição de pivôs em operação de inserção.

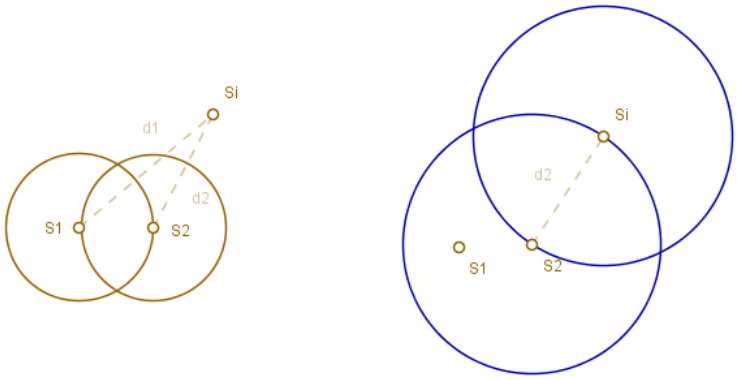


Figura 2.9: Onion-tree - Substituição de pivôs

A política *Keep-small* de substituição de pivôs mantém o mesmo objetivo da política *Keep-small* usada na construção da árvore. Nesta política, a escolha dos novos pivôs determina que a região externa ao nó seja a menor possível. A política

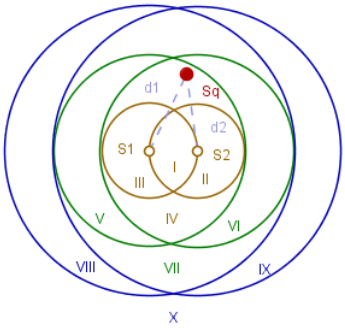
Maximize-expansions seleciona os pivôs mais próximos, maximizando a quantidade de expansões do nó, e a política *Minimize-expansions* seleciona como pivôs os elementos mais distantes, minimizando a quantidade de expansões aplicadas ao nó. A Tabela 2.3 contém um resumo das opções disponíveis para a construção da estrutura de indexação da Onion-tree.

Tabela 2.3: Onion-tree - Resumo das opções para a construção

Expansion Procedure	*Fixa: F-Onion-tree*	*Aplica uma quantidade fixa de expansões a todos os nós*
	Variável: V-Onion-tree	*Varia a quantidade de expansões do nó conforme a necessidade*
Replacement Technique	*Keep-small*	*Manter a região externa do nó o menor possível*
	Maximize expansions	*Maximiza a quantidade de expansões aplicadas ao nó*
	Minimize expansions	*Minimiza a quantidade de expansões aplicadas ao nó*

Busca

Em face a toda dinâmica de construção e particionamento do espaço métrico proposta pela Onion-tree, seus algoritmos de busca também foram necessariamente estendidos. Os algoritmos da MM-tree foram concebidos para considerar uma quantidade fixa de quatro regiões em cada nó. Dependendo da parametrização de sua construção, a Onion-tree pode gerar uma diversidade de configurações de regiões em seus nós. Assim, os algoritmos para *Range Query (RQ)* e *K-vizinhos mais próximos (KNNQ)* foram estendidos para considerar qualquer quantidade de regiões. Além disso, em busca de maior eficiência, o algoritmo *K-vizinhos mais próximos (KNNQ)* ainda introduz uma nova sequência de visitação para as regiões do nó. A Figura 2.10 ilustra a ordem de visitação dos nós e a Tabela 2.4 apresenta como se determina a região a percorrer em uma consulta *KNNQ*.



KNNQ - Ordem de visitação das regiões dos nós
1) **Expansão 1: Regiões IV,VI,V**
2) **Expansão 0: Regiões I,III,II**
3) **Expansão 2: Regiões VII, IX, VIII, X**

Figura 2.10: Onion-tree: KNNQ - Sequência de visitação dos nós

Tabela 2.4: Onion-tree: KNNQ - Sequência de visitação dos nós

Região de S_q mod 3	Condição	Ordem de visitação			
		1º	2º	3º	4º
1	$d_1 \leq d_2$	3E + 1	3E + 2	3E + 3	3E + 4
1	$d_1 > d_2$	3E + 1	3E + 3	3E + 2	3E + 4
2	$d_2 - R \leq R - d_1$	3E + 2	3E + 1	3E + 4	3E + 3
2	$d_2 - R > R - d_1$	3E + 2	3E + 4	3E + 1	3E + 3
3	$d_1 - R \leq R - d_2$	3E + 3	3E + 4	3E + 1	3E + 2
3	$d_1 - R > R - d_2$	3E + 3	3E + 1	3E + 4	3E + 2
4	$d_1 \leq d_2$	3E + 4	3E + 2	3E + 1	3E + 3
4	$d_1 > d_2$	3E + 4	3E + 3	3E + 1	3E + 2

Para um elemento s_q, as expansões são visitadas na seguinte ordem: (i) expansão *E* na qual o elemento base s_q se encontra; (ii) expansões *E - 1* e *E + 1*; (iii) expansões *E - 2* e *E + 2* e assim por diante. Esta sequência de visitação, partindo de uma região mais interna (por exemplo, *E - 1*), para outra externa (*E + 1*) permite uma redução mais rápida do raio dinâmico da *K-NN*, uma vez que expansões internas são menores e possuem maior probabilidade de conter elementos próximos. Além disso, esta ordem de visitação otimiza a aplicação da poda de elementos.

Capítulo 3

TRABALHOS CORRELATOS

Geralmente o termo dinamicidade em armazenamento de dados remete à capacidade do SGDB de realizar operações de atualização e remoção de dados após a inserção inicial de um conjunto de dados. Em indexação métrica, porém, esse termo tem designado MAMs que permitem novas operações de inserção após a construção da estrutura de indexação, ou seja, após a inserção inicial de um conjunto de dados. Se considerarmos o significado mais amplo da dinamicidade, incluindo operações de atualização, e especialmente a operação de remoção, são poucos os MAMs que possuem algoritmos para essas operações. Desta forma, um grande número de MAMs existentes na literatura são considerados estáticos.

Em árvores métricas, de forma inerente à sua construção, a alteração das distâncias entre os elementos representantes de um nó necessariamente implica em reorganização da hierarquia do MAM. Remoções realizadas em nós folha ou em nós pai de folha geralmente produzem efeitos locais na estrutura de um MAM, enquanto remoções em nós internos, especialmente de níveis próximos ao nível da raiz, podem produzir efeitos globais na estrutura de um MAM requerendo a reorganização de uma grande quantidade de nós.

Neste capítulo, serão apresentados os MAMs relevantes a esta pesquisa de mestrado que implementam a remoção física de elementos no índice, e como estas questões são tratadas por estes MAMs. Desta forma, a seção 3.1 descreve o algoritmo de remoção aplicado na Slim-tree e a seção 3.2 descreve o algoritmo de remoção aplicado na Dynamic VP-tree. O capítulo é finalizado na seção 3.3, com um quadro comparativo da dinamicidade entre as árvores métricas Slim-tree, Dynamic VP-tree e Onion-tree, e com as considerações finais.

3.1 **Remoção na Slim-tree**

A Slim-tree (TRAINA JR. et al., 2000) é uma árvore multivias balanceada na altura, dinâmica e baseada em disco, cujo particionamento do espaço métrico gera

regiões não disjuntas. A Slim-tree trabalha com páginas de tamanho fixo e os elementos são inseridos sempre nas folhas em uma construção das folhas para a raiz *(bottom-up)*, assim como em uma B-tree. Esta estrutura utiliza um único elemento que representa cada particionamento do espaço de indexação. Com exceção do nó raiz, todos os outros nós possuem um representante. Assim, em uma visão de cima para baixo *(top-down)* temos uma hierarquia de representantes com os elementos nas folhas. Desta forma, os nós desta árvore podem ser de dois tipos: nó índice *(indexNode)*, e nó folha *(leafNode)*.

O algoritmo de remoção da Slim-tree (BUENO, 2009) inicia executando uma busca pontual pelo elemento a ser removido, e durante esta busca o caminho percorrido do nó raiz ao nó que possui o elemento a ser removido é armazenado. Ao encontrar o elemento, o algoritmo executa a remoção e inicia um procedimento recursivo *"bottom-up"* de manutenção das características da árvore. As informações armazenadas em cada nível do caminho são usadas a cada chamada recursiva em direção aos níveis superiores para executar as ações que devem ser realizadas conforme a situação propagada pelo nível inferior. E estas ações podem ser de três tipos:

- Não há alterações a serem realizadas na estrutura de representantes de um nível superior;
- Necessidade da troca do representante. Houve a troca do representante do nível inferior, portanto, é preciso substituir o representante do nó por um novo representante;
- A subárvore do nível inferior foi removida, portanto, é necessário remover sua entrada no nó índice.

Os três tipos de ações anteriormente descritos tentam preservar a taxa de ocupação mínima *(TOM)*, porém as ações para tal dependem do tipo do nó em questão. Quando a *TOM* é violada em um nó folha, o nó é removido e os demais elementos são posteriormente reinseridos em outras folhas. Quando esta violação ocorre em um nó-índice, procura-se importar a entrada necessária para manter a *TOM*, de algum nó índice irmão sem que isso viole sua própria *TOM*. Primeiramente, tenta-se encontrar uma entrada disponível sem aumento do raio de cobertura do nó importador. Caso isso não seja possível, procura-se pelo menos minimizar o aumento deste raio. Quando não for possível localizar uma entrada sem violar a *TOM* do nó exportador, isso significa que todos os nós irmãos possuem a taxa de

ocupação mínima, portanto, as entradas do nó que sofreu a remoção e violou sua *TOM*, podem ser exportadas a seus nós irmãos.

Quando o representante de um nó é removido, a sua substituição procura minimizar a variação do raio de cobertura do nó e a sua entrada correspondente no nível superior é atualizada. Quando a remoção determina que a raiz fique com apenas uma entrada, violando sua *TOM* que é de no mínimo 2, a raiz é eliminada e esta entrada passa a ser a nova raiz, diminuindo a altura da árvore.

Para minimizar a reorganização da árvore, ainda se aplica uma concessão *"controlada"* na manutenção da *TOM*. Quando o nó-índice se encontra na metade superior da árvore e todos os irmãos estão com suas ocupações mínimas, então a violação da *TOM* é permitida até um limite definido. Esta concessão baseia-se no fato de que a proporção dos nós na primeira metade da árvore é pequena e a violação da TOM não é, portanto, crítica.

Após vários experimentos deste algoritmo de remoção da Slim-tree, intercalando remoções, inserções e consultas diversas, foi percebida uma melhora no nível de sobreposição da árvore resultante. O estudo deste fato resultou em outra aplicação do algoritmo de remoção. Sua aplicação foi expandida para um procedimento de reorganização da árvore, através da qual a remoção propositada de elementos específicos acompanhada de sua posterior reinserção melhora a organização da Slim-tree e consequentemente melhora o seu desempenho no processamento de consultas por similaridade.

O método proposto tem como foco os elementos que estejam na periferia, ou seja, longe do centro do nó em que ele é armazenado. A remoção destes elementos causa a diminuição do raio de um nó, diminuindo também a chance de sobreposição com outros nós e aumentando o desempenho no processamento de consultas.

O algoritmo *Slim-down* (TRAINA JR. et al., 2002) diminui a sobreposição da árvore trabalhando em nós folha de uma mesma subárvore. A nova técnica, chamada *Push-pull* (BUENO, 2009) remove efetivamente uma quantidade definida de elementos que estejam na periferia de nós folha de quaisquer subárvores. Ademais, a reinserção destes elementos busca o menor aumento de raio, priorizando o representante que esteja seja mais próximo do elemento sendo reinserido. Portanto, o diferencial dessa técnica, é utilizar o algoritmo de remoção efetiva, e a mobilidade de elementos entre quaisquer nós folha, não limitada a nós

irmãos como na técnica *Slim-down,* para uma reorganização que possibilita melhorar a organização da árvore.

A Figura 3.1 ilustra a aplicação da técnica *Push-pull*. Neste exemplo, considera-se a ocupação mínima de dois elementos e a ocupação máxima de oito elementos. O quadrante (*a)* da figura contém a situação inicial com dois nós índice, cada um com três nós folha. O quadrante (*b)* demonstra os elementos a serem removidos assinalando-os em forma de estrela. O quadrante (*c)* demonstra a situação após a remoção dos elementos, e o quadrante (*d)* a situação após a reinserção dos mesmos elementos. Na situação final pode-se visualizar a redução da sobreposição.

O algoritmo *Push-pull* utiliza uma quantidade definida de remoções por nó folha. Também foi desenvolvida uma técnica para determinar a quantidade ótima de remoções a ser usada. O cálculo do nível de sobreposição da árvore (*fat-factor*) foi modificado para gerar as informações necessárias, e com base nas estatísticas obtidas por este cálculo, a técnica *Smart Push-pull* determina a quantidade adequada de remoções a ser aplicada na execução da técnica *Push-pull*.

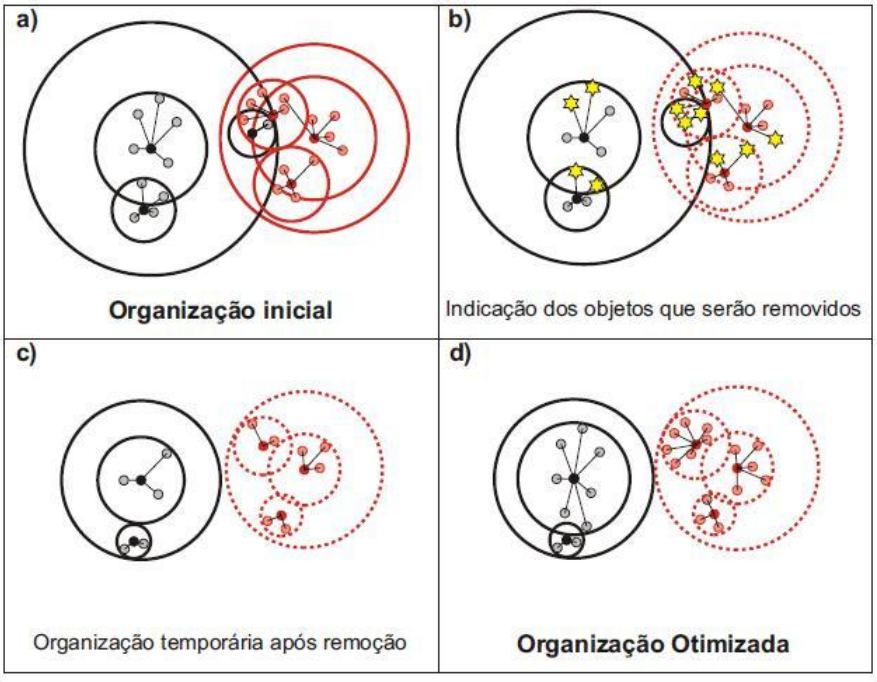


Figura 3.1: Slim-tree: Execução da técnica Push-pull removendo dois elementos por nó folha (BUENO, 2009)

3.2 **Dynamic VP-tree**

A *Dynamic VP-tree* (FU et al., 2000) é uma extensão dinâmica da *VP-tree.* A *VP-tree* é uma árvore estática, e a dinamicidade proposta pela *Dynamic VP-tree* envolve além da inserção também a remoção de elementos após a construção da estrutura.

A primeira característica na extensão à VP-tree é a quantidade de filhos permitidos em um nó interno. Como em uma *B-tree*, esta estrutura adota uma quantidade mínima e uma quantidade máxima de filhos para um nó interno. Sendo *m* a quantidade máxima (de pelo menos 2) a quantidade mínima deve ser no máximo *m*/2. O nó raiz deve ter pelo menos 2 filhos e no máximo *m* filhos, e as quantidades mínimas e máximas em um nó folha são determinadas em função do tamanho da página de disco, ou seja, em função da quantidade de elementos que podem ser armazenados em um nó.

Inserção

A inserção de um novo elemento na Dynamic VP-tree ocorre sempre nas folhas. O algoritmo percorre a árvore calculando em cada nível, a distância entre o elemento sendo inserido e o *vantage-point* do nó. O percurso é determinado pela subárvore cujo raio de abrangência contenha o elemento sendo inserido. Ao encontrar o nó folha adequado à inserção, caso o nó possua espaço a inserção é concretizada.

Quando o nó folha não possui espaço para um novo elemento, verifica-se se há espaço em algum nó irmão. Neste caso, realiza-se uma melhor distribuição de todos os elementos de todas as folhas do nó pai, de forma que se possa inserir o novo elemento na folha em que deve ser armazenado. Os elementos são organizados de acordo com suas distâncias ao *vantage-point* e redistribuídos, em igual cardinalidade, entre as mesmas folhas. Após a redistribuição, as distâncias e as quantidades mínimas e máximas do nó pai são atualizadas.

A Figura 3.2 ilustra a situação inicial da estratégia, e a Figura 3.3 ilustra a situação final, após a aplicação desta estratégia. Na Figura 3.2, o elemento *e'* necessita ser inserido no nó folha *L*, o qual não possui mais espaço para a inserção. Contudo existe espaço disponível em seus irmãos também folha. Na Figura 3.3

pode-se visualizar que o elemento *e'* foi inserido na folha *L* após uma distribuição dos outros elementos entre as mesmas folhas.

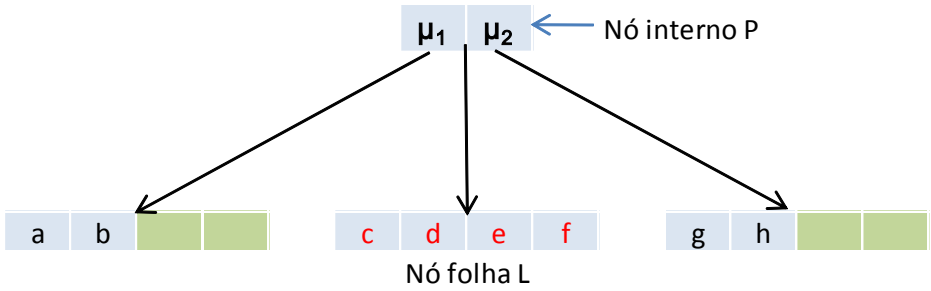


Figura 3.2: VP-tree - Inserção Situação inicial: Não há espaço na folha apropriada, porém há espaço em uma folha irmã (Figura adaptada de (FU et al., 2000)**)**

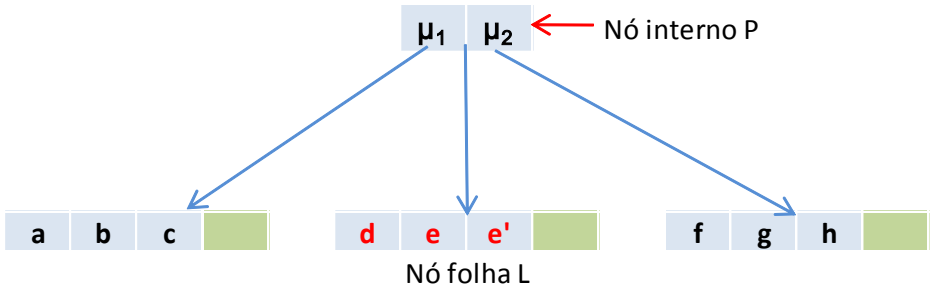


Figura 3.3: VP-tree - Inserção: Situação final: Os elementos foram redistribuídos entre as folhas, e o novo elemento inserido (Figura adaptada de (FU et al., 2000)**)**

Quando o nó folha escolhido para inserção de um novo elemento, assim como os seus nós irmãos, não possuem espaço para um novo elemento, mas o nó pai possui espaço para mais um filho, o nó folha apropriado para a inserção é particionado. Os elementos são organizados de acordo com suas distâncias ao *vantage-point* e redistribuídos, em igual cardinalidade, entre a folha antiga e a nova. Após a redistribuição, as distâncias e as quantidades mínimas e máximas do nó pai são atualizadas.

A Figura 3.4 ilustra a situação inicial estratégia, e a Figura 3.5 ilustra a situação final, após a aplicação desta estratégia. Na Figura 3.4 o elemento *e'* necessita ser inserido no nó folha *L* e não há espaço disponível entre seus nós irmãos folha. Contudo, a quantidade máxima de filhos de *P*, pai de *L*, ainda não foi atingida, e *P* ainda pode gerar mais um nó filho. Na Figura 3.5, pode-se visualizar que o nó *P* gerou novos nós folha (L_1 e L_2) através do particionamento do nó *L*. O elemento *e'* e os elementos de *L*, foram então redistribuídos entre L_1 e L_2.

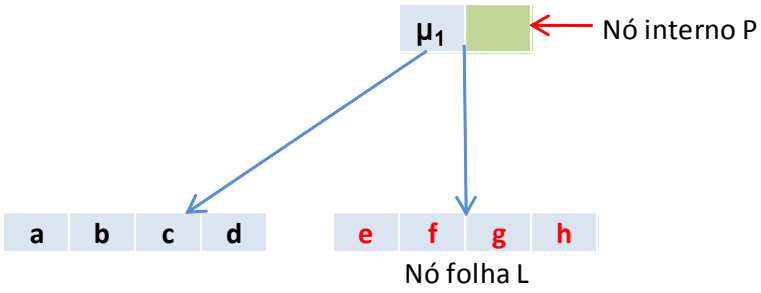


Figura 3.4: VP-tree - Inserção Situação inicial: Não há espaço na folha apropriada, porém o nó pai possui espaço para um novo filho (Figura adaptada de (FU et al., 2000)**)**

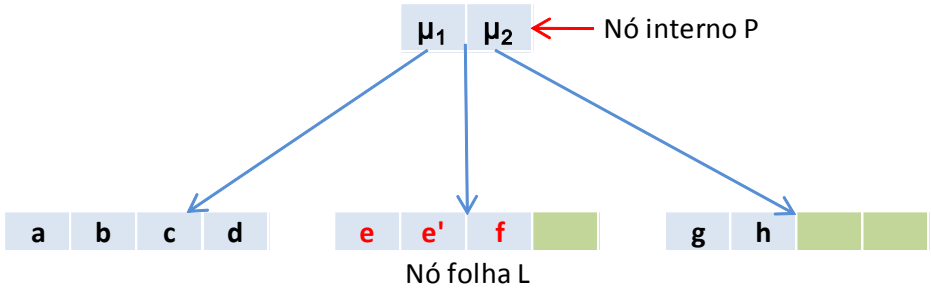


Figura 3.5: VP-tree - Inserção Situação final: Não há espaço na folha apropriada, porém o nó pai possui espaço para um novo filho (Figura adaptada de (FU et al., 2000)**)**

Quando houver espaço em uma subárvore do ancestral imediato de seu pai, deve-se redistribuir os elementos de todas as folhas da subárvore do ancestral, de maneira que se possa inserir o novo elemento na folha apropriada.

Nesta estratégia, ainda pode-se encontrar duas situações: (i) existe espaço em pelo menos uma folha na hierarquia ancestral; (ii) não existe espaço em uma folha na hierarquia ancestral, porém o ancestral pode ainda gerar mais um filho. A Figura 3.6 e a Figura 3.7 ilustram a situação (i). O elemento *e'* necessita ser inserido na folha *L* e não há espaço nas folhas irmãs. Contudo, seu ancestral *A* possui um filho *C* em cujas folhas ainda existe espaço. Na Figura 3.7 pode-se visualizar a inserção do elemento *e'* na folha *L*, e a redistribuição dos demais elementos, que provocou inclusive a movimentação dos elementos *i* e *g* para o ramo da subárvore do nó *C*. Essa movimentação provavelmente ocorreu em função das distâncias destes elementos ao *vantage-point* dos nós *B* e *C*. A quantidade de elementos a ser recolocada em cada subárvore é calculada com base na média de elementos

armazenados em cada subárvore. E a redistribuição dos elementos é feita com base em suas distâncias ao *vantage-point*.

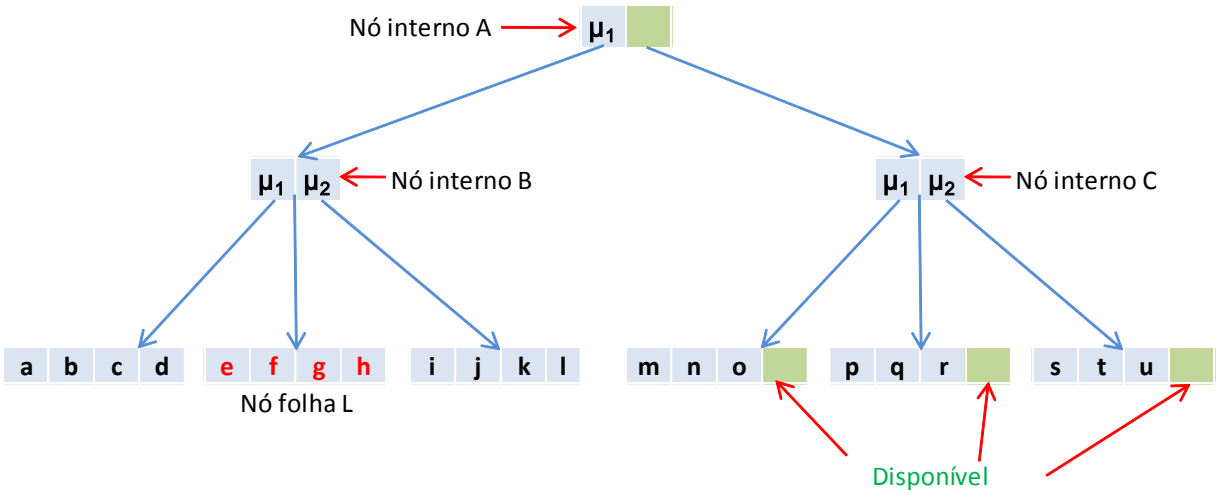


Figura 3.6: VP-tree - Inserção Situação Inicial: Não há espaço na folha apropriada, porém existe espaço em uma folha ancestral - Figura adaptada de (FU et al., 2000)

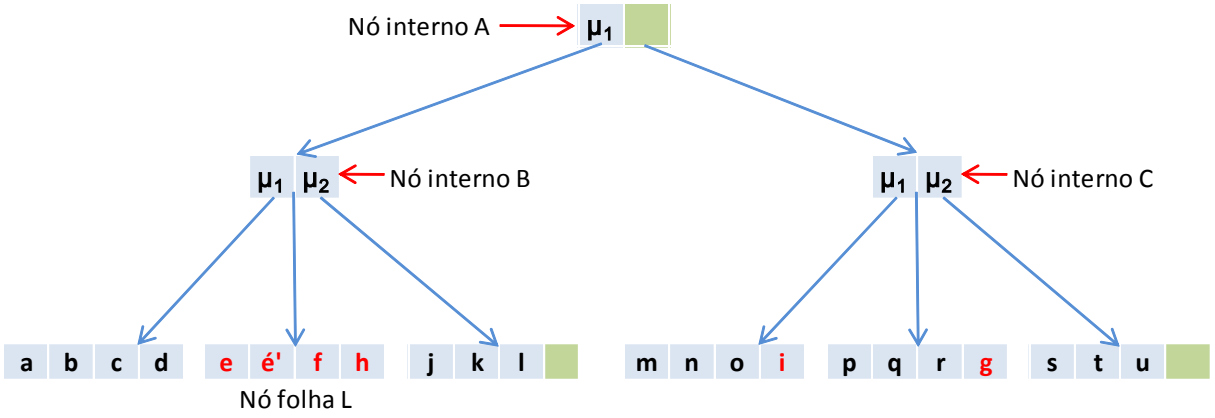


Figura 3.7: VP-tree - Inserção Situação final: Não há espaço na folha apropriada, porém existe espaço em uma folha ancestral - Figura adaptada de (FU et al., 2000)

As figuras Figura 3.8 e Figura 3.9 ilustram a situação (ii). O elemento *e'* necessita ser inserido na folha *L* e não há espaço nem em suas folhas ancestrais irmãs. Contudo, seu ancestral *A* ainda pode gerar mais um filho. Na Figura 3.9 pode-se visualizar que além do nó *A* gerar mais um filho, a redistribuição dos elementos da subárvore do nó *A* foi realizada através do particionamento do nó *B*, gerando os nós B_1 e B_2. Houve a inserção do elemento *e'*, e a redistribuição dos demais elementos entre os novos nós gerados. A subárvore *C* não foi modificada.

Uma situação particular desta última estratégia é quando o nó folha adequado à inserção é o próprio nó raiz. Neste caso, um novo nó raiz é criado, e o anterior é particionado em dois outros nós.

É interessante observar que o algoritmo de inserção proposto é baseado na redistribuição de elementos e no particionamento de nós existentes. E que a preferência recai sobre a redistribuição de elementos.

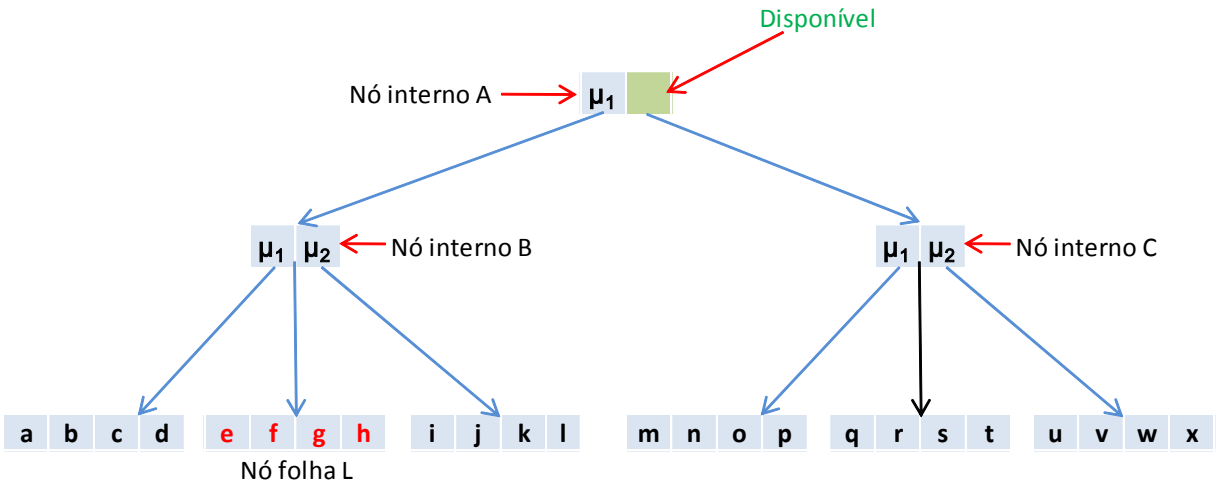


Figura 3.8: VP-tree: Inserção Situação inicial: Não há espaço na folha apropriada, porém o nó pai possui espaço para um novo filho (Figura adaptada de (FU et al., 2000)**)**

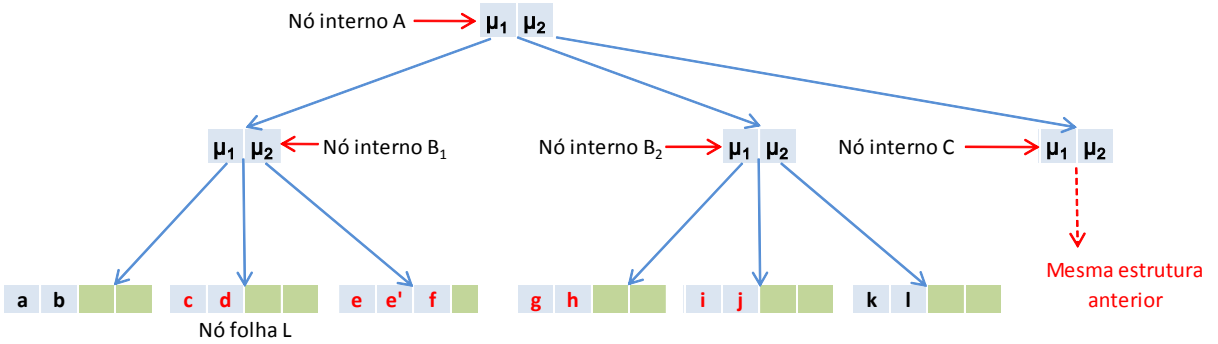


Figura 3.9: VP-tree: Inserção Situação final: Não há espaço na folha apropriada, porém o nó pai possui espaço para um novo filho - Figura adaptada de (FU et al., 2000)

Remoção

O algoritmo de remoção percorre a árvore da raiz em direção as folhas calculando em cada nível, a distância entre o elemento a ser removido e o *vantage-point* do nó. O percurso é determinado pela subárvore cujo raio de abrangência contenha o elemento a ser removido. Ao encontrar o nó folha que contém o elemento a ser removido, a remoção é concretizada caso a taxa de ocupação mínima do nó *(TOM)* não seja violada.

Além da *TOM*, a remoção também deve levar em consideração a quantidade mínima de elementos em cada subárvore. Seja *nível(E)* o nível do nó *E*. Se *E* é um nó folha, então *nível(E)=0*. Considere Min_{leaf} a quantidade mínima de elementos de

uma folha e Min_{Ifan} a quantidade mínima de subárvores de um nó interno. Então $Min_{data}(E)$ é a quantidade mínima de elementos da subárvore do nó *E* definida como:

$$Min_{data}(E) = Min_{leaf}\ X\ (MIN_{fan})^{level(e)}$$

Assim, considerando-se *L* a folha que viola a *TOM*, *P* seu nó pai e *F* a quantidade de folhas de *P*, quando houver a violação da *TOM* em um nó folha, as estratégias adotadas para a manutenção destas propriedades são descritas a seguir.

Quando *L* viola a *TOM*, porém *P* não a viola, o algoritmo verifica se há espaço em suas folhas irmãs para distribuir os elementos restantes em *L*. Contudo, quando as folhas irmãs de *L* não possuem espaço para receber estes elementos, aplica-se uma redistribuição de todos os elementos de *P* entre as *F* folhas da subárvore. Ou seja, quando houver possibilidade de redistribuição dos elementos restantes de *L* a subárvore ficará com *F - 1* folhas. Caso contrário, mantêm-se as *F* folhas, porém com seus elementos redistribuídos.

Quando *L* viola a *TOM*, porém *P* também a viola, o algoritmo localiza o ancestral imediato *A* de *L* que não viola a *TOM*. Seja *B* o filho imediato de *A*, considere ainda *B* a *k-ézima* subárvore de A:

1. Quando alguma das três condições abaixo for satisfeita, haverá uma união de árvores adjacentes:
   - Caso 1: Quando a *(K+1)ézima* subárvore tiver espaço disponível para receber os elementos de *B*, haverá a união dos elementos destas subárvores e *B* será removida.
   - Caso 2: Quando a *(K-1)ézima* subárvore tiver espaço disponível para receber os elementos de *B*, haverá a união dos elementos destas subárvores e *B* será removida.
   - Caso 3: Quando o espaço total disponível nas árvores *(K+1)* e *(K-1)* puder receber os elementos de *B*, estes elementos serão distribuídos entre as duas subárvores.
2. Quando nenhuma das três condições acima puder ser aplicada, todos os elementos de A serão redistribuídos na forma como foi feita na terceira situação do algoritmo de inserção, redistribuindo os elementos de todas as folhas da

subárvore do ancestral, de maneira que se possam inserir os elementos na folha apropriada.

3. Quando L for o nó raiz, haverá a união com um de seus filhos imediatos e o nó gerado se tornará a nova raiz.

3.3 **Conclusões**

Como síntese deste capítulo, a Tabela 3.1 resume as principais características de dinamicidade dos MAM Slim-tree, Dynamic VP-tree e Onion-tree. Atualmente, a Onion-tree (CARÉLO et al., 2009) é o método de acesso métrico baseado em memória primária mais eficiente para pesquisas por similaridade disponível na literatura. Entretanto, ainda não foi proposto um algoritmo para a remoção física de elementos. Para que este índice possa ser efetivamente incorporado a um SGBD, portanto, são necessárias a proposta e a implementação de, pelo menos, um algoritmo de remoção física de elementos.

Tabela 3.1: Comparativo entre dinamicidade da Slim-tree e Dynamic Vp-tree

	Slim-tree	Onion-tree	Dynamic VP-tree
Tipo	Multivias	Multivias	Binária
Dinamicidade	Dinâmico	Dinâmico	Dinâmico
Balanceamento	Balanceada	Não balanceada	Balanceada
Inserção	Nas folhas	Nas folhas	Nas folhas
Contrução	Botom-up	Top-Down	Top-Down
Sobreposição	Sim Oferece recursos para a avaliação do grau de sobreposição entre seus nós (Fat-Factor) e um algoritmo para pós-otimização de sua estrutura (Slim-Down)	Não	Não
Remoção	Sim	Em estudo	Sim
Armazenamento	Disco	Memória primária	Disco

Capítulo 4

REMOÇÃO LÓGICA DE ELEMENTOS NA ONION-TREE

A Onion-tree é uma estrutura de indexação construída de forma *top-down*. A remoção de elementos em nós folha não possui maiores implicações, na organização da estrutura da árvore. Porém, a remoção de elementos em nós internos pode causar uma grande reorganização da estrutura da árvore. Na Onion-tree, a remoção de nós internos ou a alteração na distância entre os representantes de um nó interno afeta todo o particionamento métrico de seus nós filhos e pode ser uma operação altamente custosa.

Uma alternativa menos custosa na remoção de elementos de nós internos é a sinalização de que o elemento encontra-se removido (ou seja, aplicar uma remoção lógica) e assim desconsiderá-lo no resultado das consultas. Contudo, vale observar que os elementos sinalizados como removidos permanecem na estrutura guiando o espaço métrico e fazendo parte dos cálculos de distância aplicados em consultas. Portanto, esta solução é uma boa alternativa para conjuntos de dados em que a quantidade de remoções é pequena. Para conjuntos de dados com grande quantidade de remoções o desempenho do processamento de consultas fica prejudicado pela realização de cálculos desnecessários envolvendo os elementos marcados como logicamente removidos. Uma forma de minimizar esta quantidade desnecessária de cálculos de distância sem grande reorganização da estrutura é reorganizar os elementos somente a partir do penúltimo nível da estrutura, isto é, reorganizar hierarquias de nós cujos filhos sejam somente folha.

Desta forma, como inicio desta pesquisa de mestrado, primeiramente foi avaliada a proposta conceitual de remoção lógica de elementos da Onion-tree apresentada em (CARÉLO et al., 2011). A avaliação desta proposta deu origem a três algoritmos de remoção lógica, denominados *LogicalDelete, ReplaceReducing* e *ReplaceGrowing*. O primeiro algoritmo aplica a remoção lógica em toda a estrutura

de indexação. Os outros dois algoritmos especializam a remoção lógica reorganizando a estrutura hierárquica de nós cujos filhos sejam somente folhas. Uma descrição mais detalhada dos algoritmos é dada a seguir:

- Algoritmo *LogicalDelete*: Consiste em sinalizar que o elemento se encontra removido e deve, portanto, ser desconsiderado no processamento de consultas;
- Algoritmo *ReplaceReducing*: Estende o algoritmo *LogicalDelete*, acrescentando tratamento na remoção em nós internos cujos filhos sejam folha, substituindo o elemento removido por outro de suas folhas, sem aumentar o raio de cobertura do nó;
- Algoritmo *ReplaceGrowing*: Assim como o algoritmo *ReplaceReducing*, estende o algoritmo *LogicalDelete,* ainda tratando a remoção de nós internos cujos filhos sejam folha e substituindo o elemento removido por outro de suas folhas, porém, desta vez permitindo o aumento do raio de cobertura do nó até um limite pré-estabelecido.

O algoritmo *LogicalDelete* aplica a remoção lógica em todos os nós, sem qualquer reorganização da estrutura originalmente construída. Os algoritmos *ReplaceReducing* e *ReplaceGrowing* estendem o algoritmo *LogicalDelete*, porém exclusivamente nos elementos armazenados em nós pai somente de nós folha. Nestes níveis, em função da substituição do elemento removido por outro de suas folhas, estes algoritmos reorganizam todos os elementos da hierarquia do nó que sofreu a remoção, mantendo um único padrão de comportamento: aumentando ou diminuindo o raio original do nó. Desta forma investigou-se o efeito de tratamentos opostos na reorganização dos elementos destes níveis no desempenho do processamento de consultas efetuadas após a remoção.

Para a avaliação do desempenho dos algoritmos *LogicalDelete*, *ReplaceReducing* e *ReplaceGrowing* os testes foram aplicados em conjuntos de dados com diferentes dimensionalidades e volumes crescentes de dados. Foram utilizadas as bases Brazilian Cities, KDD Cup 2008 e Color Histograms. As seções a seguir detalham os algoritmos, as características dos conjuntos de dados, o ambiente de teste e os resultados obtidos.

4.1 **Algoritmo *LogicalDelete***

Este algoritmo consiste em apenas sinalizar que o elemento encontra-se removido em um nó e, portanto, o elemento marcado como removido deve ser desconsiderado posteriormente no processamento de consultas de similaridade. Para isso, foram incluídos os seguintes tratamentos:

1. Em um nó folha, realizar a remoção física do elemento. Quando o nó estiver vazio efetuar sua remoção física e ajustar o ponteiro para o nó em seu nó pai;
2. Em um nó interno, apenas marcar o elemento como logicamente removido;
3. No processamento de consultas, desconsiderar um elemento marcado como logicamente removido.

Como exemplo, uma situação inicial e a sua correspondente situação final de exclusão lógica na Onion-tree são ilustradas nas figuras que se seguem. A Figura 4.1 e a Figura 4.2 ilustram as representações hierárquicas.

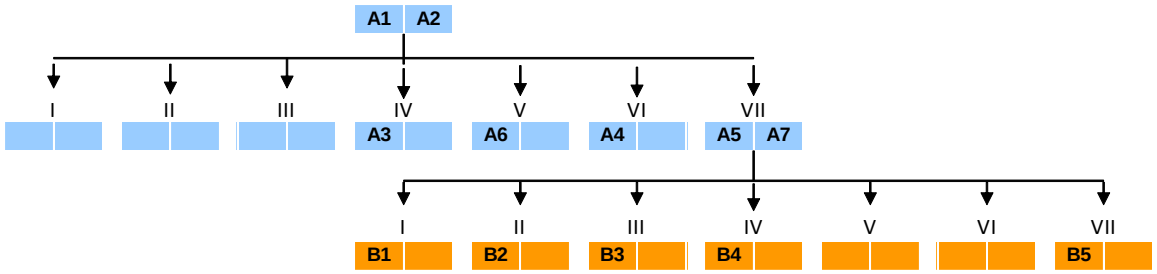


Figura 4.1: *LogicalDelete* -
Representação hierárquica de situação inicial

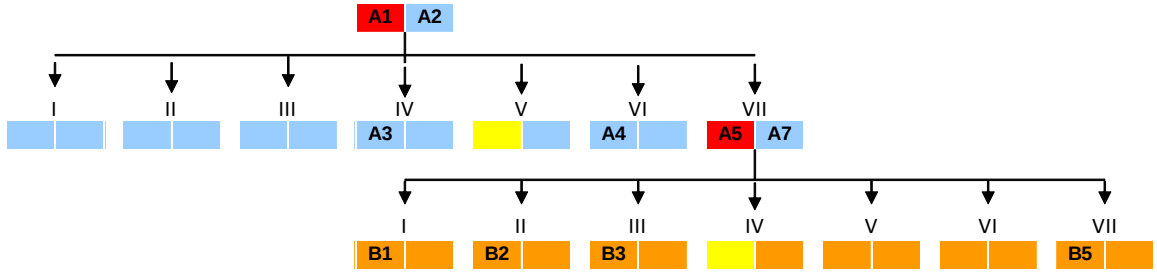


Figura 4.2: *LogicalDelete* -
Representação hierárquica de situação final

Neste exemplo, são excluídos os elementos A1, A5, A6 e B4. Na Figura 4.2 os elementos A1 e A5, armazenados em nós internos, foram marcados como logicamente excluídos, e os elementos A6 e B4, armazenados em nós folha, foram fisicamente removidos. Observe que os elementos logicamente removidos foram

destacados em vermelho, enquanto os elementos fisicamente removidos foram destacados em amarelo. A Figura 4.3 e a Figura 4.4 ilustram as correspondentes representações espaciais.

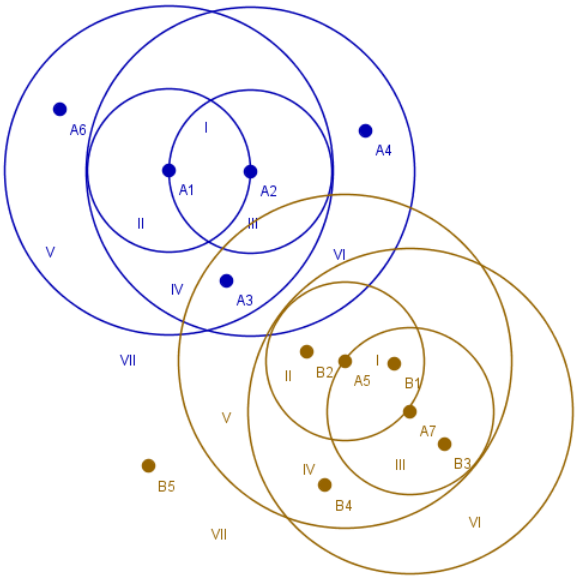


Figura 4.3: *LogicalDelete* -
Representação espacial de situação inicial

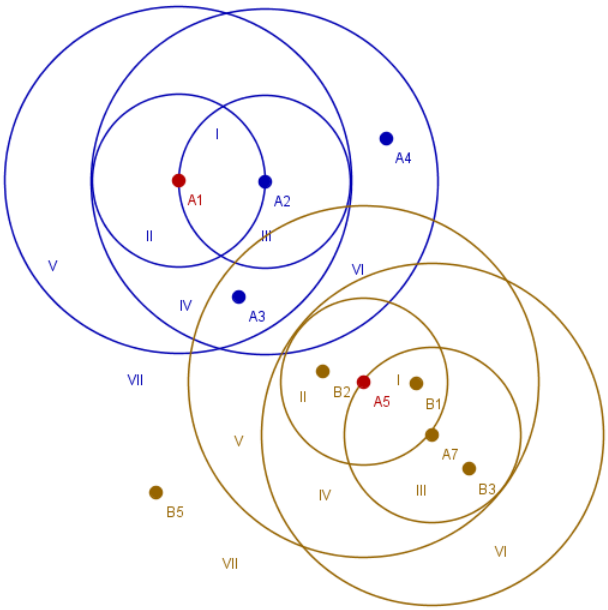


Figura 4.4: *LogicalDelete* -
Representação espacial da situação final

4.2 **Algoritmo *ReplaceReducing***

O algoritmo *ReplaceReducing* estende o algoritmo *LogicalDelete*, tratando de forma especifica a remoção em nós internos com filhos exclusivamente folha. Nestes

casos, este algoritmo substitui o elemento removido por outro de suas folhas, sem aumentar o raio de cobertura do nó interno. Ao impedir que o raio do nó aumente, evita-se que se gere uma região interna muito maior do que a atual. Os tratamentos adicionados ao algoritmo *LogicalDelete* são descritos a seguir:

1. Substituir o elemento removido por outro elemento de suas folhas sem aumentar o raio atual. Quando houver dois ou mais elementos nas folhas, apropriados para substituir o elemento removido no nó interno, escolhe-se o elemento que em substituição ao elemento removido resulte em um raio mais próximo do raio atual do nó (i.e. raio antes da substituição);
2. Quando não houver elementos nas folhas que sejam apropriados para a substituição, marcar o elemento como logicamente removido;
3. Na inserção de novos elementos, quando o nó interno tiver um elemento logicamente removido, substituí-lo pelo elemento sendo inserido, quando isto não aumentar o raio do nó.

Quando o elemento removido for o primeiro pivô, as regiões em que se encontram elementos para substituí-lo sem aumentar o raio são as regiões I e III. A Figura 4.5 ilustra estas regiões.

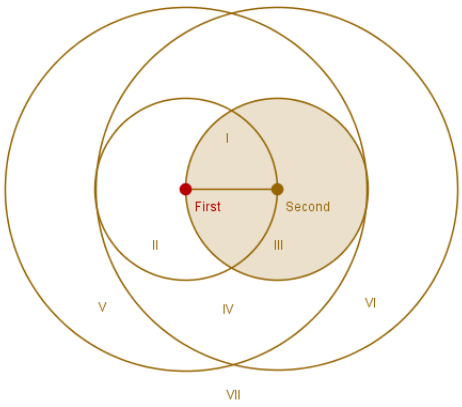


Figura 4.5: *ReplaceReducing* -
Na substituição de primeiro pivô, elementos das regiões I e III não aumentam o raio

Quando o elemento removido for o segundo pivô, as regiões em que se encontram elementos para substituí-lo sem aumentar o raio são as regiões I e II. A Figura 4.6 ilustra estas regiões.

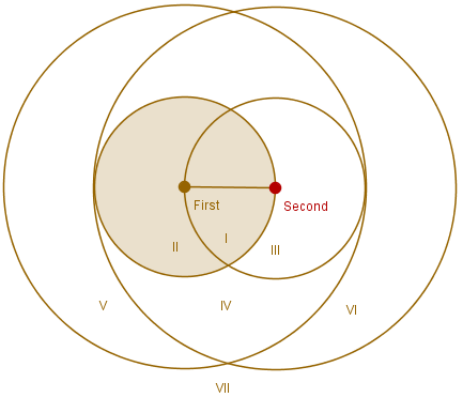


Figura 4.6: *ReplaceReducing* -
Na substituição do segundo pivô, elementos das regiões I e II não aumentam o raio

Como exemplo para o primeiro tratamento do algoritmo *ReplaceReducing*, a Figura 4.7, a Figura 4.8, a Figura 4.9 e a Figura 4.10 ilustram uma situação inicial e a situação final após a exclusão lógica. Observe a substituição do elemento removido A5 pelo elemento B3 da região III. Observe também que os elementos assinalados em vermelho foram logicamente excluídos na demonstração do algoritmo *LogicalDelete*, e que a substituição do elemento removido de um nó interno por outro em suas folhas causou a reorganização de todos os elementos da hierarquia envolvida.

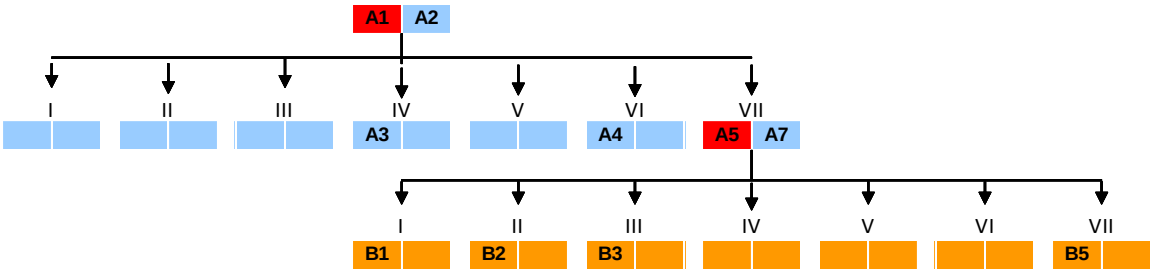


Figura 4.7: *ReplaceReducing* -
Primeiro tratamento: Representação hierárquica de situação inicial

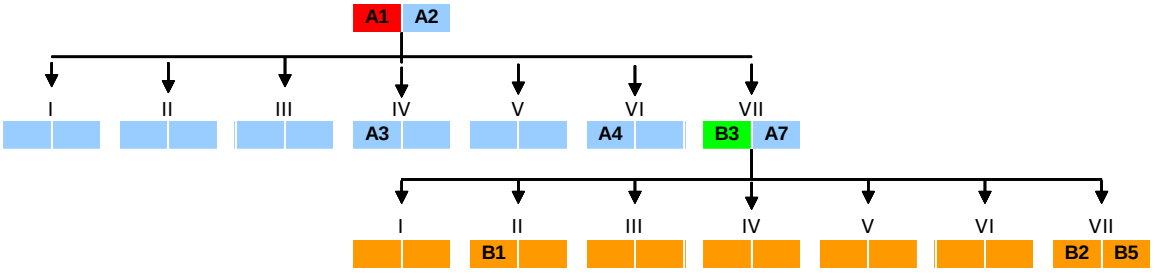


Figura 4.8: *ReplaceReducing* -
Primeiro tratamento: Representação hierárquica de situação final

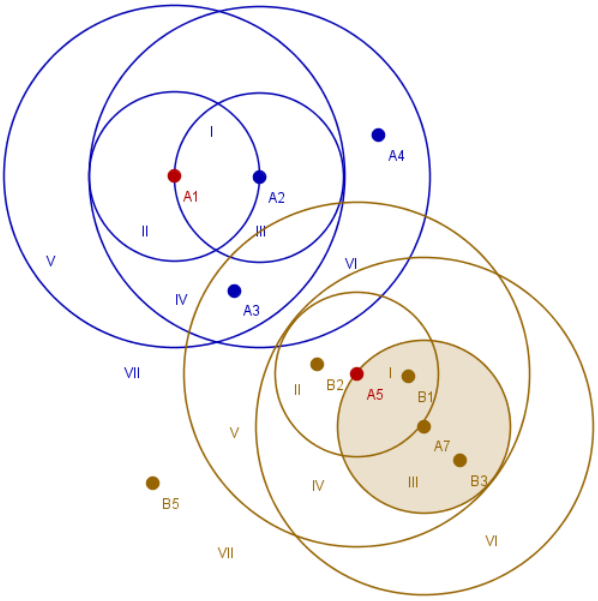


Figura 4.9: *ReplaceReducing* -
Primeiro tratamento: Representação espacial de situação inicial

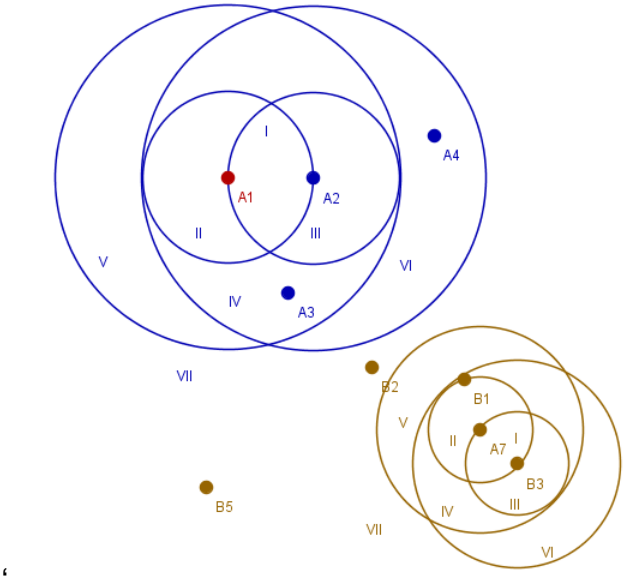


Figura 4.10: *ReplaceReducing* -
Primeiro tratamento: Representação espacial de situação final

Como exemplo para o segundo tratamento do algoritmo *ReplaceReducing*, a Figura 4.11 e a Figura 4.12 ilustram a situação final de exclusão lógica. Observe que por não haver elementos nas regiões I e III (que não provocam aumento do raio) houve a marcação lógica (em vermelho) do elemento removido A5.

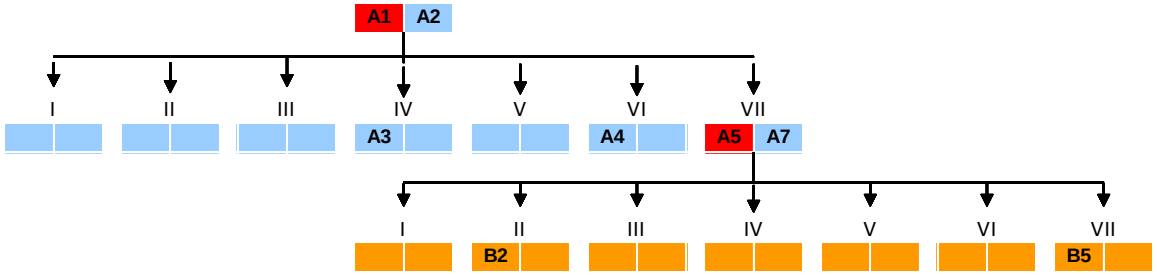


Figura 4.11: *ReplaceReducing* -
Segundo tratamento: Representação hierárquica de situação final

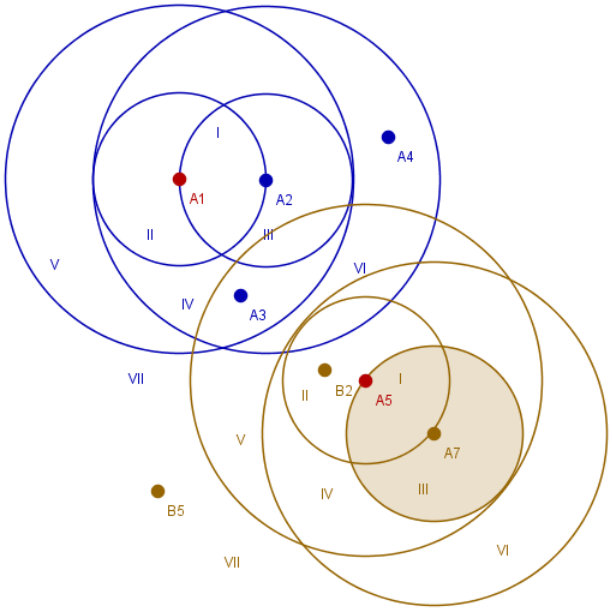


Figura 4.12: *ReplaceReducing*
Segundo tratamento: Representação espacial de situação final

Como exemplo para o terceiro tratamento do algoritmo *ReplaceReducing*, a Figura 4.13, a Figura 4.14, a Figura 4.15 e a Figura 4.16 ilustram uma situação inicial e final. O elemento B6 está em processo de inserção na região III, e ocasiona a substituição do elemento A5 pelo novo elemento B6. Observe ainda que a substituição do elemento A5 causou uma reorganização de toda a hierarquia.

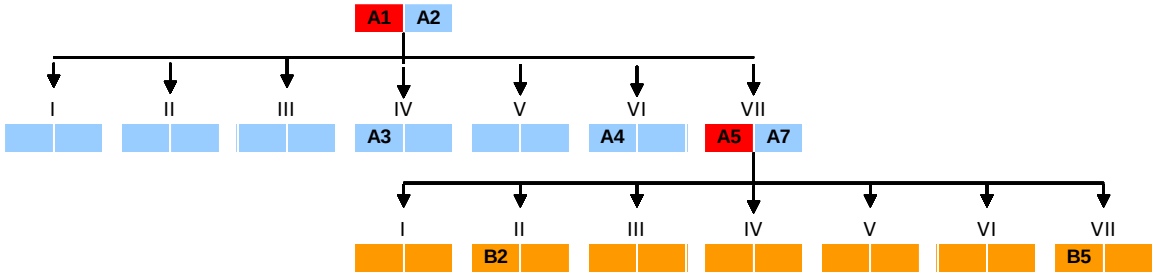


Figura 4.13: *ReplaceReducing* -
Terceiro tratamento: Representação hierárquica de situação inicial

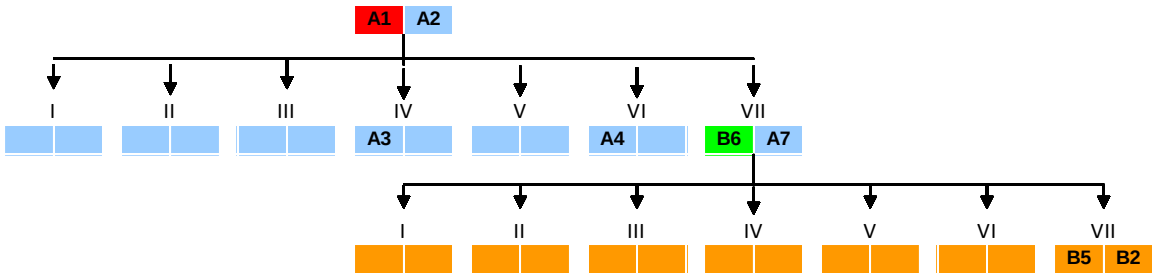


Figura 4.14: *ReplaceReducing* -
Terceiro tratamento: Representação hierárquica de situação final

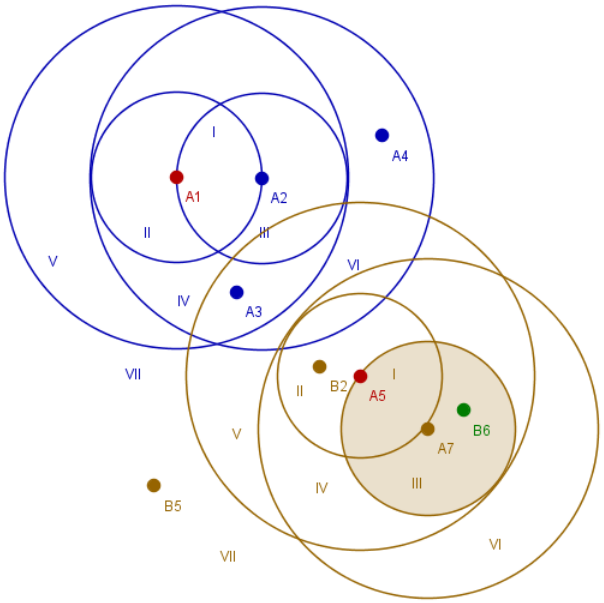


Figura 4.15: *ReplaceReducing* -
Terceiro tratamento: Representação espacial de situação inicial

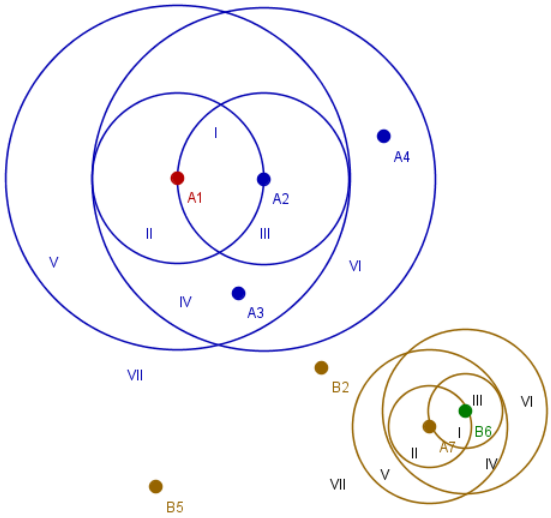


Figura 4.16: *ReplaceReducing* -
Terceiro tratamento: Representação espacial de situação final

4.3 **Algoritmo *ReplaceGrowing***

O algoritmo *ReplaceGrowing* estende o algoritmo *LogicalDelete,* da mesma forma que o algoritmo *ReplaceReducing*, tratando de forma específica nós internos cujos filhos sejam exclusivamente folha. Nestes casos, o algoritmo também substitui o elemento removido por outro de suas folhas, porém desta vez permite o aumento do raio de cobertura do nó até um limite especificado pela aplicação. Os tratamentos são descritos a seguir:

1. Quando as folhas tiverem elementos que aumentem o raio até o limite estabelecido, substituir o elemento removido por aquele que gerar um novo raio mais próximo do raio atual.
2. Quando não houver elementos nas folhas que sejam apropriados para a substituição, marcar o elemento como logicamente removido;
3. Na inserção de novos elementos, quando o nó interno cujos filhos sejam exclusivamente folha tiver um elemento logicamente removido, substituí-lo pelo elemento sendo inserido, quando isto não aumenta o raio do nó.

A Figura 4.17 e a Figura 4.18 ilustram a abrangência destas regiões respectivamente na remoção do primeiro e do segundo pivô.

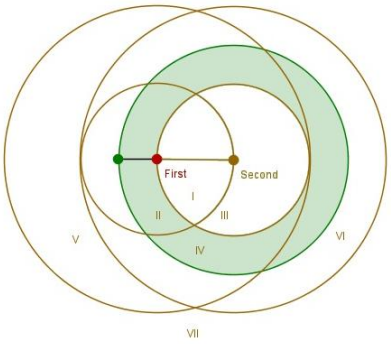


Figura 4.17: *ReplaceGrowing* - -
Área de verificação para a substituição do primeiro representante

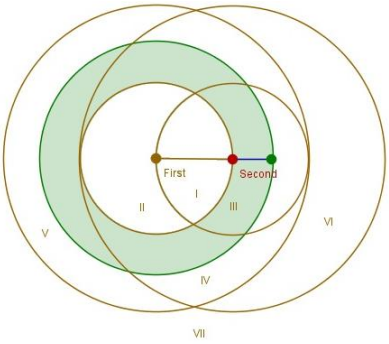


Figura 4.18: *ReplaceGrowing* -
Área de verificação para a substituição do segundo representante

Como exemplo, a Figura 4.19, a Figura 4.20, a Figura 4.21, e a Figura 4.22 ilustram uma situação inicial e final da exclusão de um representante em um nó pai de nó folha. Observe a substituição do elemento removido A5 pelo elemento B2. A substituição causou a reorganização de toda a hierarquia.

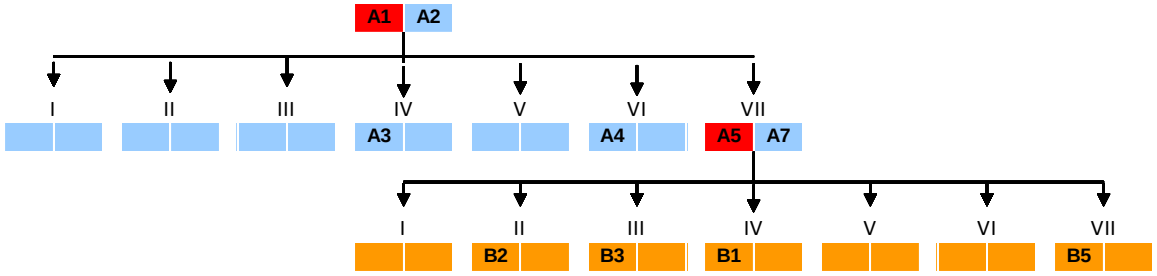


Figura 4.19: *ReplaceGrowing* -
Representação hierárquica de situação inicial

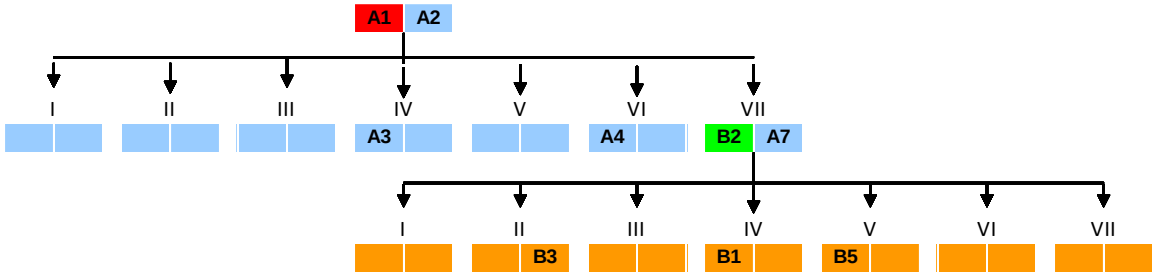


Figura 4.20: *ReplaceGrowing* -
Representação hierárquica de situação final

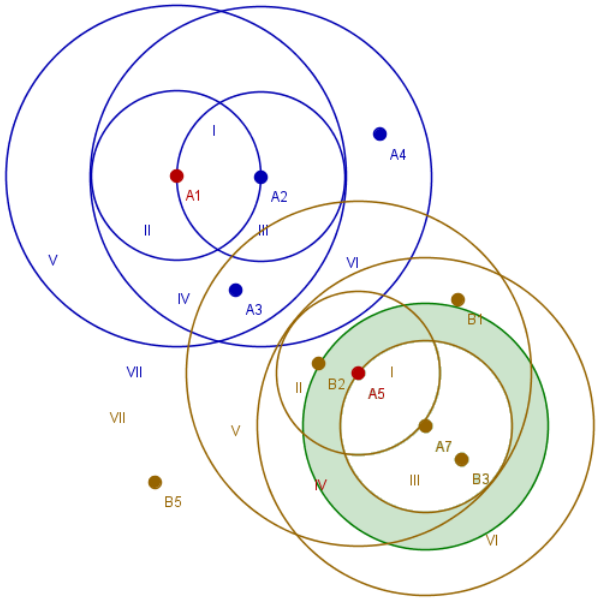


Figura 4.21: *ReplaceGrowing* -
Representação espacial de situação inicial

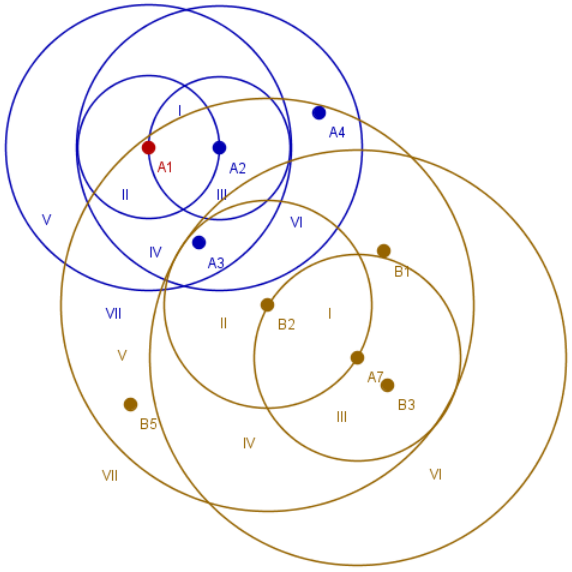


Figura 4.22: *ReplaceGrowing* -
Representação espacial de situação final

4.4 Representação Formal dos Algoritmos de Remoção Lógica

Algoritmo 1
Executado pelo algoritmo LogicalDelete - executa a remoção lógica de um elemento
DeleteElement(Node, Element)

```
      Input: Node (nó em pesquisa para encontrar elemento a ser removido), Element
      (Elemento a ser removido)
      Output: null

1     If Node = null then return; // Search is finished
2     else
3       d 1 ← distance(Element, Node.Pivot1)
4       If d 1 = 0 then Element for delete is Node.Pivot1
5         if Node is not leaf then Node.Pivot1.removed = yes
6         else
7           // Node is a leaf and Pivot2 is empty, so fisicaly delete the node
8           If Node.Pivot2 = 0 then delete Node
9           else
10              // Pivot1 is the element to be deleted and
11              // Node is a leaf and Pivot2 is not empty
12              // Delete Pivot1 and make Pivot2 the first pivot
13             delete Node.Pivot1;
14            Node.Pivot1 ← Node.Pivot2
15          end
16        end
17      end
```

```
18    d2 ← distance(Element, Node.Pivot2)
19    If d2 = 0 then Element for delete is Node.Pivot2
20      if Node is not leaf then Node.Pivot2.removed = yes;
21      else
22        // Pivot2 is the element to be deleted and
23        // Node is a leaf and there is Pivot1
24        // Only delete Pivot2
25        delete Node.Pivot2
26      end
27    end
28
29    // Continue the search
30    For Region ← 1 to all Node´s sons do
31      if Query radius intersects Node´s son region then
32        DeleteElement(Node´s son,Element);
33      end
34    end for
```

Algoritmo 2

ReplaceReducing e ReplaceGrowing

DeleteElement(N, Father, E, Algorithm, Mesure)

```
      Input: N (nó), Father (pai do nó), E (Elemento), Algorithm(Algoritmo em execução),
      Mesure(Medida para diminuição / aumento do raio)
      Output: null
1     If Node = null then return; // Search is finished
2     else
3       d1 ← distance(Element, Node.Pivot1)
4       If d1 = 0 then Element for delete is Node.Pivot1
5         if Node is not leaf then
6           if Node is dad of internal nodes then Node.Pivot1.removed = yes;
7           else
8             // Node is dad of leafs
9             ReplaceRemovedPivot(Node, 1, Node.Pivot1, Algorithm,Mesure);
10          end
11        else
12          // Node is a leaf and Pivot2 is empty, so fisicaly delete the node
13          If Node.Pivot2 = 0 then delete Node
14          else
15            // Pivot1 is the element to be deleted and
16            // Node is a leaf and Pivot2 is not empty
17            // Delete Pivot1 and make Pivot2 the first pivot
18            delete Node.Pivot1
19            Node.Pivot1 ← Node.Pivot2
```

```
20        end
21      end
22    end
23    d2 ← distance(Element, Node.Pivot2)
24    If d2 = 0 then Element for delete is Node.Pivot2
25      if Node is not leaf then
26        if N is not dad of internal nodes then Node.Pivot2.removed = yes;
27        else
28          // Node is dad of leafs
29          ReplaceRemovedPivot(Node, 2, Node.Pivot2, Algorithm, Mesure);
30        end
31      else
32        // Pivot2 is the element to be deleted and
33        // Node is a leaf and there is Pivot1
34        // Only delete Pivot2
35        delete Node.Pivot2;
36      end
37    end
38
39    // Continue the search
40    For Region ← 1 to all Node´s sons do
41      if Query radius intersects Node´s son region then
42        DeleteElement(Node´s Son, Node, Element, Algorithm, Mesure);
43      end
44    end for
```

Algoritmo 3

Executado pelos algoritmos ReplaceReducing e ReplaceGrowing, substitui um elemento removido em um nó pai somente de folhas

ReplaceRemovedPivot(Node, PivotNumber, Pivot, Algorithm, Mesure)

```
      Input: Node (nó que sofrerá a substituição), PivotNumber (número do pivot em
      substituição), Pivot (elemento a ser substituido), Algorithm (Algoritmo em execução),
      Mesure (Medida para diminuição / aumento do raio)
      Output: null
1     If Algorithm = "ReplaceReducing" then
2       Limit = Node.radius - Mesusure
3     else
4       Limit = Node.radius + Mesusure
5     end;
```

```
6   // Save all childs and delete all leafs
7   For 1 to all Leafs
8     Save Leaf.Pivot1;
9     Save Leaf.Pivot2;
10    Delete Leaf;
11  end for;
12  // Look for another element to replace the deleted one
13  For 1 to all Leafs
14    // If element to replace is Pivot1, look in the leafs for an element
15    // considering the radius between this element and Pivot2
16    If Pivot = 1 then
17      d1 ← distance(Leaf.Pivot1, Node.Pivot2)
18      d2 ← distance(Leaf.Pivot2, Node.Pivot2)
19      If d1 is the smaller diference to the limit then
20        // Leaf.Pivot1 will replace Node.Pivot1
21        Node.Pivot1 = Leaf.Pivot1
22      else
23        If d2 is the smaller diference to the limit then
24          // Leaf.Pivot2 will replace Node.Pivot1
25          Node.Pivot1 = Leaf.Pivot2
26        end
27      end
28    else
29    // If element to replace is Pivot2, look in the leafs for an element
30    // considering the radius between this element and Pivot1
31      d1 ← distance(Leaf.Pivot1, Node.Pivot1)
32      d2 ← distance(Leaf.Pivot2, Node.Pivot1)
33      If d1 is the smaller diference to the limit then
34        // Leaf.Pivot1 will replace Node.Pivot2
35        Node.Pivot2 = Leaf.Pivot1
36      else
37        If d2 is the smaller diference to the limit then
38          // Leaf.Pivot2 will replace Node.Pivot2
39          Node.Pivot2 = Leaf.Pivot2
40        end
41      end
42    end
43  end for
44  For 1 to all saved Leafs
45    Reinsert(Leaf.Pivot1)
46    Reinsert (Leaf.Pivot2)
47  end for
```

4.5 **Ambiente de Teste**

Para a análise de desempenho dos algoritmos *LogicalDelete, ReplaceReducing e ReplaceGrowing*, foram utilizados conjuntos de dados com diferentes volumes (quantidades de elementos) e dimensionalidades.

Em todos os conjuntos de dados, para a construção da Onion-tree foram utilizadas a política de expansão *F-Onion-tree* e a política de substituição de representantes *Keep-small* (ver *Tabela 2.3)*. A quantidade de expansões utilizada foi determinada em (CARÉLO et al., 2011). A Tabela 4.1 relaciona para cada conjunto de dados utilizado nos testes, a quantidade de elementos, sua dimensionalidade, o diâmetro do conjunto de dados e a quantidade de expansões usada para a construção da Onion-tree (seção 2.7). Com exceção do conjunto de dados *Ozone*, a qual utilizou a métrica *Dynamic Time Warping*, todas as demais utilizaram a métrica L_2 para determinação das distâncias entre seus elementos.

Tabela 4.1: Bases de dados analisadas, suas características e quantidade de expansões para a construção da Onion-tree

Base	Qt.Elem.	Dim.	Qt.Exp.	Diam.	Descrição
Ozone	2536	73	7	41,7298	Series temporais da medição de ozônio entre 1998 e 2004
Brazilian Cities	5508	2	5	40,5461	Coordenadas geográficas das cidades brasileiras (www.ibge.gov.br)
Color Histograms	68027	32	7	1,41318	Histogramas das imagens do repositório KDD da Universidade da Califórnia em Irvine (kdd.ics.uci.edu)
KDD Cup 2008	102240	117	11	34,5868	Base de dados contendo imagens com suspeita de câncer de mama (www.kddcup2008.com)

O procedimento para avaliação dos algoritmos foi elaborado com o objetivo de medir o desempenho no processamento de consultas posterior à remoção dos elementos, e para isso foram executadas as etapas abaixo:

1. Construção da estrutura de indexação com 100% dos elementos do conjunto de dados;
2. Realização de consultas por abrangência;
3. Remoção de 10% de elementos do índice, sendo os elementos escolhidos localizados nas folhas e em nós pai exclusivamente de folhas;

4. Reinserção dos elementos removidos na etapa 3;
5. Realização de consultas por abrangência (mesmas consultas da etapa 2).

O desempenho dos algoritmos foi avaliado comparando-se os totais de tempo e quantidade de cálculos de distância entre as consultas antes e após a reinserção dos elementos (etapas 2 e 5).

Para que fosse possível medir com maior precisão o desempenho dos algoritmos, foram selecionados para remoção aqueles elementos que estivessem em nós folha, ou em nós cujos filhos fossem exclusivamente nós folha. Foram selecionados 10% de elementos do conjunto de dados, sendo 50% destes elementos em nós folhas e 50% destes elementos em nós pai exclusivamente de folhas.

Com relação às consultas, foram executadas consultas *Range Query (RQ)* variando o raio de duas maneiras: (i) com um raio fixo determinado por um percentual do diâmetro do conjunto de dados, e (ii) com um raio dinâmico, previamente calculado por uma consulta KNN (k-vizinhos mais próximos), que retornasse até 10 elementos mais próximos a cada elemento consultado. Para a *RQ* com raio fixo, o percentual utilizado do diâmetro do conjunto foi de 31% conforme indicado em (CARÉLO et al., 2011) o qual retorna aproximadamente 1% dos elementos do conjunto de dados.

Foram selecionados os elementos que seriam efetivamente afetados pelo algoritmo de remoção. Após a operação de remoção estes elementos são alocados em outros locais da estrutura de indexação e desta forma é possível verificar o impacto desta movimentação no processamento posterior de consultas. Assim, estes elementos foram selecionados especificamente em função do algoritmo em análise. Para o algoritmo *LogicalDelete*, que somente sinaliza que o elemento está removido ainda mantendo-o na estrutura, os elementos foram selecionados aleatoriamente a partir do conjunto utilizado na remoção. Para os algoritmos *ReplaceReducing* e *ReplaceGrowing*, os elementos foram selecionados a partir do conjunto de elementos que seriam efetivamente movimentados na reorganização da estrutura, caso o elemento removido tenha sido substituído por outro de suas folhas.

A quantidade de consultas executadas também variou conforme o raio usado para a consulta. Na consulta *RQ* com raio fixo foi utilizado um subconjunto de 500 elementos do total de elementos movimentados pelo algoritmo de remoção. Quando

a movimentação de elementos não atingiu esta quantidade, o conjunto foi completado com elementos escolhidos aleatoriamente do conjunto de dados. Na consulta *RQ* com o raio dinâmico foram consultados 100% dos elementos do conjunto de dados, ou seja, cada elemento foi consultado tendo si próprio como centro desta consulta. Para cada elemento consultado, o tempo e a quantidade de cálculos de distância considerados foi a média na execução de 10 consultas. A Tabela 4.2 resume as condições executadas para cada raio da consulta *RQ*.

Tabela 4.2: Consulta RQ - Raio e quantidade de consultas executadas

Base	Raio fixo		Raio dinâmico	
	Qtd. Retornada	Qtd. Consultada	Qtd. Retornada	Qtd. Consultada
Ozone	1%	500	10	100% base
Brazilian Cities	1%	500	10	100% base
Color Histograms	1%	500	10	100% base
KDD Cup 2008	31%	500	10	100% base

Para os algoritmos *ReplaceReducing* e *ReplaceGrowing*, o conjunto *KDD Cup 2008* foi utilizado na avaliação de dois limites para a variação do raio do nó provocado pela substituição de um elemento removido de um nó pai de nós exclusivamente folha. Ainda observando, conforme mencionado na seção 4.2 e na seção 4.3, que ao variar o raio do nó, os algoritmos sempre priorizaram a medida mais próxima do raio original, foram avaliados os seguintes limites para a diminuição ou aumento do raio do nó: (i) diminuir / aumentar o valor de seu raio em até 0,5 do raio de seu nó pai (novo raio <= raio atual +/- 0,5); (ii) diminuir sem limite (novo raio <= raio atual) ou aumentar em até 10% do valor de seu raio (novo raio <= raio atual * 1,1). A Tabela 4.3 descreve os limites avaliados na diminuição e no aumento do raio do nó quando a remoção provoca a substituição de um elemento por outro de suas folhas.

Tabela 4.3: *ReplaceReducing* e *ReplaceGrowing* – Variações de raio analisadas

Medida	Elementos removidos	Teste	Qtd. Retornada na RQ
Seu raio -/+ 0,5 do raio de seu pai	Os 10% últimos elementos da base	1	Raio dinâmico - 1022 elementos
	10% -> 50% folha e 50% pai de folha	2	Raio dinâmico - 10 elementos
		3	Raio fixo 1% do diâmetro do conjunto
Diminuir sem limite /aumentar 10% de seu raio	10% -> 50% folha e 50% pai de folha	4	Raio dinâmico - 10 elementos
		5	Raio fixo - 31% do diâmetro do conjunto

Em um primeiro momento (teste 1 da Tabela 4.3) não foram utilizados, tanto para remoção quanto para consultas, elementos especialmente selecionados de nós folha e pai de nós exclusivamente folha. Trabalhou-se com o conjunto dos 10% últimos elementos inseridos no conjunto de dados. No caso do conjunto *KDD Cup 2008*, os últimos 1022 elementos que foram inseridos no conjunto. A partir do resultado do primeiro teste, verificou-se a necessidade de identificar se o algoritmo havia ou não movimentado uma quantidade de elementos de forma determinante para a verificação do desempenho do algoritmo no tratamento de elementos em nós folhas ou em nós pais de nós folhas, e a partir disso trabalhou-se apenas com os elementos efetivamente afetados pelos algoritmos em análise.

Os experimentos foram executados em um computador com um processador Intel Core Duo 3.00GHz e 8GB 533 MHz DDR2 de memória principal.

4.6 **Resultados**

Em função da quantidade de elementos e da alta dimensionalidade dos dados, o conjunto de dados *KDD Cup 2008* foi utilizado em uma quantidade maior de avaliações. Os demais conjuntos de dados foram utilizados nos dois últimos dos cinco testes executados. Todavia, nestes últimos os mesmos resultados verificados para o conjunto *KDD Cup 2008* também foram verificados para os demais conjuntos de dados. Todos os resultados obtidos estão descritos no Apêndice deste documento. A Tabela 4.4 descreve o desempenho dos três algoritmos

implementados para o conjunto *KDD Cup 2008* em consultas após a remoção, tendo como base as consultas antes da aplicação da remoção.

Tabela 4.4: KDD Cup 2008 –
Resultados apurados nos testes - Tempos totais de consulta

Medida	Seq.	LogicalDelete		ReplaceReducing		ReplaceGrowing	
		Qtd. Mov.	% Após reinserção	Qtd. Mov.	% Após reinserção	Qtd. Mov.	% Após reinserção
Seu raio -/+ 0,5 do raio de seu pai	1	NA	100,05	NA	100,77	NA	97,85
	2	0	99,77	2	99,76	3828	102,09
	3	0	99,94	2	99,49	3828	99,90
Diminuir sem limite /aumentar 10% de seu raio	4	0	99,77	1838	100,09	2355	100,61
	5	0	99,78	1838	99,87	2355	99,82

O algoritmo *LogicalDelete,* de forma inerente a seu algoritmo, uma vez que apenas marca o elemento como logicamente removido e este elemento ainda permanece na estrutura guiando os operadores de busca, apresenta o mesmo desempenho em consultas antes e após a reinserção dos elementos removidos.

Avaliando-se o algoritmo *ReplaceReducing*, observa-se nos testes 2 e 3, os quais avaliam respectivamente as consultas *RQ* de raio dinâmico e de raio fixo, que a diminuição do raio em até 0,5 do raio de seu nó pai quase não altera a estrutura original movimentando apenas 2 elementos da estrutura, mantendo a maior parte dos elementos removidos apenas sinalizados como excluídos. Em contrapartida, observa-se nos testes 3 e 4, os quais também avaliam respectivamente as consultas *RQ* de raio dinâmico e de raio fixo, que permitindo que o raio diminua sem limite, há uma movimentação maior de elementos, ou seja, 1.838 elementos, o que corresponde a 18,36% do total de 10.006 elementos removidos, contudo sem alterar de forma significativa o desempenho original das consultas. Mesmo com uma maior movimentação de elementos, o desempenho dos dois limites de variação do raio é praticamente o mesmo.

Avaliando-se o algoritmo *ReplaceGrowing*, observa-se nos testes 2 e 3 que o aumento do raio em até 0,5 do raio de seu nó pai movimenta 38,25% do total de elementos removidos enquanto que nos testes 3 e 4, seu aumento em até 10% movimenta 23,53% do total de elementos removidos. Entre os dois limites de aumento do raio não há diferença significativa a partir do desempenho original das

consultas, e novamente, o desempenho dos dois limites de alteração de raio é praticamente o mesmo.

4.7 **Validação dos Resultados**

A comparação das medições de tempo apuradas em processamentos distintos sempre foi uma preocupação presente nos experimentos. Como forma de se verificar efetivamente a validade dos resultados, as medições foram sempre conciliadas entre as variáveis observadas.

A primeira observação foi conciliar o tempo total de consulta de todos os elementos do conjunto de dados antes da remoção, e o tempo total de consulta dos elementos restantes na base, após a remoção de 10% de seus elementos. Desta maneira, o tempo total de consulta dos 90% de elementos restantes no conjunto de dados sempre se comportou como esperado, diminuindo proporcionalmente à quantidade de elementos removidos. A Tabela 4.5 descreve as variáveis observadas no teste 3 da Tabela 4.3, onde se avaliou os tempos de consulta de uma *RQ* com o raio fixo de 1% do diâmetro do conjunto de dados da base *KDD Cup 2008*.

Tabela 4.5: KDD Cup 2008 – Resultados do teste 3 da Tabela 4.3.
Tempo total de consulta após remoção de 10% dos elementos da base

	Elementos na base	Tempo total de consulta		
		Logical Delete	Replace Reducing	Replace Growing
Após construção	**100%**	45.500,21	47.984,02	48.592,74
Após remoção	**90%**	40.990,94	43.220,72	43.774,32
		90,09%	**90,07%**	**90,08%**
Após Reinserção	**100%**	45.474,55	47.737,84	48.546,43
Após remoção	**90%**	40.990,94	43.220,72	43.774,32
		90,14%	**90,54%**	**90,17%**

A primeira avaliação foi entre o tempo total de consulta de todos os elementos após a construção da estrutura, e o tempo total de consulta dos elementos restantes no conjunto, após a remoção de 10% de seus elementos. Esta avaliação demonstrou que o tempo se manteve coerente. Houve uma redução proporcional à quantidade de elementos removidos, ou seja, após a remoção de 10% dos elementos, o tempo total também foi reduzido em 10%.

A segunda avaliação comparou o tempo total de consulta de todos os elementos do conjunto de dados após a reinserção dos elementos anteriormente removidos, com o tempo total de consulta dos elementos restantes na base após a remoção de 10% de seus elementos. Aqui também houve uma redução proporcional à quantidade de elementos removidos. O tempo total de consulta após a remoção de 10% dos elementos foi 10% menor que o tempo total de consulta após a reinserção dos elementos anteriormente removidos.

Para a analise do desempenho dos algoritmos de remoção *LogicalDelete*, *ReplaceReducing* e *ReplaceGrowing*, foram apurados separadamente, os totais do tempo de consulta e dos cálculos de distância após a construção da estrutura Onion-tree e após a reinserção dos elementos anteriormente removidos, em três conjuntos de elementos: (i) para o total dos elementos da base; (ii) para os elementos excluídos; e (iii) para os elementos movimentados pelo algoritmo em questão. A Tabela 4.6 e a Tabela 4.7 descrevem os resultados apurados. Observa-se que a variação no resultado final é produzida pelos elementos afetados pelos algoritmos. O que comprova que os resultados verificados são decorrência da execução dos algoritmos.

Tabela 4.6: KDD2008 – Resultados do teste 4 da Tabela 4.3. Tempos totais de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,77%	100,09%	100,61%
Elementos excluidos	99,77%	100,10%	100,62%
Elementos movimentados	99,76%	100,10%	100,62%

Tabela 4.7: KDD2008 - Resultados do teste 4 da Tabela 4.3. Total de cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,08%	99,90%
Elementos excluidos	100,00%	100,08%	99,90%
Elementos movimentados	100,00%	100,08%	99,91%

4.8 **Conclusões**

Para o penúltimo nível da estrutura de indexação, ou seja, para nós pai exclusivamente de folha, nenhum dos algoritmos que avaliaram a substituição de um elemento removido por outro de suas folhas, ou seja, os algoritmos *ReplaceReducing* e *ReplaceGrowing*, apresentaram desempenho que justifique o processamento de uma quantidade maior de instruções do que aquela aplicada na remoção lógica. Para estes níveis, a melhor relação custo/benefício é indicada pelo algoritmo *LogicalDelete.*

.

Capítulo 5

REMOÇÃO FÍSICA DE ELEMENTOS NA ONION-TREE

Analisando o que é satisfatório para o desempenho em uma operação de remoção, podemos ressaltar dois pontos: (i) sobretudo, a remoção de elementos não deve degradar o desempenho no processamento de consultas, ou seja, as consultas realizadas após as operações de remoção devem ter um custo similar às mesmas consultas realizadas antes das operações de remoção; (ii) adicionalmente, o custo da operação de remoção deve ser satisfatório, isto é, o tempo de execução do algoritmo de remoção deve ser viável. Além disso, o custo de um conjunto de operações de remoção deve ser inferior, se possível bem inferior, ao custo de reconstruir o índice com os elementos remanescentes após a remoção.

Em relação ao desempenho no processamento de consultas posteriores a remoção, a proposta de remoção lógica feita em (CARÉLO et al., 2011) e analisada no Capítulo 4 demonstrou que o algoritmo de remoção lógica *LogicalDelete* tem a melhor relação custo/benefício quando comparado com os algoritmos *ReplaceReducing* e *ReplaceGrowing*. Embora o ponto forte desta relação seja o menor custo, o elemento removido ainda permanece no índice e à medida que cresce a quantidade de elementos removidos, cálculos de distância desnecessários podem deteriorar o desempenho no processamento de consultas aos elementos efetivamente válidos que estão armazenados no índice. Se ao invés de logicamente removidos, os elementos forem fisicamente removidos, o desempenho no processamento de consultas para os elementos que permanecem no índice será melhor, caso o algoritmo de remoção garanta um efeito local na reorganização do índice e que o custo desta reorganização seja baixo.

Quanto à viabilidade da execução da operação de remoção, temos que em árvores métricas de construção *top-down* a inclusão ou remoção física em nós internos, ou a alteração na distância entre os elementos pivôs destes nós obriga a

reorganização de toda a hierarquia (ou seja, a reorganização dos elementos de todos os nós descendentes do nó que possui o elemento removido, por exemplo). Ainda, quanto menor o nível em que o nó com o elemento removido se encontra na estrutura, maior será o custo de reorganização de sua hierarquia, ou seja, quanto mais próximo o nó com o elemento removido estiver à raiz, maior será o custo de reorganização da sua subárvore.

Levando em consideração as questões anteriormente discutidas nesta seção, é necessária a proposta de algoritmos de remoção física de elementos na Onion-tree em que seja viável a reorganização do índice após a remoção dos elementos, procurando manter o desempenho no processamento de consultas o mais próximo do verificado antes das remoções serem aplicadas.

Este Capítulo apresenta a proposta de dois algoritmos de remoção física para a Onion-tree, chamados *ReorgAll* e *PromoteNode*. Em atenção ao desempenho posterior no processamento de consultas, o algoritmo *ReorgAll*, reorganiza toda a hierarquia afetada pela remoção, entretanto, procurando manter o índice o mais próximo possível de sua estrutura original. Em atenção à viabilidade da operação de remoção, o algoritmo *PromoteNode* especializa o algoritmo *ReorgAll*, procurando por um nó que possa ser promovido no lugar do nó que sofreu a remoção, assim tentando diminuir o custo da reorganização de sua hierarquia. As seções 5.1, 5.2 e 5.3 descrevem estes algoritmos, as seções 5.4, 5.5, 5.6, e 5.7 descrevem os testes de desempenho realizados para validar a propostas dos algoritmos *ReorgAll* e *PromoteNode* e os resultados obtidos.

5.1 **Algoritmo *ReorgAll***

O algoritmo *ReorgAll* reorganiza toda a hierarquia do nó que possui o elemento removido, removendo fisicamente este nó e também removendo fisicamente todos os nós de suas sub-árvores (ou seja, removendo os seus nós descendentes), e reinserindo no índice os elementos de todos os nós removidos por meio do uso do algoritmo de inserção da Onion-tree. A única observação é a ordem em que o algoritmo executa as reinserções dos elementos removidos. O algoritmo percorre a hierarquia afetada pela remoção em sua largura, armazenando em sua sequência os elementos de todas as regiões do nó em um vetor. Desta forma, armazena-se todos os elementos de um nível para somente então percorrer o

próximo nível. Após armazenar todos os elementos da hierarquia afetada pela remoção, o algoritmo percorre o vetor e reinsere novamente os elementos na Onion-tree.

A estrutura de indexação Onion-tree não é determinística. Em função da ordem dos elementos na inserção, o mesmo conjunto de dados pode produzir estruturas de indexação distintas. Assim, o armazenamento dos elementos no vetor de acordo com a largura original da árvore procura minimizar a chance de que a nova estrutura seja radicalmente diferente da original, o que aumentaria a chance de um desempenho muito diferente no processamento de consultas posteriores à remoção.

Como exemplo, uma situação inicial e a sua correspondente situação final de exclusão física na Onion-tree utilizando o algoritmo *ReorgAll* são ilustradas nas figuras que se seguem. A Figura 5.1 ilustra a representação hierárquica de uma situação inicial de remoção. Neste exemplo, o elemento B5 (destacado em vermelho) está em remoção.

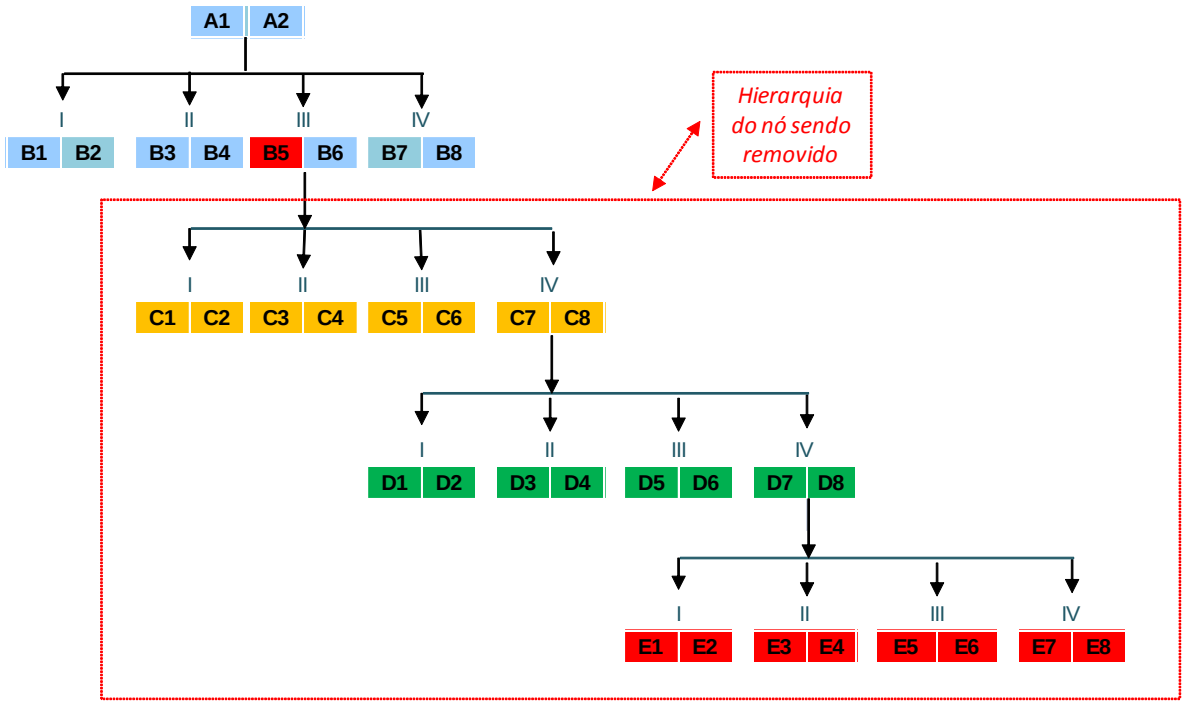


Figura 5.1: *ReorgAll* – Representação hierárquica de situação inicial de remoção

A Figura 5.2 ilustra a representação hierárquica após a remoção do nó contendo o elemento B5 e sua hierarquia. A Figura 5.3 ilustra o armazenamento dos elementos da hierarquia do nó que continha o elemento B5 conforme a largura da

estrutura de indexação. A Figura 5.4 ilustra a situação final após a reinserção dos elementos na hierarquia.

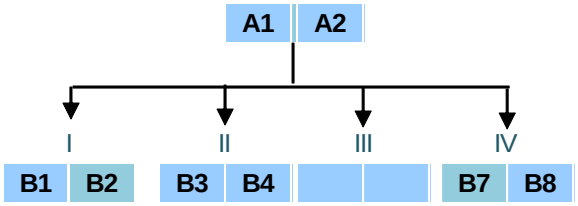


Figura 5.2: *ReorgAll* - Representação hierárquica após todos os elementos da hierarquia do nó em remoção ter sido fisicamente removidos

Figura 5.3: *ReorgAll* - Representação do vetor contendo os elementos armazenados conforme a largura da estrutura de indexação

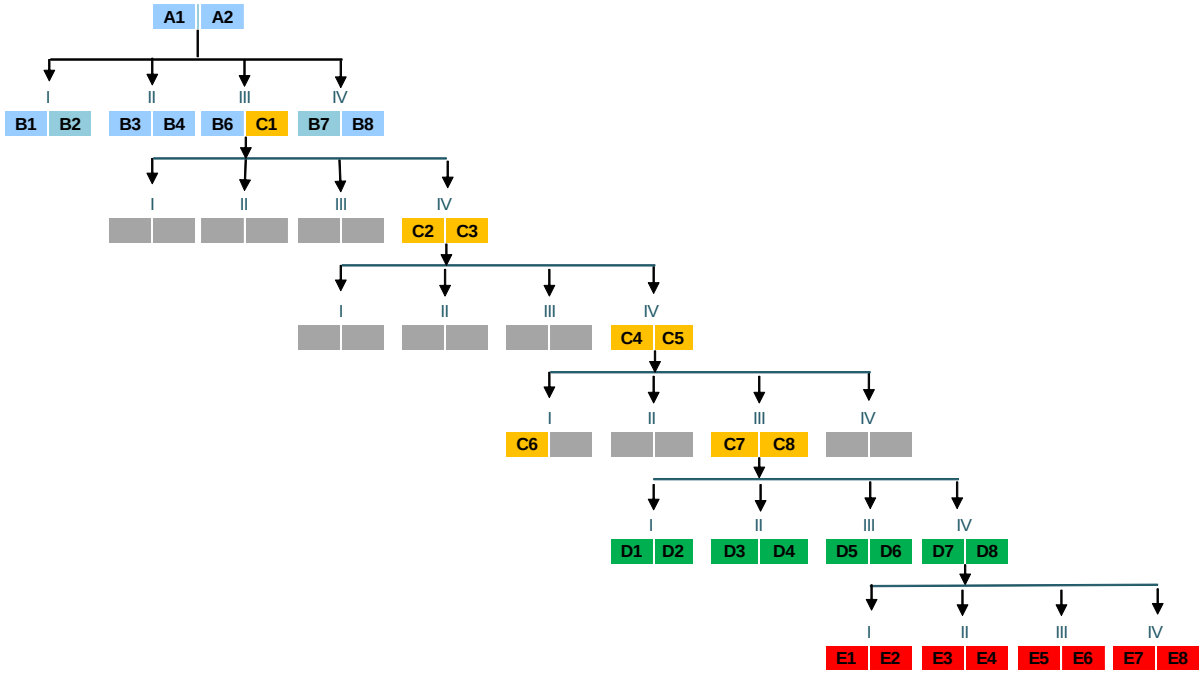


Figura 5.4: ReorgAll – Representação hierárquica de situação final de remoção

A Figura 5.5 e a Figura 5.6 ilustram as respectivas representações espacial inicial e final para este exemplo.

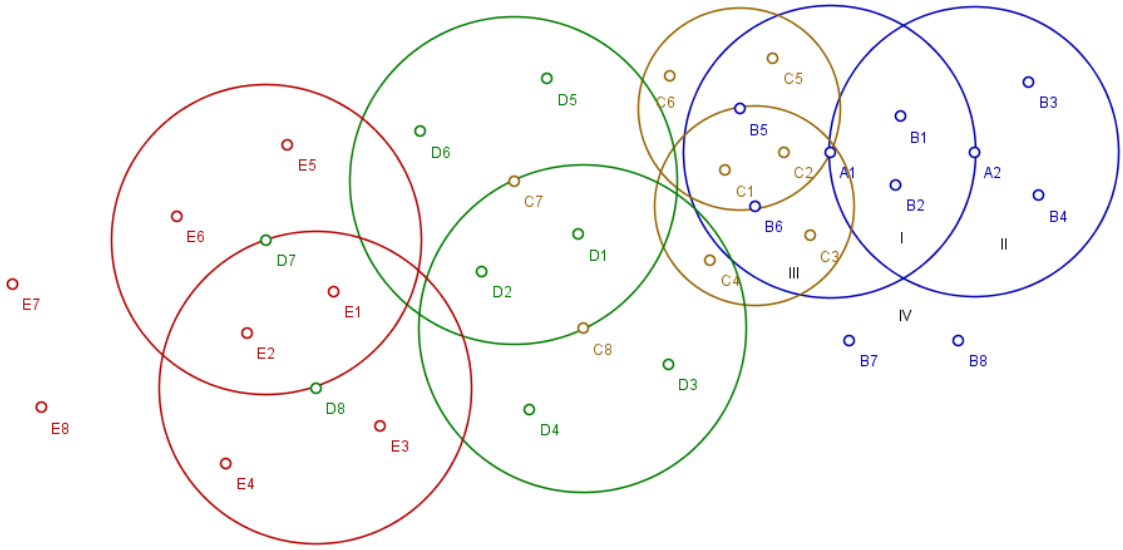


Figura 5.5: *ReorgAll* – Representação espacial de situação inicial de remoção

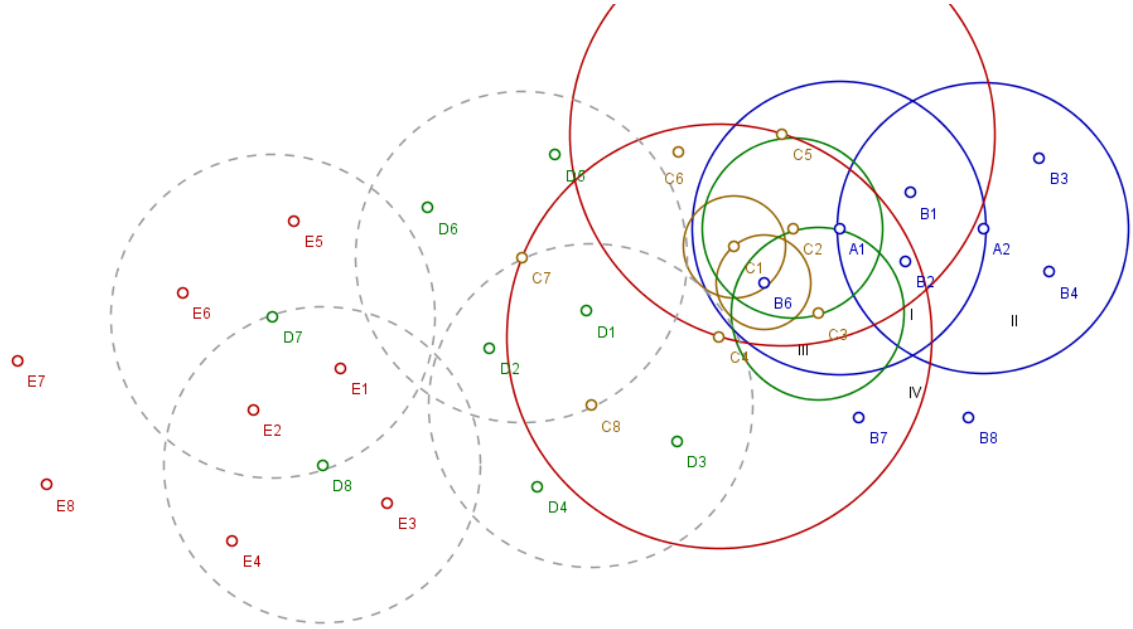


Figura 5.6: ReorgAll – Representação espacial de situação final de remoção

Embora os elementos C1, C2, C3, C4, C5, C6, C7 e C8 inicialmente no terceiro nível da estrutura, após a remoção tenham sido alocados de forma mais esparsa, os elementos dos quarto e quinto níveis, ou seja, os elementos D1, D2, D3, D4, D5, D6, D7, D8, E1, E2, E3, E4, E5, E6, E7 e E8 foram alocados com a mesma distribuição inicial, porém em outros níveis da estrutura. Note que neste exemplo a

estrutura de indexação após a remoção teve aumentada em 2 níveis a sua altura. Porém, para outros casos espera-se que na reinserção de elementos, o algoritmo de inserção consiga organizar melhor o índice e assim melhorar a taxa de ocupação dos nós e consequentemente reduzir o aumento da altura da Onion-tree.

5.2 **Algoritmo *PromoteNode***

O algoritmo *PromoteNode* estende o algoritmo *ReorgAll*. Durante o armazenamento dos elementos para posterior reinserção, este algoritmo verifica se a reorganização pode ser abreviada através da promoção de outro nó, dentre seus nós descendentes, que possa substituir o nó que possui o elemento que sofreu a remoção.

Na Onion-tree, uma hierarquia necessita de reorganização quando a distância entre os elementos pivôs de um nó é alterada. Se esta distância não se altera, a hierarquia se mantém. A ideia é localizar um nó substituto àquele que sofreu a remoção de forma a aproveitar toda sua hierarquia e reorganizar apenas os elementos no caminho entre o nó removido e aquele que o substituirá. Para isso, as distâncias entre os pivôs do nó pai do nó que possui o elemento removido e as distâncias dos pivôs do nó substituto devem atender as condições necessárias para ocupar a região em que se encontra o nó que possui o elemento removido. Espera-se com isso, minimizar o custo de execução da operação de remoção.

Assim, sejam A e B os pivôs do nó pai do nó que possui o elemento removido. Sejam C e D os pivôs de um nó candidato à substituição (ou seja, de um nó descendente do nó que possui o elemento removido). Seja Dist(x, y) a distância de um elemento a outro. Para que o nó candidato possa substituir o nó que possui o elemento removido, as distâncias Dist(C,A), Dist(C, B), Dist(D,A) e Dist(D,B) devem obedecer aos relacionamentos de <, <=, >, >= em relação a Dist(A,B), ou seja, estes relacionamentos devem estar de acordo com os relacionamentos estabelecidos para ocupar a região que o nó que possui elemento removido ocupa.

Por exemplo, para um nó pai de 4 regiões, se o nó que possui o elemento removido estiver na região 4, todas as distâncias entre os pivôs do nó candidato à substituição e os pivôs do nó pai do nó que possui o elemento removido devem ser maior ou igual a Dist(A,B).

A Figura 5.7, a Figura 5.8 e a Figura 5.9 ilustram o funcionamento do algoritmo *PromoteNode*. Como no exemplo anteriormente citado, para simplificação do entendimento e clareza gráfica, o exemplo destas figuras também considera uma Onion-tree com política fixa sem expansão, gerando 4 regiões em cada nó. A Figura 5.7 ilustra as condições a serem satisfeitas para que um dado elemento ocupe cada uma das regiões do nó contendo os representantes A1 e A2. A distância entre eles é 0,83.

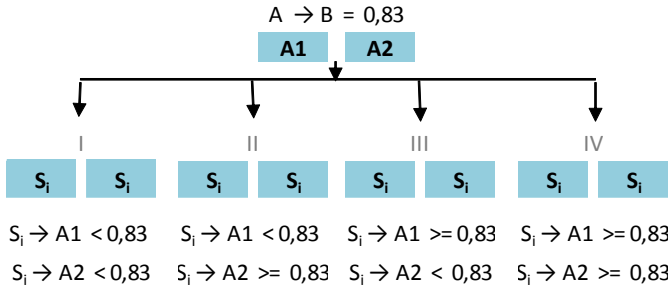


Figura 5.7: *PromoteNode* – Representação das condições para ocupação de cada região do nó que contém os elementos A e B

A Figura 5.8 ilustra para o nó acima, uma hierarquia e uma situação de remoção com condições para a promoção de um nó descendente em substituição ao nó que contém o elemento removido. O elemento B7 (em vermelho) está sendo removido. Dentre seus descendentes existe o nó que contém elementos C7 e C8. Os elementos deste nó satisfazem as condições descritas na Figura 5.7 para ocupar a região IV na qual se encontra o nó que contém o elemento B7. Em relação ao raio do nó pai do nó que contém o elemento B7, as distâncias dos elementos C7 e C8 são, em relação ao elemento A1 e em relação ao elemento A2, maiores ou iguais que 0,83.

A Figura 5.9 ilustra a hierarquia final após a promoção do nó que contém os elementos C7 e C8 em substituição ao nó que contém o elemento removido B7. Toda hierarquia do nó contendo os elementos C7 e C8 foi mantida. Assim como o elemento B8, pivô parceiro do elemento removido B7, os seis elementos no caminho entre o nó que contém o elemento B7 e o nó que contém os elementos C7 e C8 foram reinseridos, ou seja, os elementos C9, C10, C11, C12, C13 e C14. É importante observar que os elementos reinseridos podem ser alocados em qualquer nó do índice.

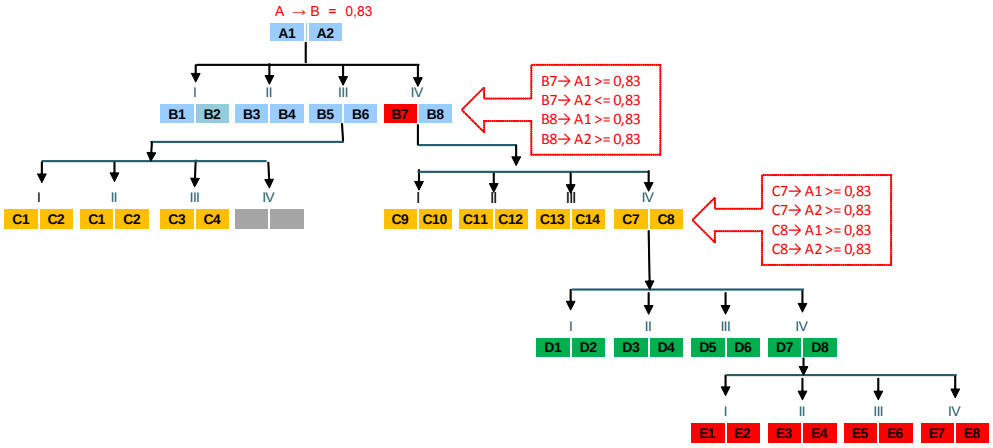


Figura 5.8: *PromoteNode* – Representação hierárquica da situação inicial para uma remoção com condições de promoção de um nó descendente

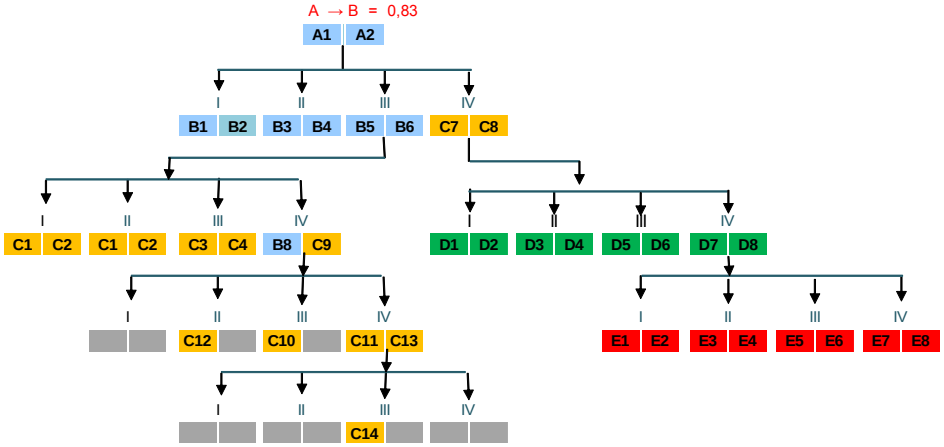


Figura 5.9: *PromoteNode* – Representação hierárquica da situação final para uma remoção com condições de promoção de um nó descendente

A Figura 5.10 e a Figura 5.11 ilustram as respectivas representações espaciais inicial e final para este exemplo.

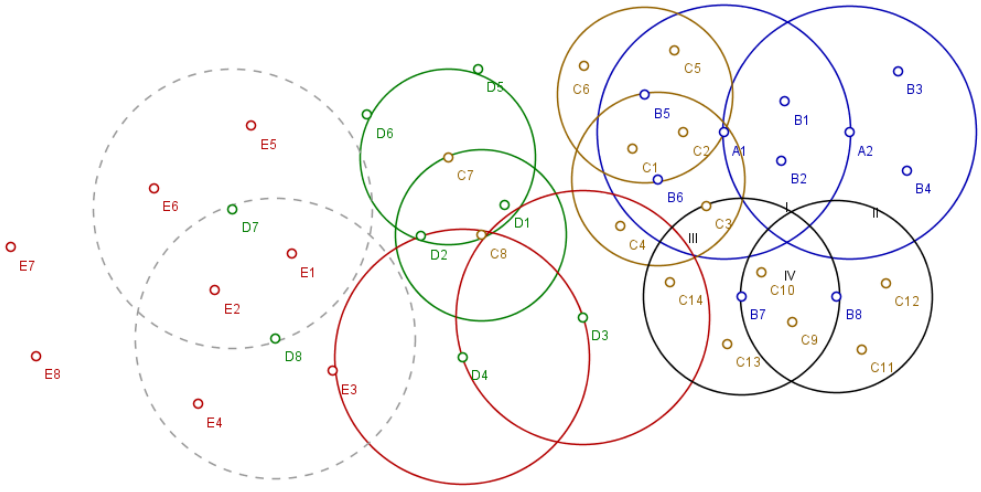


Figura 5.10: *PromoteNode* – Representação espacial da situação inicial para uma remoção com condições de promoção de um nó descendente

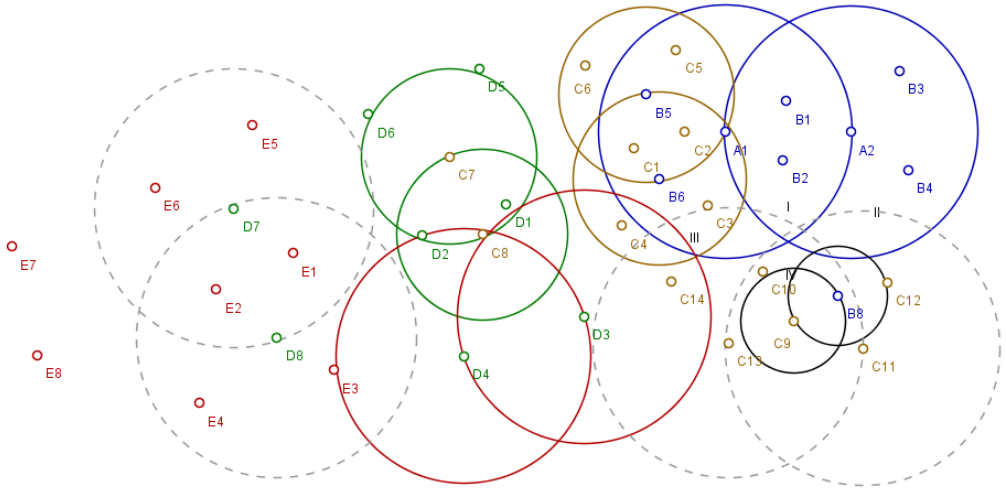


Figura 5.11: *PromoteNode* – Representação espacial da situação final para uma remoção com condições de promoção de um nó descendente

É importante observar, que a promoção de uma hierarquia tem maior probabilidade de ocorrer quando a remoção acontece nos primeiros níveis da estrutura. Além disso, esta probabilidade também é maior quando o nó que possui o elemento removido está na região mais externa de seu nó pai, ou seja, encontra-se, por exemplo, na região IV de um nó sem expansão ou na região VII de um nó com uma expansão. Como a região mais externa é a região mais extensa, há maior probabilidade de encontrar, entre os descendentes do nó removido, candidatos que estejam, em relação ao nó pai do nó que possui o elemento removido, também na região externa.

5.3 **Representação Formal dos Algoritmos**

Algoritmo 4
Remove fisicamente um elemento da estrutura de indexação
DeleteElement(Node, Father, Element, Algorithm)

	Input: *Node (nó em pesquisa para encontrar elemento a ser removido), Father (nó pai), Element (elemento a remover), Algorithm(algoritmo em execução)*
	Output: null
1	**If** *Node* **= null then return;**
2	**else**

```
3     d1 ← distance(Element, Node.Pivot1);
4     If d1 = 0 then Element for delete is Node.Pivot1
5           // Pivot1 is the element to be deleted then
6           // Save Pivot2
7           ElementToKeep ← Node.Pivot2
8           if Algorithm = PromoteNode then
9             // If algorithm is PromoteNode then identify the father´s region of this node
10            RegionSub ← Region of Node in his father;
11          end
12          // Reorganize this Node´s hierarchy
13          ReorgHierarchy(Node, Father, ElementToKepp, Algorithm, RegionSub)
14          Return
15    end
16
17    d2 ← distance(Element, Node.Pivot2)
18    If d2 = 0 then Element for delete is Node.Pivot2
19          // Pivot2 is the element to be deleted then
20          // Save Pivot1
21          ElementToKeep ← Node.Pivot1;
22          if Algorithm = PromoteNode then
23            // If algorithm is PromoteNode then identify the father´s region of this node
24            RegionSub ← Region of Node in his father;
25          end
26          // Reorganize this Node´s hierarchy
27          ReorgHierarchy(Node, Father, ElementToKepp, Algorithm, RegionSub)
28          Return
29    end
30
31    // Continue the search
32    For Region ← 1 to all Node´s sons do
33      if Query radius interesects Node´s son region then
34        DeleteElement(Node´s son, Node, Element, Algorithm);
35      end
36    end for
37  end
```

Algoritmo 5

Executado pelos algoritmos RegorgAll e PromoteNode reorganiza a hierarquia do nó que teve um elemento removido

ReorgHierarchy(NodeBeingDeleted, Father, ElementToKeep, Algorithm, RegionSub)

Input: *NodeBeingDeleted (nó em remoção), Father (pai do nó em remoção), ElementToKeep (par do elemento que está em remoção), Algorithm(algoritmo em execução), RegionSub (Região do nó sendo removido em seu nó pai)*

Output: null

```
1   // Save ElementToKeep
2    Add ElementToKeep to the ElementsToReinsert;
3
4   // NodesToVerify will store the elements of the same level
5   //  for verify if there is a substitute for NodeBeingDeleted
6   //  First, save NodeBeingDeleted´s child
7   For NodeBeingDeleted.Son ← 1 to to all NodeBeingDeleted´s sons do
8      add NodeBeingDeleted.son to NodesToVerify;
9   end for
10
11  // Read the elements of the level and verify if there is a substitute
12  While NodesToVerify is not empty
13
14    NodeToVerify = The first element of the queue
15
16    if Algorithm = PromoteNode then
17      // Verify if this node can replace NodeBeingDeleted
18      if IsSub(NodetoVerify, FatherNodeBeingDeleted, RegionNodeBeingDeleted) then
19        // This node (NodeToVerify) can replace NodeBeingDeleted
20        FatherNodeBeingDeleted.Son[RegionSub] ← NodeToVerify; // Replace
21      else
22        // This node (NodeToVerify) can´t replace NodeBeingDeleted, so store it for reinsert
23        Add NodeToVerify to ElementsToReinsert
24      end
25    else
26      // The algorithm is ReorgAll, so store this node for futher reinsert
27      Add NodeToVerify to ElementsToReinsert
28    end
29
30    // Save all childs of NodeToVerify In the end of the queue
31    // for verify the next level
32    For NodeToVerify.Son ← 1 to to all NodeToVerify sons do
33      add NodeToVerfy.Son to NodesToVerify;
34    end for
35
36     // This node has already being verified, so remove it from the queue
37     Remove NodeToVerify from the queue
38
39   end while
40
41   For i = 1 to all ElementsToReinsert do;
42     Reinsert (ElementsToReinsert[i]);
43   end for
44
```

Algoritmo 6

Executado apenas pelo algoritmo PromoteNode, verifica se o nó pode substituir o nó em remoção

IsSub(Node, Father)

Input: *NodeSub (nó em análise para substituição), Father (pai do nó em remoção), RegionSub (Região do nó em remoção em seu nó pai)*

```
        Output: boolean

1       d1 ← distance(NodeSub.Pivot 1, Father.Pivot1)
2       d2 ← distance(NodeSub.Pivot1, Father.Pivot2)
3       // What region could be assigned for NodeSub.Pivot1 in Father´s NodeBeingDeleted
4       RegionS1 ← ChooseRegion(Father,d1,d2)
5
6       d1 ← (NodeSub.Pivot2, Father.Pivot1)
7       d2 ← (NodeSub.pivot2, Father.Pivot2)
8       // What region could be assigned for NodeSub.Pivot2 in Father´s NodeBeingDeleted
9       RegionS2 ← ChooseRegion(Father,d1,d2)
10
11      // If both regions are the same as NodeBeingDeleted region in his father
12      // this node can substitute NodeBeingDeleted
13      If RegionS1 = RegionSub && RegionS2 = RegionSub then
14        return true;
15      end
```

Algoritmo 7 - Determina com base no nó pai, a região de um elemento
ChooseRegion (Father, d_1, d_2)

```
        Input: Father (nó), d 1 and d 2 (distâncias entre o elemento a ser inserido e os
        representantes)
        Output: número da região em que o elemento deverá ser inserido

1       Region ← 0; R ← 0; // (variáveis auxiliares)
2       for Expansion ← 0 to all Father.Expansions do
3         R ← R + Father.radius;
4         if d 1 < R and d 2 < R then Region ← 1; break;
5         if d 1 < R and d 2 >= R then Region ← 2; break;
6         if d 1 >= R and d 2 < R then Region ← 3; break;
7         if d 1 >= R and d 2 >= R then Region ← 4; break;
8       end for
9       if Region = 0 then
10        Expansion ← Expansion - 1;
11        Region ← 4;
12      end
13      return (Expansion x 3) + Region;
```

5.4 **Ambiente de Testes**

Para a análise de desempenho dos algoritmos *ReorgAll e PromoteNode* foi utilizado o conjunto de dados *KDD Cup 2008*. O computador utilizado nos testes e

os parâmetros utilizados para a construção da Onion-tree foram os mesmos utilizados nos testes descritos no Capítulo 4 (Seção 4.5).

Para avaliação dos algoritmos foram elaborados procedimentos de teste com o objetivo de medir, além do desempenho no processamento de consultas posteriores à remoção, também o custo da operação de remoção determinado pela reorganização da hierarquia afetada pela remoção. Assim como nos testes descritos no Capítulo 4, os procedimentos executaram consultas por abrangência com raio dinâmico, previamente calculado por uma consulta aos k-vizinhos mais próximos, que retornou os 10 elementos mais próximos a cada elemento consultado.

A primeira configuração de teste (Conf1) foi elaborada com foco na avaliação do desempenho no processamento de consultas posteriores a operação de remoção e executou grande quantidade de remoções. Foram selecionados aleatoriamente 30.000 elementos do conjunto de dados *KDD Cup 2008*, observando-se que este conjunto possuísse elementos de todos os níveis da estrutura. Neste procedimento foram consultados após as remoções, todos os 72.240 elementos que restaram no conjunto de dados, tendo si próprio como centro da consulta.

A segunda configuração de teste (Conf2) teve foco na avaliação do custo da operação de remoção e removeu uma quantidade menor de elementos, porém em níveis específicos da estrutura. Assim, com base na altura média da estrutura inicialmente construída, que é de 12 níveis, foram selecionados 500 elementos para remoção em três extratos da estrutura de indexação: (i) elementos dos níveis 1, 2 e 3, ou seja, elementos no nó raiz e nos níveis imediatamente abaixo do nó raiz; (ii) elementos dos níveis 6, 7 e 8, ou seja, elementos de níveis intermediários do índice; e (iii) elementos folha e pai exclusivamente de folha. Para as consultas foi utilizado um único conjunto de 1.000 elementos, selecionados aleatoriamente a partir do conjunto de dados *KDD Cup 2008,* dentre aqueles que não sofreriam remoção em quaisquer dos níveis avaliados.

Em cada configuração de teste, foram analisados o algoritmo *LogicalDelete*, e os algoritmos de remoção física *ReorgAll* e *PromoteNode*. Para isso, o desempenho no processamento destas estruturas após a remoção foi analisado sob dois aspectos: (i) com base no algoritmo *LogicalDelete;* e (ii) com base na estrutura reconstruída com os elementos não removidos. A primeira requer cálculos de distância desnecessários. A segunda, assim como as estruturas resultantes da remoção física, é livre de cálculos de distância desnecessários. Assim, a análise

tendo como *baseline* o algoritmo *LogicalDelete,* possibilitou observar o efeito da quantidade desnecessária de cálculos de distância no processamento de consultas posteriores à remoção, e a análise tendo como *baseline* a reconstrução do índice possibilitou observar o desempenho dos algoritmos de remoção física, também livres de cálculos de distância desnecessários.

As configurações de teste executaram as seguintes etapas:

A. Execução do algoritmo de remoção lógica *LogicalDelete* e dos algoritmos de remoção física *RegorgAll* e *PromoteNode*
    1. Construção da estrutura de indexação dos 102.240 elementos do conjunto de dados *KDD Cup 2008*;
    2. Remoção de elementos
        a) Na Conf1, remoção de 30.000 elementos;
        b) Na Conf2, remoção de 500 elementos do nível em análise;
    3. Realização de consultas por abrangência
        a) Na Conf1, consultar os 72.240 elementos restantes no conjunto de dados;
        b) Na Conf2, consultar 1.000 elementos dentre aqueles não selecionados para remoção em qualquer dos níveis avaliados.

B. Reconstrução do índice
    1. Construção da estrutura de indexação com os elementos não removidos
        a) Na Conf1, construção de 73.240 elementos;
        b) Na Conf2, construção de 101.740 elementos;
    2. Realização de consultas por abrangência
        a) Na Conf1, 72.240 elementos;
        b) Na Conf2, consultar 1.000 elementos dentre aqueles não selecionados para remoção em qualquer dos níveis avaliados.

Desta maneira, o desempenho no processamento de consultas foi avaliado, comparando-se entre os algoritmos e a construção com os elementos não removidos, os totais de tempo e quantidade de cálculos de distância das consultas, e o perfil final da árvore, representado pelas variáveis de tamanho em bytes da estrutura, quantidade total de nós e de nós folha, altura máxima e altura média da árvore. As seções 5.5, 5.6 e 5.7 descrevem os resultados obtidos.

5.5 Impacto da Remoção no Desempenho da Onion-tree no Processamento de Consultas

Conforme mencionado na Seção 5.4, esta seção discute o desempenho da Onion-tree no processamento de consultas após a remoção sob dois aspectos: (i) os efeitos de cálculos de distância desnecessários no desempenho de consultas após a remoção, e para isso usa como base de comparação o algoritmo *LogicalDelete;* e, (ii) o desempenho dos algoritmos de remoção física *ReorgAll* e *PromoteNode*, usando como base de comparação, a reconstrução da estrutura com os elementos não removidos. O desempenho da Onion-tree no processamento de consultas após a remoção é analisado através do tempo total do processamento das consultas e quantidade de cálculos de distância.

5.5.1 Os Efeitos de Cálculos de Distância Desnecessários no Desempenho da Onion-tree no Processamento de Consultas após a Remoção

A Figura 5.12 mostra os resultados obtidos na Conf1, que remove 29,34% do total de elementos do conjunto de dados *KDD Cup 2008*. Os resultados do algoritmo *LogicalDelete* são comparados com os resultados obtidos para as estruturas livre de cálculo de distância desnecessários (algoritmos *ReorgAll*, *PromoteNode* e reconstrução da estrutura).

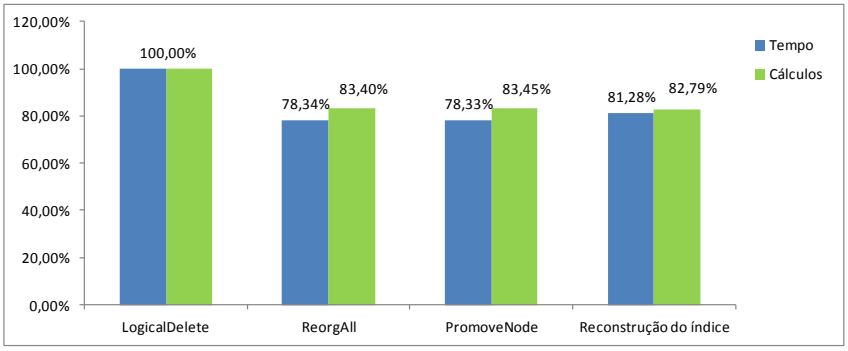


Figura 5.12: Remoção física - Com base no algoritmo *LogicalDelete,* após remoção de 30.000 elementos

Em cálculos de distância, os algoritmos de remoção física e a reconstrução do índice com os elementos não removidos necessitaram quantidade entre 16,66% e 17,21% menor que o necessário à remoção lógica. O tempo total das consultas também foi menor que o necessário à remoção lógica. Para os algoritmos *ReorgAll* e

PromoteNode foi menor em aproximadamente 21,66%, e para a reconstrução do índice, menor em 18,72%.

Em outra visão para os mesmos resultados, a Figura 5.13 ilustra a comparação com base na estrutura reconstruída com os elementos não removidos. A quantidade de cálculos de distância requerida pela remoção lógica foi 20,79% maior, e as consultas consumiram 23,04% mais tempo que a reconstrução dos elementos não removidos.

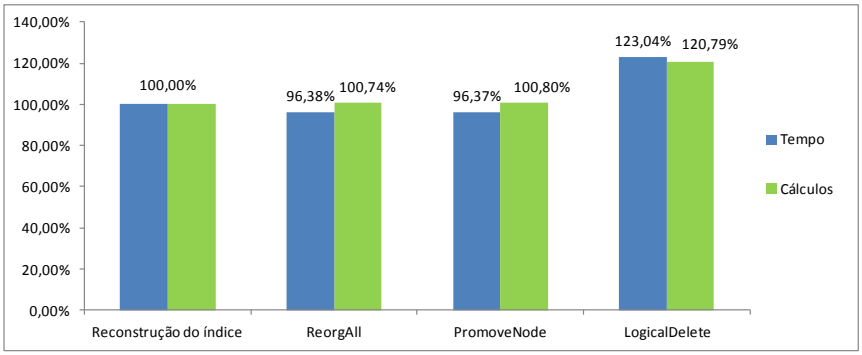


Figura 5.13: Remoção física - Com base na reconstrução dos 72.240 elementos não removidos, após remoção de 30.000 elementos

A Figura 5.14 e a Figura 5.15 ilustram os resultados obtidos na Conf2 que remove 0,48% do total de elementos em níveis específicos do índice tendo como base de comparação o algoritmo *LogicalDelete*. Com menor quantidade de remoções, verifica-se que a remoção lógica tem melhor desempenho que na Conf1.

A Figura 5.14 mostra que em cálculos de distância, para os três níveis analisados, a reconstrução da estrutura com os elementos não removidos necessitou em média, de quantidade maior em 1,57%. Com relação aos algoritmos *ReorgAll* e *PromoteNode*, nos níveis intermediários (níveis 6,7 e 8) e folha e pai de folha, a quantidade de cálculos de distância necessária à remoção lógica é praticamente a mesma que a necessária para estes algoritmos. Porém, nos primeiros níveis da estrutura, ou seja, os níveis 1,2 e 3 verifica-se que a remoção lógica necessita em relação ao algoritmo ReorgAll de quantidade menor em cálculos de distância, em 2,69%, e em relação ao algoritmo PromoteNode de quantidade maior em 4,35%.

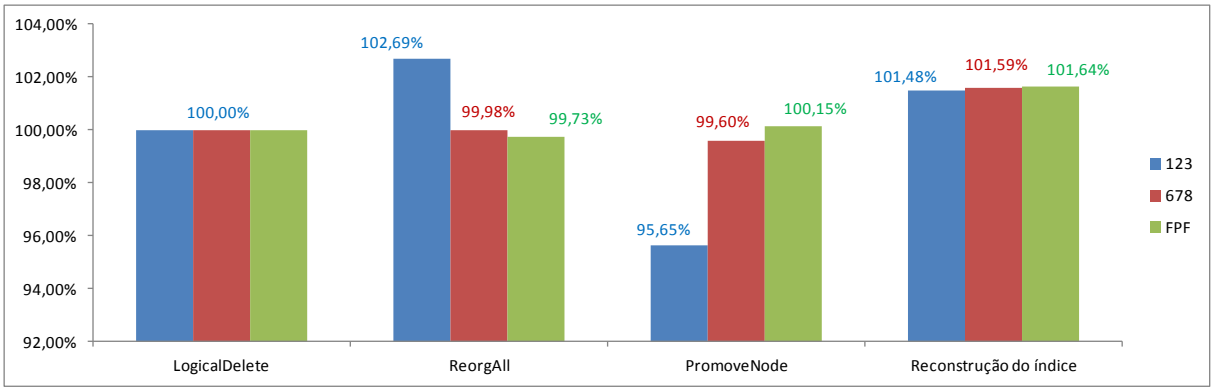


Figura 5.14: Remoção física - Com base no algoritmo *LogicalDelete*, cálculos de distância das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A Figura 5.15 mostra que no desempenho do processamento posterior de consultas, para os três níveis analisados, a reconstrução da estrutura com os elementos não removidos necessitou em média, 1,99% mais tempo que a remoção lógica. Com relação aos algoritmos de remoção física, a remoção lógica apresenta praticamente o mesmo desempenho que o algoritmo *ReorgAll*. Porém em relação ao algoritmo *PromoteNode*, a remoção lógica necessita de mais tempo maior para processar as consultas após a remoção. Nos primeiros níveis da estrutura, ou seja, os níveis 1, 2 e 3, tempo de processamento das consultas é maior em 7,57%, e nos níveis intermediários (níveis 6,7 e 8) e folha e pai de folha, maior em 1,56%.

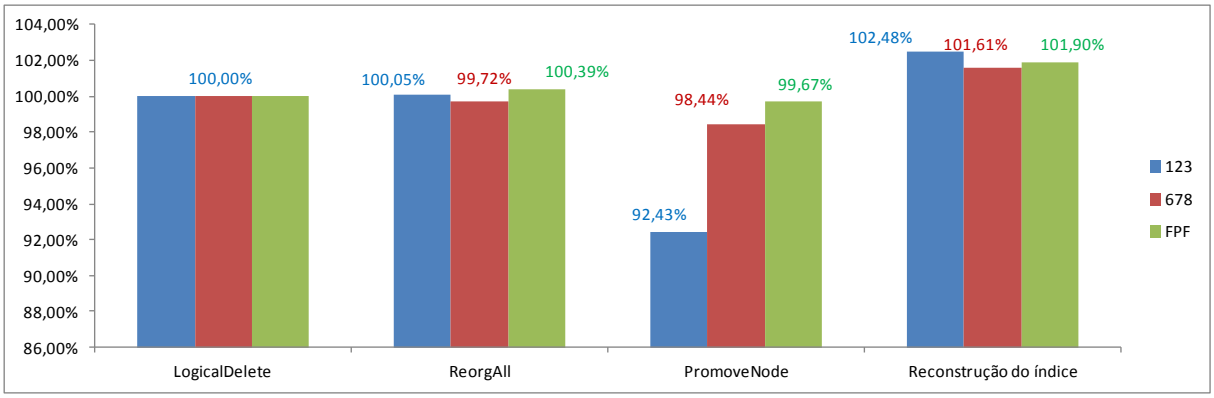


Figura 5.15: Remoção física - Com base no algoritmo *LogicalDelete*, tempo total das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A Figura 5.16 e a Figura 5.17 ilustram os mesmos resultados comparados com base na estrutura reconstruída com os elementos não removidos. Nos níveis analisados, a remoção lógica necessita em média de quantidade 1,54% menor em

cálculos de distância, e em média 1,96% menos tempo nas consultas, sendo que nos níveis 1, 2 e 3 o ganho foi de 2,42%.

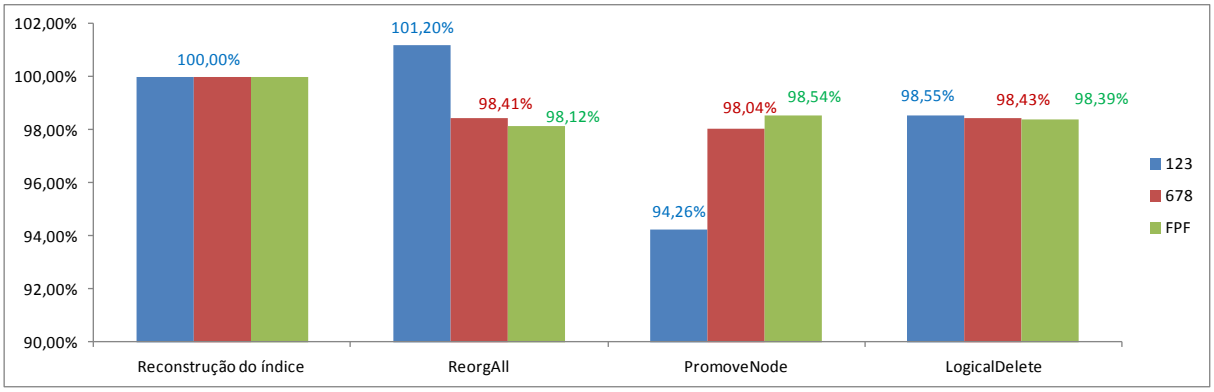


Figura 5.16: Remoção física - Com base na reconstrução dos 101.740 elementos não removidos, cálculos de distância das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

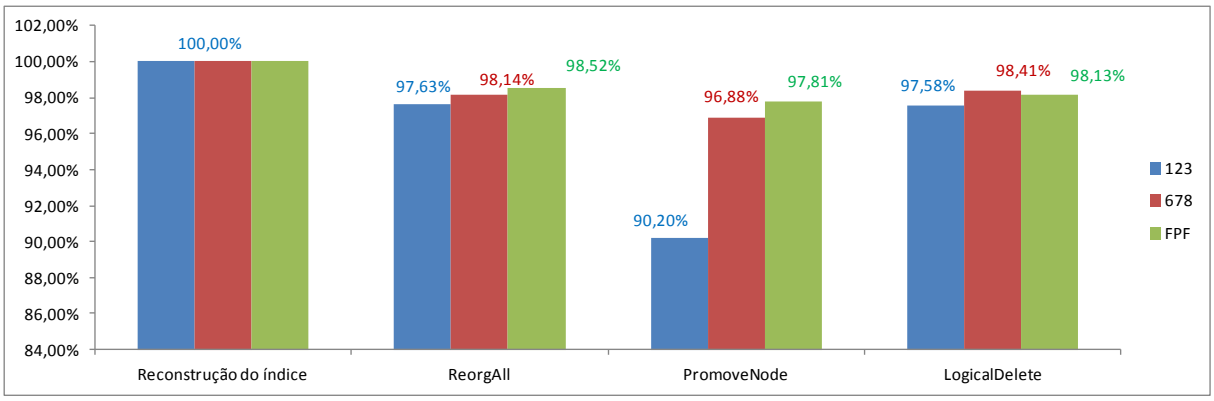


Figura 5.17: Remoção física - Com base na reconstrução dos 101.740 elementos não removidos, tempo total das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

Assim, a grande diferença na quantidade de remoções entre as configurações de teste possibilitou observar melhor o comportamento de cálculos de distância desnecessários no processamento de consultas após a remoção. Analisando o algoritmo *LogicalDelete* na Conf1, que remove 29,34% do total de elementos da base, verifica-se que o desempenho da remoção lógica no processamento de consultas após a remoção fica bem abaixo daquele verificado para as estruturas livre de cálculos de distância desnecessários. Entretanto, na Conf2 que executa pequena quantidade de remoções, apenas 0,48% do total de elementos da base, a

execução desnecessária de cálculos de distância no processamento de consultas não tem efeito determinante no desempenho total das consultas.

5.5.2 Análise dos algoritmos *ReorgAll* e *PromoteNode*

Uma vez que o algoritmo *PromoteNode* é uma extensão do algoritmo *ReorgAll*, para uma correta avaliação dos algoritmos, primeiramente foi necessário certificar-se que o algoritmo *PromoteNode* tenha encontrado situações que possibilitaram a substituição de um nó removido por outro de sua hierarquia. Assim, na Conf1, comparando-se entre as estruturas resultantes as posições de cada elemento após a remoção, verificou-se que o algoritmo *PromoteNode* produziu uma estrutura com apenas 10,13% dos elementos na mesma posição que aquela resultante pelo algoritmo *ReorgAll*. Além disso, 35,39% dos elementos foram alocados em níveis inferiores e 54,48% em níveis superiores àqueles resultantes pelo algoritmo *ReorgAll.* Esta análise comprovou que o algoritmo *PromoteNode* efetivamente executou a condição de promover outro nó (e sua hierarquia) em substituição ao nó removido. A Tabela 5.1 mostra a porcentagem de elementos que em relação à base diminuíram, permaneceram ou aumentaram de nível após a remoção.

Tabela 5.1: Remoção física - Elementos movimentados entre os níveis

Base	Comparação	(-)	(=)	(+)
Após construção do índice	ReorgAll	37,57%	8,27%	54,17%
	PromoteNode	30,62%	7,09%	62,29%
ReorgAll	PromoteNode	35,39%	10,13%	54,48%

Analisando o desempenho os algoritmos *ReorgAll* e *PromoteNode*, em grande quantidade de remoções verificamos que os algoritmos de remoção física obtiveram, em cálculos de distância, desempenho igual àquele verificado para as consultas na estrutura reconstruída com os elementos não removidos. Este resultado pode ser verificado na Figura 5.18. Conforme mencionado na seção 5.1, a reorganização da hierarquia afetada pela remoção na mesma ordem que a inserção dos elementos na construção inicial mantém a estrutura resultante mais próximo possível da estrutura original, aumentando a probabilidade de manter o desempenho original.

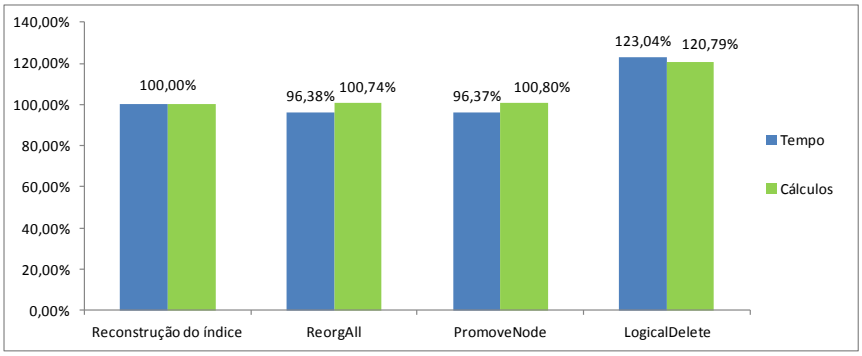


Figura 5.18: Remoção física - Com base na reconstrução dos 72.240 elementos não removidos, após remoção de 30.000 elementos

Em relação ao tempo total gasto pelas consultas, conforme a Figura 5.18, os algoritmos *ReorgAll* e *PromoteNode* obtiveram 3,62% de ganho. Todavia, a distribuição da quantidade de elementos conforme o desempenho, ilustrada pela Tabela 5.2, revela que para os algoritmos *ReorgAll* e *PromoteNode,* respectivamente 66,96% e 66,00% dos elementos que permaneceram no índice tiveram tempo de consulta menor ou igual à reconstrução dos elementos não removidos. Ainda respectivamente, apenas 14,61% e 15,18% destes elementos tiveram tempo de consulta 10% maior que o verificado para as consultas na estrutura reconstruída com os elementos não removidos.

Tabela 5.2: Remoção física (tempo) - Distribuição dos elementos conforme o desempenho em relação à reconstrução dos 72.240 elementos não removidos

Tempo de consulta	ReorgAll	PromoteNode
Menor ou igual	66,96%	66,00%
Até 10% maior	18,43%	18,82%
Maior 10%	14,61%	15,18%

A Figura 5.24, Figura 5.25, Figura 5.26 e a Figura 5.31 mostram que com pequena quantidade de remoções, verificamos na Conf2, onde a remoção é aplicada em níveis específicos da estrutura, que tanto com base no algoritmo *LogicalDelete*, quanto com base na reconstrução dos elementos não removidos, o algoritmo *PromoteNode* obteve o melhor desempenho. Sobretudo, este algoritmo tem ainda melhor desempenho quando avaliado com base na reconstrução dos 101.740 elementos não removidos. A Figura 5.24 e a Figura 5.25 mostram os resultados com base no algoritmo *LogicalDelete*. A Figura 5.26 e a Figura 5.31 ilustram os resultados com base na reconstrução com elementos não removidos.

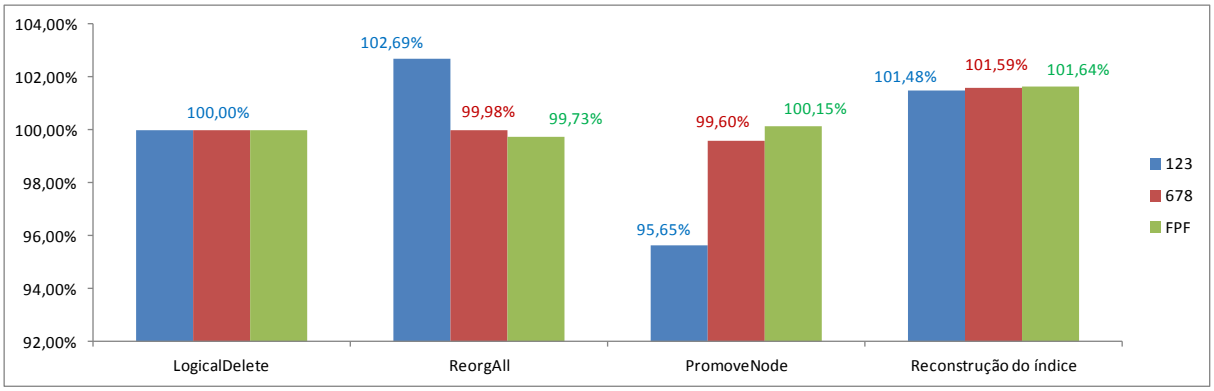


Figura 5.19: Remoção física - Com base no algoritmo *LogicalDelete*, cálculos de distância das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

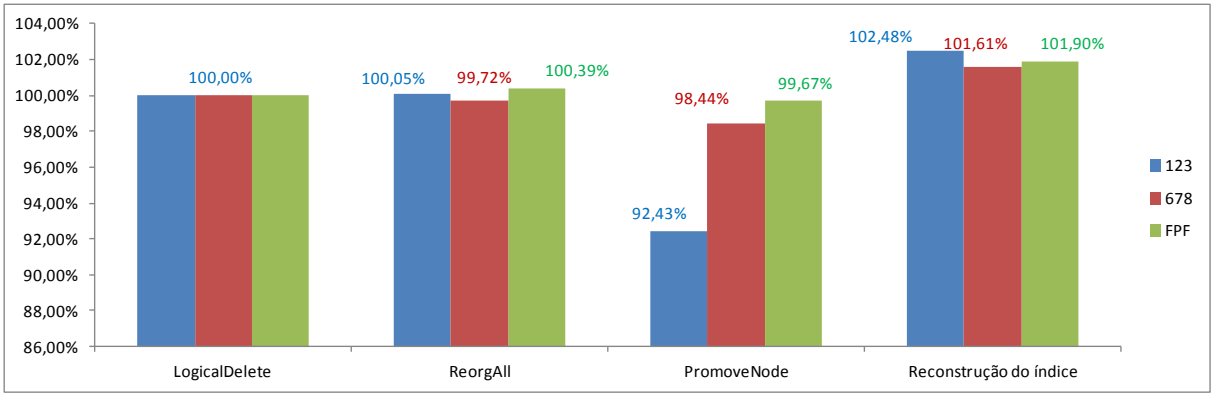


Figura 5.20: Remoção física - Com base no algoritmo *LogicalDelete*, tempo total das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

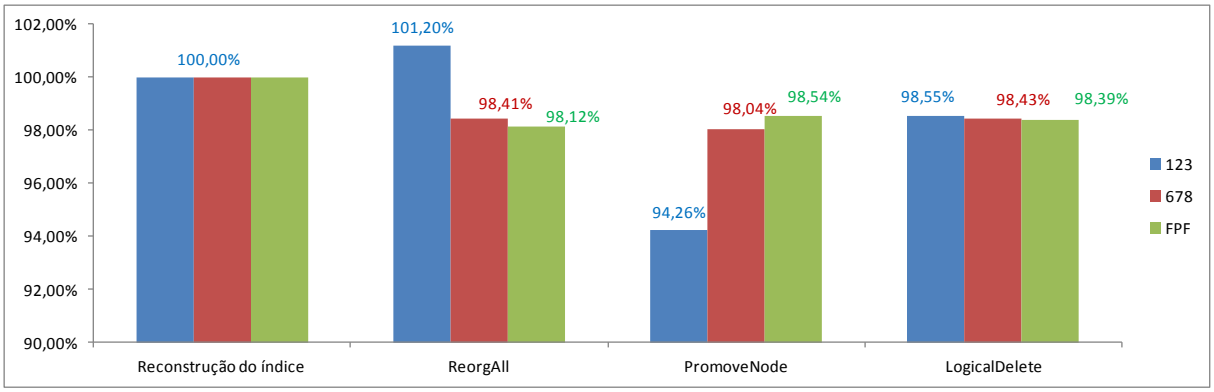


Figura 5.21: Remoção física - Com base na reconstrução dos 101.740 elementos não removidos, cálculos de distância das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

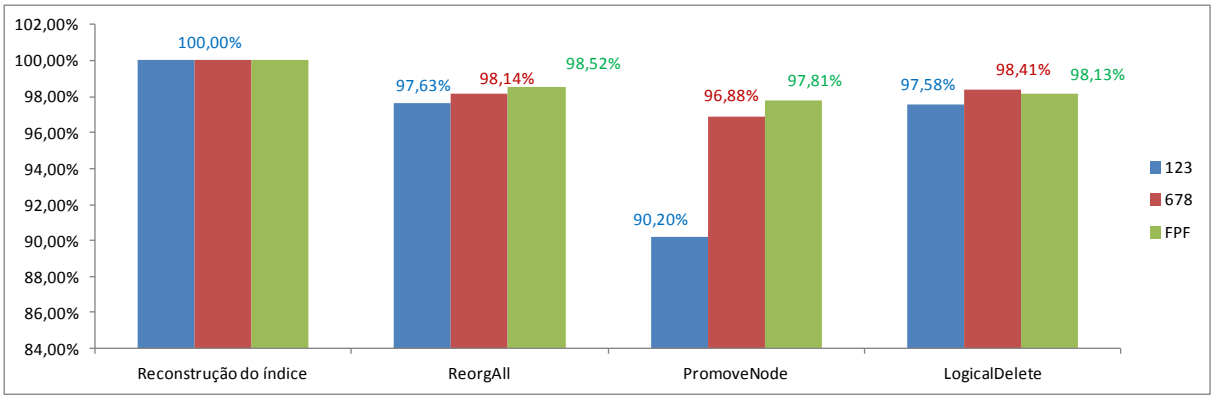


Figura 5.22: Remoção física - Com base na reconstrução dos 101.740 elementos não removidos, tempo total das consultas após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A Figura 5.23 e a Figura 5.24 mostram que no desempenho do algoritmo *PromoteNode* em relação ao algoritmo *LogicalDelete*, o maior ganho de desempenho do foi nos primeiros níveis da estrutura (extratos 1, 2 e 3), nos quais há maior probabilidade de se promover uma hierarquia em substituição ao nó removido. Nestes níveis, este algoritmo obteve 4,35% de ganho em cálculos de distância, e 7,57% de ganho em tempo das consultas. Nos níveis intermediários (os níveis 6, 7 e 8) houve um ganho de 1,56% no tempo das consultas. Nos níveis folha e pai de folha, de forma inerente à condição a ser avaliada, pois nestes níveis não se encontra oportunidade de promover uma hierarquia em substituição ao nó que sofreu a remoção, o algoritmo *PromoteNode* obteve o mesmo desempenho que o algoritmo *LogicalDelete*.

Com relação à estrutura reconstruída com os elementos não removidos, o algoritmo *PromoteNode* apresentou o mesmo desempenho verificado na comparação ao algoritmo *LogicalDelete,* porém com ganhos mais significativos. Em cálculos de distância o ganho foi de 5,74% nos níveis 1, 2 e 3; de 1,96% nos níveis intermediários, e 1,46% nos níveis folha e pai de folha. No tempo total das consulta o ganho foi de 9,8% nos níveis 1, 2 e 3; de 3,14% nos níveis intermediários, e 2,19% nos níveis folha e pai de folha. A Figura 5.23 e a Figura 5.24 ilustram estes resultados.

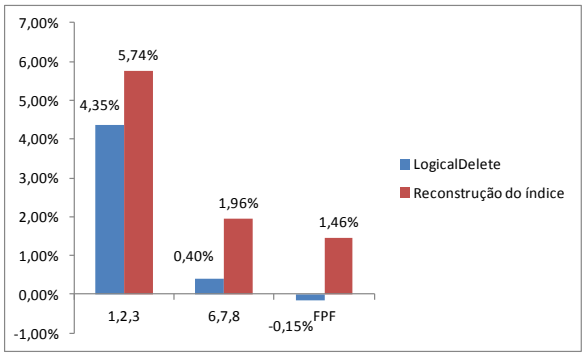


Figura 5.23: Remoção física – PromoteNode: Ganhos em cálculos de distância após remoção de 500 elementos em níveis específicos. O rótulo 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. O rótulo 678 indica o desempenho nos níveis intermediários, e o rótulo PFP indica o desempenho nos níveis pai de nós folha e folha do índice

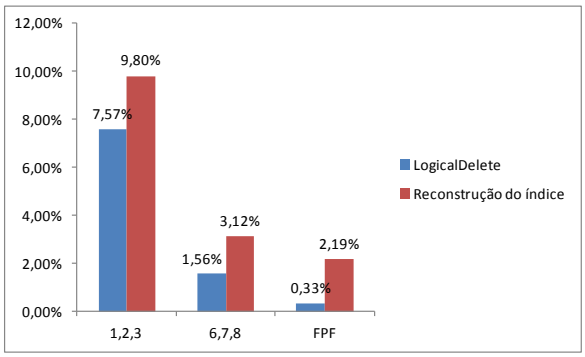


Figura 5.24: Remoção física – PromoteNode: Ganhos em tempo de consulta após remoção de 500 elementos em níveis específicos. O rótulo 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. O rótulo 678 indica o desempenho nos níveis intermediários, e o rótulo PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A Figura 5.25 e a Figura 5.26 comparam o desempenho do algoritmo *ReorgAll* em relação ao algoritmo *LogicalDelete. O* algoritmo *ReorgAll* obteve nos níveis intermediários e nos níveis folha e pai de folha o mesmo desempenho em cálculos de distância e tempo total de consulta. Nos níveis 1, 2 e 3 o tempo de consulta foi o mesmo, porém houve piora nos cálculos de distância em 2,69%. Com relação à reconstrução dos elementos não removidos, em cálculos de distância o ganho foi de 1,59% nos níveis intermediários e de 1,88% nos níveis folha e pai de folha. No tempo total das consultas o ganho foi de 1,86% nos níveis intermediários, e 1,48% nos níveis folha e pai de folha. Nos níveis 1, 2 e 3, embora com ganho de tempo de 2,37%, houve piora de 1,20% em cálculos de distância.

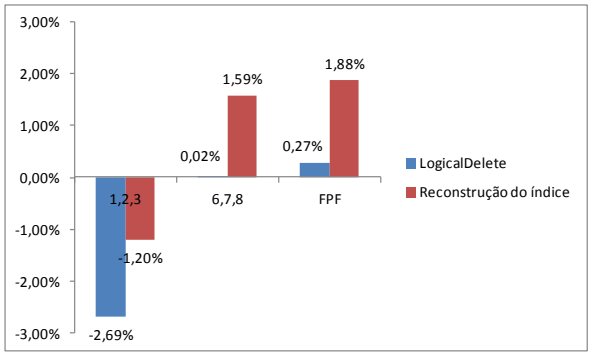


Figura 5.25: Remoção física – *ReorgAll*: Ganhos em cálculos de distância após remoção de 500 elementos em níveis específicos. O rótulo 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. O rótulo 678 indica o desempenho nos níveis intermediários, e o rótulo PFP indica o desempenho nos níveis pai de nós folha e folha do índice

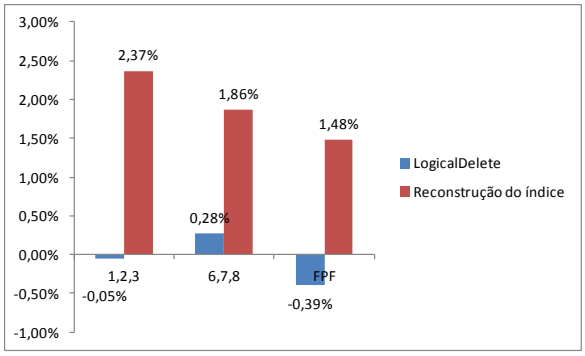


Figura 5.26: Remoção física – *ReorgAll*: Ganhos em tempo de consulta após remoção de 500 elementos em níveis específicos. O rótulo 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. O rótulo 678 indica o desempenho nos níveis intermediários, e o rótulo PFP indica o desempenho nos níveis pai de nós folha e folha do índice

Para o algoritmo *ReorgAll*, nos níveis intermediários, a menor quantidade de remoções combinada com a reorganização da hierarquia afetada pela remoção na ordem de inserção dos elementos na construção inicial (seção 5.1) fez com que este algoritmo, tanto para o tempo das consultas, quanto para a quantidade de cálculos de distância, atingisse desempenho muito próximo àquele verificado para o algoritmo *LogicalDelete*.

Em síntese, comparados à reconstrução dos elementos não removidos, com relação ao desempenho no processamento de consultas posterior à operação de remoção física, em pequena quantidade de remoções, os algoritmos de remoção física *ReorgAll* e *PromoteNode* obtiveram praticamente o mesmo desempenho em cálculos de distância. Com relação ao tempo de processamento, estes algoritmos obtiveram desempenho, em média, 3,62% superior à reconstrução dos elementos não removidos. Entretanto, para os extratos analisados da estrutura, o algoritmo

PromoteNode obteve melhor desempenho que o algoritmo *ReorgAll*. O algoritmo *PromoteNode* obteve ganhos, de 9,8% nos primeiros níveis da estrutura (extratos 1,2 e 3), de 3,12% nos níveis intermediários (extratos 6,7 e 8), e de 2,19% nos níveis folha e pai de folha. O algoritmo *ReorgAll* obteve ganhos, de 2,7% nos primeiros níveis da estrutura (extratos 1,2 e 3), de 3,12% nos níveis intermediários (extratos 6,7 e 8), e de 2,19% nos níveis folha e pai de folha.

5.6 **Custo da Operação de Remoção Física**

Uma forma de se determinar o custo dos algoritmos de remoção física é analisar a quantidade de cálculos de distância necessários para realizar esta operação.

A quantidade de cálculos de distância necessários ao algoritmo *ReorgAll* é a soma das quantidades necessárias para encontrar o elemento a ser removido, e para a reinserção no índice de todos os elementos da hierarquia a ser reorganizada. Este custo varia em função do comprimento e da quantidade de elementos da hierarquia em reorganização. No caso do algoritmo *PromoteNode*, que é uma extensão do algoritmo *ReorgAll* e promove outro nó no lugar daquele que sofreu a remoção, o custo é calculado da mesma forma que no algoritmo *ReorgAll*, porém, incide somente na quantidade de elementos a serem reorganizados entre o nó que sofreu a remoção e aquele que o substituirá.

O custo da operação de remoção dos algoritmos de remoção física *ReorgAll* e *PromoteNode* foi analisado com base na remoção lógica. Para esta análise, é importante observar que a quantidade de cálculos de distância necessária ao algoritmo *LogicalDelete* é somente aquela necessária para encontrar o elemento a ser removido. Além disso, nas configurações de teste há a situação onde um mesmo elemento pode ser incluído em reorganizações tantas vezes quantas forem as remoções em níveis superiores de sua hierarquia. Por exemplo, um elemento em um nível folha pode ser incluído na reorganização de hierarquias em remoções que aconteçam em cada um de seus níveis superiores. Isso impõe um alto custo para as operações de remoção física.

A Tabela 5.3 demonstra, para a Conf1 que remove 30.000 elementos da base, que o algoritmo *ReorgAll* manteve apenas 8,27% na mesma posição que o algoritmo *LogicalDelete*, e o algoritmo *PromoteNode* manteve apenas 7,09% dos

elementos na mesma posição que o algoritmo *LogicalDelete*. Esta grande movimentação imputou alto custo aos algoritmos de remoção física. Nesta configuração, para remoção de 29,34% dos elementos, o algoritmo *ReorgAll* movimentou 91,73% e o algoritmo *PromoteNode* 92,91% dos elementos. A Figura 5.27 ilustra o custo desta movimentação. O algoritmo *ReorgAll* necessitou de 5.603%, e o algoritmo *PromoteNode* necessitou de 5.436% mais cálculos de distância que o necessário à remoção lógica. A Figura 5.28 ilustra os resultados para o tempo gasto na operação de remoção. O algoritmo *ReorgAll* necessitou de 1.632% e o algoritmo *PromoteNode* necessitou de 1.588% mais tempo que a remoção lógica.

Tabela 5.3: Remoção física: Elementos movimentados entre os níveis após a remoção de 30.000 elementos

Base	Comparação	(-)	(=)	(+)
Após construção do índice	ReorgAll	37,57%	8,27%	54,17%
	PromoteNode	30,62%	7,09%	62,29%
ReorgAll	PromoteNode	35,39%	10,13%	54,48%

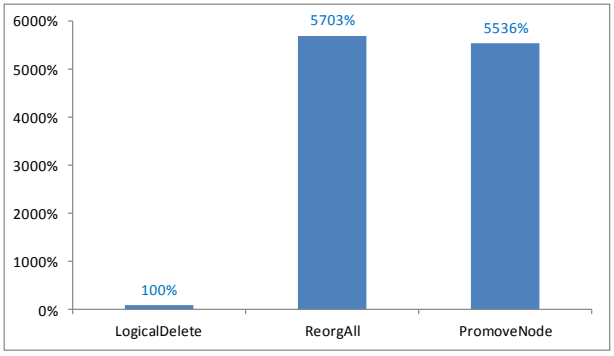


Figura 5.27: Custo da remoção - Cálculos de distância na remoção de 30.000 elementos

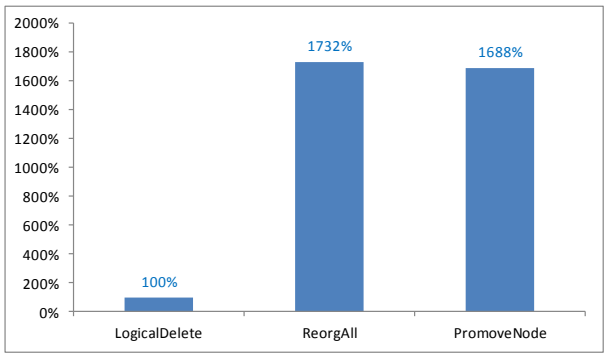


Figura 5.28: Custo da remoção - Tempo total da operação na remoção de 30.000 elementos

A Conf2 que remove 500 elementos em níveis específicos fornece mais detalhes sobre o custo da operação de remoção. A Figura 5.29 ilustra os resultados

em cálculos de distância. Nesta configuração pode-se confirmar o alto custo de reorganização dos níveis superiores da estrutura. Nestes níveis, o algoritmo *ReorgAll* necessitou de 248.187% e o algoritmo *PromoteNode* de 155.334% mais cálculos de distância que a remoção lógica. Nos demais níveis o custo ainda é bem maior do que o verificado para a remoção lógica, porém cai expressivamente em relação àqueles verificados na remoção física em níveis superiores da estrutura. Nos níveis intermediários estes custos foram maiores que aqueles para a remoção lógica em 2.941% para o algoritmo *ReorgAll,* e 2.487% para o algoritmo *PromoteNode*. Nos níveis folha e pai de folha, para o algoritmo *ReorgAll* maior em 113% e para o algoritmo *PromoteNode* maior em 72%.

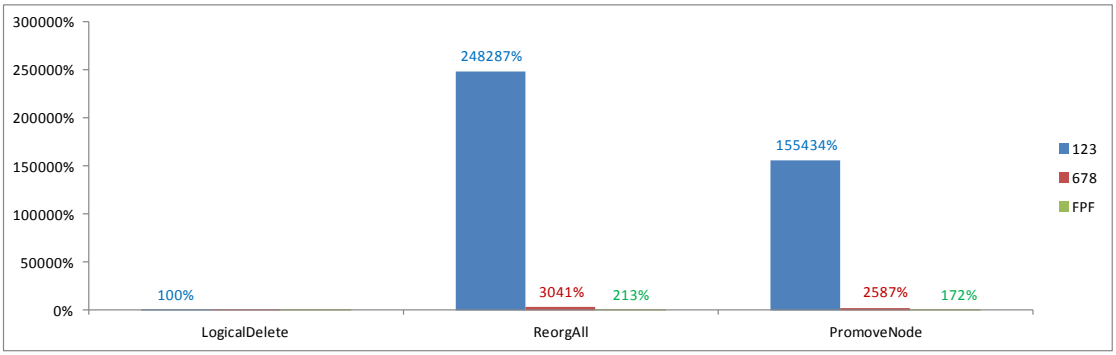


Figura 5.29: Custo da remoção – Cálculos de distância na remoção de 500 elementos de níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A Figura 5.30 ilustra, ainda para a Conf2, os resultados para o tempo gasto na operação de remoção. Consequência da quantidade de cálculos de distância necessária à operação de remoção, o tempo de execução da remoção apresenta o mesmo padrão de comportamento. Em todos os níveis analisados, os algoritmos de remoção física necessitaram mais tempo de execução que na remoção lógica. Nos primeiros níveis da estrutura o algoritmo *ReorgAll* necessitou 37.235% e o algoritmo *PromoteNode* 23.438% mais tempo que na remoção lógica. Nos níveis intermediários o tempo de execução foi maior para o algoritmo *ReorgAll* em 800% e para o algoritmo *PromoteNode* em 675%. Nos níveis folha e pai de folha para o algoritmo *ReorgAll* em 44% e o algoritmo *PromoteNode* em 33%.

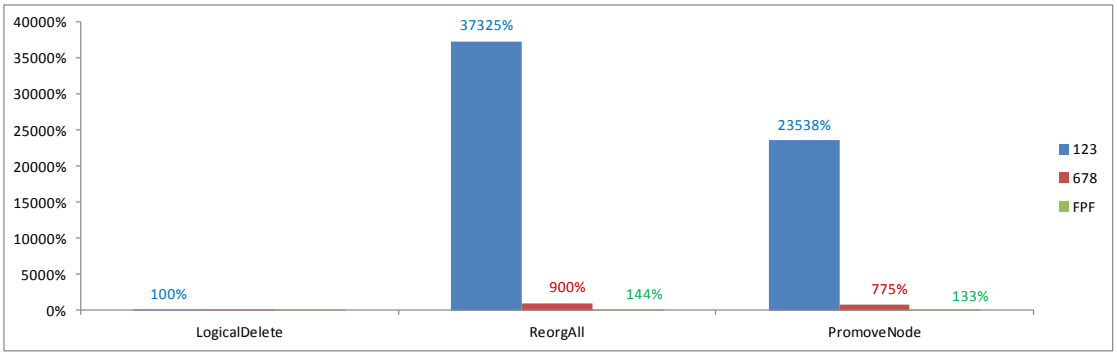


Figura 5.30: Custo da remoção - Tempo total da remoção de 500 elementos de níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

Observa-se que conforme diminui o comprimento e a quantidade de elementos da hierarquia a ser reorganizada, diminui progressivamente o custo de sua reorganização. Portanto, o custo da operação de remoção é maior quanto maior o comprimento e a quantidade de elementos da hierarquia em reorganização. A remoção física nos níveis superiores da estrutura tem um custo significativamente mais alto que nos níveis intermediários. Não obstante a este custo, o processamento das consultas após a remoção física apresentou melhor desempenho que o verificado na reconstrução da estrutura dos elementos não removidos. Considerando que na utilização regular de conjuntos de dados, é executada uma quantidade muito maior de consultas que a quantidade de operações de remoção, a relação custo/benefício da remoção física se torna satisfatória.

5.7 **Perfil das Estruturas**

Esta Seção discute o perfil do índice, representado pelos parâmetros de tamanho, quantidade total de nós e de nós folha do índice, após a aplicação da operação de remoção. Para avaliação do efeito da remoção física nestes parâmetros, a base de comparação utilizada foi a construção inicial da estrutura de indexação, ou seja, a estrutura antes da aplicação dos algoritmos de remoção.

As Figuras de 5.26 a 5.31 mostram que para os algoritmos de remoção física, os parâmetros de tamanho, quantidade total de nós e de nós folha do índice, diminuíram proporcionalmente à quantidade de remoções. Na Conf1 que removeu 29,34% dos elementos da base, estas variáveis diminuíram aproximadamente 30%. Na Conf2 que removeu 0,48% dos elementos, estas variáveis diminuíram

aproximadamente 0,4%. O único parâmetro que não variou proporcionalmente à quantidade de elementos removidos foi a altura máxima. Em ambas as configurações de teste, após a remoção física a altura máxima da estrutura cresceu significativamente em relação à construção inicial com 29 níveis.

A Figura 5.31 mostra a altura máxima após a aplicação da remoção na Conf1 que remove 29,34% dos elementos do conjunto. Antes das remoções, a estrutura possui 29 níveis. A remoção aplicada pelo algoritmo *ReorgAll* produziu em uma estrutura de 34 níveis, 5 níveis maior que construção inicial. A remoção aplicada pelo algoritmo *PromoteNode* produziu uma estrutura de 32 níveis, 3 níveis maior que construção inicial do índice. Em comparação com a reconstrução da estrutura com os elementos não removidos, que resultou em uma estrutura de 25 níveis, o algoritmo *ReorgAll* produz uma estrutura 9 níveis maior, e o algoritmo *PromoteNode* produz uma estrutura 7 níveis maior.

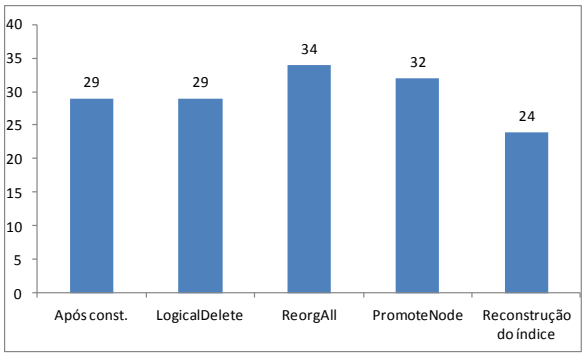


Figura 5.31: Altura máxima da estrutura após a remoção de 30.000 elementos

A Conf2 que remove 0,48% dos elementos da base fornece mais detalhes para a compreensão do comportamento da altura máxima após grande quantidade de remoções. A Figura 5.32 mostra que a remoção de 500 elementos nos níveis superiores da estrutura já resulta em aumento significativo da altura máxima. Nos demais níveis o aumento da altura máxima não foi significativo.

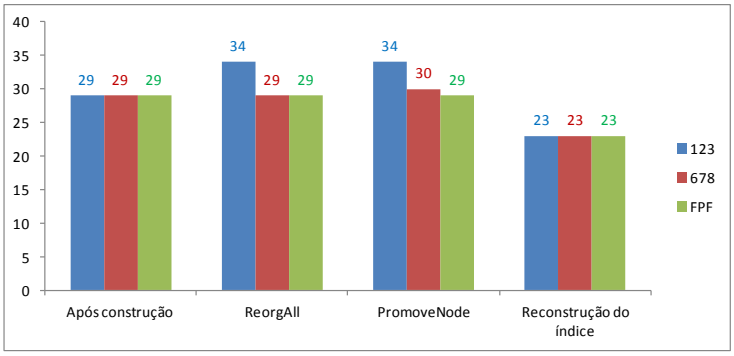


Figura 5.32: Altura máxima da estrutura após a remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois

níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

Resultado da altura máxima, a altura média apresenta o mesmo comportamento em relação à remoção em níveis superiores da estrutura. A Figura 5.33 e a Figura 5.34 demonstram que a altura média é resultado da remoção nos níveis superiores da estrutura.

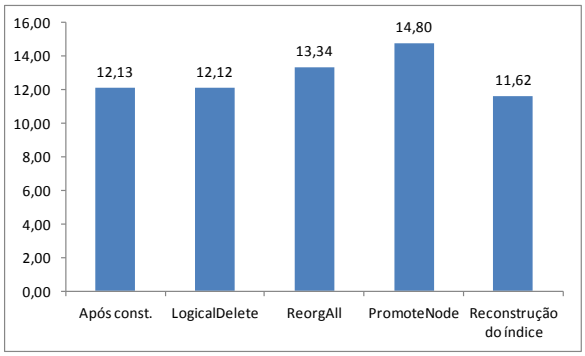


Figura 5.33: Altura média da estrutura após a remoção de 30.000 elementos

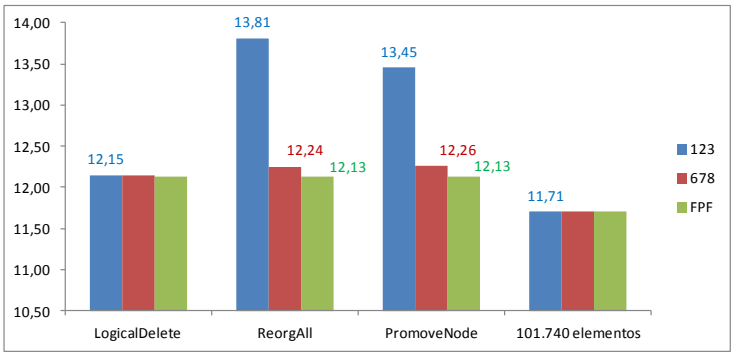


Figura 5.34: Altura média da estrutura após a remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

A despeito do aumento significativo da altura máxima, o desempenho global no processamento das consultas não foi afetado porque a ocupação dos nós da estrutura melhorou. Na Onion-tree, a taxa de ocupação dos nós tem influência no desempenho global do processamento das consultas. Apesar dos algoritmos de remoção física terem aumentado significativamente a altura máxima da estrutura, a melhor ocupação dos nós proporcionou melhor desempenho no processamento das consultas.

A Tabela 5.4 e a Tabela 5.5 mostram que com relação à reconstrução dos elementos não removidos, na Conf1 que remove 29,34% dos elementos da base, a quantidade de nós com apenas um representante diminuiu, para o algoritmo

ReorgAll em 5,07% e para o algoritmo *PromoteNode* em 3,39%. Considerando que a reconstrução do índice produz uma estrutura com 44.436 nós, esta diminuição representa, para o algoritmo *ReorgAll* 3,11%, e para o a algoritmo *PromoteNode* 2,12% do total de 72.240 elementos. Na reconstrução dos elementos não removidos estes elementos estão armazenados sozinhos em um nó. Após a execução dos algoritmos *ReorgAll* e *PromoteNode*, estes elementos passaram a ser alocados junto com outro elemento. Ademais, a quantidade de nós com dois representantes também cresceu. Para o algoritmo *ReorgAll* cresceu em 1,52%, e para o algoritmo *PromoteNode* em 1,01% do total de 72.240 elementos. Desta forma, para o algoritmo *ReorgAll* 4,63%, e para o algoritmo *PromoteNode* 3,13% dos elementos foram melhor alocados do que na reconstrução do índice com os elementos não removidos. Verificamos na Seção 5.5.2 que na nesta mesma configuração de teste, também comparado à reconstrução dos elementos não removidos o ganho de desempenho dos algoritmos *ReorgAll* e *PromoteNode* no processamento de consultas foi em média de 3,62%.

Tabela 5.4: Ocupação dos nós, após remoção de 30.000 elementos

Algoritmo	Ocupação dos nós			
	100%		50%	
	%	Dif%	%	Dif%
72.240 elementos	100,00%		100,00%	
ReorgAll	101,52%	1,52%	94,93%	-5,07%
PromoteNode	101,01%	1,01%	96,61%	-3,39%

Tabela 5.5: Ocupação dos nós, após remoção de 500 elementos

Versao	Operacao	Ocupação dos nós			
		100%		50%	
		%	Dif	%	Dif
101.740 elementos	**Após construção**	100,00%		100,00%	
ReorgAll	**Após remoção niveis 123**	100,37%	0,37%	98,74%	-1,26%
	Após remoção niveis 678	100,29%	0,29%	99,03%	-0,97%
	Após remoção niveis FPF	100,20%	0,20%	99,32%	-0,68%
PromoteNode	**Após remoção niveis 123**	100,08%	0,08%	99,73%	-0,27%
	Após remoção niveis 678	100,27%	0,27%	99,11%	-0,89%
	Após remoção niveis FPF	100,20%	0,20%	99,32%	-0,68%

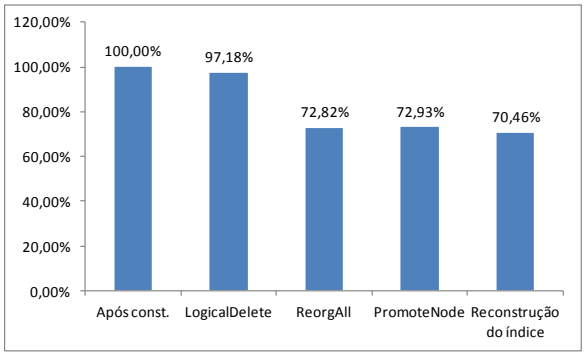


Figura 5.35: Tamanho final da estrutura após remoção de 30.000 elementos

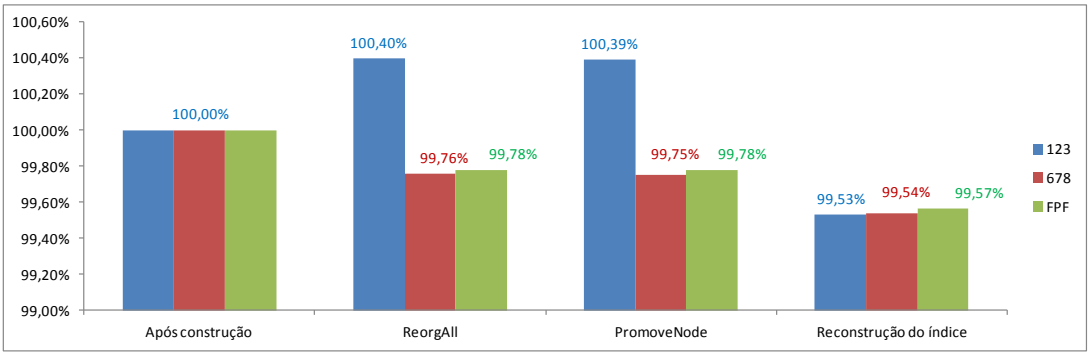


Figura 5.36: Tamanho final da estrutura após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

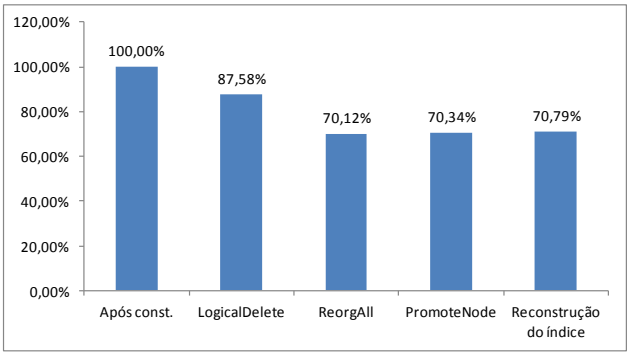


Figura 5.37: Quantidade total de nós após remoção de 30.000 elementos

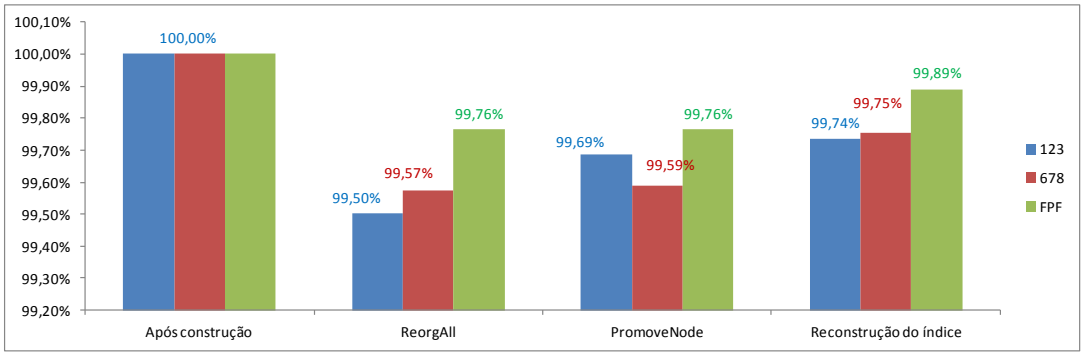


Figura 5.38: Quantidade total de nós após remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do

nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

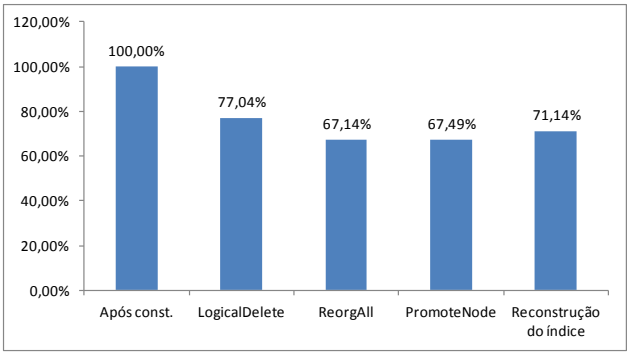


Figura 5.39: Quantidade total de nós folha após a remoção de 30.000 elementos

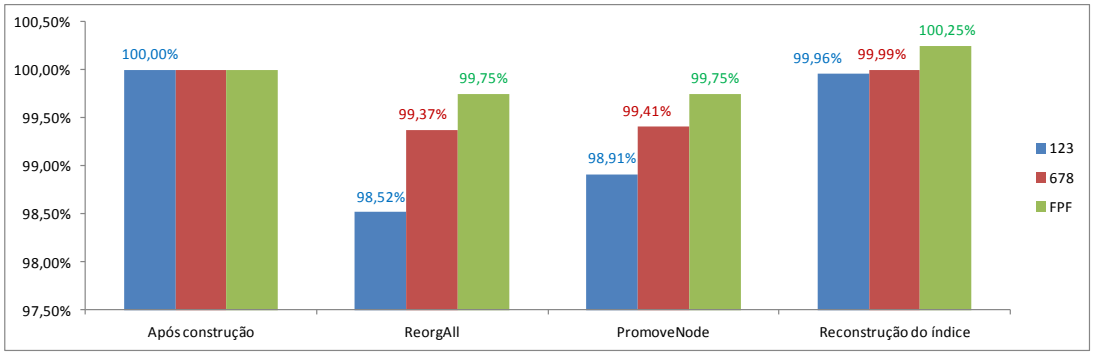


Figura 5.40: Quantidade total de nós folha após a remoção de 500 elementos em níveis específicos. A legenda 123 indica o desempenho da remoção no nível do nó raiz e nos dois níveis abaixo do nó raiz. A legenda 678 indica o desempenho nos níveis intermediários, e a legenda PFP indica o desempenho nos níveis pai de nós folha e folha do índice

Capítulo 6

CONCLUSÃO

A estrutura de indexação métrica Onion-tree já possui algoritmos para a inserção de elementos e o processamento de consultas por similaridade dos tipos *Range Query* (consulta por abrangência) e *KNN* (consulta aos *k-vizinhos* mais próximos). Entretanto, ainda não foi proposto na literatura um algoritmo para a remoção de elementos na Onion-tree, e assim esta pesquisa enfocou a remoção lógica e física de elementos na estrutura de indexação métrica Onion-tree.

Primeiramente, foi implementado e avaliado o algoritmo de remoção lógica proposta em (CARÉLO et al., 2011). A proposta feita em (CARÉLO et al., 2011) deu origem à implementação de três algoritmos de remoção lógica, denominados *LogicalDelete, ReplaceReducing* e *ReplaceGrowing.* O primeiro algoritmo aplica a remoção lógica, que consiste em apenas marcar, pelo uso de um campo extra, que o elemento encontra-se removido e, portanto, deve ser desconsiderado no processamento de consultas. Já os algoritmos *ReplaceReducing* e *ReplaceGrowing* são especializações do primeiro, adicionando tratamento especial para a remoção em nós internos com filhos exclusivamente folha. O algoritmo *ReplaceReducing* permite a diminuição do raio do nó que sofreu a remoção. De forma antagônica, o algoritmo *ReplaceGrowing* permite o aumento deste raio.

Os testes experimentais dos algoritmos de remoção lógica demonstraram que para o penúltimo nível da estrutura de indexação, ou seja, para nós pai exclusivamente de folha, nenhum dos algoritmos que avaliaram a substituição de um elemento removido por outro de suas folhas, no caso os algoritmos *ReplaceReducing* e *ReplaceGrowing*, apresentaram desempenho que justifique o processamento de uma quantidade maior de instruções do que aquela aplicada na remoção lógica. Para estes níveis, a melhor relação custo/benefício é indicada pelo algoritmo *LogicalDelete.*

Adicionalmente, foram propostos e avaliados dois algoritmos de remoção física que podem ser aplicados em qualquer nível da estrutura da Onion-tree. O algoritmo *ReorgAll* reorganiza todos os elementos da hierarquia do nó que possui o elemento que sofreu a remoção, removendo fisicamente os nós e reinserindo os elementos removidos destes nós por meio da aplicação do algoritmo de inserção de elementos. O algoritmo *PromoteNode*, o qual estende o algoritmo *ReorgAll*, promove, quando houver condições para tal, outro nó em substituição àquele que sofreu a remoção.

Para testar a eficiência dos algoritmos de remoção física propostos, foram realizados testes de desempenho para analisar o custo da operação de remoção e o impacto da operação de remoção no processamento de consultas posteriores. Como *baseline* foram usados os algoritmos de remoção lógica e a reconstrução do índice com os elementos não removidos.

Os testes demonstraram que, em grande quantidade de remoções, o desempenho da remoção lógica no processamento de consultas posterior à remoção fica bem abaixo do verificado para as estruturas livre de cálculos de distância desnecessários, ou seja, as estruturas geradas pela remoção dos algoritmos *ReorgAll* e *PromoteNode,* e a estrutura gerada pela reconstrução dos elementos não removidos. Nesta comparação, os algoritmos de remoção física *ReorgAll* e *PromoteNode* produziram melhora respectivamente de 21,67% e 21,66%, e a reconstrução da estrutura com os elementos não removidos produziu melhora de 18,72% do desempenho no processamento de consultas após a remoção. Entretanto, os testes também demonstraram que em pequena quantidade de remoções, a execução desnecessária de cálculos de distância no processamento de consultas não tem efeito determinante no desempenho da Onion-tree no processamento de consultas.

Com relação aos algoritmos de remoção física *ReorgAll* e *PromoteNode*, os testes mostraram que a promoção de um nó, em substituição àquele que sofreu a remoção, reorganizando apenas os elementos no caminho entre o nó removido e aquele que o substituirá, apresenta vantagens significativas em relação à simples reorganização da hierarquia que sofreu a remoção. Além disso, o armazenamento dos elementos da hierarquia afetada pela remoção conforme a largura da estrutura de indexação, armazenando todos os elementos de um nível para somente então

percorrer o próximo nível mantém o desempenho mais próximo possível do desempenho original antes das operações de remoção.

A promoção de uma hierarquia tem maior probabilidade de ocorrer quando a remoção acontece nos primeiros níveis da estrutura. Esta probabilidade também é maior quando o nó que possui o elemento removido estiver na região mais externa de seu nó pai. Como esta é a região mais extensa, há maior probabilidade de encontrar, entre os descendentes do nó removido, candidatos que estejam em relação ao nó pai, também na região externa.

Desta forma, o custo da operação de remoção física é maior quanto maior for o comprimento e a quantidade de elementos da hierarquia em reorganização. Os testes experimentais demonstraram que a remoção física nos níveis superiores da estrutura tem um custo significativamente mais alto que nos níveis intermediários. Assim, por reorganizar apenas os elementos no caminho entre o nó removido e aquele que o substituirá, o algoritmo *PromoteNode* apresentou menor custo que o algoritmo *ReorgAll,* que reorganiza todos os elementos da hierarquia afetada pela remoção.

Com relação ao perfil da estrutura da Onion-tree após a remoção física, os testes experimentais mostraram que a taxa de ocupação dos nós tem influência no desempenho global do processamento das consultas. Apesar dos algoritmos de remoção física terem aumentado a altura máxima da estrutura, a melhor ocupação dos nós proporcionou melhor desempenho no processamento das consultas.

Comparados à reconstrução dos elementos não removidos, com relação ao desempenho no processamento de consultas posterior à operação de remoção física, em pequena quantidade de remoções, os algoritmos de remoção física *ReorgAll* e *PromoteNode* obtiveram praticamente o mesmo desempenho em cálculos de distância. Com relação ao tempo de processamento, estes algoritmos obtiveram desempenho, em média, 3,62% superior à reconstrução dos elementos não removidos. Entretanto, para os extratos analisados da estrutura, o algoritmo *PromoteNode* obteve melhor desempenho que o algoritmo *ReorgAll*. O algoritmo *PromoteNode* obteve ganhos, de 9,8% nos primeiros níveis da estrutura (extratos 1,2 e 3), de 3,12% nos níveis intermediários (extratos 6,7 e 8), e de 2,19% nos níveis folha e pai de folha. O algoritmo *ReorgAll* obteve ganhos, de 2,7% nos primeiros níveis da estrutura (extratos 1,2 e 3), de 3,12% nos níveis intermediários (extratos 6,7 e 8), e de 2,19% nos níveis folha e pai de folha.

Não obstante ao custo da operação de remoção, o processamento das consultas após a remoção física apresentou melhor desempenho que o verificado na reconstrução da estrutura dos elementos não removidos. Considerando situações nas quais são executadas uma enorme quantidade de consultas e de operações de remoção, sendo que a quantidade de consultas é muito maior do que a quantidade de operações de remoção (por exemplo, de 3 a 4 ordens de grandeza maior), a relação custo/benefício da remoção física se torna satisfatória.

Dentre os algoritmos de remoção física apresentados, o algoritmo *PromoteNode* é o que apresenta melhor desempenho no processamento de consultas após a remoção de elementos, e menor custo na operação de remoção dos elementos na Onion-tree.

Com a proposta de algoritmos de remoção lógica e física de elementos nesta dissertação e com a proposta de algoritmos de bulk-loading em outra pesquisa de mestrado realizada na USP/ICMC pelo aluno Arthur Emmanuel de oliveira Carósia, a Onion-tree possui o conjunto de algoritmos necessários para a sua implementação em um SGBD, ou seja, possui o algoritmos de inserção de elementos, de remoção de elementos, de bulk-loading (carga intensiva de dados) e de pesquisa de similaridade (range query e K-nearest neighbor queries). Desta forma, trabalhos futuros visam principalmente a adaptação da Onion-tree para a implementação em um SGBD e incluem: a incorporação de mecanismos de controle de concorrência, incluindo a investigação de diferentes mecanismos de travas. Outras pesquisas futuras visam o seu uso em ambientes paralelos, distribuídos e em computação em nuvem, que requer a extensão da sua estrutura de dados e a adaptação de seus algoritmos.

REFERÊNCIAS

BAYER, R. AND MCCREIGHT, E. M. Organization and maintenance of large ordered indexes. **Acta Informatica**, v. 1, n. 3, p. 173–189, 1972. Springer-Verlag New York, Inc.

BECKMANN, N.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. The R*-tree: an efficient and robust access method for points and rectangles. (H. Garcia-Molina & H. V Jagadish, Eds.)**ACM SIGMOD Record**, v. 19, n. 2, p. 322–331, 1990. ACM.

BÖHM, C.; BERCHTOLD, S.; KEIM, D. A. Searching in high-dimensional spaces - index structures for improving the performance of multimedia databases. **ACM Computing Surveys**, v. 33, n. 3, p. 322–373, 2001.

BOZKAYA, T.; OZSOYOGLU, M. Distance-Based Indexing for High-Dimensional Metric Spaces. SIGMOD Conference. **Anais...** v. 26, p.357–368, 1997. ACM Press.

BRIN, S. Near Neighbor Search in Large Metric Spaces. VLDB. **Anais...** p.574–584, 1995. INSTITUTE OF ELECTRICAL \& ELECTRONICS ENGINEERS (IEEE).

BUENO, R. **Tratamento do tempo e dinamicidade em dados representados em espaços métricos**, 2009. Universidade de São Paulo: Tese.

BURKHARD, W. A.; KELLER, R. M. Some approaches to best-match file searching. **Communications of the ACM**, v. 16, n. 4, p. 230–236, 1973. ACM.

CARÉLO, C. C. M.; POLA, I. R. V.; CIFERRI, R. R.; et al. The Onion-tree: Quick Indexing of Complex Data in the Main Memory. Proceedings of the 13th East European Conference on Advances in Databases and Information Systems. **Anais...** v. 5739, p.235–252, 2009.

CARÉLO, C. C. M.; POLA, I. R. V.; CIFERRI, R. R.; et al. Slicing the metric space to provide quick indexing of complex data in the main memory. **Information Systems**, v. 36, n. 1, p. 79–98, 2011.

CHÁVEZ, E.; NAVARRO, G.; BAEZA-YATES, R.; MARROQUÍN, J. L. Searching in metric spaces. **ACM Computing Surveys**, v. 33, n. 3, p. 273–321, 2001. ACM Press.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. Proceedings of the International Conference on Very Large Data Bases. **Anais...** v. 25, p.426–435, 1997. Citeseer.

CIFERRI, R. R. **Análise da influência do fator distribuição espacial dos dados no desempenho de métodos de acesso multidimensionais**, 2002. Universidade Federal de Pernambuco: Doutorado.

FALOUTSOS, C. **Searching Multimedia Databases by Content**. Kluwer Academic Publishers, 1996.

FELIPE, J. C. **Desenvolvimento de métodos para extração e análise de características intrínsecas de imagens médicas, visando a recuperação perceptual por conteúdo**, 9. Aug. 2005. Universidade de São Paulo: Doutorado.

FU, A. W.-C.; CHAN, P.; CHEUNG, Y.-L.; MOON, Y. S. Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances. **The VLDB Journal**, v. 9, n. 2, p. 154–173, 2000.

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In: B. Yormack (Ed.); Proceedings of the 1984 ACM SIGMOD international conference on Management of data. **Anais...** v. 14, p.47–57, 1984. ACM.

PETRAKIS, E. G. M.; FALOUTSOS, C. Similarity Searching in Medical Image Databases. **IEEE Transactions on Knowledge and Data Engineering**, v. 9, n. 3, p. 435–447, 1997.

PETRAKIS, E. G. M.; FALOUTSOS, C.; LIN, K. I. ImageMap: An Image Indexing Method Based on Spatial Similarity. **IEEE Transactions on Knowledge and Data Engineering**, v. 14, n. 5, p. 979–987, 2002. IEEE Computer Society.

POLA, I. R. V. **Indexação em domínios métricos generalizáveis**, May. 2005. Universidade de São Paulo: Mestrado.

POLA, I. R. V. **Explorando conceitos da teoria de espaços métricos em consultas por similaridade sobre dados complexos**, 9. Aug. 2010. Universidade de São Paulo: Doutorado.

POLA, I. R. V.; TRAINA JR., C.; TRAINA, A. J. M. The MM-Tree: A Memory-Based Metric Tree Without Overlap Between Nodes. Proceedings of the 12th East European Conference on Advances in Databases and Information SystemsADBIS. **Anais...** p.157–171, 2007.

SELLIS, T.; ROUSSOPOULOS, N.; FALOUTSOS, C. The R+-tree: A dynamic index for multi-dimensional objects. Proceedings of 13th International Conference on Very large Data Bases. **Anais...** p.507–518, 1987. Morgan Kaufmann.

SMEULDERS, A. W. M.; WORRING, M.; SANTINI, S.; GUPTA, A.; JAIN, R. Content-based image retrieval at the end of the early years. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 22, n. 12, p. 1349–1380, 2000.

SMEULDERS, A. W. M.; WORRING, M.; SANTINI, S.; GUPTA, A.; JAIN, R. Content-based image retrieval at the end of the early years. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 22, n. 12, p. 1349–1380, 2000. IEEE Computer Society.

SOUSA, E. P. M. DE. **Identificação de Correlações Usando a Teoria dos Fractais**, 8. Aug. 2006. Biblioteca Digital de Teses e Dissertações da USP.

TALAVERA, L. Feature Selection as a Preprocessing Step for Hierarchical Clustering. In: I. Bratko; S. Dzeroski (Eds.); Proceedings of the 25th international conference on Machine learning. **Anais...** p.389–397, 1999. Morgan Kaufmann.

TRAINA JR., C.; TRAINA, A. J. M.; SANTOS FILHO, R. F.; FALOUTSOS, C. How to improve the pruning ability of dynamic metric access methods. Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management McLean VA USA November 49 2002. **Anais...** p.219–226, 2002. ACM Press.

TRAINA JR., C.; TRAINA, A. J. M.; SEEGER, B.; FALOUTSOS, C. Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes. Advances in Database Technology EDBT 2000 7th International Conference on Extending Database Technology Konstanz Germany March 2731 2000 Proceedings. **Anais...** v. 1777, p.51–65, 2000. Springer.

UHLMANN, J. K. Satisfying general proximity/similarity queries with metric trees. **Information Processing Letters**, v. 40, n. 4, p. 175–179, 1991. Elsevier.

WILSON, D. R.; MARTINEZ, T. R. Improved Heterogeneous Distance Functions. **Journal of Artificial Intelligence Research**, v. 6, n. 1-34, p. 1–34, 1996.

YIANILOS, P. N. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. Proceedings of the fourth annual ACMSIAM Symposium on Discrete algorithms. **Anais...** p.311–321, 1993. Society for Industrial and Applied Mathematics.

APÊNDICE - RESULTADOS APURADOS

Resultados apurados nos testes da Tabela 4.3: *ReplaceReducing* e *ReplaceGrowing* – Variações de raio analisadas

A. KDD Cup 2008

Teste 1

Parâmetros do teste:

- Variação do raio: Para o algoritmo ReplaceReducing, e para o algoritmo ReplaceGrowing, variação máxima de 0,5 do raio de seu pai;
- Elementos removidos: Os 10% últimos elementos inseridos na base;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico - 1022 elementos.

KDD Cup 2008: Resultados do Teste 1 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,05%	100,77%	97,85%
Elementos excluidos	100,52%	100,94%	98,00%
Elementos movimentados	100,65%	101,19%	98,35%

Teste 2

Parâmetros do teste:

- Variação do raio: Para o algoritmo ReplaceReducing e para o algoritmo ReplaceGrowing, variação máxima de 0,5 do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico que retorne 1022 elementos.

KDD Cup 2008: Resultados do Teste 2 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,77%	99,76%	102,09%
Elementos excluidos	99,78%	99,76%	102,09%
Elementos movimentados	99,77%	99,76%	102,09%

KDD Cup 2008: Resultados do Teste 2 da Tabela 4.3.
Tempo de consulta após remoção de 10% da base

Elementos	Logical Delete	Replace Reducing	Replace Growing
Após construção	100,00%	100,00%	100,00%
Após remoção	90,25%	90,25%	90,24%
Após reinserção	100,00%	100,00%	100,00%
Após remoção	90,45%	90,47%	88,40%

Teste 3

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing* e para o algoritmo *ReplaceGrowing*, variação máxima de 0,5 do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio fixo de 1% do diâmetro do conjunto de dados.

KDD Cup 2008: Resultados do Teste 3 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,94%	99,49%	99,90%
Elementos excluidos	99,98%	99,34%	99,84%
Elementos movimentados	99,93%	99,31%	99,84%

KDD Cup 2008: Resultados do Teste 3 da Tabela 4.3.
Tempo de consulta após remoção de 10% da base

Elementos	Logical Delete	Replace Reducing	Replace Growing
Após construção	100,00%	100,00%	100,00%
Após remoção	90,09%	90,07%	90,08%
Após reinserção	100,00%	100,00%	100,00%
Após remoção	90,14%	90,54%	90,17%

Teste 4

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing*, diminuir o raio do nó sem imposição de limite, e para o algoritmo *ReplaceGrowing*, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico de 10 elementos.

KDD Cup 2008: Resultados do Teste 4 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,77%	100,09%	100,61%
Elementos excluidos	99,77%	100,10%	100,62%
Elementos movimentados	99,76%	100,10%	100,62%

KDD Cup 2008: Resultados do Teste 4 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,08%	99,90%
Elementos excluidos	100,00%	100,08%	99,90%
Elementos movimentados	100,00%	100,08%	99,91%

Teste 5

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing*, diminuir o raio do nó sem imposição de limite, e para o algoritmo *ReplaceGrowing*, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 500 elementos selecionados entre os elementos movimentados;
- Quantidade retornada pela RQ: Raio fixo de 31% do diâmetro do conjunto de dados.

KDD Cup 2008: Resultados do Teste 5 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,78%	99,87%	99,82%

KDD Cup 2008: Resultados do Teste 5 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,07%	99,93%

B. Color Histograms

Teste 4

Parâmetros do teste:

- Variação do raio: Para o algoritmo ReplaceReducing, diminuir o raio do nó sem imposição de limite, e para o algoritmo ReplaceGrowing, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico de 10 elementos.

Color Histograms: Resultados do Teste 4 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,23%	101,12%	101,54%
Elementos excluidos	100,22%	100,99%	101,60%
Elementos movimentados	99,92%	101,07%	102,06%

Color Histograms: Resultados do Teste 4 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,05%	99,88%
Elementos excluidos	100,00%	100,06%	99,88%
Elementos movimentados	100,00%	100,09%	99,92%

Teste 5

Parâmetros do teste:

- Variação do raio: Para o algoritmo ReplaceReducing, diminuir o raio do nó sem imposição de limite, e para o algoritmo ReplaceGrowing, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 500 elementos selecionados entre os elementos movimentados;
- Quantidade retornada pela RQ: Raio fixo de 31% do diâmetro do conjunto de dados.

Color Histograms: Resultados do Teste 5 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,08%	101,58%	100,80%

Color Histograms: Resultados do Teste 5 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,71%	100,63%

C. Ozone

Teste 4

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing*, diminuir o raio do nó sem imposição de limite, e para o algoritmo *ReplaceGrowing*, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico de 10 elementos.

Ozone: Resultados do Teste 4 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,02%	100,01%	99,85%
Elementos excluidos	100,01%	100,05%	99,84%
Elementos movimentados	99,95%	100,02%	99,74%

Ozone: Resultados do Teste 4 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	99,99%	99,86%
Elementos excluidos	100,00%	100,04%	99,87%
Elementos movimentados	100,00%	100,03%	99,83%

Teste 5

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing*, diminuir o raio do nó sem imposição de limite, e para o algoritmo *ReplaceGrowing*, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;

- Quantidade de elementos consultados: 500 elementos selecionados entre os elementos movimentados;
- Quantidade retornada pela RQ: Raio fixo de 31% do diâmetro do conjunto de dados.

Ozone: Resultados do Teste 5 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,07%	100,00%

Ozone: Resultados do Teste 5 da Tabela 4.3:
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,12%	99,99%

D. Brazilian Cities

Teste 4

Parâmetros do teste:

- Variação do raio: Para o algoritmo *ReplaceReducing*, diminuir o raio do nó sem imposição de limite, e para o algoritmo *ReplaceGrowing*, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 100% do conjunto de dados;
- Quantidade retornada pela RQ: Raio dinâmico de 10 elementos.

Brazilian Cities: Resultados do Teste 4 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	98,92%	100,10%	99,94%
Elementos excluidos	98,98%	100,20%	99,86%
Elementos movimentados	99,22%	100,87%	100,02%

Brazilian Cities: Resultados do Teste 4 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,08%	99,98%
Elementos excluidos	100,00%	100,13%	100,02%
Elementos movimentados	100,00%	100,19%	99,97%

Teste 5

Parâmetros do teste:

- Variação do raio: Para o algoritmo ReplaceReducing, diminuir o raio do nó sem imposição de limite, e para o algoritmo ReplaceGrowing, variação máxima de 10% do raio de seu pai;
- Elementos removidos: 10% do total da base sendo deste total, 50% folha e 50% pai de folha;
- Quantidade de elementos consultados: 500 elementos selecionados entre os elementos movimentados;
- Quantidade retornada pela RQ: Raio fixo de 31% do diâmetro do conjunto de dados.

Brazilian Cities: Resultados do Teste 5 da Tabela 4.3.
Tempo de consulta

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	99,94%	101,54%	98,29%

Brazilian Cities: Resultados do Teste 5 da Tabela 4.3.
Cálculos de distância

Elementos	Logical Delete	Replace Reducing	Replace Growing
100% da base após construção	100,00%	100,00%	100,00%
100% da base após reinserção	100,00%	100,06%	100,01%