Algoritmos de *bulk-loading* para o método de acesso métrico Onion-tree

Arthur Emanuel de Oliveira Carosia

Algoritmos de *bulk-loading* para o método de acesso métrico Onion-tree [1]

Arthur Emanuel de Oliveira Carosia

Orientadora: Profa. Dra. Cristina Dutra de Aguiar Ciferri

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

USP – São Carlos
Abril de 2013

[1] Trabalho realizado com apoio financeiro da FAPESP - Processo 2010/13756-8

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

C292a

Carosia, Arthur Emanuel de Oliveira
Algoritmos de Bulk-loading para o Método de Acesso Métrico Onion-tree / Arthur Emanuel de Oliveira Carosia; orientador Cristina Dutra de Aguiar Ciferri. -- São Carlos, 2013.
111 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2013.

1. bulk-loading. 2. Onion-tree. 3. método de acesso métrico. 4. consulta por similaridade. I. Ciferri, Cristina Dutra de Aguiar, orient. II. Título.

Agradecimentos

Agradeço aos meus pais, Ana e Arthur, pelo amor, educação e carinho dados a mim. Aos meus irmãos, Rafael e Thales, pelo apoio, mesmo à distância. Aos meus avós, Geraldo e Teresa, que partiram recentemente.

À minha orientadora e amiga, Prof. Dra. Cristina Dutra de Aguiar Ciferri, agradeço pela motivação, confiança, apoio e paciência que sempre demonstrou a mim.

Ao Prof. Dr. Ricardo Rodrigues Ciferri pela ajuda e conselhos.

Aos meus amigos do grupo de bases de dados, GBDI, com os quais discuti e compartilhei experiências.

Aos meus grandes amigos, Matheus, Ederaldo, Filipe, Bruno e Thiago, pela companhia, momentos de descontração e conselhos.

À minha namorada, Gleice, que com seu amor me fortalece e traz paz.

À FAPESP pelo apoio financeiro.

Resumo

Atualmente, a Onion-tree [Carélo et al., 2009] é o método de acesso métrico baseado em memória primária mais eficiente para pesquisa por similaridade disponível na literatura. Ela indexa dados complexos por meio da divisão do espaço métrico em regiões (ou seja, subespaços) disjuntas, usando para isso dois pivôs por nó. Para prover uma boa divisão do espaço métrico, a Onion-tree introduz as seguintes características principais: (i) procedimento de expansão, o qual inclui um método de particionamento que controla o número de subespaços disjuntos gerados em cada nó; (ii) técnica de substituição, a qual pode alterar os pivôs de um nó durante operações de inserção baseado em uma política de substituição que garante uma melhor divisão do espaço métrico, independente da ordem de inserção dos elementos; e (iii) algoritmos para a execução de consultas por abrangência e aos k-vizinhos mais próximos, de forma que esses tipos de consulta possam explorar eficientemente o método de particionamento da Onion-tree.

Entretanto, a Onion-tree apenas oferece funcionalidades voltadas à inserção dos dados um-a-um em sua estrutura. Ela não oferece, portanto, uma operação de *bulk-loading* que construa o índice considerando todos os elementos do conjunto de dados de uma única vez. A principal vantagem dessa operação é analisar os dados antecipadamente para garantir melhor particionamento possível do espaço métrico. Com isto, a carga inicial de grandes volumes de dados pode ser melhor realizada usando a operação de *bulk-loading*. Este projeto de mestrado visa suprir a falta da operação de *bulk-loading* para a Onion-tree, por meio da proposta de algoritmos que exploram as características intrínsecas desse método de acesso métrico. No total, são propostos três algoritmos de *bulk-loading*, denominados *GreedyBL*, *SampleBL* e *HeightBL*, os quais utilizam respectivamente as seguintes abordagens: gulosa, amostragem e de estimativa da altura do índice.

Testes experimentais realizados sobre conjuntos de dados com volume variando de 2.536 a 102.240 imagens e com dimensionalidade variando de 32 a 117 dimensões mostraram que os algoritmos propostos introduziram vantagens em relação à estrutura criada pelo algoritmo de inserção um-a-um da Onion-tree. Comparado com a inserção um-a-um, o tamanho do índice foi reduzido de 9% até 88%. Em consultas por abrangência, houve redução de 16% até 99% no número de cálculos de distância e de 9% a 99% no tempo gasto em relação à inserção. Em consultas aos k-vizinhos mais próximos, houve redução de 13% a 86% em número de cálculos de distância e de 9% até 63% no tempo gasto.

Abstract

The main-memory Onion-tree [Carélo et al., 2009] is the most efficient metric access method to date. It indexes complex data by dividing the metric space into several disjoint regions (i.e. subspaces) by using two pivots per node. To provide a good division of the metric space, the Onion-tree introduces the following characteristics: (i) expansion procedure, which provides a partitioning method that controls the number of disjoint subspaces generated at each node; (ii) replacement technique, which can replace the pivots of a leaf node during insert operations based on a replacement policy that ensures a better division of the metric space, regardless of the insertion order of the elements; and (iii) algorithms for processing range and k-NN queries, so that these types of query can efficiently use the partitioning method of the Onion-tree.

However, the Onion-tree only performs element-by-element insertions into its structure. Another important issue is the mass loading technique, called bulk-loading, which builds the index considering all elements of the dataset at once. This technique is very useful in the case of reconstructing the index or inserting a large number of elements simultaneously. Despite the importance of this technique, to the best of our knowledge, there are not in the literature bulk-loading algorithms for the Onion-tree. In this master's thesis, we fill this gap. We propose three algorithms for bulk-loading Onion-trees: the GreedyBL algorithm, the SampleBL algorithm and the HeightBL algorithm. These algorithms are based on the following approaches, respectively: greedy, sampling and estime height of the index.

Performance tests with real-world data with different volumes (ranging from 2,536 to 102,240 images) and different dimensionalities (ranging from 32 to 117 dimensions) showed that the indices produced by the proposed algorithms are very compact. Compared with the element-by-element insertion, the size of the index reduced from 9% up to 88%. The proposed algorithms also provided a great improvement in query processing. They required from 16% up to 99% less distance calculations and were from 9% up to 99% faster than the element-by-element insertion to process range queries. Also, they required from 13% up to 86% less distance calculations and were from 9% up to 63% faster than the element-by-element insertion to process k-NN queries.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um método de acesso métrico (MAM) visa prover acesso eficiente a um grande número de aplicações que exigem comparação entre dados complexos, tais como imagens, áudio e vídeo. Para melhorar o acesso aos dados complexos, MAMs reduzem o espaço de busca, guiando a busca a porções do conjunto de dados nas quais os elementos armazenados têm, provavelmente, maior similaridade com o elemento da consulta.

A medida de semelhança entre dois elementos pode ser expressa como uma métrica que se torna menor à medida que os elementos são mais semelhantes [Hjaltason and Samet, 2003]. Portanto, os MAMs particionam o espaço métrico em subespaços para que as consultas não tenham que acessar o conjunto de dados completo.

Formalmente, um espaço métrico é um par ordenado $< \mathbb{S}, d >$, onde $\mathbb{S}$ é o domínio dos elementos de dados e $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ é a métrica. Para quaisquer $s_1, s_2, s_3 \in \mathbb{S}$, a métrica deve possuir as seguintes propriedades: (i) identidade: $d(s_1, s_1) = 0$; (ii) simetria: $d(s_1, s_2) = d(s_2, s_1)$; (iii) não-negatividade: $d(s_1, s_2) \geq 0$; e (iv) desigualdade triangular: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$ [Chávez et al., 2001]. Por exemplo, elementos do conjunto $S \subset \mathbb{S}$, os quais podem ser representados por números, vetores, matrizes, gráficos ou mesmo funções, podem ser indexados com um MAM usando métricas tais como a distância Manhattan (L_1) ou a distância Euclidiana (L_2) [Wilson and Martinez, 1997].

Procurar por elementos próximos a um elemento de consulta $s_q \in \mathbb{S}$ é um problema central em aplicações que gerenciam dados complexos. Os dois tipos mais úteis de consultas por similaridade são a de abrangência (ou seja, *range queries*) e a de k-vizinhos mais próximos (ou seja, *k-NN queries*), as quais são definidas a seguir.

- **Consulta por abrangência**: dado um raio de consulta r_q, essa consulta retorna cada elemento $s_i \in S$ que satisfaz à condição $d(s_i, s_q) \leq r_q$. Um exemplo é: "Selecione as imagens que são similares à imagem P até cinco unidades de similaridade".

- **Consulta aos k-vizinhos mais próximos**: dada uma quantidade $k \geq 1$, essa consulta retorna os k elementos em S que são os mais próximos do centro de consulta s_q. Um exemplo é: "Selecione as três imagens mais similares à imagem P".

Na literatura existem MAMs baseados em memória secundária e MAMs baseados em memória primária. São várias as motivações para o desenvolvimento de um MAM baseado em memória primária [Carélo et al., 2009]. Uma primeira motivação refere-se ao fato de que MAMs baseados em memória primária são úteis para aplicações que requerem a construção de índices repetidas vezes e de forma rápida. Por exemplo, esses métodos de acesso são usualmente aplicados pelo sistema gerenciador de base de dados (SGBD) para otimizar subconsultas no processamento de consultas complexas. Além disso, MAMs baseados em memória primária não precisam minimizar o número de acessos a disco, como é feito pelos MAMs baseados em memória secundária. Como resultado, MAMs baseados em memória primária podem proporcionar uma melhor divisão do espaço métrico, permitindo que consultas por similaridade sejam respondidas mais rapidamente. Finalmente, o aumento da capacidade da memória primária disponível e a diminuição dos seus custos também motivam o uso de MAMs baseados em memória primária. Com base nessas motivações, este projeto de mestrado enfoca o MAM Onion-tree, que é baseado em memória primária.

Atualmente, a Onion-tree [Carélo et al., 2009, Carélo et al., 2011] é o MAM baseado em memória primária mais eficiente para pesquisa por similaridade disponível na literatura. Ela indexa dados complexos por meio da divisão do espaço métrico em regiões (ou seja, subespaços) disjuntas, usando para isso dois pivôs por nó. Para prover uma boa divisão do espaço métrico, a Onion-tree introduz as seguintes características principais: (i) procedimento de expansão, o qual inclui um método de particionamento que controla o número de subespaços disjuntos gerados em cada nó; (ii) técnica de substituição, a qual pode alterar os pivôs de um nó durante operações de inserção baseado em uma política de substituição que garante melhor divisão do espaço métrico, independente da ordem de inserção dos elementos; e (iii) algoritmos para a execução de consultas por abrangência e aos k-vizinhos mais próximos, de forma que esses tipos de consulta possam explorar eficientemente o método de particionamento da Onion-tree.

1.1 Motivações e Contribuições

Uma limitação da Onion-tree é apenas oferecer um algoritmo voltado à inserção dos dados um-a-um em sua estrutura. Ela não oferece, portanto, uma operação de *bulk-loading* que construa o índice considerando todos os elementos do conjunto de dados de uma única vez. A principal vantagem da operação de *bulk-loading* é analisar os dados antecipadamente para garantir melhor particionamento possível do espaço métrico e assim obter uma estrutura final que possa garantir melhor desempenho em consultas em relação à estrutura criada pela inserção de dados um-a-um. O algoritmo de *bulk-loading* também pode ser invocado sempre que o índice precisar ser reconstruído.

Visando suprir essa limitação, nesta dissertação de mestrado são propostos três algoritmos de *bulk-loading* para o MAM Onion-tree. Esses algoritmos organizam os dados antecipadamente, de forma a definir qual a melhor ordem dos elementos para serem inseridos no índice. Além disso, os algoritmos utilizam a construção do índice no sentido *top-down*. Os algoritmos propostos, bem como as suas principais características são:

- *GreedyBL*: algoritmo de *bulk-loading* que utiliza a abordagem gulosa para definir qual o melhor par de pivôs a ser selecionado para cada nó da Onion-tree. Ou seja, a cada nível da estrutura, todos os pares de pivôs são testados para a escolha de qual o melhor par de pivôs por meio da avaliação do resultado de uma função objetivo.

- *SampleBL*: algoritmo de *bulk-loading* que utiliza a técnica de amostragem para escolha dos pivôs. Assim, ao invés de testar todos os pares de pivôs possíveis, essa abordagem apenas testa os pares de pivôs coletados na amostra por meio da avaliação de uma função objetivo.

- *HeightBL*: algoritmo de *bulk-loading* que utiliza uma estratégia que se baseia na diferença de alturas entre uma estrutura estimada antes da execução do algoritmo e a estrutura que de fato está sendo construída à medida em que o algoritmo é executado.

Testes experimentais realizados para conjuntos de dados com diferentes volumes de dados e dimensionalidades a fim de validar os algoritmos de *bulk-loading* propostos mostraram que esses algoritmos introduziram diversas vantagens quando comparados com o algoritmo de inserção um-a-um da Onion-tree. Primeiro, a Onion-tree construída pelos algoritmos propostos possuem tamanho de 9% até 88% menores do que a Onion-tree construída pelo algoritmo de inserção um-a-um. Em consultas por abrangência, houve redução de 16% até 99% no número de cálculos de distância e de 9% a 99% no tempo gasto. Em consultas aos k-vizinhos mais próximos, houve redução de 13% a 86% em número de cálculos de distância e de 9% até 63% no tempo gasto.

1.2 Estrutura da Dissertação

Além desse capítulo de introdução, esta dissertação está estruturada em mais cinco capítulos:

- No capítulo 2 é descrita a fundamentação teórica necessária ao entendimento do projeto, com destaque para os conceitos básicos, que são espaço métrico, funções de distância e consultas por similaridade. Além disso, também são abordados os métodos de acesso mais relevantes ao trabalho proposto, como a R-tree, M-tree, Slim-tree, MM-tree e Onion-tree.

- No capítulo 3 são resumidos trabalhos correlatos, os quais abordam propostas de *bulk-loading* para os seguintes MAMs: R-tree, M-tree e Slim-tree. Nesse capítulo também são descritos trabalhos que abordam formas de *bulk-loading* genéricas.

- No capítulo 4 são descritos detalhes relacionados à proposta de mestrado. Em especial, o capítulo detalha a os algoritmos de *bulk-loading* propostos: *GreedyBL*, *SampleBL*, e *HeightBL*.

- No capítulo 5 são descritos os resultados dos experimentos realizados com os algoritmos de *bulk-loading* propostos, comparando-os com o algoritmo de inserção um-a-um da Onion-tree.

- O capítulo 6 conclui a dissertação e identifica propostas para trabalhos futuros.

Capítulo 2

Espaço Métrico e Métodos de Acesso

As consultas em um SGBD normalmente levam em consideração a relação de ordem total entre os elementos como critério para busca e comparação no método de acesso. Essa relação é aplicada normalmente entre tipos de dados tradicionais, como números e cadeias de caracteres. No entanto, quando se trata de dados complexos, como exemplo dados multimídia, sua comparação é feita por similaridade, ao contrário do uso da relação de ordem total. Dados multimídia são representados facilmente em um espaço métrico, a partir do instante em que uma métrica é definida.

Em especial, imagens também não estão sujeitas à relação de ordem total e, portanto, não podem ser comparadas utilizando o operador de igualdade ou os demais operadores relacionais. Para a recuperação de imagens em um SGBD, normalmente se utilizam predicados que levam em consideração a similaridade entre a imagem consultada e as imagens armazenadas na base de dados. Desse modo, algumas características da imagem são utilizadas como parâmetros de comparação de similaridade entre as imagens, como exemplo cor, forma e textura. Isso é conhecido como "recuperação de dados por conteúdo"(*Content-based retrieval - CBR*) [Smeulders et al., 2000]. Essas características das imagens são extraídas com o uso de extratores de características e sua representação é na forma de vetores de características. Um exemplo de vetor de característica é um histograma de cores de uma imagem.

Os vetores de características podem possuir tanto dimensão fixa quanto variada. No caso em que apresentam dimensão fixa, os dados podem ser indexados em um espaço multidimensional. No entanto, quando os vetores de características possuem dimensão variada, eles são chamados adimensionais e podem ser indexados em um espaço métrico se houver uma métrica definida [Marrach, 2011].

Neste capítulo é descrita a fundamentação teórica básica para o entendimento deste projeto de mestrado. Inicialmente são abordados nas seções 2.1 a 2.3, respectivamente, conceitos sobre espaço métrico, funções de distância e consultas por similaridade. Em seguida, na seção 2.4 são resumidos conceitos envolvendo métodos de acesso, com ênfase em métodos de acesso multidimensionais e métricos. Por fim, a seção 5.5 discute as considerações finais sobre o capítulo.

2.1 Espaço Métrico

Um espaço métrico $\mathbb{M}$ é definido como o par $< \mathbb{S}, d >$, no qual $\mathbb{S}$ é o domínio de objetos possíveis, e d é a função de distância, que se aplica para os elementos definidos em $\mathbb{S}$. Essa função de distância, $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$, retorna um valor com a distância entre dois objetos do domínio. Desse modo, existe uma relação da distância entre dois objetos e a similaridade envolvida entre eles: quanto menor a distância, mais similares os objetos são, e vice-versa. No entanto, o par $< \mathbb{S}, d >$ é apenas um domínio métrico quando a função de distância d atender às seguintes propriedades:

- Simetria: $d(s_1, s_2) = d(s_2, s_1)$.
- Não-negatividade: $d(s_1, s_2) \geq 0$.
- Identidade: $d(s_1, s_1) = 0$.
- Desigualdade triangular: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$.

A primeira propriedade, simetria, mostra que a distância entre o objeto s_1 e o objeto s_1 é a mesma, independentemente da ordem em que são dispostos os elementos. A segunda propriedade, não-negatividade, mostra que a distância entre dois elementos é sempre um valor maior ou igual a zero. A terceira propriedade, identidade, mostra que a distância de um elemento para ele mesmo sempre é zero. Por fim, a quarta propriedade ilustra a desigualdade triangular, a qual define que a distância de um elemento s_1 a outro elemento s_2 é sempre menor ou igual à soma das distâncias de s_1 e s_2 usando um terceiro elemento s_3.

Em especial, a propriedade da desigualdade triangular, em um MAM, se utiliza dos valores das distâncias que já foram previamente armazenadas para inferir os limites superior e inferior de descarte de uma busca. Desse modo, é necessário conhecer apenas duas distâncias entre elementos para realizar a poda de elementos no instante de uma consulta, que utiliza todas as propriedades destacadas. Como resultado, não é necessário calcular todas as distâncias entre o elemento de consulta e os elementos armazenados no MAM, reduzido assim a quantidade de cálculos de distância e evitando-se percorrer ramos desnecessários da estrutura de indexação [Bueno, 2009].

2.2 Funções de Distância

As funções de distância também são chamadas de métricas quando atendem às propriedades descritas na seção 2.1. Vetores com valores numéricos podem ser comparados utilizando-se funções métricas. Dentre as mais utilizadas, estão as da família *Minkowsi (Lp)* [Pola, 2005]. Essas métricas são normalmente empregadas em espaços vetoriais, que possuem dados que são vistos como pontos com dimensão fixa E, isto é, todos os elementos do espaço vetorial possuem a mesma quantidade de dimensões. Sendo a = $\{a_1, a_2, ..., a_E\}$ e b = $\{b_1, b_2, ..., b_E\}$ dois vetores que possuem dimensão E, a família de distâncias *Lp* é definida como:

$$Lp(a,b) = (\sum_{i=1}^{E} |a_i - b_i|^p)^{1/p}$$

Dessa família de funções, as mais conhecidas são a L_1, L_2 e L_{inf}. A métrica L_1 é conhecida como a distância *Manhattan* ou *distância de bloco*, e seu valor é referente ao somatório dos módulos das diferenças entre as coordenadas correspondentes dos vetores. A métrica L_2 é conhecida como a *distância euclidiana*, e é a função usual para calcular distância entre vetores. Por fim, a métrica L_{inf} é conhecida como *Chebyshev*, e é obtida com o cálculo do limite da equação quando p tende ao infinito. A Figura 2.1 ilustra a região de cobertura para cada função de distância *Minkowsi* e as suas respectivas fórmulas são exibidas a seguir.

- $L_1(a,b)(City\text{-}Block) = (\sum_{i=1}^{E} |ai - bi|$
- $L_2(a,b)(Euclidean) = (\sum_{i=1}^{E} |ai - bi|^2)^{1/2}$
- $L_{inf}(a,b)(Chebyshev) = max_{i=1}^{E} |ai - bi|$

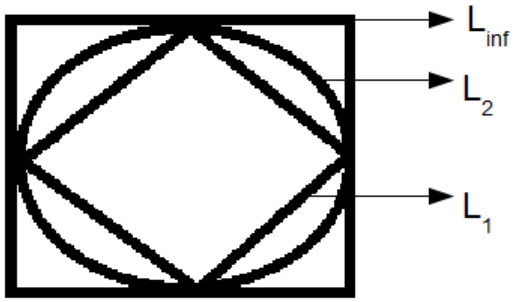

Figura 2.1: Métricas Minkowski. Figura reproduzida de [Bueno, 2009]

Pode-se dizer que o domínio contendo todas as palavras de uma linguagem não pode ser representado em espaços multidimensionais, pois se trata de um domínio adimensional. Nesse tipo de conjunto, a função de distância de *Levenshtein*, L_{edit}(a,b), é utilizada para medir a similaridade entre duas palavras [Bueno, 2009]. A versão sem atribuição de pesos às posições das cadeias de caracteres retorna o número mínimo de operações de edições necessárias para transformar a cadeia de caracteres a na cadeia de caracteres b. Desse modo, a distância mínima é zero e a distância máxima é o tamanho da maior palavra do domínio. A versão com pesos dessa função associa a cada operação de edição um custo. Por exemplo, L_{edit}("casa","asia") = 2 indica que são necessárias uma operação de remoção da letra “c” e uma operação de inserção da letra “i” na cadeia de caracteres “casa” para que ela se torne a cadeia de caracteres “asia”.

2.3 Consultas por Similaridade

O tipo de consulta utilizada em um espaço métrico é a consulta por similaridade, desde que ela requer uma função de distância métrica, que mede a dissimilaridade entre dois objetos do espaço. Esse tipo de consulta envolve o objeto consultado, um espaço e ainda parâmetros, que variam conforme o tipo da consulta por similaridade em questão. Como introduzido no capítulo 1, as consultas por similaridade mais utilizadas são as consultas pelos k-vizinhos mais próximos (*k-nearest neighbor queries*) e as consultas por abrangência (*range queries*). Enquanto que a consulta pelos k-vizinhos mais próximos

tem como objetivo retornar a quantidade k de objetos mais similares ao objeto de consulta, a consulta por abrangência tem o objetivo de recuperar os objetos similares ao objeto de consulta até um dado limite.

As seções 2.3.1 e 2.3.2 detalham definições formais para consultas por abrangência e aos k-vizinhos mais próximos, respectivamente. Nessas seções, são considerados o espaço métrico $\mathbb{M} = < \mathbb{S}, d >$ e o conjunto $\mathbb{S} = \{t_1, t_2, ..., t_n\} \subseteq \mathbb{M}$ [Chávez et al., 2001].

2.3.1 Consultas por Abrangência

A consulta por abrangência é realizada a partir de um elemento dado como referência $s_q \in \mathbb{S}$, um conjunto de elementos S $\subset \mathbb{S}$, uma função de distância d e um raio de cobertura r_q. A consulta $RQ(s_q, r_q)$ retorna os elementos contidos em $\mathbb{S}$ e que possuem no máximo distância r_q em relação ao elemento s_q. Assim, a resposta resultante, S´ $\subseteq \mathbb{S}$, é $\{s_i \in S | d(s_q, s_i) \leq r_q\}$.

Um exemplo de consulta por abrangência pode ser ilustrado pela seguinte sentença: "Selecione todas as palavras que sejam diferentes da palavra p em até 3 caracteres, onde o universo $\mathbb{S}$ é o conjunto de todas as palavras". Nesse exemplo, ilustrado na Figura 2.3, s_q é a palavra p, o raio de busca r_q é 3 caracteres, a métrica d é a função L_{edit} e $\mathbb{S}$ é uma base de dados contendo as palavras conhecidas [Marrach, 2011].

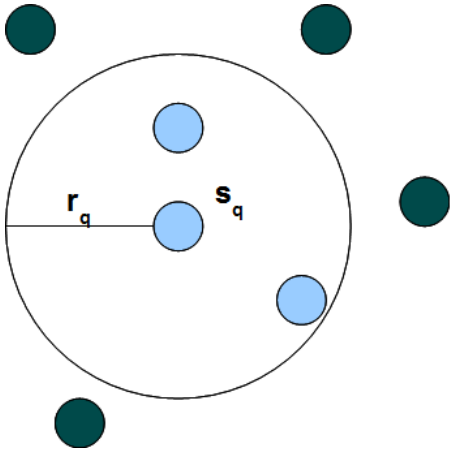

Figura 2.2: Exemplo de Consulta por Abrangência. Figura adaptada de [Bueno, 2009]

2.3.2 Consulta aos K-Vizinhos Mais Próximos

A consulta aos k-vizinhos mais próximos é realizada a partir de um objeto de referência $s_q \subset \mathbb{S}$, um conjunto de elementos S $\subset \mathbb{S}$ e o valor k de vizinhos mais próximos que se deseja recuperar. Assim, o resultado da consulta $KNN(s_q, k)$ é o conjunto dos k elementos mais próximos ao elemento s_q em $\mathbb{S}$. A resposta resultante, S´ $\subseteq \mathbb{S}$, é $(s_i | s_i \in$ S||S'| = k e $\forall s_i \in S', \forall s_j \in [S - S'], d(s_q, s_i) \leq d(s_q, s_j))$ [Bueno, 2009].

Um exemplo desse tipo de consulta pode ser ilustrado pela sentença: "Selecione as 3 palavras mais semelhantes à palavra p". Ao contrário de um raio de busca, utiliza-se a quantidade k de elementos que se deseja obter na resposta. Nesse exemplo, ilustrado na Figura 2.4, a quantidade k de elementos atribuída é 3 e a imagem i faz o papel do elemento de consulta s_q.

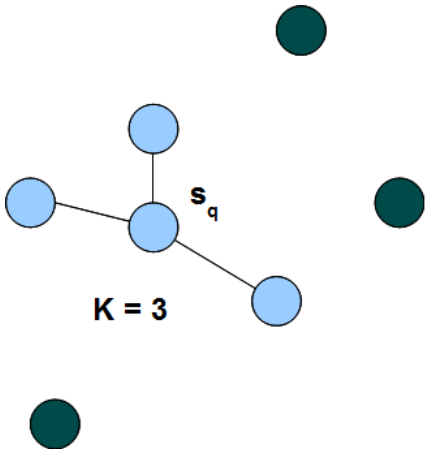

Figura 2.3: Consulta aos *k*-vizinhos mais próximos. Figura adaptada de [Bueno, 2009]

2.4 Métodos de Acesso

O uso de estruturas de dados para armazenar, recuperar e localizar dados é muito comum e, desde o princípio da informática, existe a tentativa de se obter estruturas mais eficientes para que essas funcionalidades sejam oferecidas. Uma das abordagens mais simples de resolver o problema de se buscar em um conjunto de dados é a chamada busca sequencial, sendo que o arquivo é percorrido até que o elemento de busca seja encontrado. Se o item a ser procurado não existir no arquivo, a busca sequencial deve percorrer todos os elementos do arquivo para garantir que isto é verdade [Pola, 2005]. Ou seja, o pior caso é O(n) para realizar a busca sequencial, onde n é o numero de elementos do arquivo. Esse processo pode ser adequado quando a quantidade de dados é pequena, mas pode ser inviável quando o volume de dados é muito grande.

Desde que consultas em SGBDs usualmente têm como resposta uma pequena quantidade de dados, é possível descartar partes dos elementos sem efetivamente compará-los, reduzindo assim o tempo necessário para realizar a busca como um todo. Inicialmente, estruturas como a *B-tree* [Bayer and McCreight, 1972] foram criadas para indexar elementos simples, tais como números e caracteres. Essas estruturas utilizam a relação de ordem total entre os elementos para que eles sejam indexados. Elas possuem um bom desempenho, realizando a consulta de elementos com complexidade sub-linear [Pola, 2005]. Como exemplo, uma *B-tree* possui complexidade $O(\log n)$ no pior caso em uma busca por uma única entrada.

Por outro lado, considere domínios em que os elementos são coleções de números. Supondo que essas coleções de números possuam sempre o mesmo tamanho, ou seja, dimensionalidade, podem ser usados os métodos de acesso multidimensionais para melhorar o desempenho no processamento de consultas a esses tipos de dados. Métodos de acesso multidimensionais usam estruturas geométricas n-dimensionais para indexar os elementos da base de dados. Diversos métodos de acesso multidimensionais foram propostos na literatura [Carélo et al., 2009], sendo que um método bastante conhecido é a R-tree [Guttman, 1984], voltada para indexar dados espaciais que são representados por geometrias compostas por um conjunto de coordenadas.

Métodos de acesso multidimensionais foram desenvolvidos para lidar com espaços de baixa dimensionalidade. Nesse sentido, eles sofrem do problema da maldição da dimensionalidade, o que significa

que em altas dimensões as distâncias entre os elementos tendem a se homogenizar. A quantidade de atributos aumenta o espaço de busca e as distâncias entre pares de objetos tendem a ser muito próximas. Assim, conforme a dimensionalidade aumenta, os espaços multidimensionais tendem a ser mais esparsos [Talavera, 1999]. Devido à maldição da dimensionalidade, métodos de acesso multidimensionais têm seu desempenho degenerado quando a dimensionalidade aumenta. Portanto, em se tratando de dados complexos, como imagens, os métodos de acesso mais amplamente utilizados são aqueles que foram desenvolvidos para espaços métricos, nos quais existe uma métrica que mede a dissimilaridade entre elementos do domínio.

As seções 2.4.1 e 2.4.2 detalham conceitos relacionados aos métodos de acesso multidimensionais e métricos, respectivamente. Essas seções também resumem os principais métodos de acesso relevantes para o desenvolvimento deste trabalho de mestrado. Note que, embora o enfoque do mestrado seja em métodos de acesso métricos, na seção 2.4.1 são detalhados conceitos de métodos de acesso multidimensionais, sendo descrito o método R-tree, devido ao fato desse método ser amplamente utilizado no contexto de base de dados, e prover uma abordagem de *bulk-loadin*g com conceitos fundamentais para o desenvolvimento deste projeto de mestrado.

2.4.1 Métodos de Acesso Multidimensionais

Métodos de acesso multidimensionais correspondem a uma estrutura de dados com o objetivo de indexar coleções de números que representam objetos espaciais. Esse tipo de método de acesso pode organizar os objetos espaciais em uma estrutura *hash*, em uma estrutura hierárquica, formando uma árvore, ou ainda utilizando uma combinação dessas formas. O principal objetivo dos métodos de acesso multidimensionais é prover a recuperação rápida de objetos que satisfaçam relacionamentos topológicos. O método de acesso multidimensional atua como uma etapa de filtragem dos objetos espaciais. Desse modo, ele reduz a quantidade de objetos espaciais a serem comparados em uma busca para uma quantidade muito inferior à quantidade total de objetos armazenados em um arquivo de dados [Ciferri, 2002].

A definição de um método de acesso multidimensional é feita a partir do seguintes elementos:

- uma estrutura de dados que gerencia objetos espaciais em disco.
- algoritmos de busca que permitem a consulta por relacionamentos topológicos.
- algoritmos para alterar a estrutura de dados, como algoritmos de inserção e remoção de objetos da estrutura. A operação de alteração de uma estrutura do método de acesso multidimensional é implementada por meio da operação de remoção seguida por uma inserção do objeto modificada.

A Figura 2.4 ilustra exemplos de tipos de dados espaciais, os quais são indexados por um método de acesso multidimensional. O ponto é considerado um objeto sem extensão e pode ser usado para representar uma cidade em um mapa, por exemplo. Uma linha é uma sequência de pontos conectados retilineamente e pode ser utilizada para representar, por exemplo, uma estrada. Uma linha poligonal se trata de uma linha na qual os pontos não necessariamente estão dispostos de modo retilíneo. Já um polígono é formado por uma sequência de linhas ou linhas poligonais na qual o primeiro ponto é o mesmo que o último e, assim, pode ser utilizado para representar, por exemplo, um estado. Polígonos complexos, por sua vez, podem possuir partes disjuntas ou buracos, como exemplo um país que contenha

ilhas. Por fim, um poliedro é limitado por quatro ou mais polígonos (faces) que possuem arestas que têm intersecção entre suas faces. Um poliedro pode ser usado para representar um sólido no espaço, como um membro do corpo humano.

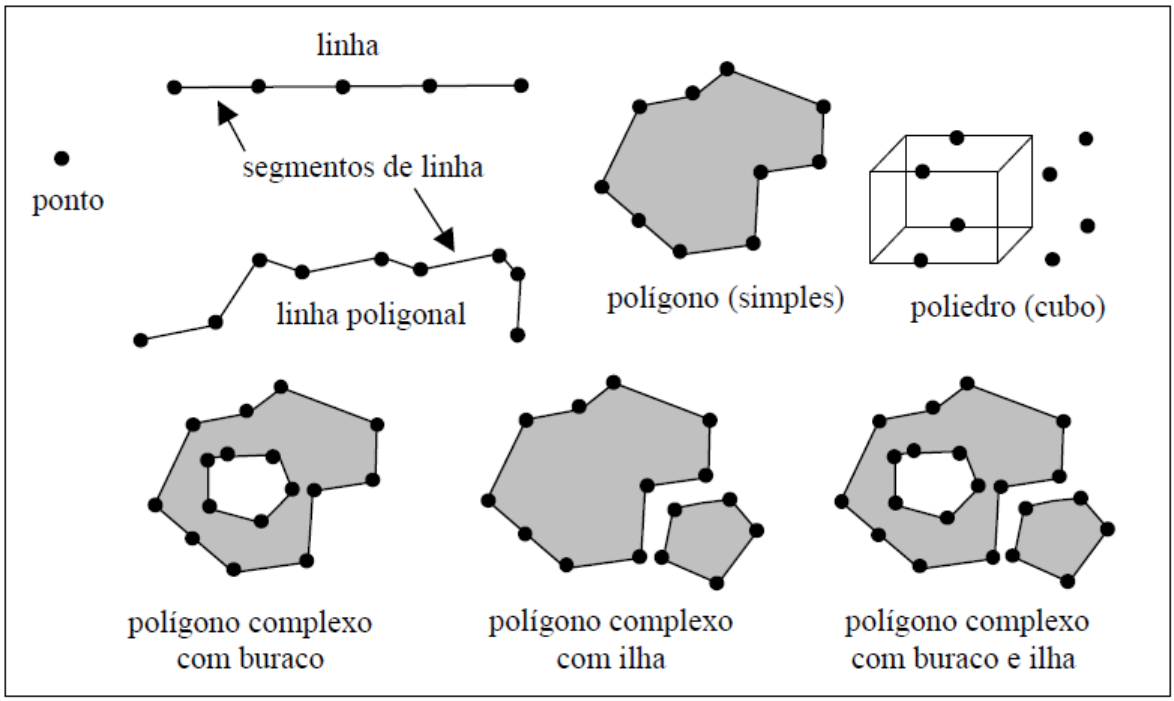

Figura 2.4: Tipos de dados espaciais. Figura reproduzida de [Ciferri, 2002]

Um método de acesso multidimensional utiliza-se de aproximações para representar os objetos que são armazenados no arquivo de dados. Uma primeira motivação para isso refere-se ao fato de que armazenar a geometria exata do objeto pode ser muito custoso quando há a necessidade de verificar os relacionamentos topológicos. Além disso, a quantidade de pontos necessárias para representar a geometria exata de objetos complexos pode ser muito grande, de centenas a milhares. Aproximações, por outro lado, simplificam a satisfação de relacionamentos topológicos, ao mesmo tempo que requerem poucos pontos para serem armazenados.

Uma aproximação deve ser conservativa, ou seja, ela deve conter cada ponto da geometria do objeto espacial que ela deve abstrair. Uma aproximação conservativa faz com que as propriedades do objeto armazenado sejam conservadas. Uma aproximação muito utilizada é o MBB (*Minimum Bounding Box*), ou MBR (*Minimum Bounding Rectangle*) ou "retângulo envolvente mínimo". O MBR consiste no menor retângulo d-dimensional com lados paralelos aos eixos e que contém completamente o objeto espacial. Ele necessita de poucos pontos para ser armazenado e também facilita a avaliação de predicados espaciais, devido às propriedades dos retângulos. Outras formas de aproximações conservativas são exibidas na Figura 2.5: retângulo envolvente mínimo rotacionado, círculo envolvente mínimo, polígono envolvente mínimo, casco convexo e elipse envolvente mínima.

Um *survey* sobre métodos de acesso multidimensionais é apresentado em [Gaede and Günther, 1998]. Dentre os métodos de acesso multidimensionais existentes, um dos

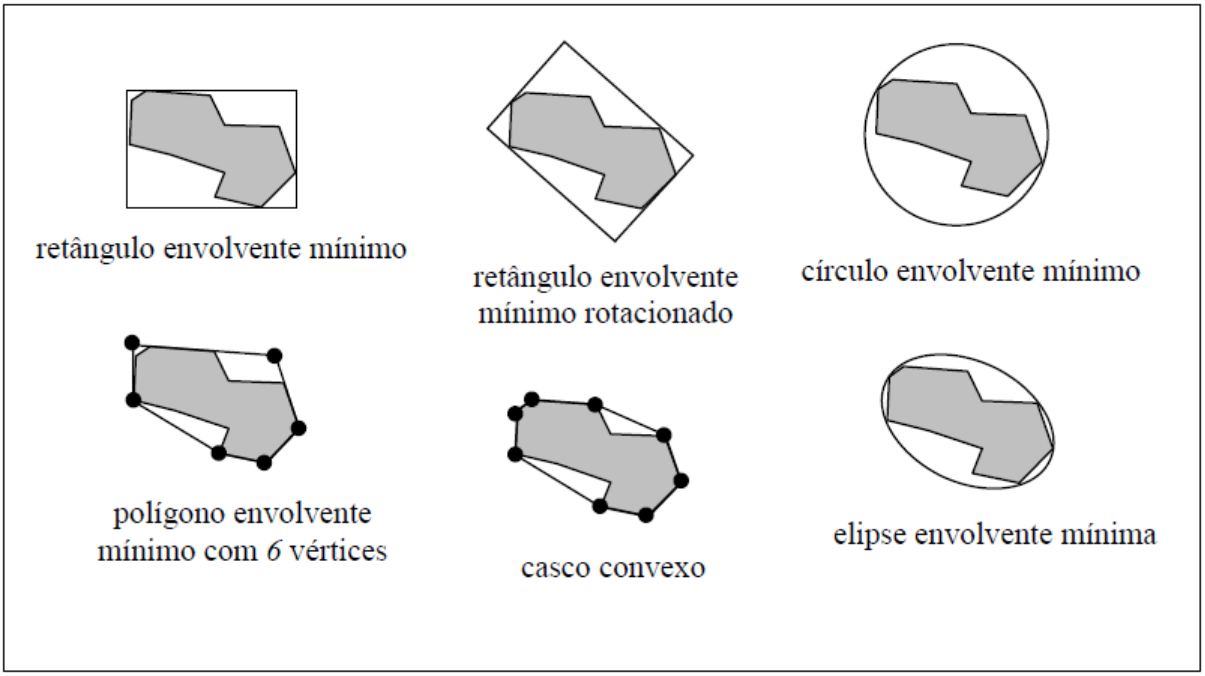

Figura 2.5: Aproximações conservativas. Figura reproduzida de [Ciferri, 2002]

mais utilizados é a R-tree. Assim, a sua estrutura de dados e algoritmos de busca e inserção são detalhados nesta monografia na seção 2.4.1.1.

2.4.1.1 R-tree

O método de acesso multidimensional R-tree [Guttman, 1984] é uma estrutura de dados com as seguintes características. Ela é balanceada, com os registros do índice nos nós folhas contendo ponteiros para os dados, os seus nós correspondem às páginas no disco e a estrutura é elaborada para que em uma busca espacial acesse um número pequeno de nós. Além disso, o índice é dinâmico, permitindo inserções e remoções a qualquer instante.

Estrutura de Dados da R-tree

Uma base de dados espacial consiste de uma coleção de tuplas representando objetos espaciais, na qual cada tupla tem um identificador único que pode ser usado para recuperá-la. Os nós folha de uma R-tree contêm entradas da seguinte forma:

- I: é um retângulo n-dimensional que contorna o objeto indexado.
- *Identificador da tupla*: é uma referência para a tupla na base de dados.

Os nós internos de uma R-tree possuem a seguinte forma:

- I: retângulo n-dimensional que cobre todos os retângulos dos nós dos níveis inferiores da estrutura.
- *Ponteiro para o nó filho*: endereço de um nó inferior na R-tree.

Considerando que M seja o número máximo de entradas que um nó pode armazenar e $m \leq M/2$ especifique a número mínimo de entradas em um nó, uma R-tree satisfaz às seguintes propriedades:

- Cada nó folha contém entre m e M elementos, a menos que ele seja a raiz.
- Para cada elemento em um nó folha, I é o menor retângulo que o contém espacialmente.
- Cada nó interno tem entre m e M elementos, a menos que ele seja o nó raiz.
- O nó raiz tem pelo menos dois filhos, a menos que ele seja um nó folha.
- Todas os nós folhas estão no mesmo nível da estrutura.

A Figura 2.6 ilustra a estrutura de dados de uma R-tree, enquanto que a Figura 2.7 exibe a respectiva distribuição espacial dos dados que a compõe.

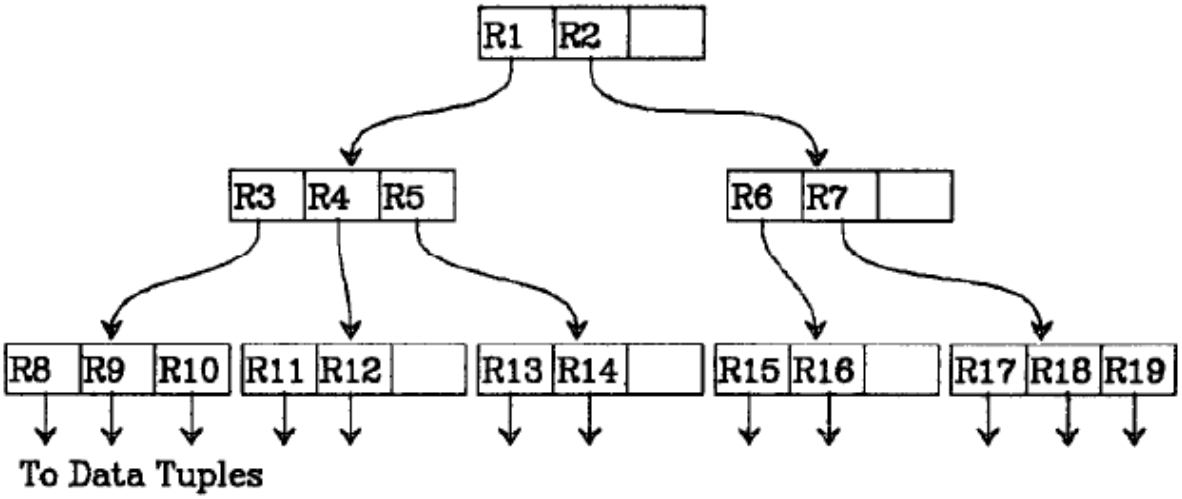

Figura 2.6: R-tree: Estrutura de dados. Figura reproduzida de [Guttman, 1984]

Inserção na R-tree

A inserção de elementos na R-tree é baseada nos seguintes princípios: os dados são inseridos nas folhas, os nós cuja capacidade foram excedidas se dividem em dois e essa divisão pode se propagar pela árvore.

O algoritmo de inserção na estrutura funciona do seguinte modo. Primeiro, ocorre, até que se chegue ao nó folha, a procura para a posição do novo registro, na qual é invocada a função *ChooseLeaf* que determina um nó folha adequado para inserir o elemento. Quando o nó folha correto for encontrado, verifica-se se ele possui espaço para inserção. Se possuir, o registro é inserido e, caso contrário, o nó é dividido em dois com o uso do algorimo *SplitNode* para que possa ocorrer a inserção. Se ocorreu a divisão, a estrutura deve ser reorganizada, verificando se a divisão se propagou nos níveis superiores. Finalmente, se o nó raiz se dividiu, é criado então uma nova raiz cujos filhos são os dois nós resultantes do último processo de divisão e a estrutura cresce em um nível.

Em se tratando do algoritmo *ChooseLeaf*, seu objetivo é determinar o nó folha no qual o novo elemento a ser inserido irá pertencer. Para isso, o algoritmo atua escolhendo os nós filhos tal que o

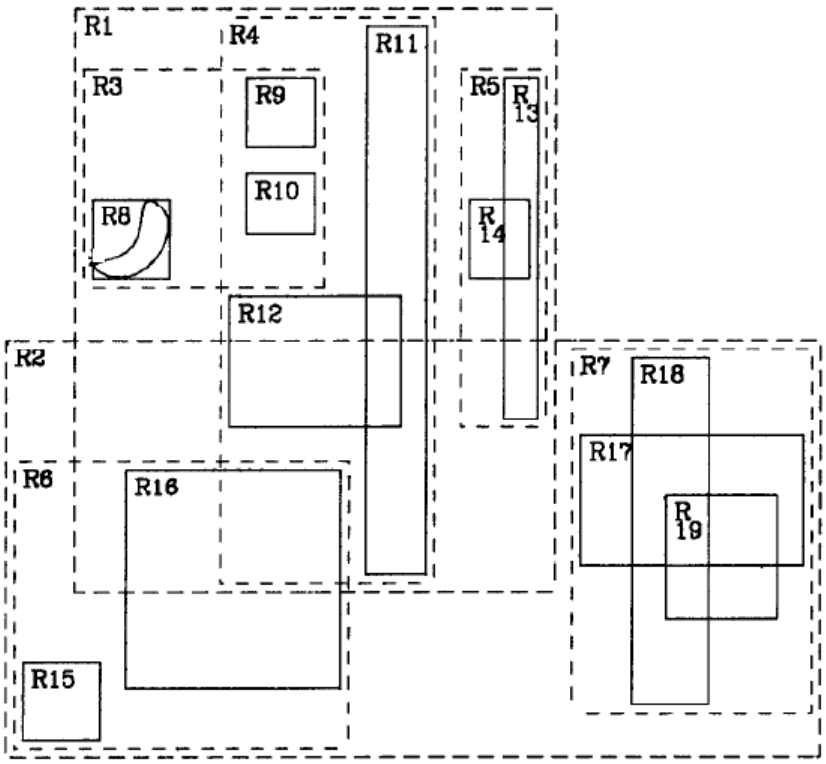

Figura 2.7: R-tree: Distribuição dos dados no espaço. Figura reproduzida de [Guttman, 1984]

MBR precise do menor aumento possível para que o elemento seja inserido. Em caso de muitos nós filhos atenderem a esse critério, o empate é resolvido escolhendo a entrada cujo MBR possui a menor área. Esse processo se repete até que seja alcançado um nó folha, que é o nó em que o novo elemento poderá ser inserido.

Com relação ao algoritmo de divisão do nó, *SplitNode*, ele possui três políticas para que o espaço seja particionado no instante da divisão do nó. Essas políticas são descritas a seguir.

1. *Algoritmo exaustivo*: é o método que testa todas as combinações possíveis de áreas de nós após a divisão para que se obtenha a área mínima após o processo. Esse algoritmo possui custo exponencial.

2. *Algoritmo de custo quadrático*: esse algoritmo tenta encontrar uma área de divisão do nó que seja pequena, mas não necessariamente a menor possível. Ele funciona escolhendo duas das *M+1* entradas para serem os primeiros elementos dos dois novos grupos, de tal modo que o par desperdiçaria a maior parte da área se ambos fossem colocados no mesmo grupo. A cada passo, a área da expansão requerida para adicionar cada entrada restante a cada grupo é calculada e a entrada atribuída é aquela que possui a maior diferença de área entre os dois grupos.

3. *Algoritmo de custo linear*: esse algoritmo é uma simplificação do algoritmo de custo quadrático. Ele simplesmente escolhe qualquer entrada do conjunto de entradas restantes para a sua execução.

Busca na R-tree

O algoritmo de busca da R-tree percorre a estrutura da raiz às folhas, podendo, nesse processo, percorrer

mais de uma subárvore se necessário. A R-tree é mantida de modo que permite ao algoritmo de busca eliminar regiões irrelevantes do espaço indexado e examinar apenas dados próximos à área de busca.

O algoritmo de busca funciona do seguinte modo. A partir de uma estrutura cujo nó raiz seja definido como T, o objetivo é encontrar todos os registros cujos retângulos sobreponham o retângulo de busca S. O primeiro passo é procurar nas subárvores. Se T não é um nó folha, cada entrada do nó deve ser investigada para verificar se o retângulo de busca S sobrepõe algum retângulo do nó. Por fim, se T é um nó folha, o algoritmo verifica todas as entradas que determinam se há sobreposição entre o retângulo de busca e algum retângulo do nó. Se o resultado for verdadeiro, então essa entrada é adicionada ao conjunto de resposta.

2.4.2 Métodos de Acesso Métricos

MAMs têm como objetivo realizar o particionamento do espaço de dados em regiões, escolher elementos representantes e agrupar os demais elementos ao seu redor. Essa organização dos dados forma uma estrutura hierárquica na qual a distância dos demais elementos aos seus representantes é utilizada para realizar operações como busca e inserção.

A classificação dos MAMs pode ser feita da seguinte forma [Chávez et al., 2001]. Existem aqueles que são baseados em métricas discretas e métricas contínuas. Eles podem ser também estáticos ou dinâmicos, de acordo com o fato de oferecerem suporte a inserções e remoções a partir do instante em que a estrutura é criada. Por fim, MAMs podem ser classificados ainda como armazenados em memória primária ou disco.

Vários MAMs têm sido desenvolvidos com o objetivo de aumentar a eficiência na construção e também nas formas de atualizar e realizar consultas aos dados complexos [Uhlmann, 1991, Yianilos, 1993, Bozkaya and Ozsoyoglu, 1997]. Com relação aos conceitos principais usados por esses MAMs, o trabalho de [Burkhard and Keller, 1973], descreve formas de particionamento do espaço recursivamente e com o uso de representantes para guiar a busca no espaço métrico. A primeira delas encontra um elemento representante e agrupa os elementos restantes de acordo com a distância deles a esse representante. A segunda técnica determina uma quantidade fixa de conjuntos para divisão, cada qual com o seu representante. Outro trabalho neste sentido é o realizado por [Uhlmann, 1991], que define duas formas de estruturar o domínio métrico por meio da decomposição por "bolas" (*ball decomposition*). Uma delas considera a divisão por "hiper-planos generalizados" (*generalized hiperplanes*), o que leva a uma divisão do espaço de busca sem sobreposições, enquanto que a outra forma considera que a "bola" métrica é uma região determinada por um centróide e um raio de cobertura.

Nas subseções a seguir são resumidos os MAMs mais importantes para o desenvolvimento deste projeto. Primeiramente serão descritos aqueles baseados em disco (ou seja, M-tree e Slim-tree) e em seguida, aqueles baseados em memória primária (ou seja, MM-tree e Onion-tree).

2.4.2.1 M-tree

O MAM M-tree [Ciaccia et al., 1997] particiona os objetos levando em consideração as suas distâncias relativas e os armazena em nós de tamanho fixo, que correspondem às regiões do espaço métrico. Esse MAM tem as seguintes características:

- voltado para disco.

- dinâmico, ou seja, aceita remoções ou inserções após a estrutura ser construída.
- balanceado, pois é de construção *bottom-up*.
- é baseado em “bolas"métricas.
- pode ter sobreposições, ou seja, a intersecção das regiões do espaço métrico, abrangidas por duas subárvores do mesmo nível, pode não ser vazia.

A estrutura dos nós da M-tree

Enquanto os nós folhas na M-tree armazenam todos os objetos indexados, representados pelos seus vetores de características, os nós internos armazenam os objetos representantes de cada nó. Um objeto representante é aquele que pertence aos dados, mas que foi atribuído o papel de representante do nó por algum algoritmo de promoção de objetos.

Em relação ao nó interno, para cada objeto representante O_r, existe um ponteiro associado, denotado $ptr(T(O_r))$, que referencia a raiz da subárvore, *T(O_r)*, que é chamada da árvore de cobertura de O_r. Assim, todos os objetos que pertençam ao raio de cobertura de O_r estão dentro da distância $r(O_r)$ de O_r, que é chamado de *raio de cobertura*. Por fim, o objeto representante O_r é associado com a distância para $P(O_r)$, seu objeto pai, que é o objeto que referencia o nó em que O_r está armazenado. Resumindo, tem-se que:

- O_r: objeto representante.
- *ptr(T(O_r)*: ponteiro para a raiz de *T(O_r)*.
- *r(O_r)*: raio de cobertura de O_r.
- *d(O_r,P(O_r))*: distância de O_r ao seu pai.

Em se tratando do nó folha, há uma entrada para o objeto O_j e o campo de ponteiro armazena o identificador do objeto, O_{id}, que pode ser usado para prover acesso ao objeto de fato, que está armazenado em um arquivo de dados separado. Além disso, há também a distância do objeto ao seu pai, definida por *d(O_j,P(Oj))*. Resumindo, tem-se que:

- O_j: objeto da base de dados.
- *O_{id}(O_j)*: identificador do objeto.
- *d(O_j,P(O_j))*: distância de O_j ao seu pai.

Construção da M-tree

O algoritmo de inserção da M-tree percorre recursivamente a árvore para localizar o nó folha mais adequado para inserir o novo objeto. Possivelmente, pode ocorrer divisão (*split*) do nó quando o nó estiver totalmente preenchido (*overflow*). O método utilizado para escolher o nó mais adequado para inserção é o seguinte: descer, a cada nível da árvore, pela subárvore na qual não é necessário aumentar o raio de cobertura do elemento representante. Se múltiplas subárvores possuem essa característica, a escolhida é a que o elemento representante possui a menor distância ao elemento a ser inserido. No entanto, se não há uma subárvore tal que não seja possível não aumentar o raio do elemento representante

do nó, a escolha da subárvore segue o critério de escolher aquela em que ocorre minimização do aumento do raio de cobertura.

O algoritmo de construção da M-tree atua fazendo com que essa árvore cresça no sentido *bottom-up*, isto é, das folhas para a raiz, assim como as estruturas dinâmicas e balanceadas mais comuns. Um nó *N*, quando tem sua capacidade máxima superada (*overflow*), tem seus elementos divididos entre o próprio nó, *N*, e um novo nó alocado, *N'*, ao mesmo nível de *N* na árvore. Além disso, há *promoção* para o nó pai, N_p, de dois objetos representantes para referenciar os nós *N* e *N'*. Vale lembrar também que esse processo de divisão de elementos entre os nós, quando ocorre *overflow*, é recursivo e pode ser propagado das folhas para a raiz da árvore. Assim, somente quanto a raiz possui *overflow* e divide os seus elementos entre dois nós, a M-tree aumenta em um nível.

Em relação ao algoritmo *split*, que realiza a divisão de elementos entre dois nós quando ocorre *overflow* em um nó, ele faz uso de um outro método, chamado *Promote*, que depende de algum critério para escolher dois objetos representantes, O_{p1} e O_{p2}, para inserir no nó pai. Além disso, o método chamado *Partition*, também utilizado pelo algoritmo de *Split*, é o responsável por dividir a entrada do nó com *overflow* em dois subconjuntos, que são armazenados nos nós *N* e *N'*. Desse modo, uma implementação dos métodos *Promote* e *Partition* depende da política de divisão de nós adotada (*split policy*).

A Figura 2.8 ilustra o processo de divisão de um nó com capacidade excedida em dois nós. O processo de divisão ocorre para acomodar todos os objetos. Desse modo, um novo nó é alocado no mesmo nível e os objetos são realocados conforme a sua distância ao representante de cada nó, que é promovido de acordo com algum algoritmo de promoção introduzido pelo método *Promote*.

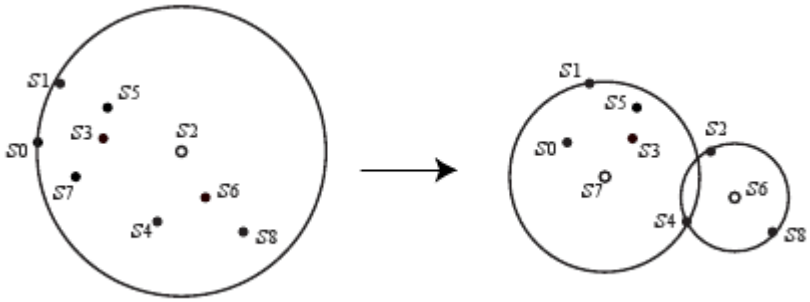

Figura 2.8: Exemplo de particionamento de um nó na M-tree. Figura reproduzida de [Carelo, 2009]

O método *Promote* determina quais os dois elementos que são promovidos ao nó pai de acordo com uma das seguintes políticas:

- *mRAD*: é a política mais complexa em termos de cálculos de distância, pois considera todos os possíveis pares de objetos e promove o par de objetos para os quais a soma do raio de cobertura $r(O_{p1}) + r(O_{p2})$ é mínimo.
- *mMRAD*: similar a *mRAD*, mas minimiza o máximo do raio de cobertura.
- *MLBDIST*: usa apenas distâncias que já foram pré-computadas e armazenadas na estrutura. O seu objetivo é escolher dois novos representantes, O_{p1} e O_{p2}, considerando que O_{p2} é equivalente a

O_p, além de determinar O_{p2} como o objeto mais distante de O_p: $d(O_{p2},O_p) = max\ d(O_j,O_p)$.

- *RANDOM*: escolhe os objetos representantes aleatoriamente. Apesar de não ser uma política inteligente, é rápida e seus resultados podem ser usados como base para comparar com outras políticas.

- *SAMPLIG*: é idêntica à política *RANDOM*, mas aplicada ao conjunto de objetos a ser redistribuído, gerando uma amostra de objetos. Para cada par dessa amostra, os objetos são redistribuídos e o raio de cobertura de ambos representantes é estimado, e é escolhido o par cuja soma dos raios é a mínima.

Por fim, o método *Partition*, que determina a divisão dos elementos de um nó com *overflow* em dois conjuntos, pode usar uma das duas estratégias descritas a seguir:

- *Hiperplano generalizado*: atribui cada objeto $Oj \in N$ ao elemento representante mais próximo: se $d(O_j,O_{p1}) \leq d(O_j,O_{p2})$, então O_j é atribuído para N_1, senão O_j é atribuído para N_2.

- *Balanceado*: calcula o valor de $d(O_j,O_{p1})$ e $d(O_j,O_{p2})$ para todos $O_j \in N$ e executa o seguinte algoritmo. N_1 é atribuído ao vizinho mais próximo de O_{p1} e é removido de N. Do mesmo modo, N_2 é atribuído ao vizinho mais próximo de O_{p2} e é removido de N.

O método balanceado associa a cada objeto representante o objeto mais próximo dele, forçando assim que todos os representantes cubram aproximadamente o mesmo número de objetos. Em relação ao método do hiperplano generalizado, ele enfoca a distribuição nos objetos não representantes, associando cada elemento ao seu representante mais próximo. Além disso, o método balanceado pode aumentar a sobreposição entre as regiões do espaço métrico, mas aumenta o uso do espaço nodal e, portanto, pode reduzir a altura da árvore quando comparado com o outro método. Por fim, o método hiperplano generalizado pode garantir que os dados sejam mais bem distribuídos na árvore, mas no entanto pode gerar a nós pouco utilizados.

2.4.2.2 Slim-tree

A Slim-tree [Traina-Jr et al., 2000] é um MAM balanceado, dinâmico, baseado em disco e que tem o seu crescimento no sentido *bottom-up*. Além disso, o seu particionamento do espaço métrico leva a regiões não disjuntas, sujeitas a sobreposições. A principal característica dessa estrutura é prover meios para a avaliação do grau de sobreposição dos seus nós (*Fat-factor*) e um algoritmo de otimização da estrutura (*Slim-down*).

Na Slim-tree, os dados são armazenados em páginas em disco de tamanho fixo, na qual cada página representa um nó da árvore. Todos os objetos são armazenados nas folhas. O objetivo da Slim-tree é organizar os objetos em uma estrutura hierárquica usando representantes como centro de cada região, que cobrem os objetos de uma subárvore.

A estrutura dos nós da Slim-tree

A estrutura de dados da Slim-tree comporta dois tipos de nós: os nós de dados, que são os nós folhas, e os nós internos. Cada nó tem uma capacidade máxima definida como C. Os nós folhas armazenam

todos os objetos que existem na estrutura da Slim-tree. Seus componentes são um vetor dos elementos resumidos a seguir:

- S_i: contém os atributos que compõem o elemento armazenado.
- O_{id}: é o identificador do objeto S_i.
- $d(S_i,S_{rep}))$: é a distância entre o objeto S_i e o objeto representante S_{rep}.

A estrutura de um nó interno da Slim-tree é um vetor dos seguintes elementos:

- S_i: objeto representante de sua hierarquia apontada por Ptr_i.
- R_i: raio de cobertura da região correspondente.
- $d(S_i,S_{rep})$: distância entre o representante do nó índice atual e o seu pai.
- $Ptr(TS_i)$: *link* que aponta para o nó raiz da subárvore cujo representante é S_i.
- $NEntries(Ptr(TS_i))$: número de entradas do nó apontado por $Ptr(TS_i)$.

A Figura 2.9 ilustra a estrutura lógica da Slim-tree, contendo sete cadeias de caracteres. A função de distância usada é a L_{edit} [Pola, 2010]. Outras duas ilustrações da Slim-tree são exibidas nas Figuras 2.10 e 2.11. Enquanto a primeira ilustra o particionamento do espaço métrico, a segunda ilustra a estrutura hierárquica correspondente.

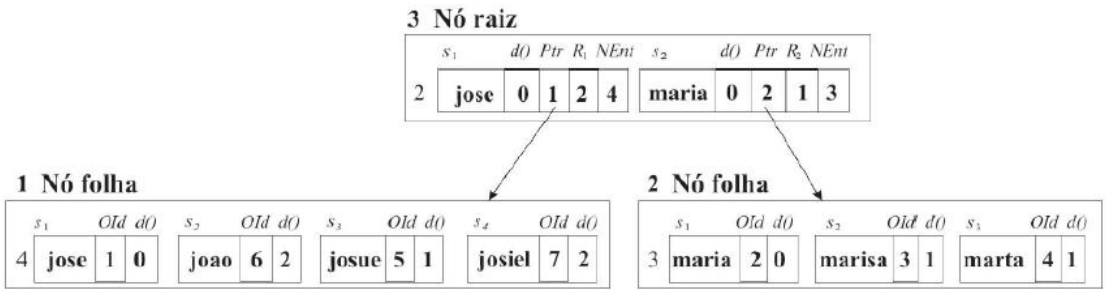

Figura 2.9: Slim-tree: Estrutura de indexação. Figura reproduzida de [Pola, 2010]

Construção da Slim-tree

A inserção de objetos na Slim-tree é realizada do seguinte modo. A partir do nó raiz, o algoritmo tem o objetivo de localizar um nó cujo raio possa cobrir o novo objeto. Se nenhum nó se encaixar nesse propósito, é escolhido aquele cujo centro é o mais próximo do novo objeto. No entanto, se por acaso mais de um nó puder ser escolhido, o algoritmo *ChooseSubtree* é invocado para escolher um deles. Assim, o processo é repetido recursivamente para cada nível da árvore.

Outra situação que pode ocorrer durante o processo de inserção na Slim-tree é o *overflow* de um nó, ou seja, quando o nó excede a sua capacidade máxima. Assim, quando isso ocorre, um novo nó *m'* deve

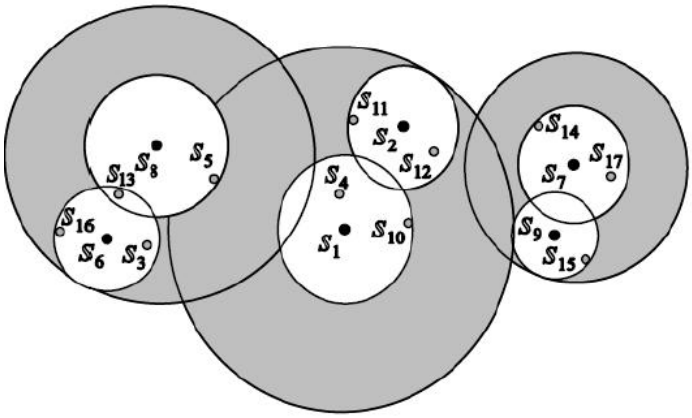

Figura 2.10: Slim-tree: Particionamento métrico. Figura reproduzida de [Bueno, 2009]

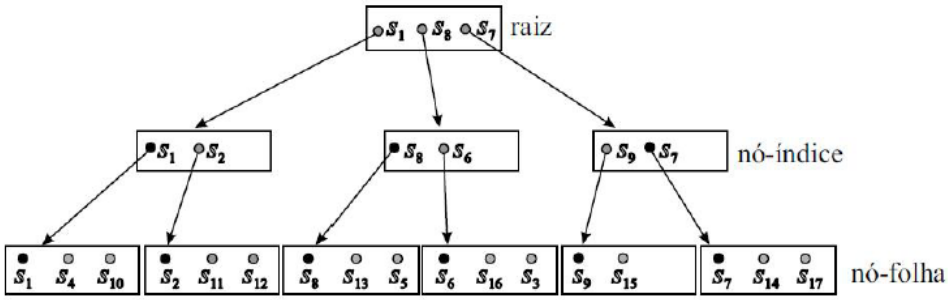

Figura 2.11: Slim-tree: Estrutura hierárquica. Figura reproduzida de [Bueno, 2009]

ser alocado no mesmo nível e os objetos são distribuídos entre os nós. Quando o nó raiz se divide, um novo nó raiz é criado e então a árvore cresce em um nível.

Com relação ao algoritmo *ChooseSubtree*, há três opções disponíveis na Slim-tree para que ele funcione:

- *random*: escolhe aleatoriamente um dos nós qualificados.
- *mindist*: escolhe um nó que tem a distância mínima do novo objeto ao centro do nó.
- *minoccup*: escolhe o nó cuja ocupação é a menor entre os nós qualificados. Ele utiliza o elemento *NEntries* do nó, que armazena número de entradas que um nó possui.

Em se tratando do algoritmo *Split* da Slim-tree, as seguintes opções são as disponíveis:

- *random*: escolhe aleatoriamente um dos dois objetos centrais novos e divide os demais elementos entre eles. Cada elemento é armazenado no nó cuja distância ao centro é a menor.
- *minMax*: todos os possíveis pares de objetos são considerados como representantes em potencial e o par escolhido é aquele que minimiza o raio de cobertura. Esse algoritmo tem custo elevado, cúbico, apesar de encontrar o melhor par de objetos possível.

- *MST*: uma *minimal spannimg tree* [Gallager et al., 1983] dos objetos é gerada e a maior das distâncias da árvore é removida. São obtidos dois agrupamentos e cada um é associado a um nó. Assim, o representante de cada nó é o elemento central. Esse algoritmo produz uma Slim-tree quase tão boa quanto a gerada pelo algoritmo *minMax* em tempo de processamento bem menor.

O particionamento do espaço métrico realizado pela Slim-tree gera regiões com sobreposição, o que pode impactar o desempenho da busca, por exemplo. Desse modo, a estrutura oferece meios para avaliar o grau de sobreposição dos nós e também meios para reorganizar a estrutura caso seja necessário. Para isso, existe o *Fat-Factor*, que é uma medida que retorna o grau de sobreposição entre os nós da árvore. Ademais, existe também o algoritmo *Slim-down*, que tem como objetivo reduzir a sobreposição das regiões, atuando nos nós folha da árvore. O seu princípio básico de funcionamento é não aumentar o raio de cobertura e utilizar os espaços disponíveis nos nós folhas para realocar os objetos reduzindo a sobreposição e a quantidade de acessos a disco necessárias para realizar as buscas. Esse processo, por ser custoso, é recomendado apenas quanto o valor de *Fat-factor* demonstra necessidade de execução ou após uma grande carga de dados.

Busca na Slim-tree

A busca na Slim-tree é realizada usando como base a propriedade desigualdade triangular. Assim, a poda pode ser aplicada sem precisar de cálculos suplementares. O algoritmo de consulta por abrangência começa a partir da raiz e percorre a árvore enquanto calcula a distância do elemento da busca e o representante de cada nó, nível a nível. Inicialmente, é calculada a desigualdade triangular, e o conjunto de elementos relevantes à consulta é determinado. Por fim, apenas para esses elementos são realizados os cálculos de distância ao elemento de referência da consulta para então determinar o conjunto final da resposta [Bueno, 2009].

A consulta aos k-vizinhos mais próximos faz uso de um raio dinâmico, que começa com valor infinito, e uma lista de páginas válidas que contém ponteiros para as subárvores que podem conter elementos na resposta. Essas páginas têm que ser visitadas em uma determinada ordem a fim de minimizar o tempo da realização da consulta. O critério escolhido é que a próxima página a ser percorrida seja aquela cujo representante está mais próximo do elemento da consulta. Os elementos que satisfazem à consulta são armazenados em uma lista, e a cada novo elemento essa lista é avaliada e atualizada com os k melhores elementos encontrados [Bueno, 2009].

2.4.2.3 MM-tree

O MAM MM-tree (*Metric Memory tree*) [Pola et al., 2007] é uma estrutura de indexação métrica que é utilizada em memória primária. Todos os seus nós possuem a mesma estrutura, não havendo diferenciação entre nós folha e não-folha. Esse MAM foi desenvolvido com o objetivo de atender à necessidade de responder consultas por similaridade com grande eficiência, em um conjunto de dados que caiba em memória primária.

Estrutura de Dados da MM-tree

A construção da MM-tree é realizada a partir de um particionamento recursivo do espaço em 4 regiões disjuntas. Para isso, são utilizados dois elementos, s_1 e s_2, para definí-las, que são os representantes de cada nó. A partir de então, cada elemento inserido s_i é atribuído a apenas uma região, o

que evita a sobreposição entre regiões. A fim de determinar a qual região um novo elemento inserido deve ser associado, há o cálculo das distâncias entre o novo elemento e os representantes do nó e, dependendo do resultado obtido, o elemento é atribuído a uma dessas 4 regiões. A Tabela 2.1 ilustra a região à qual um novo elemento é atribuído no instante de sua inserção com base em sua distância aos pivôs. Nessa tabela, r significa a distância entre pivôs, ou seja, $r = d(s_1,s_2)$.

Tabela 2.1: MM-tree: Atribuição dos elementos inseridos às regiões. Tabela reproduzida de [Pola et al., 2007]

$d(s_i, s_1)\ \theta\ r$	$d(s_i, s_2)\ \theta\ r$	*Região*
$<$	$<$	*I*
$<$	$\geq$	*II*
$\geq$	$<$	*IV*
$\geq$	$\geq$	*V*

Todos os nós da MM-tree, sejam eles nós folhas ou nós internos, possuem os mesmos componentes. Cada nó armazena no máximo dois elementos, que são utilizados como representantes de suas regiões. A Figura 2.12 ilustra a divisão do espaço métrico, enquanto que a Figura 2.13 ilustra a sua respectiva estrutura de dados. Os componentes que um nó da MM-tree armazena são definidos a seguir:

- s_1 e s_2: são os pivôs do nó.
- $d(s_2,s_2)$: distância entre os pivôs.
- Ptr_1,...,Ptr_4: são os ponteiros para as quatro regiões que armazenam os elementos filhos do nó atual.

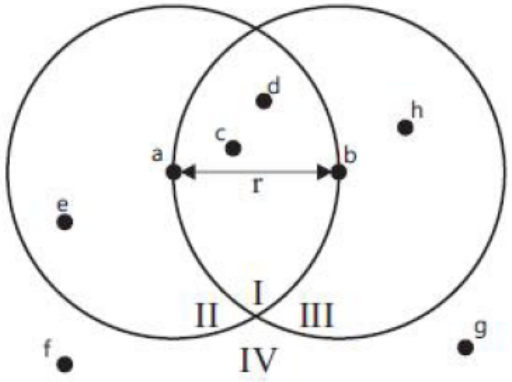

Figura 2.12: MM-tree: Divisão do espaço métrico. Figura reproduzida de [Pola, 2005]

A Figura 2.12 ilustra a distribuição dos elementos a, b,..., h no espaço métrico. Vale destacar que os elementos a e b são os pivôs e também que existe uma distância entre eles que é definida como sendo o

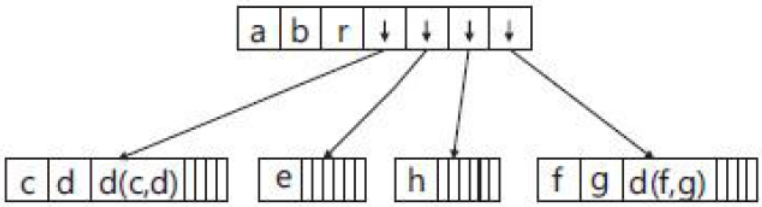

Figura 2.13: MM-tree: Estrutura de dados. Figura reproduzida de [Pola, 2005]

raio *r* da estrutura. A partir do raio, são definidas regiões circulares ao redor dos pivôs, o que determina cada uma das 4 regiões do espaço. Já a Figura 2.13 mostra a estrutura de dados da MM-tree por meio da hierarquia gerada pelos elementos ilustrados na Figura 2.12. Nela, pode-se verificar que os pivôs *a* e *b* estão no primeiro nível da estrutura, juntamente como valor de raio *r* e também os ponteiros para as quatro regiões por eles determinadas. O segundo nível da estrutura armazena os demais elementos, de *c* a *h*.

Construção da MM-tree

A MM-tree é uma estrutura dinâmica, o que significa que inserções na estrutura podem ocorrer a qualquer instante. Nessa estrutura, a inserção é iniciada a partir do nó raiz e o novo objeto é, a cada nível da estrutura, atribuído a uma região que pode ser definida como ilustrado na Figura 2.1. Esse processo se repete até que um nó folha da estrutura hierárquica possa ser utilizado para a sua inserção. Se por acaso o elemento já estiver na MM-tree, a inserção é interrompida para evitar duplicidade de elementos.

A MM-tree tende a se degenerar quando uma sequência de objetos inseridos mantém um padrão tal que a distância dos objetos contíguos é muito pequena [Pola, 2005]. Nesse caso, a estrutura pode se tornar desbalanceada, e semelhante a uma lista encadeada, o que resulta em uma queda no desempenho no momento da busca. Portanto, é importante considerar a questão do balanceamento da MM-tree, e, nesse sentido, ela introduz um algoritmo de semi-balanceamento, o qual é usado para que a estrutura seja balanceada o tanto quanto possível.

A Figura 2.14 ilustra o funcionamento do algoritmo de semi-balanceamento da MM-tree. Na Figura 2.14(a) é exibida a situação em que a árvore cresce em um nível, após a inserção do elemento indicado por *j*. Isso faz com que a árvore cresça em um nível, apesar de existirem espaços vazios e que podem armazenar o elemento em outras regiões. O algoritmo proposto atua na estrutura da Figura 2.14(a) da seguinte maneira. Primeiramente, os elementos de *a* até *h* são organizados em uma lista e uma matriz de distâncias entre os objetos. A partir disso, o algoritmo considera cada par de elementos e verifica se é possível reorganizar todos os elementos em um mesmo nível da estrutura. A condição de parada do algoritmo é quando o primeiro par de objetos representantes é encontrado ou quando esgotam-se todas as possibilidades de combinações possíveis. Na Figura 2.14(b) é mostrada a configuração final após a execução do algoritmo sobre os elementos exibidos na Figura 2.14(a). Percebe-se que uma melhor distribuição dos elementos foi encontrada ao trocar os elementos representantes (a,b) por (d,e) no primeiro nível da árvore.

O algoritmo de balanceamento da MM-tree só deve ser ativado durante a inserção e somente quando o objeto inserido fizer com que a subárvore alvo da inserção cresça em um nível. Ainda, o algoritmo

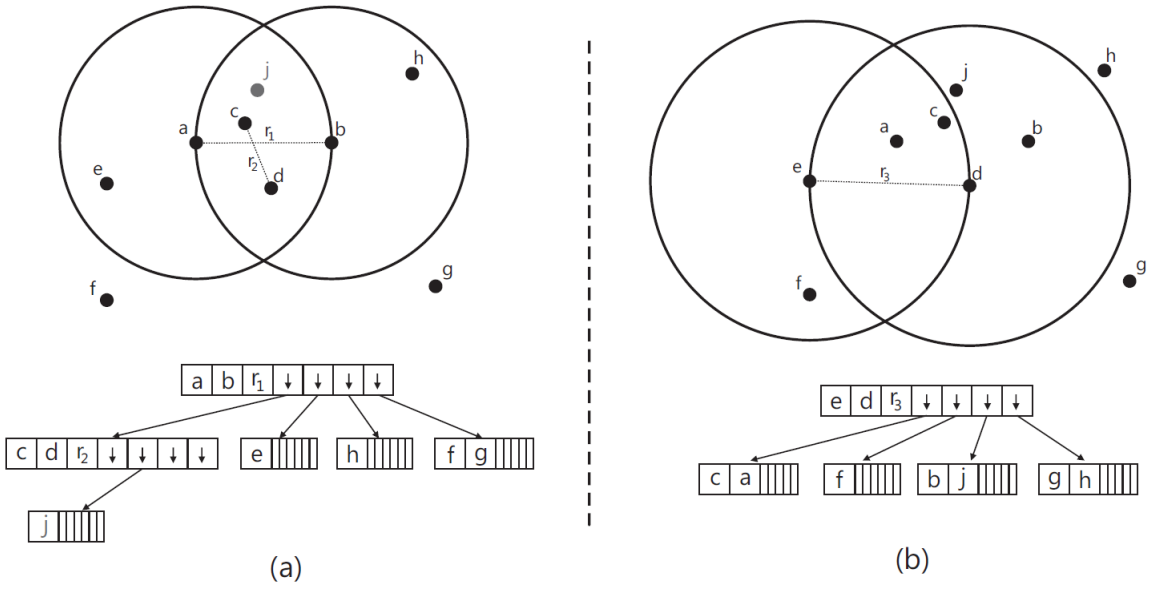

Figura 2.14: MM-tree: Algoritmo de semi-balanceamento. Figura reproduzida de [Pola, 2005]

somente é aplicado quando o número de objetos naquele nível está entre dois e oito. Desse modo, a matriz de distância que é criada nunca é maior do que 8x8 e sua criação é feita para evitar a repetição de cálculos de distância.

Busca de elementos na MM-tree

A MM-tree responde consultas pontuais e também responde às consultas por abrangência e aos k-vizinhos mais próximos usando como base quais regiões intersectam a região da consulta [Marrach, 2011] (2.2). A consulta por abrangência recebe como parâmetros o elemento base para a consulta s_q e o raio r_q. A partir disso, a cada nó visitado, se a distância de s_q ao pivô for menor do que o raio de consulta, o pivô é adicionado ao conjunto de resposta. O próximo passo consiste em verificar as regiões subordinadas ao pivô e em que há intersecção do raio r_q de acordo com o que é ilustrado na Tabela 2.2.

Tabela 2.2: MM-tree: Regra para visitação das regiões. Tabela reproduzida de [Pola et al., 2007]

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 aos k-vizinhos mais próximos, os parâmetros necessários são o elemento da consulta, s_q, e a quantidade k de elementos mais próximos do elemento base. Nesse tipo de consulta, o raio r_q é dinâmico e tem o seu valor alterado de acordo com a execução do algoritmo, assim como uma lista de elementos que serão parte da resposta. O ponto inicial é a raiz da MM-tree. Ademais, o raio r_q tem seu valor inicial definido como infinito e a lista é vazia. A cada instante em que um elemento mais próximo a s_q é encontrado, o raio da consulta é diminuído e o elemento da lista que é mais distante a s_q

é substituído pelo elemento encontrado. Esse algoritmo também consulta as regras definidas na Tabela 2.2 para determinar as regiões que devem ser analisadas.

2.4.2.4 Onion-tree

A Onion-tree [Carélo et al., 2009] é um MAM que estende as funcionalidades da MM-tree. Assim, a Onion-tree também é voltada para memória primária e divide o espaço métrico em regiões disjuntas com base em dois representantes. No entanto, a Onion-tree melhora algumas funcionalidades da MM-tree, que pode ficar com a sua estrutura desbalanceada por gerar sub-regiões com diferentes tamanhos - o algoritmo de semi-balanceamento requer tempo de processamento quadrático, sendo portanto, muito custoso.

As principais características da Onion-tree são listadas a seguir:

- Nova forma de particionamento do espaço métrico para controlar o número de regiões disjuntas geradas em cada nó. Esse é o procedimento de expansão da Onion-tree, e pode gerar mais do que 4 regiões disjuntas.
- Capacidade de mudança dos pivôs dos nós folhas durante as operações de inserção. Essa é a política de substituição da Onion-tree, que leva a um melhor particionamento do espaço métrico e evita tanto que as regiões fiquem muito grandes quanto muito pequenas. Isso melhora o custo para construção do índice e o custo para processar consultas por similaridade.
- Novos algoritmos de consulta por abrangência e consulta aos k-vizinhos mais próximos, de forma que esses ofereçam suporte ao novo método de particionamento do espaço métrico. Ademais, a Onion-tree também incorpora uma nova definição de uma ordem apropriada de visita para as consultas aos k-vizinhos mais próximos.

Estrutura de dados da Onion-tree

A estrutura do nó da Onion-tree é a mesma para nós folhas e também para nós internos, a exemplo da MM-tree. Seus componentes são descritos a seguir:

- $N.s_1$ e $N.s_2$: representam os dois pivôs armazenados no nó.
- $N.r$: representa a distância entre os dois pivôs.
- *N.Expansion*: número de expansões que o nó possui.
- *N.Region*: número total de regiões que o nó possui.
- *N.Parent*: ponteiro para o nó pai.
- *N.Child[1...N.Region]*: vetor de ponteiros para os nós filhos.

A estrutura de dados da Onion-tree permite que o espaço métrico seja dividido em mais do que quatro regiões disjuntas. A partir de uma estrutura semelhante à MM-tree, a Onion-tree realiza o particionamento do nó em mais regiões utilizando a multiplicidade do raio inicial. A Figura 2.15 ilustra o procedimento de particionamento dessa estrutura. Nela, pode-se observar na Figura 2.15(a) a estrutura inicial do nó N com apenas quatro regiões. Na Figura 2.15(b), o nó N foi expandido uma vez, gerando o

nó *N'*. Nesse processo, a Onion-tree duplicou o seu raio inicial, que é determinado pela distância entre os pivôs. Desse modo, a região *IV*, a mais externa, foi particionada em outras regiões, denominadas *IV', V', VI' e VII'*. Na Figura 2.15(c) é ilustrada a situação final do nó, após mais uma expansão, aplicada sobre o nó *N'*, gerando o novo nó *N"*. Nesse exemplo, o raio inicial, definido entre os pivôs, foi triplicado e a região *VII* gerou mais regiões: *VII", VIII", IX" e X"*.

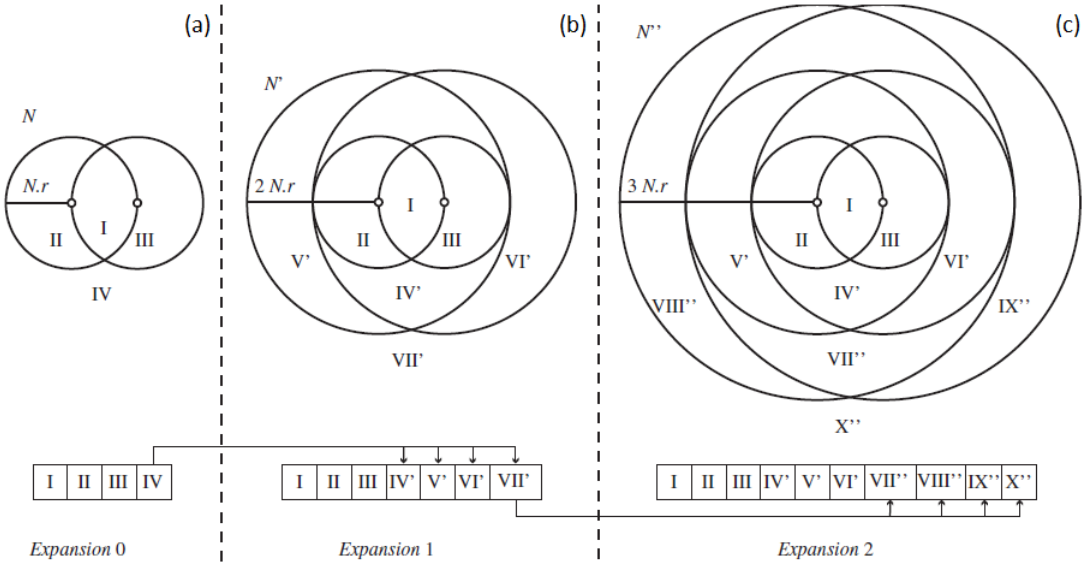

Figura 2.15: Onion-tree: Particionamento do nó. Figura reproduzida de [Carélo et al., 2009]

Um aspecto importante relacionado à expansão na Onion-tree é a definição da quantidade de expansões que devem ser aplicadas a um nó. Assim, o procedimento de expansão introduz duas propostas: ser fixo ou variado. A primeira proposta, gera a estrutura chamada *F-Onion-tree*, enquanto que a segunda proposta gera a estrutura chamada *V-Onion-tree*. Enquanto a *F-Onion-tree* aplica a mesma quantidade de expansões por nó, de acordo com um parâmetro de entrada, a *V-Onion-tree* aplica diferentes números de expansões para cada nó. No entanto, essa política somente realiza a expansão quando o raio entre os pivôs tem valor menor do que o raio do nó pai, pois de outro modo as regiões do nó já cobrem essa área e não há necessidade de mais expansões. Essa abordagem é chamada *keep-small strategy* e seu objetivo é manter a região externa do nó a menor possível. Desse modo, pode-se observar que a política de expansão exerce um papel muito importante na construção da estrutura.

O algoritmo responsável pelas expansões da Onion-tree é chamado de *CreateRegions*. Ele determina o número de expansões que devem ser aplicadas a um nó N, em função de um parâmetro de entrada *E*. Se $E > 0$, a política de expansões é determinada para fixa e o valor de *E* é utilizado como o número de expansões para cada nó. Senão, a política é determinada como sendo variável e o número de expansões é determinado pela política *keep-small*.

Outra questão a ser considerada na Onion-tree é o algoritmo que realiza a associação de um elemento com a sua região adequada no momento de sua inserção na estrutura. O algoritmo responsável por isso é chamado *ChooseRegion*. A entrada desse algoritmo é um nó *N* e as distâncias d_1 e d_2, que são as distâncias do elemento a ser inserido aos pivôs do nó. Assim, o algoritmo verifica com base nessas distâncias qual a região adequada para a inserção do novo elemento. Se nenhuma região for encontrada,

o elemento é associado à região externa ao nó *N*.

Inserção de Elementos na Onion-tree

A Onion-tree se trata de um MAM dinâmico. Desse modo, ela aceita inserções após a sua criação e também possui uma política para evitar que a árvore fique desbalanceada. Em relação a isso, a estrutura também possui três políticas de substituição de pivôs, chamada *Replacement Technique*, que tentam buscar um melhor particionamento do espaço: *Keep-small*, *Maximize-expansions* e *Minimize-expansions*, as quais são descritas a seguir.

- *Keep-small*: essa política mantém o mesmo objetivo da política *Keep-small* usada para construir a árvore. Assim, seu objetivo é escolher os pivôs de modo que a região externa ao nó seja a menor possível.
- *Maximize-expansions*: essa política seleciona os pivôs mais próximos, maximizando a quantidade de expansões do nó.
- *Minimize-expansions*: essa política seleciona os pivôs mais distantes, minimizando a quantidade de expansões do nó.

A aplicação da política de substituição de pivôs ocorre quando está em curso a inserção de um novo elemento e ocorre a verificação de que o subespaço do nó pode ser melhor particionado a partir de outros representantes. Essa política atua apenas antes de inserir um elemento s_i em um nó folha que está cheio. Ela também atualiza o raio do nó com a distância dos novos pivôs após a troca dos elementos. Para que o processo ocorra, é realizada uma análise combinatória entre o elemento novo e os dois representantes do nó folha, e é aplicada a política de substituição de pivôs parametrizada para a árvore.

A Figura 2.16 ilustra a substituição de pivôs durante a operação de inserção. Nela, verifica-se que no instante da inserção do elemento s_i, o espaço métrico é melhor particionado se o pivô s_1 do nó for trocado de posição com o elemento s_i. Para isso, os valores r e d_2 devem ser atualizados adequadamente.

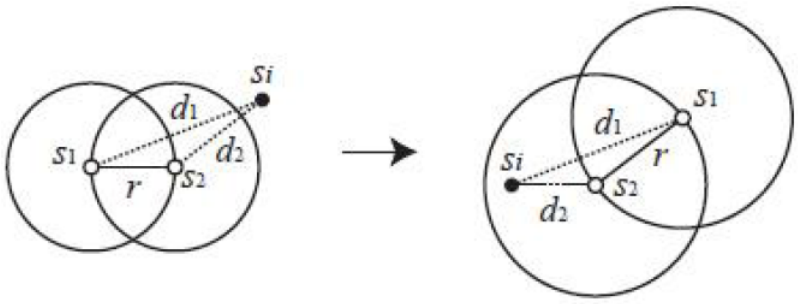

Figura 2.16: Onion-tree: Substituição de pivôs. Figura reproduzida de [Carélo et al., 2009]

Busca de Elementos na Onion-tree

Os algoritmos de busca que a Onion-tree oferece foram estendidos em relação aos algoritmos da MM-tree. A MM-tree considerava apenas quatro regiões fixas em um nó para realizar a sua busca, ao contrário da Onion-tree, que pode ter mais regiões dependendo dos parâmetros de sua construção.

Desse modo, o algoritmos para consulta por abrangência e o algoritmo de consulta aos *k*-vizinhos mais próximos foram estendidos para considerar qualquer quantidade de regiões que forem criadas pelo procedimento de expansão.

Uma novidade que a Onion-tree introduz é que, no algoritmo de consulta aos *k*-vizinhos mais próximos, há uma nova sequência de visitação para as regiões do nó a fim de conseguir maior eficiência no processamento da consulta. Esse processo é visualizado na Figura 2.17, que ilustra a ordem de visitação das expansões e regiões para um elemento de consulta *Sq* que está na região IV' de um nó com duas expansões. De acordo com a política de visitação proposta, a ordem das expansões visitadas é definida por 1, 0 e 2. E também, desde que o elemento de consulta esteja na região IV' e mais próximo de s_2 do que s_1, a ordem de visita das regiões na expansão 1 é IV', VI' e V'. Na Tabela 2.3 é descrito o modo utilizado para determinar quais as regiões visitadas em uma consulta aos *k*-vizinhos mais próximos.

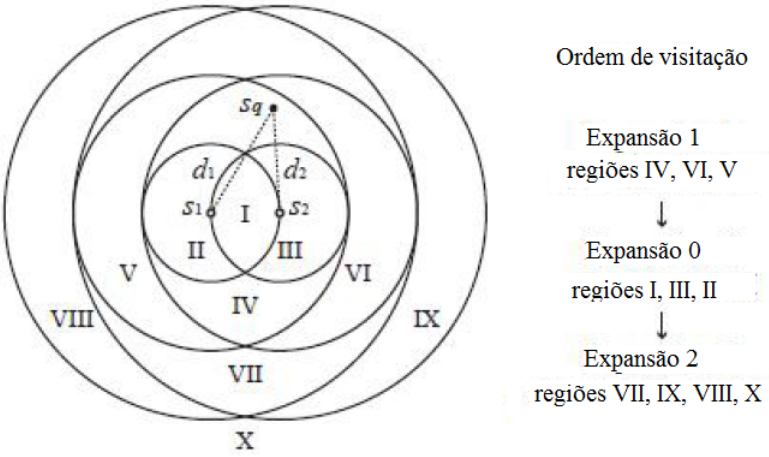

Figura 2.17: Onion-tree: Sequência de visitação dos nós na consulta aos *k*-vizinhos mais próximos. Figura reproduzida de [Carélo et al., 2009]

Tabela 2.3: Onion-tree: Tabela para determinar a sequência de visitação dos nós na consulta aos *k*-vizinhos mais próximos. Figura reproduzida de [Carélo et al., 2009]

Região de S_q mod 3	Condição	Ordem de visitação			
		1°	2°	3°	4°
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	$d_1 > d_2$	$3E+4$	$3E+3$	$3E+1$	$3E+2$

Formalmente, a ordem de visitação das expansões é a seguinte. Primeiro, é visitada a região do nó no qual o elemento de consulta se encontra. Então, são visitadas as demais regiões do nó de acordo com a sua proximidade ao elemento da consulta. Essa política determina tanto a ordem de visita das expansões

do nó como também a ordem de visita das suas regiões. As expansões são visitadas na seguinte ordem: (i) a 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. Essa sequência de visitação definida, partindo de uma região interna para uma externa, ou seja, de $E - 1$ para $E + 1$, permite uma redução mais rápida do raio dinâmico, já que as regiões internas são menores e têm mais chance de conter elementos próximos. Ainda, essa ordem de visitação otimiza a aplicação da poda de elementos.

2.5 Considerações Finais

Nesse capítulo foram descritos conceitos básicos relacionados ao projeto de mestrado. Os aspectos abordados incluem espaço métrico, funções de distância e consultas por similaridade. Além disso, também foram descritos os seguintes métodos de acesso existentes na literatura que são de importância dentro do contexto do projeto: R-tree, M-tree, Slim-tree, MM-tree e Onion-tree. A Tabela 2.4 descreve os principais conceitos que foram abordados para cada método de acesso.

Tabela 2.4: Resumo dos métodos de acesso abordados

Nome	Tipo de Método de Acesso	Memória	Conceito de Divisão do Espaço	Algoritmos Descritos	Tratamento de Sobreposição	Dinâmico
R-tree	Multidimensional	Secundária	Aproximação (MBR)	Inserção e Busca	Sim	Sim
M-tree	Métrico	Secundária	Bola	Inserção e Busca	Sim	Sim
Slim-tree	Métrico	Secundária	Bola	Inserção e Busca	Sim	Sim
MM-tree	Métrico	Primária	Bola	Inserção, Semi-Balanceamento e Busca	Desnecessário, sem sobreposição	Sim
Onion-tree	Métrico	Primária	Bola	Inserção, Substituição de Pivôs e Busca	Desnecessário, sem sobreposição	Sim

No próximo capítulo, capítulo 3, são resumidos trabalhos correlatos, os quais abordam propostas de *bulk-loading* para os MAMs R-tree, M-tree e Slim-tree. Com relação à MM-tree e à Onion-tree, esses MAMs não oferecem operações de *bulk-loading*. Adicionalmente, no capítulo 3 também são descritos trabalhos que abordam formas de *bulk-loading* genéricas.

Capítulo 3

Trabalhos Correlatos

Normalmente os métodos de acesso possuem operações que realizam a inserção de elementos um-a-um. No entanto, existe uma segunda opção mais eficiente a se considerar, que é a inserção de elementos em grande quantidade, que é chamada *bulk-loading*. O algoritmo de *bulk-loading* consiste na criação da estrutura de indexação considerando todos os elementos do conjunto de dados de uma vez. Sua principal vantagem é analizar com antecedência os elementos para organizá-los de modo a obter uma estrutura de dados que possa responder às consultas com maior eficiência em relação à estrutura de dados criada a partir da inserção aleatória de elementos.

Existem algumas formas de se realizar o algoritmo de *bulk-loading*, dentre elas estão a forma baseada em ordenação, baseada em *buffer* e baseada em amostragem. A forma de *bulk-loading* baseada em ordenação é a mais utilizada para métodos de acesso tradicionais. Nesse caso, os dados são ordenados e a estrutura é construída de modo *bottom-up*. Em relação ao *bulk-loading* baseado em *buffer*, ele faz uso de uma estrutura de dados chamada *buffer-tree* e funciona selecionando um subcojunto de dados que caiba em memória primária. A desvantagem desse método é que o seu desempenho depende da ordem na qual são inseridos os elementos de entrada. Além disso, ele ignora os custos de processamento, tentando reduzir apenas os custos de acessos a disco. A forma de *bulk-loading* baseada em amostragem escolhe uma amostra com tamanho suficiente para que caiba em memória primária e constrói uma estrutura para a base de dados inteira usando como base estimativas providas pela amostra. Esse método tem o grande problema de depender da qualidade da amostra coletada. Se a amostra for ruim, o algoritmo de *bulk-loading* pode demorar muito mais do que o tempo necessário para a inserção sequencial [Vespa et al., 2010].

Para que um método de acesso seja utilizado por um SGBD, é necessário que ele proporcione inserção sequencial, isto é, que os elementos sejam inseridos um por vez. O algoritmo de *bulk-loading* não é obrigatório, mas pode ser significantemente mais rápido do que a inserção sequencial quando um índice é criado a partir de uma grande quantidade de elementos ou quando um índice é destruído para depois ser reconstruído.

Neste capítulo são descritos trabalhos existentes na literatura que possuem relação com o presente projeto de mestrado. Desse modo, são abordados trabalhos que tratam da proposta do algoritmo de

bulk-loading tanto para métodos de acesso multidimensionais quanto para métodos de acesso métricos. A seção 3.1 resume trabalhos que realizam *bulk-loading* na *R-tree* [Guttman, 1984], a seção 3.2 descreve o trabalho de *bulk-loading* da M-tree [Ciaccia et al., 1997], e a seção 3.3 sintetiza o trabalho de *bulk-loading* da *Slim-tree* [Traina-Jr et al., 2000]. A seção 3.4 discute dois trabalhos de *bulk-loading* genéricos. O capítulo é finalizado na seção 3.5, com as considerações finais.

3.1 Bulk-loading da R-Tree

A maioria dos trabalhos que realizam *bulk-loading* na R-tree possui a mesma idéia básica: os MBRs de entrada são ordenados de acordo com algum critério global de uma dimensão (como a coordenada x ou o valor do centro do MBR, por exemplo) e colocados nos nós folhas nessa ordem. Em seguida, o restante do índice é construído recursivamente no sentido *bottom-up*.

Nesta seção são descritos três desses trabalhos, que tratam de diferentes abordagens para realizar o algoritmo de *bulk loading*. A seção 3.1.1 descreve o algoritmo *Top Down Greedy Splitting Algorithm*, enquanto que a seção 3.1.1 aborda o algoritmo *Overlap Minimizing Top-Down bulk-loading* e, finalmente, a seção 3.1.3 é a detalha o algoritmo de *bulk-loading* da R-tree com o uso de *buffers*.

3.1.1 TGS: Top Down Greedy Splitting Algorithm

O algoritmo *TGS* (*Top Down Greedy Splitting Algorithm*)[García R et al., 1998] de *bulk-loading* da R-tree tem duas características principais: (i) os dados de entrada são particionados em subárvores no sentido *top-down*; e (ii) a cada nível da árvore, ele considera todos os cortes ortogonais aos eixos coordenados e os escolhe a partir da otimização de uma função de custo determinada pelo usuário. Isso é, o algoritmo cria a estrutura no sentido *top-down* por meio de um passo que reorganiza os elementos da base de dados que devem ser posicionados abaixo do nó sendo construído. O algoritmo *TGS* tem duas idéias que motivaram o seu desenvolvimento:

- Minimizar os primeiros níveis da estrutura, já que o potencial para redução de custo é maior.
- Considerar todas as partições criadas por cortes tais que as subárvores resultantes estejam totalmente preenchidas.

O funcionamento do algoritmo é o seguinte. *TGS* aplica recursivamente o passo da divisão, que particiona um conjunto de *n* MBRs em dois subconjuntos aplicando-se um corte ortogonal aos eixos. O corte realizado deve satisfazer às seguintes condições: (i) o custo da função objetivo *f(r1,r2)* deve ser mínimo (onde *r1* e *r2* são MBRs); ii) um subconjunto tem cardinalidade i*S, para algum *i* e *S* é fixado por nível, e diz a respeito ao número máximo de MBRs por subárvore. Por fim, a recursão é aplicada em ambos subconjuntos até que haja um subconjunto para cada nó filho. O algoritmo da divisão é ilustrado na Figura 3.1.

3.1.2 OMT: Overlap Minimizing Top-Down bulk-loading Algorithm for R-Tree

O algoritmo de *bulk-loading* da R-tree chamado OMT (*Overlap Minimizing Top-Down bulk-loading*) [Lee and Lee, 2003] é realizado no sentido *top-down* e tem o objetivo de minimizar a área de sobreposição entre os nós internos para melhorar o desempenho no processamento de consultas na

```
Seja n = número de retângulos de entrada
Seja S = número máximo de retângulos por sub-árvore
Seja M - número máximo de entradas por nó
Seja f(r1,r2) a função de custo
Se n ≤ S retorne {condição de parada}
Para cada dimensão d
    Para cada ordenação considerada na dimensão d
        Para i de 1 até n/M - 1
            Seja B0 = MBR dos primeiros i*S retângulos
            Seja B1 = MBR dos demais retângulos
            Armazenar i se f(B0,B1) é o melhor valor
Dividir o conjunto de entrada e ordenar na melhor posição
```

Figura 3.1: TGS: Algoritmo da divisão. Figura adaptada de [García R et al., 1998]

estrutura. A principal idéia desse trabalho é que quando ocorre sobreposição em nós internos, ela é mais crítica para o desempenho das consultas do que quando ela ocorre em nós folhas. Resumidamente, esse trabalho determina a topologia da R-tree resultante com alguns cálculos e então agrupa os dados para criar as entradas do nó raiz. Já que a árvore é criada da raiz para as folhas, existe a possibilidade de controlar a região de sobreposição nos nós internos. Por fim, ocorre o particionamento recursivo de cada entrada para criar os níveis inferiores com a restrição de 100% de utilização do espaço.

Detalhadamente, o primeiro passo do algoritmo consiste em determinar a topologia da árvore resultante da aplicação do algoritmo de *bulk-loading*. Essa etapa é possível devido ao uso da restrição de 100% de utilização do espaço. Assim, a altura h da árvore é calculada para que em seguida possa ser determinada a quantidade de itens que podem ser inseridos em uma subárvore que é filha de uma entrada no nó raiz. Já que existe a restrição de 100% de preenchimento do espaço, a maioria dos nós terão M entradas e, a partir disso, pode ser calculado o tamanho *Nsubtree* da subárvore. A seguir são detalhadas as fórmulas para determinar h e *Nsubtree*, nas quais N representa a quantidade total de itens de dados e M representa a capacidade máxima de um nó da R-tree.

- $h = \lceil logM^N \rceil$
- $Nsubtree = M^{h-1}$

A partir dessas fórmulas são determinados o número de entradas do nó raiz e o número de cortes verticais no nível superior, chamado de S. A fórmula também representa o número de nós a serem gerados a partir de um corte vertical. A fórmula para esse cálculo é descrita a seguir.

- $S = \left\lfloor \sqrt{\lceil N/Nsubtree \rceil} \right\rfloor$

Por fim, com o valor obtido de S, pode-se criar as entradas do nó raiz por meio do particionamento dos itens de dados em S grupos. Cada corte vertical é particionado em S grupos e cada grupo será uma entrada do nó raiz. Em seguida, cada entrada do nó raiz é particionada em M grupos, sendo que cada grupo novamente será a entrada do nó inferior. Assim, recursivamente, cada grupo do nível inferior é dividido em M grupos até que cada grupo contenha somente M itens. A Figura 3.2 ilustra

o funcionamento do algoritmo *OMT* e da sua divisão *top-down* de um conjunto de 9 MBRs. Nela, pode-se notar que inicialmente o nó raiz é dividido em dois MBRs, um à esquerda e um à direita, e que armazenam respectivamente cinco e quatro MBRs no seu nível inferior.

No algoritmo OMT, o particionamento em M grupos realizado a cada nível é feito por meio da ordenação dos dados em relação às coordenadas x e y, em turnos de níveis. Por exemplo, como inicialmente a ordenação é feita em relação a x, quando for o nível *h-2* da árvore, os dados são ordenados em relação à coordenada y. Já no nível *h-4*, os dados são ordenados em relação à coordenada x, e assim por diante.

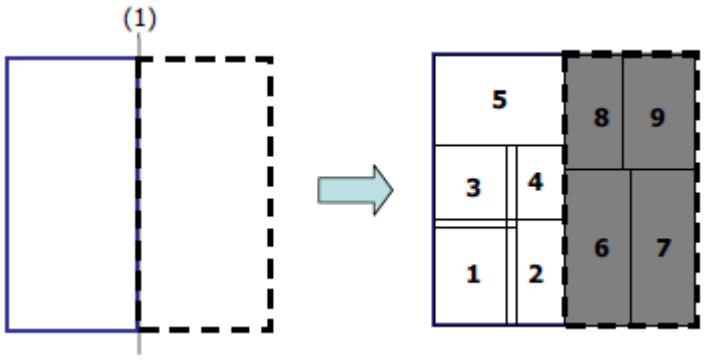

Figura 3.2: OMT: Exemplo do *bulk-loading top down*. Figura reproduzida de [Lee and Lee, 2003]

3.1.3 Efficient Bulk Operations on Dynamic R-trees

[Arge et al., 1999] descreve um algoritmo de *bulk-loading* para a R-tree com o uso de uma estrutura chamada *buffer tree*. Além disso, ele introduz técnicas para realizar uma grande quantidade de consultas (*bulk-queries*) e de remoções (*bulk deletions*). O algoritmo considera que a R-tree para o qual ele irá ser executado possui as duas funções básicas a seguir:

- *Route(r,v)*: para um determinado nó v da R-tree e um MBR r a ser inserido, retorna a melhor (de acordo com alguma heurística) subárvore v_s para inserir r. Se necessário, a função também atualiza o MBR armazenado em v que corresponde a v_s.

- *Split(v)*: dado um nó v, divide v em dois novos nós: v' e v''. A função também atualiza as entradas que apontam para os novos nós nos seus correspondentes nós pais.

Do mesmo modo, o algoritmo considera que, para realizar consultas, a R-tree possui a seguinte função básica:

- *Search(r,v)*: a partir de um MBR r e um nó v, retorna o conjunto de subárvores v_s cujos MBRs associados em v interceptam r.

Esse algoritmo de *bulk-loading* é uma variação da técnica de *buffer tree*, descrita em [Arge, 1995]. Sejam B o número de MBRs por bloco de disco e M o número de MBRs que cabem em memória primária, a idéia principal do algoritmo é a seguinte. Inicialmente, *buffers* são adicionados a todos os

nós da R-tree em cada $\lfloor logB^{M/4B} \rfloor$-ésimo nível da árvore. Isto é, considerando que os nós folhas estão no nível 0, *buffers* de tamanho M/2B blocos são atribuídos aos nós do nível i*$\lfloor logB^{M/4B} \rfloor$ para i = 1,2,.... A Figura 3.3 ilustra uma R-tree com *buffers*. Desse modo, as operações na estrutura são feitas de modo "atrasado". Por exemplo, considere que está sendo executado um conjunto grande de inserções. No instante da inserção de um MBR *r*, a função *Route(r,v)* não é imediatamente utilizada para procurar qual a subárvore adequada para a sua inserção. Ao invés disso, espera-se até que pelo menos *B* MBRs a serem inseridos sejam armazenados em um bloco de *buffer* no nó raiz (que está armazenada em disco). Quando um *buffer* está cheio, inicia-se o seu processo de esvaziamento. Para cada MBR *r* do *buffer*, repetidamente é chamada a função *Route(r,v)* para que esse MBR chegue até o próximo nó *buffer v'*. Desse modo, o MBR é inserido no nó, que pode ser esvaziado do mesmo modo quando estiver cheio. Ao chegar a um nó folha, o MBR é inserido e, se necessário, o índice é reestruturado usando a função *Split*. O algoritmo apenas move os primeiros *M/4* MBRs a cada processo de esvaziamento, a fim de evitar que a função de *Split* desencadeie um efeito cascata de reorganização do índice.

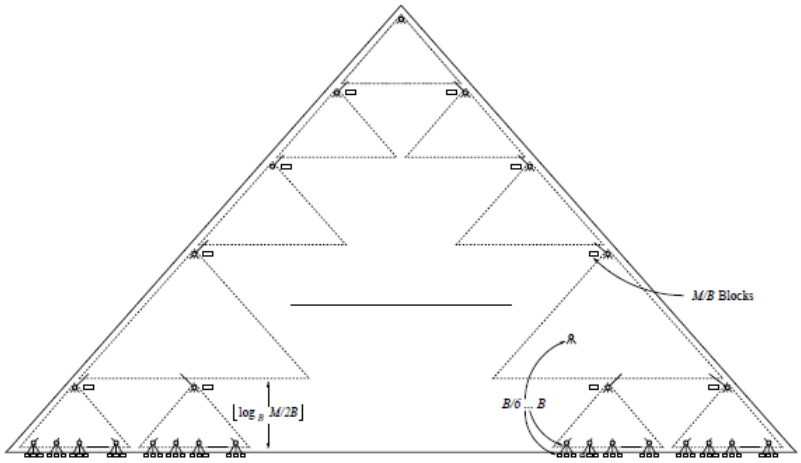

Figura 3.3: Algoritmo de *bulk-loading* da R-tree com *buffers*: Exemplo de R-tree com *buffers* em seus nós. Figura reproduzida de [Arge et al., 1999]

É importante ainda detalhar um ponto do processo do algoritmo de *bulk-loading* descrito: como a reestruturação é realizada, isto é, como o *buffer* de um nó é esvaziado. O índice é reestruturado a partir das chamadas à função *Split* e, para isso é tomado um cuidado especial quanto às chamadas a ela. Primeiramente, a R-tree cuja raiz é *v* é carregada em memória primária. Então, todos os MBRs do *buffer* são carregados e cada um deles é dirigido para um nó folha e lá inserido. Se o nó folha tiver a sua capacidade excedida e, o que significa *overflow*, a função de *Split* é invocada para dividir esse nó e também, se necessário, os demais nós que estão em níveis superiores, até o nó raiz. Se a divisão de nós se propagar até o nó raiz, então o processo atuará em partes da árvore que não estão em memória primária ainda. Quando isso acontece, o processo de esvaziar o *buffer* é interrompido temporariamente e as árvores cujos pais são os dois novos nós gerados inicialmente pela função *Split* são escritos em disco. Para cada MBR do nó que apresentou *overflow*, a função *Route* é chamada para decidir em qual *buffer* dos dois novos nós criados ele deverá ser inserido e, então, cada MBR é escrito de volta ao seu *buffer* em disco. Por fim, o processo de reestruturação é propagado recursivamente aos níveis superiores do índice

por meio da função *Split*.

O trabalho também apresenta uma forma de realizar grandes quantidades de consultas, chamada de *bulk queries*. Para isso, a solução abordada é adicionar *buffers* à R-tree e realizar as consultas de modo "atrasado", assim como são realizadas as inserções. Para realizar uma consulta, é inserido então o MBR de consulta no *buffer* do nó raiz. Quando o *buffer* é esvaziado, uma cópia do MBR de uma consulta é recursivamente guiada pelas subárvores e em cada um dos *buffers* que são relevantes para a resposta à consulta. Quanto o MBR chega ao nó folha, os MBRs relevantes, que atendem aos parâmetros da consulta, são retornados.

Em se tratando de remoções em massa, *bulk deletions*, primeiramente é realizada uma consulta, como descrito anteriormente, para o MBR *r* a ser removido e, quando esse MBR chega ao nó folha, a remoção é realizada. Essa remoção pode fazer com que o nó folha atinja uma capacidade inferior à sua capacidade mínima, o que exige que o índice seja reestruturado por meio da união de nós, que pode se propagar para os níveis superiores da árvore.

3.2 Bulk-loading da M-Tree

O algoritmo de *bulk-loading* da M-tree [Ciaccia and Patella, 1998] tem como idéia básica a geração de agrupamentos de objetos, que são definidos como $S = \{O_1, ...O_n\}$, com as restrições de capacidade mínima (u_{min}) e máxima, (u_{max}) de utilização dos nós. O algoritmo funciona do seguinte modo. A partir de um conjunto de objetos *S*, *k* objetos são aleatoriamente selecionados e inseridos no conjunto amostra *F*. O próximo passo consiste em atribuir cada elemento restante do conjunto *S* ao seu elemento mais próximo do conjunto de amostras, produzindo *k* conjuntos: F_1,...,F_k. Em seguida, o algoritmo de *bulk-loading* é invocado em cada um desses *k* conjuntos, gerando *k* subárvores. Por fim, o algoritmo é invocado mais uma vez sobre o conjunto *F*, para obter a árvore superior, T_{sup}. Assim, cada subárvore obtida é então inserida como folha de T_{sup} de acordo com o objeto da amostra O_f correspondente e a árvore final é obtida.

Nesse processo podem acontecer duas situações indesejadas: ou as subárvores têm diferentes alturas ou nós com ocupação inferior à mínima permitida (*underflow*). Para resolver esses problemas, existem duas técnicas utilizadas:

- Reinserir os objetos dos conjuntos que estão com *underflow* em outros conjuntos e remover a amostra correspondente de **F**.

- Dividir as subárvores com alturas maiores, obtendo subárvores menores. As raízes das subárvores obtidas são inseridas no conjunto de amostras **F**.

Enquanto a primeira heurística gera um número menor de amostras e garante que cada nó das subárvores tenha a utilização mínima permitida, a segunda técnica aumenta o número de amostras ao dividir as subárvores de maior altura. Se um conjunto de objetos é o resultado final dessa fase, o processo de *bulk-loading* é totalmente repetido, no entanto, a partir de uma nova amostra.

A Figura 3.4 ilustra o funcionamento do algoritmo de *bulk-loading* descrito. Supondo *umin* = 1/2, os nós A' e I são removidos e seus objetos associados aos nós D e H, respectivamente. Considerando que a subárvore de altura mínima tem raiz em B, as subárvores cuja raiz são A e C são divididas, gerando as

subárvores cujas raízes são D, E, C"', H, F e G. Por fim, o algoritmo constrói a árvore superior a partir de todas as subárvores. A estrutura final da M-tree após as operações é ilustrada na Figura 3.5.

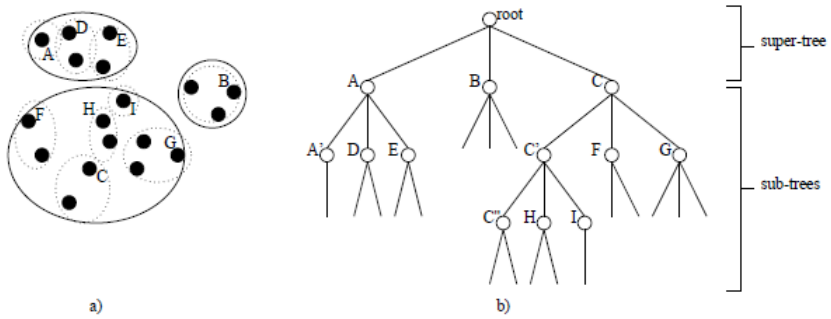

Figura 3.4: Exemplo de execução do algoritmo de *bulk-loading* da M-tree. Figura reproduzida de [Ciaccia and Patella, 1998]

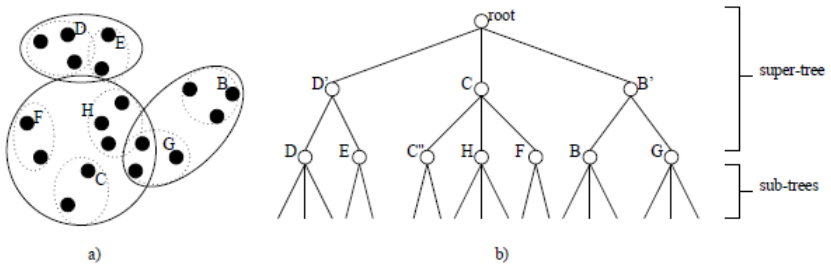

Figura 3.5: Exemplo de funcionamento da fase de redistribuição do algoritmo de *bulk-loading* da M-tree. Figura reproduzida de [Ciaccia and Patella, 1998]

3.3 Bulk-loading da Slim-Tree

O algoritmo de *bulk-loading* da Slim-tree [Vespa et al., 2010] tem o objetivo de melhorar o processo de criação da estrutura de dados, assim como também criar estruturas que permitem melhor desempenho na consulta. Ele constrói a estrutura no sentido *top-down*, baseado em técnicas de amostragem, e cria árvores balanceadas com reduzida sobreposição nos nós. No entanto, dependendo do conjunto a ser indexado, o algoritmo pode produzir estruturas com alta sobreposição, o que faz com que a estrutura responda às consultas com desempenho pior do que a inserção dos elementos sequencialmente. Nesse sentido, os autores [Vespa et al., 2010] propõem, além do algoritmo de *bulk-loading*, uma técnica para determinar previamente se o algoritmo está construindo uma estrutura considerada ruim.

A idéia principal do *bulk-loading* da Slim-tree é prever a estrutura final antes de realizar o algoritmo de *bulk-loading* propriamente dito. Isto é feito com o uso de amostras como representantes dos nós. Assim, a proposta é estimar o número de nós e de elementos em cada nó antecipadamente e então distribuir os elementos entre os nós. O algoritmo busca construir uma árvore balanceada desde o início,

ao contrário do algoritmo de *bulk-loading* da M-tree que inicialmente constrói a árvore para depois balancear.

O primeiro passo do algoritmo é estimar a quantidade de nós que irão compor a estrutura final. Os parâmetros considerados são: a quantidade *N* de elementos a serem indexados, a capacidade máxima *CM* de elementos por nó e a ocupação mínima *Cm* de elementos por nó. Para fazer as estimativas, existem três abordagens a serem consideradas: (i) nós de tamanho fixo para toda a árvore; (ii) nós de tamanho fixo a cada nível; e (iii) nós de tamanho limitado, isto é, o número de elementos que podem ser armazenados nesses nós é sempre o mesmo. A seguir, essas três abordagens são detalhadas.

- *Fixed-size bulk-loading*: essa proposta considera que os nós são de tamanho fixo. Ela pode ser executada tanto de modo *top-down* quanto *bottom-up*. A execução *top-down* tem grandes chances de gerar uma árvore com muitos nós folhas em relação à abordagem *bottom-up*. No entanto, não pode ser garantido que a abordagem *bottom-up* resulte em árvores com bom desempenho no processamento das consultas. Enquanto a abordagem *top-down* constrói a árvore escolhendo os novos elementos para inserir em um nó que possui o representante mais próximo, na abordagem *bottom-up* não há meios de prever como escolher os elementos próximos para construir os níveis superiores.

- *Level-fixed-size bulk-loading*: essa proposta tem o objetivo de fixar o tamanho de cada nó dependendo do seu nível na árvore. Ela utiliza fórmulas para que seja feita a estimativa do número de nós por nível e do número de elementos por nó. Embora essa abordagem resulte em um bom uso do espaço e menor quantidade de nós, ela pode produzir uma estrutura com algum nível de sobreposição devido ao fato da sua construção ser no sentido *top-down*. Ademais, também é difícil explorar a proximidade e a distribuição dos elementos sobre o espaço métrico. Isto implica que os nós que armazenam elementos em regiões densas seguem o mesmo processo de construção daqueles que armazenam elementos em regiões esparsas.

- *Bounded-size bulk-loading*: essa proposta, ao contrário das anteriores, leva em consideração a distribuição dos dados no espaço, armazenando mais elementos nos nós da árvore que cobrem regiões mais densas e menos elementos em regiões esparsas. A cada nó, é verificado um limite para definir quantos elementos são necessários para manter a árvore balanceada. O objetivo dessa abordagem é avaliar a similaridade de cada elemento à sua amostra mais próxima para definir o número de elementos por nó, forçando que um número de elementos mínimo seja mantido nesse nó.

Para que a técnica *Bounded-size bulk-loading* obtenha bom desempenho, é necessário que haja uma escolha adequada de elementos amostras visando-se obter melhores agrupamentos. No entanto, como essa restrição nem sempre é garantida, o limite mínimo de elementos por nó garante que nós cujas amostras possuam poucos elementos associados sejam eliminadas. O algoritmo define limites de ocupação mínima, *balancedMin*, e máxima, *balancedMax*, por nó para que a árvore mantenha a sua altura como sendo $H = \lceil log_{CM}^{N} \rceil$, no qual C_M é a capacidade máxima de elementos por nó e N representa a quantidade de elementos total.

Os parâmetros *balancedMin* e *balancedMax* são determinados dinamicamente para cada nó e indicam o mínimo e o máximo limite de elementos para cada nó. Para esses parâmetros, o algoritmo

de *bulk-loading* retorna verdadeiro se a quantidade de elementos no conjunto *S* irá construir uma árvore balanceada com a sua respectiva ocupação ou falso caso contrário. Assim, essas funções permitem controlar dinamicamente a quantidade máxima e mínima de elementos em cada nó, o que faz com que se obtenha uma árvore final balanceada.

O funcionamento do *Bounded-size bulk-loading* é ilustrada na Figura 3.6. Existem três fases definidas: amostragem, atribuição e refinamento. A fase de amostragem (Figura 3.6(a)) escolhe elementos para que sejam amostras e o algoritmo se inicie. A fase de atribuição usa a função *balancedmMax* e a distância entre cada elemento do conjunto de dados e as amostras para criar um conjunto associado a cada amostra (Figura 3.6(b)). A fase de refinamento aloca novamente os elementos que pertencem aos conjuntos que não estiveram de acordo com a função *balancedMin* (Figura 3.6(c)).

Em detalhes, o algoritmo inicia escolhendo elementos aleatoriamente no conjunto de dados para que sejam determinados como sendo representantes. Em seguida, o conjunto de dados é particionado, associando cada subconjunto a um representante. Para isso, cada elemento é atribuído à partição cujo representante é o seu elemento mais próximo, respeitando a ocupação máxima do nó para manter a árvore balanceada. Se por acaso o conjunto da partição não possui a quantidade mínima de elementos necessária, a partição é descartada e os seus elementos inseridos em outros conjuntos, mas sempre mantendo a árvore balanceada. Por fim, um novo nó é criado com esses elementos, um representante é escolhido para eles e o processo pode ser repetido, construindo a árvore recursivamente até que o valor nulo seja encontrado para o representante, o que indica que chegou ao nó raiz, e o processo é encerrado.

Um ponto negativo é que esse algoritmo tem queda de desempenho quando os valores limites de ocupação máxima e mínima de um nó são muito próximos. Isso faz com que o algoritmo realize muitas reorganizações para construir a estrutura, o que leva muito tempo de processamento. Além disso, a estrutura final exige grande sobreposição nos nós. O pior caso desse algoritmo ocorre quando os valores de ocupação máxima e mínima são iguais. Para evitar esse problema, também é proposta uma técnica que aumenta a altura da árvore e reduz a densidade dos elementos na estrutura. Isso é feito por meio da inserção de elementos no nó de tal modo que a quantidade, C_v, seja inferior à máxima ocupação do nó. Resultados experimentais apontaram que um valor adequado para C_v é dado pela seguinte fórmula: C_v = (C_M - C_m)/2, onde C_M é a máxima ocupação do nó e C_m é a ocupação mínima. Desse modo, o valor de C_v é utilizado também para estimar a altura da árvore resultante e também o número de elementos por nó.

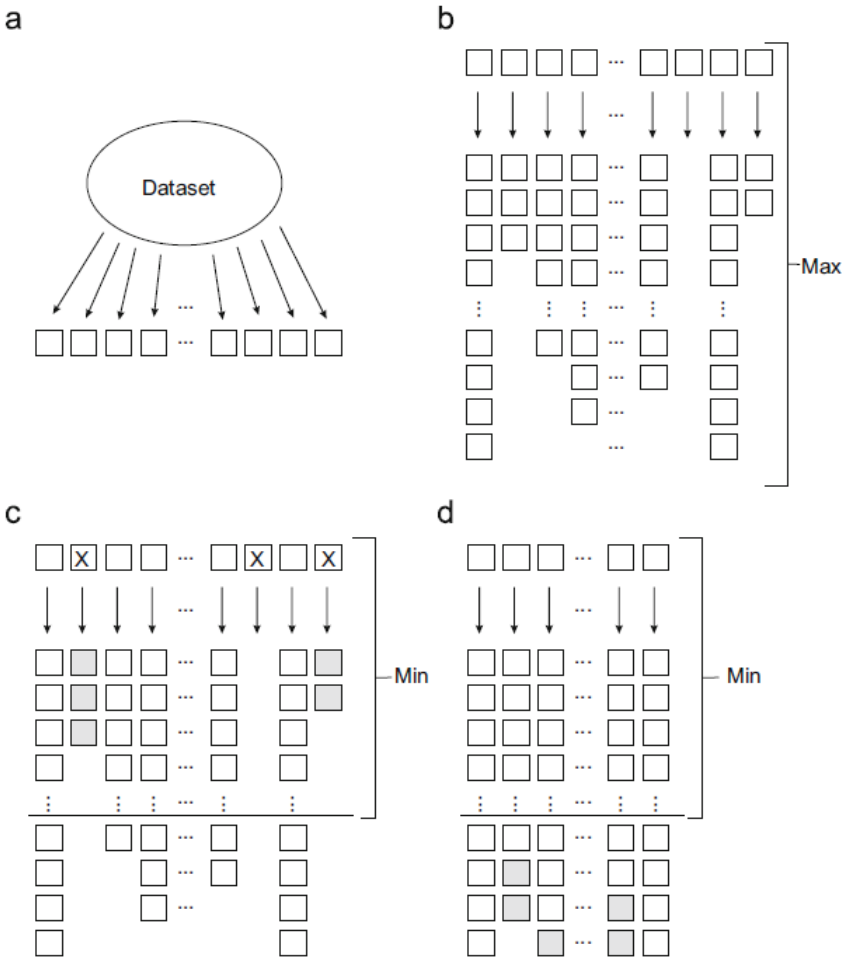

Figura 3.6: Estágios do algoritmo *Bounded-size bulk-loading*. Figura reproduzida de [Vespa et al., 2010]

3.4 Técnicas de Bulk-loading Genéricas

Esta seção descreve dois trabalhos que envolvem técnicas de *bulk-loading* que podem ser aplicadas a diversas estruturas de dados, tais como *B-tree* [Bayer and McCreight, 1972] e R-tree [Guttman, 1984]. Em ambos casos, considera-se que o índice não cabe inteiro em memória primária. Os trabalhos são detalhados nas seções 3.4.1 e 3.4.2.

3.4.1 An Evaluation of Generic Bulk-loading Techniques

Em [Bercken and Seeger, 2001], são introduzidos dois algoritmos de *bulk-loading* genéricos, que podem ser aplicados aos métodos de acesso baseados em árvore. O primeiro algoritmo é chamado *PathBased-BulkLoading*, e se aplica às estruturas conhecidas como *GP-trees (Grow & Post-tree)*. *GP-trees* são estruturas em forma de árvore balanceada e que crescem no sentido *bottom-up*, como exemplo *B-tree*. As estruturas às quais os algoritmos são dedicados devem apresentar as seguintes operações:

- *chooseSubtree*: dado um registro e um nó índice, determina a referência para a subárvore na qual a operação de inserção de um registro deve ser encaminhada.
- *grow*: dado um registro e um nó de dados, insere o registro nesse nó.

- *split post*: a partir de um nó com a sua capacidade máxima excedida, divide (*split*) o nó em dois e introduz (*post*) no nó pai a informação de que ocorreu uma operação de divisão.
- *search*: dado uma consulta e um nó do índice, retorna todos os registros armazenados naquele nó e que são relevantes para a consulta.

O algoritmo de *bulk-loading Path-Based bulk-loading* constrói o índice no sentido *top-down*, particionando os dados recursivamente até que a partição preencha a memória primária. O algoritmo de *bulk-loading* considera a situação em que não há memória primária suficiente para construir o índice totalmente antes de transferí-lo para disco.

Esse algoritmo funciona do seguinte modo. Na primeira fase, registros de uma amostra dos dados são inseridos em um índice mantido em memória primária até que o índice fique totalmente preenchido. Em seguida, a cada nó folha do índice criado é associado um *bucket* armazenado em disco. Os registros que não foram inseridos no índice até o momento são então inseridos no índice mas, ao chegarem aos nós folhas, são associados aos seus respectivos *buckets* em disco. A partir do momento em que todos os registros foram processados, os nós armazenados na memória primária são escritos em disco. Os pares constituídos de *buckets* não vazios e referências aos seus correspondentes nós folhas são então escritos em uma *to-do list* armazenada em disco. A segunda fase do algoritmo inicia-se com um par sendo removido da *to-do list* e o seu *bucket* correspondente sendo usado como fonte de registros de dados, que são processados recursivamente, como descrito na primeira etapa.

A Figura 3.7 ilustra o funcionamento do algoritmo, assumindo que a memória primária tem o tamanho de 4 páginas. O lado esquerdo da figura ilustra a situação em que o índice está construído na memória mas ainda existem registros a serem processados. O lado direito se refere à situação após o processamento do conjunto de dados inteiro.

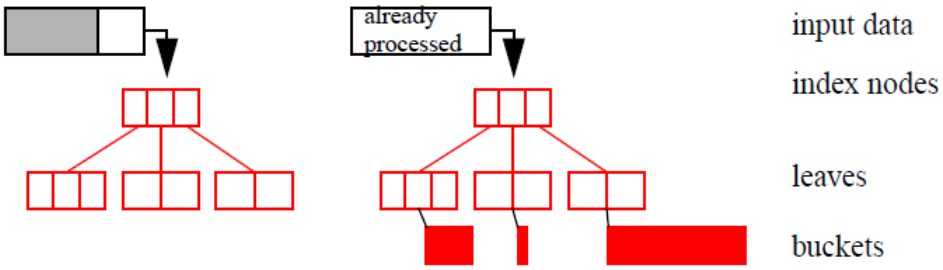

Figura 3.7: Exemplo de funcionamento do algoritmo *Path-Based bulk-loading* para m=4. Figura reproduzida de [Bercken and Seeger, 2001].

A segunda proposta de *bulk-loading* é o algoritmo chamado *QuickLoad*, o qual é destinado a uma classe de estruturas conhecidas como *OP-trees* (por exemplo, R-tree e M-tree), que consistem de um predicado P e um ponteiro para uma subárvore na qual cada um dos registros satisfazem P. Nesse tipo de estrutura os predicados dentro de um mesmo índice podem se sobrepor, isto é, um registro pode satisfazer a mais de um predicado.

O algoritmo *QuickLoad* tem a sua funcionalidade baseada no particionamento dos dados de entrada em um grande número de partições a partir de uma amostra dos dados. Desse modo, o algoritmo é aplicado recursivamente em cada uma das partições, mantendo uma *OP-tree* em memória para organizar

a amostra dos dados. O tamanho da amostra utilizada para particionar os dados pode ser tão grande quanto o tamanho da memória primária disponível. Assim como o algoritmo *PathBasedBulkLoading*, o algoritmo *QuickLoad* é voltado para quando não há memória primária suficiente para construir o índice totalmente em memória antes de transferí-lo para disco.

O algoritmo *QuickLoad* atua inserindo registros em sua estrutura armazenada em memória primária (*OP-tree*) até que a memória seja totalmente preenchida. Em seguida, *buckets* são associados aos nós folhas da estrutura e a inserção não é mais guiada até os nós folhas, mas aos *buckets* correspondentes. Quando todos os registros da entrada são processados, existem dois casos possíveis. Se um *bucket* está vazio, a referência para a sua folha correspondente é inserida em um arquivo em disco. Essas referências são utilizadas no próximo passo do algoritmo, para construir o nível superior da árvore. Por outro lado, se o *bucket* contém registros, um par consistindo de uma referência para o *bucket* e para o correspondente nó folha é inserido em uma *to-do-list*. O algoritmo é então aplicado recursivamente nos elementos da *to-do-list*. Quando a lista fica vazia, o arquivo em disco contém exatamente as referências para as folhas do índice final. Essas referências são fonte para o algoritmo criar o nível superior do índice. O algoritmo pára quando houver apenas uma referência restante, que está apontando para a raiz do índice final.

A Figura 3.8(a) ilustra o funcionamento do algoritmo, no qual uma *OP-tree* é construída a partir dos registros de entrada $R_1, ...R_{13}$. Nesse exemplo, assume-se que cabem três nós folhas em memória. Na parte esquerda, a memória foi completamente utilizada, apesar de alguns registros ainda não terem sido processados (R_9) a R_{13}. Os elementos S_1, S_2 e S_3 representam predicados que cobrem os registros nos seus nós folhas correspondentes. O lado direito da figura ilustra a situação após os elementos restantes serem inseridos em seus respectivos *buckets*. Assim, o nó folha que pertence a S_2 já representa o nó folha do índice alvo porque seu *bucket* está vazio, enquanto que o algoritmo deve ser executado novamente para os demais nós folhas, usando os *buckets* como entrada. A ilustração do processamento do nó folha e do seu correspondente *bucket*, indicado por S_1* e que está à direita da Figura 3.8(a), é ilustrado na Figura 3.8(b). O resultado da execução são duas páginas que irão ser parte do índice final.

3.4.2 A Generic Approach to Bulk-loading Multidimensional Index Structures

[Bercken et al., 1997] descrevem um algoritmo genérico para realizar o *bulk-loading* de estruturas multidimensionais. Exemplos de estruturas para as quais o algoritmo pode ser aplicado são *B+-tree* [Comer, 1979], R-tree [Guttman, 1984], *TV-trees* [Lin et al., 1994] e *LSD-trees* [Henrich et al., 1989]. O método descrito se baseia no uso das operações *split* e *merge* da estrutura de índice ao qual o *bulk-loading* irá ser aplicado, além de fazer uso de uma estrutura de dados chamada *buffer-tree*. No entanto, a estrutura *buffer-tree* utilizada é diferente da estrutura apresentada originalmente em dois pontos: (i) cada nó interno da *buffer-tree* contém um *buffer* adicional em que os registros são armazenados e (ii) inserções múltiplas são processadas simultaneamente na *buffer-tree*. Um nó nessa árvore adia o processo de inserção ao armazenar o registro a ser inserido na estrutura em seu *buffer*. No instante em que um certo limite de registros é excedido, o processo de inserção de todos os registros do *buffer* avança ao próximo nível da árvore.

A estrutura de índice que os autores consideram para o seu desenvolvimento é uma árvore. Além disso, a estrutura deve oferecer suporte para as seguintes operações:

- *InsertIntoNode*: insere um registro em um nó do índice, seja ele um nó folha ou nó interno;

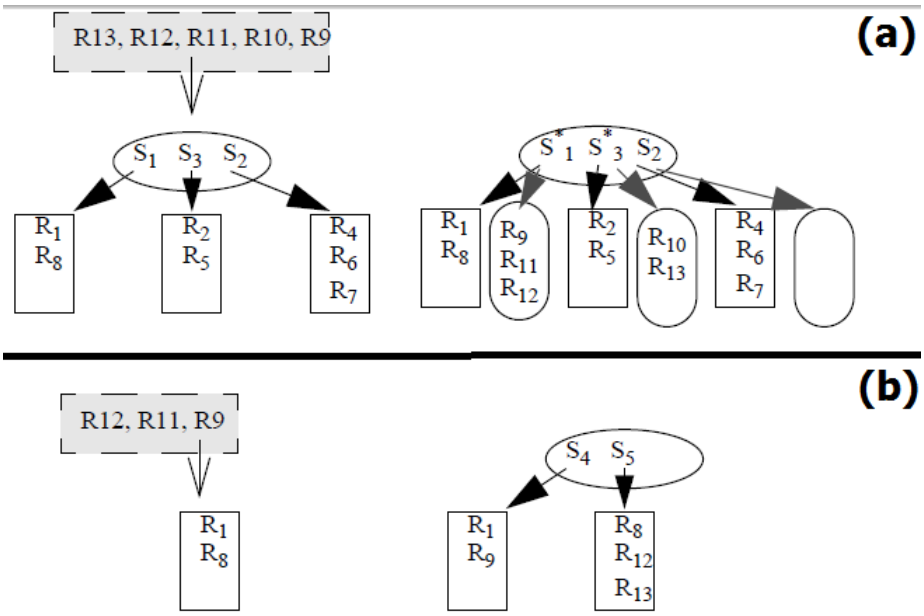

Figura 3.8: *Quick Load bulk-loading*: (a) Exemplo de funcionamento do algoritmo quando a maioria dos nós folhas cabem em memória primária. (b) Exemplo de funcionamento recursivo do algoritmo. Figura reproduzida de [Bercken and Seeger, 2001].

- *Split*: divide um nó com a capacidade excedida em dois nós.
- *ChooseSubtree*: para um registro de dados em um certo nó do índice, escolhe uma subárvore para inserir o registro.

A idéia do algoritmo consiste em inserir os elementos na *buffer-tree*. Nessa estrutura, uma operação de inserção atravessa a árvore da raiz às folhas, enquanto que as operações de reestruturação ocorrem das folhas para a raiz. Uma operação de reestruturação acontece quando um nó de dados da *buffer-tree* possui *overflow* em um nó interno. Assim, ocorre a divisão desse nó em dois e a inserção de uma nova entrada no nó pai. Esse processo pode gerar novamente *overflow* no nó pai, fazendo com que o processo de reestruturação seja necessário várias vezes até que chegue ao nó raiz da árvore.

O algoritmo de *bulk-loading* inicia o processo de inserir elementos na raiz da *buffer-tree*. Inicialmente, os registros vão sendo armazenados um-a-um no *buffer* do nó raiz até que ele seja totalmente preenchido, bloqueando o processo de inserção. O algoritmo considera três casos de *overflow*: quando ele ocorre no *buffer*, na página de dados ou nas entradas do nó que pertence ao índice. Quando ocorre *overflow* no *buffer*, uma mudança estrutural é necessária na *buffer-tree*, a qual consiste em uma operação para limpar o *buffer*, e faz com que os processos de inserção dos elementos que estavam bloqueados nele sejam reativados. Assim, cada elemento do *buffer* avança para o próximo nível da árvore. Se algum *buffer* dos nós dos níveis inferiores também estiver com *overflow*, a operação de limpá-los também será invocada. No entanto, se uma página de dados está com a sua capacidade máxima excedida, ela é então dividida em duas e as entradas correspondentes são armazenadas no nó pai. Do mesmo modo, um novo nó do índice é criado quando ocorre *overflow* em suas entradas, ou seja, também é realizada uma divisão do nó índice com capacidade excedida em dois nós e suas entradas são divididas.

Após inserir todos os elementos na *buffer-tree* e todos os processos de reestruturação da árvore, pode ser que alguns dos elementos ainda não foram inseridos no nó de dados, pois ainda estão bloqueados em algum nó do índice. Desse modo, um processo para limpar em profundidade todos os *buffers* não-vazios é realizado, iniciando da raiz, e atuando até que todas as entradas cheguem às páginas de dados. Finalmente, para gerar a estrutura final, todas as entradas geradas na primeira *buffer-tree*, e que estão em suas folhas, são inseridas em uma segunda *buffer-tree*, a qual representa a configuração final do método de acesso multidimensional após a operação do algoritmo de *bulk-loading* descrito.

A Figura 3.9 ilustra um conjunto de 25 retângulos $\{r_1, ..., r_{25}\}$ a serem inseridos em uma R-tree com o uso do algoritmo de *bulk-loading* descrito nessa seção. Nesse exemplo, considere que a capacidade da página de dados é três, que existem duas páginas no *buffer* e que cada nó armazena quatro elementos.

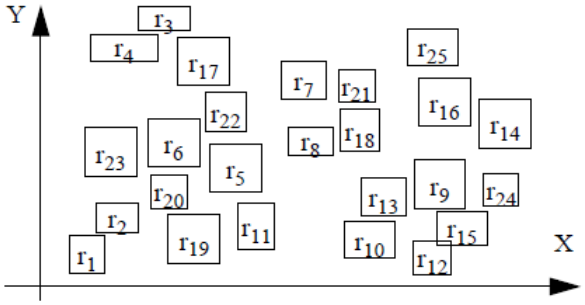

Figura 3.9: *Generic bulk-loading*: Amostra contendo 25 retângulos. Figura reproduzida de [Bercken et al., 1997]

A Figura 3.10 ilustra a *buffer-tree* após a inserção de 23 desses retângulos, na ordem dos seus índices, em uma árvore inicialmente vazia. A estrutura possui três nós, N_1, N_2 e N_3 e cinco páginas em disco, P_1 a P_5, além dos *buffers* associados a cada nó. Existem também doze processos de inserção nos nós folhas e, portanto, terminados. Os processos que ainda estão armazenados nos *buffers* estão ativos, mas bloqueados. Após a inserção do retângulo r_{24}, cada nó armazena quatro elementos. O resultado desse algoritmo é ilustrado na Figura 3.11, que mostra a configuração da árvore após a ativação de todos os processos de inserção que estavam bloqueados no nó raiz e avançaram para o próximo nível da árvore.

Quando algum *buffer* armazenar mais do que duas páginas, como o nó N3 na Figura 3.11, o *overflow* deve ser eliminado, limpando os *buffers* um a um. Desse modo, os elementos que o *buffer* contém também avançam para o próximo nível. A Figura 3.12 exibe o nó N3 sendo dividido em dois nós no instante em que sua capacidade máxima é excedida devido aos processos de inserção vindos do nível superior. Após todos os processos de inserção serem ativados, pode ser que alguns deles não cheguem aos nós folhas e, portanto, são ativados dos seus respectivos *buffers* em profundidade. Como resultado, é gerada a estrutura final, a qual é ilustrada na Figura 3.13

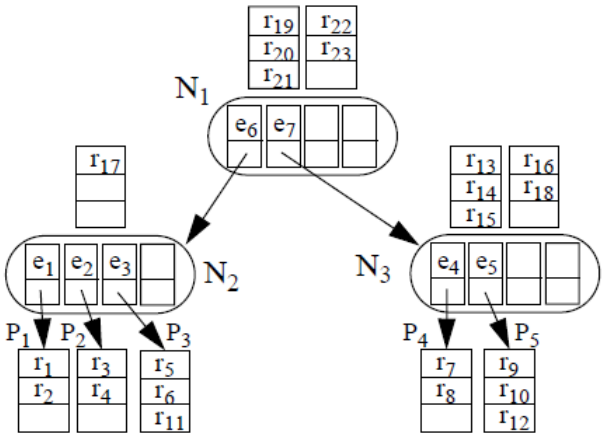

Figura 3.10: *Generic bulk-loading*: Exemplo de árvore após 23 inserções. Figura reproduzida de [Bercken et al., 1997]

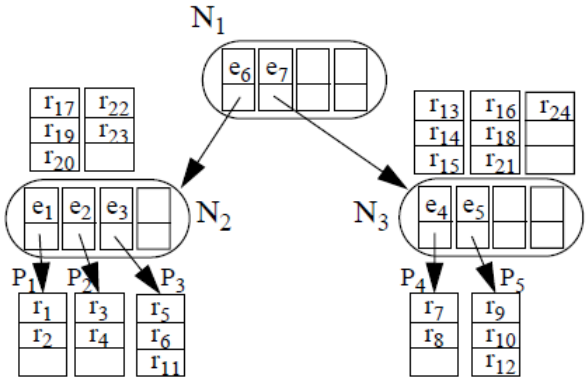

Figura 3.11: *Generic bulk-loading*: Exemplo de árvore após limpar o *buffer* do nó raiz. Figura reproduzida de [Bercken et al., 1997]

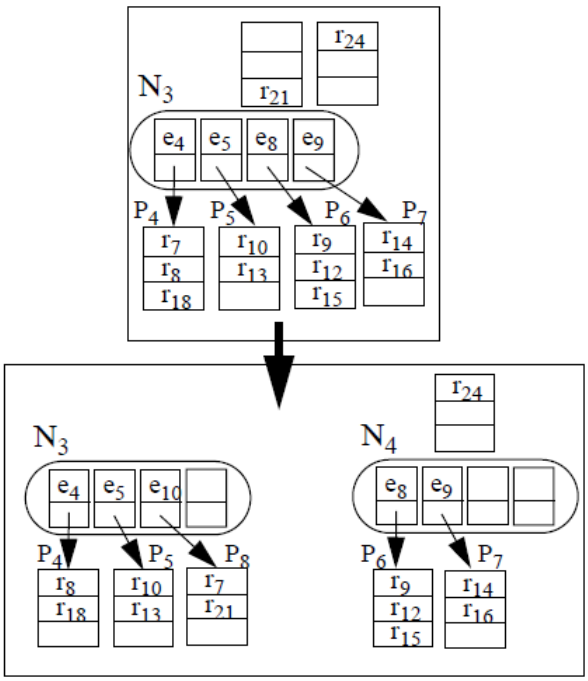

Figura 3.12: *Generic bulk-loading*: Exemplo de divisão do nó N3. Figura reproduzida de [Bercken et al., 1997]

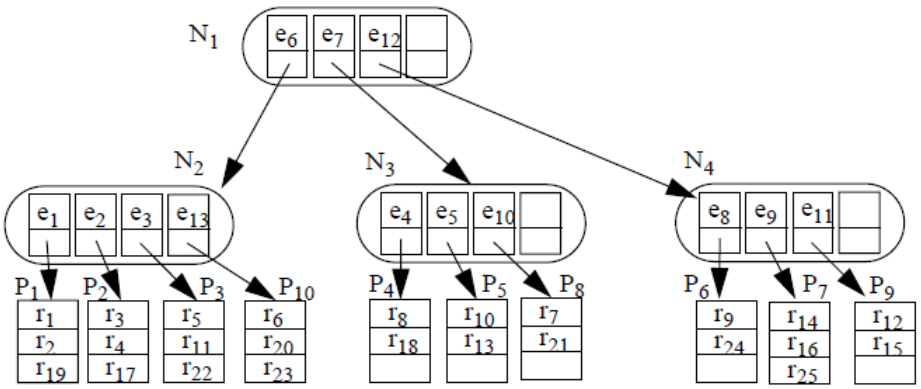

Figura 3.13: *Generic bulk-loading*: Exemplo da árvore após terminar os processos de inserção. Figura reproduzida de [Bercken et al., 1997]

3.5 Considerações Finais

Neste capítulo foram descritos trabalhos relacionados ao presente projeto de mestrado, os quais abordam propostas para a operação de *bulk-loading* para os métodos de acesso R-tree, M-tree e Slim-tree. Também foram resumidas duas propostas de *bulk-loading* genéricas. A Tabela 3.1 ilustra uma síntese de todos os trabalhos relacionados descritos neste capítulo.

Tabela 3.1: Descrição dos algoritmos Bulk Loading descritos no capítulo

Nome	Método de Acesso	Construção da Estrutura	Princípio	Característica Explorada
TGS	R-tree	Top-Down	Particionamento recursivo do conjunto de dados	Aproximações (MBRs)
OMT	R-tree	Top-Down	Determinar a topologia da estrutura com antecedência	Aproximações (MBRs)
Efficient Bulk On Dynamic R-trees	R-tree	Bottom-Up	Uso de variação da *buffer-tree*	*Buffers* associados aos nós da R-tree
Bulk Loading da M-tree	M-tree	Bottom-Up	Agrupamentos de objetos	Distância dos elementos ao representante
Bulk Loading da Slim-tree	Slim-tree	Top-Down	Uso de amostras como representantes e estimativas da estrutura com antecedência	Distância dos elementos ao representante
An Evaluation of Generic Bulk-Loading Techniques	GP-trees e OP-trees	Top-Down	Particionamento recursivo dos dados e associar *buckets* aos nós folhas do índice	Parte do índice é mantida em memória primária
A Generic Approach to bulk-loading Multidimensional Index Structures	B+tree, R-tree, TV-trees, LSD-trees	Top-Down	Uso da *buffer-tree*	Usa operações do próprio método de acesso, como exemplo a função *split*

Em se tratando das propostas descritas, elas não podem ser diretamente aplicadas à Onion-tree. As propostas de *bulk-loading* para a R-tree fazem uso das características próprias desse método de acesso, como o uso de MBRs para aproximar os dados espaciais armazenados, ordenação dos MBRs de acordo com o critério de uma dimensão e tentativa de reduzir a sobreposição existente entre os MBRs. Os algoritmos de *bulk-loading* propostos para os MAM M-tree e Slim-tree são voltados para a memória secundária, explorando as características desse tipo de MAM, como a sobreposicão dos nós. Além disso, o algoritmo de *bulk-loading* para a M-tree possui a deficiência de não garantir que o índice seja criado sempre. Do mesmo modo, as propostas de *bulk-loading* genéricas detalhadas neste capítulo são voltadas para estruturas especiais de métodos de acesso, como GP-trees e OP-trees, e exploram as características

dessas estruturas, bem como são baseados na premissa de que a quantidade de dados a ser inserida no índice não cabe completamente em memória primária.

Portanto, para que a operação de *bulk-loading* seja direcionada à Onion-tree, essa operação deve ser voltada à memória primária e atender às características desse MAM, como a forma na qual a Onion-tree particiona o espaço métrico. Ainda não existem na literatura algoritmos de *bulk-loading* para a Onion-tree que atendam a essas necessidades, o que consiste no objetivo deste projeto de mestrado. Os resultados obtidos para atingir esse objetivo são descritos no próximo capítulo.

Capítulo 4

Bulk-loading na Onion-tree

Vários trabalhos têm surgido na literatura propondo técnicas para a operação de *bulk-loading* para métodos de acesso. Dentro do contexto desta dissertação de mestrado, técnicas que tratam da operação de *bulk-loading* para MAMs são as mais relevantes, as quais foram descritas no capítulo 3. Diferentes fundamentos têm sido usados para o desenvolvimento dessas técnicas, com destaque para as diferentes formas de realizar a operação de *bulk-loading*: (i) *bulk-loading* baseado em ordenação; (ii) *bulk-loading* baseado em *buffer*; (iii) *bulk-loading* baseado em amostragem. A técnica baseada em ordenação trabalha ordenando os dados e construindo a estrutura de indexação no sentido *bottom-up*. A técnica baseada em *buffer* faz o uso da estrutura de dados *buffer-tree* para armazenar um subconjunto dos dados em memória primária a fim de construir a estrutura de indexação por partes. Por fim, a técnica baseada em amostragem escolhe uma amostra dos dados para ser armazenada em memória primária e constrói toda a estrutura baseando-se em estimativas. Como em um espaço métrico não há relação de ordem total dos dados, as técnicas de *bulk-loading* tradicionais baseadas em ordenação não podem ser diretamente aplicadas para MAMs. Além disso, vale destacar que, em oposição à construção da estrutura de indexação no sentido *bottom-up*, ou seja, dos nós folhas para o nó raiz, também existe a construção da estrutura de indexação no sentido *top-down*, ou seja, do nó raiz para os nós folhas.

A Onion-tree é o MAM baseado em memória primária mais eficiente disponível na literatura. Entretanto, ela possui algumas limitações, dentre as quais se destaca o fato dela apenas oferecer um algoritmo voltado à inserção dos dados um-a-um em sua estrutura. Ou seja, a Onion-tree não oferece uma operação de *bulk-loading* que construa o índice considerando todos os elementos do conjunto de dados de uma única vez. A operação de *bulk-loading* consiste na carga em massa de elementos e pode ser útil nos casos de reconstrução do índice ou inserção de uma grande quantidade de elementos simultaneamente, visando-se prover melhor desempenho no processamento das consultas.

Neste capítulo são apresentadas as características e o funcionamento dos algoritmos de *bulk-loading* desenvolvidos para o MAM Onion-tree, que são as principais contribuições desta dissertação. Os algoritmos de *bulk-loading* propostos organizam os elementos a serem inseridos na estrutura de indexação antecipadamente, de forma a definir qual a melhor ordem de inserção desses elementos. Outra característica dos algoritmos é que eles enfocam na construção do índice no sentido *top-down*,

e aplicam diferentes formas de realização da operação, como abordagem gulosa, abordagem baseada em amostragem e abordagem baseada na altura da estrutura final. Por fim, os três algoritmos propostos são voltados às duas versões da Onion-tree, ou seja, à F-Onion-tree, a qual considera que todos os nós da estrutura de indexação possuem o mesmo número de expansões, e à V-Onion-tree, a qual considera que cada nó da estrutura de indexação pode ter um número diferente de expansões.

Os três algoritmos propostos, bem como as seções nas quais eles são descritos são:

- Algoritmo *GreedyBL*, o qual utiliza a estratégia gulosa para realizar o *bulk-loading* na Onion-tree. Descrito na seção 4.1.
- Algoritmo *SampleBL*, o qual utiliza a estratégia de amostragem para realizar o *bulk-loading* na Onion-tree. Descrito na seção 4.2.
- Algoritmo *HeightBL*, o qual utiliza a estratégia de estimativa da altura da estrutura final para realizar o *bulk-loading* na Onion-tree. Descrito na seção 4.3.

O capítulo é encerrado na seção 4.4 com as considerações finais.

4.1 Algoritmo de Bulk-Loading Guloso

O primeiro algoritmo de *bulk-loading* desenvolvido para a Onion-tree e que serve como base para o desenvolvimento deste trabalho utiliza a estratégia gulosa. O algoritmo proposto, chamado *GreedyBL*, realiza a construção da estrutura no sentido *top-down*, verificando qual o melhor par de pivôs a ser escolhido para cada nó da estrutura. O algoritmo utiliza como premissa a construção de uma estrutura balanceada e, para isso, introduz uma função objetivo que deve ser minimizada. A seção 4.1.1 detalha os princípios nos quais o algoritmo *GreedyBL* é baseado. A seção 4.1.2 detalha o algoritmo. A seção 4.1.3 ilustra um exemplo de execução do algoritmo.

4.1.1 Considerações iniciais

A premissa utilizada como base para o desenvolvimento do algoritmo *GreedyBL* leva em consideração o fato de que a Onion-tree possui uma estrutura altamente dependente da ordem de inserção dos dados. Desse modo, uma estrutura altamente desbalanceada pode ser gerada dependendo da ordem de inserção dos dados, fazendo com que a Onion-tree tenha o seu desempenho degenerado no processamento de consultas. O objetivo do algoritmo é construir a estrutura do índice de tal forma que a árvore fique totalmente balanceada ou aproximadamente balanceada. Com isso, espera-se que a estrutura resultante garanta melhor desempenho no processamento das consultas quando comparada à estrutura gerada pela inserção dos dados um-a-um.

O algoritmo é guloso, desde que testa todas as possibilidades de pares de elementos para determinar quais deles distribuem mais uniformemente os dados entre as regiões da Onion-tree. No entanto, existe a possibilidade de nenhum par de pivôs distribuir os elementos de forma completamente uniforme entre as regiões e, portanto, o par de elementos escolhido é aquele que mais se aproxima dessa distribuição. A quantidade de elementos esperada por região é dada pela Equação 4.1. Para que a escolha do par de pivôs seja avaliada, a Equação 4.2 é utilizada para verificar o quão distante a distribuição dos elementos entre as regiões da Onion-tree está da quantidade de elementos esperada por região. Nessas equações, o

termo *QtdEl* representa a quantidade de elementos na região *r*, o termo *QtdReg* representa o número de regiões definidas para a Onion-tree em um dado nó e o termo *QtdElNivel* define o número de elementos a serem inseridos em um dado nível. O objetivo do algoritmo é encontrar o par de pivôs que minimiza a Equação 4.2.

$$ValorEsperado = \lceil (QtdElNivel)/(QtdReg) \rceil \tag{4.1}$$

$$Objetivo = \Sigma_{r=0}^{r=n} \sqrt{(QtdEl[r] - ValorEsperado)^2}/QtdReg \tag{4.2}$$

A determinação dos melhores pivôs por região é relizada da seguinte forma. Para cada par de elementos de uma dada região, é analisado se a quantidade de elementos que serão alocados em cada uma das regiões da *Onion tree* é uniforme ou aproximadamente uniforme, isto é, se cada região irá possuir aproximadamente a mesma quantidade de elementos. Isso é feito por meio da minimização da Equação 4.2. Esse processo é feito inicialmente para o nó raiz e, assim que seus pivôs são determinados, ele é repetido recursivamente para cada uma das regiões geradas na estrutura. O mesmo processo se repete para cada nível gerado na árvore até que cada uma das regiões possua apenas dois ou menos elementos, que são os elementos que compõem os nós folhas.

4.1.2 Algoritmo GreedyBL

O algoritmo proposto *GreedyBL* (Algoritmo 1) funciona como descrito a seguir. Ele recebe como entrada um vetor contendo a lista de elementos que serão utilizados para construir a Onion-tree, o número da região à qual esses elementos pertencem, o número de expansões que a Onion-tree possui e também uma referência ao nó pai do nó atual. Para determinar o par de pivôs que melhor divide os elementos entre as regiões, inicialmente o valor de *minObjetivo* é inicializado como infinito (linha 1) e todos os pares de elementos são analisados (linhas 2 a 15).

Após a escolha dos pares de pivôs determinados pelos índices *i* e *j* (linha 4), verifica-se a condição de que a estrutura a ser construída é uma V-Onion ou uma F-Onion. Se a entrada do algoritmo determinar que *NumeroExpansoes* é maior ou igual a zero, então será construída uma F-Onion-tree. Caso contrário, será construída uma V-Onion-tree. A identificação da construção de uma V-Onion-tree é feita quando o valor do parâmetro *NumeroExpansões* é igual a -1. Nesse caso, a quantidade de expansões a ser criada para o nó representado pelos pivôs identificados pelos índices *i* e *j* é determinada pelo Algoritmo 2 (linhas 5 a 7).

O próximo passo é verificar a distribuição dos elementos por região da Onion-tree e avaliar essa distribuição usando a Equação 4.2 (linhas 8 e 9). Se o valor da Equação 4.2 obtido for o mínimo até o momento, então ele é armazenado, assim como também os pivôs e a distribuição dos elementos entre as regiões (linhas 10 a 13).

Para criar o nó atual, o par de pivôs que minimizou a Equação 4.2 é selecionado (linha 16) e se o nó pai fornecido como parâmetro é vazio, então o nó raiz da Onion-tree é criado contendo os pivôs selecionados (linhas 17 a 19). Caso contrário, um nó filho é inserido na região determinada pelo nó pai contendo os pivôs selecionados (linhas 20 a 22). Para o restante dos elementos distribuídos entre as regiões, analisa-se o tamanho dos conjuntos de elementos de cada região *r* (linhas 23 a 31). Se o

tamanho de algum desses conjuntos for maior do que 2, o algoritmo é chamado recursivamente contendo os elementos associados à região *r* como dados de entrada (linhas 25 a 27). Caso contrário, os elementos contidos nessa região compõem um nó folha a ser armazenado na estrutura (linhas 28 a 30).

O Algoritmo 2 determina o método de particionamento de um nó de uma V-Onion-tree durante a execução do algoritmo *GreedyBL*. Seu funcionamento é semelhante ao método de particionar um nó utilizado pela Onion-tree. Seus parâmetros de entrada são os dois pivôs do nó e também uma referência ao nó pai. Se metade do raio do nó pai é maior do que o raio definido pela distância dos dois pivôs, então haverá expansões determinadas para esse nó (linha 1). O número de expansões é calculado a partir da divisão entre o raio do nó pai e o raio determinado pelos dois pivôs (linha 2). Se a condição não é verificada, não são aplicadas expansões ao nó (linha 5).

Algorithm 1: `GreedyBL` (*Elementos*, *NumeroExpansoes*, *NoPai*)

Input : *Elementos* {elementos do conjuto de dados},
NumeroExpansoes {número de expansões da Onion-tree},
NoPai {referência ao nó pai do nó atual},
Output: *Onion − tree* {Onion-tree depois do bulk-loading}

```
1  minObjetivo ← MAXINT;
2  for (i = 0 ; i < tamanho(Elementos); i++) do
3      for (j = 0 ; j < tamanho(Elementos); j++) do
4          pivos.pivo1 ← selecionarPivos(Elementos, i);
5          pivos.pivo2 ← selecionarPivos(Elementos, j);
6          if NumeroExpansoes = -1 then
7              expansoes = VOnionExpansions(pivos[i], pivos[j],NoPai);
8          end
9          elementoPorRegiao ← verificarRegioes(pivos, Elementos - pivos, expansoes);
10         valorObjetivo ← funcaoObjetivo(elementoPorRegiao);
11         if valorObjetivo < minObjetivo then
12             minObjetivo ← valorObjetivo;
13             armazena(pivos, elementoPorRegiao[][])
14         end
15     end
16 end
17 melhoresPivos ← pivosArmazenados();
18 if NoPai = null then
19     noAtual ← inserirRaiz(melhoresPivos);
20 end
21 else
22     noAtual ← inserirNo(melhoresPivos, NoPai);
23 end
24 for (r = 0 ; r < numeroDeRegioes(expansoes) ; r++) do
25     entrada[] = elementosRegiao(r);
26     if tamanho(entrada) > 2 then
27         GreedyBulkLoading(entrada[r], r, NumeroExpansoes, noAtual);
28     end
29     else
30         inserirNo(entrada, r);
31     end
32 end
```

Algorithm 2: `VOnionExpansions` (*pivoS*1, *pivoS*2,*NoPai*)

Input : *pivoS*1 {primeiro pivô do nó}, *pivoS*2 {segundo pivô do nó}, *NoPai* {referência ao nó pai do nó atual}, **Output**: *NumeroExpansoes* {número de expansões aplicado ao nó}

```
1 if distancia(pivoS1, pivoS2) <= raio(NoPai)/2 then
2 |   NumeroExpansoes ← raio(NoPai)/distancia(pivoS1, pivoS2);
3 end
4 else
5 |   NumeroExpansoes ← 0;
6 end
7 return NumeroExpansoes;
```

O algoritmo *GreedyBL* é considerado guloso porque avalia cada par de elementos do conjunto de dados de acordo com a Equação 4.2, escolhendo o par que minimiza o valor dessa equação. No entanto, a escolha de todos os pares possíveis de pivôs possui complexidade elevada, $O(n^2)$. Ademais, a checagem de qual região os demais elementos devem ser associados resulta em uma complexidade O(n). Portanto, o algoritmo *GreedyBL* possui complexidade $O(n^3)$. Como resultado, o algoritmo possui grande queda de desempenho no momento da criação do índice à medida em que o número de elementos a ser inserido na Onion-tree aumenta, o que não é adequado para um MAM baseado em memória.

4.1.3 Exemplo de Execução

Nesta seção é ilustrado um exemplo de execução do Algoritmo 1. Para facilitar a visualização dessa execução, ele é mostrada usando como base uma F-Onion-tree sem expansões. Uma F-Onion-tree sem expansões divide o espaço métrico em apenas 4 regiões, representadas por I, II, III e IV.

Considere o conjunto de elementos $S = s_1, ..., s_{22}$ ilustrado na Figura 4.1(a). A chamada inicial para o algoritmo consiste na invocação da função *GreedyBL(Elementos,NumeroExpansoes,NoPai)* com os seguintes parâmetros, na sequência: (i) conjunto S com 22 elementos; (ii) número de expansões igual a 0; (iii) nó pai com valor nulo.

O primeiro passo consiste no teste de todos os pares de elementos sendo verificados como pivôs. Suponha que o primeiro par de pivôs a ser testado é composto pelos elementos s_1 e s_2, ilustrados na Figura 4.1(b). Para a escolha desse par, ocorre a seguinte distribuição dos elementos entre as quatro regiões definidas pela Onion-tree: quatro elementos na região I, cinco elementos na região II, quatro elementos na região III e sete elementos na região IV. Em seguida, é realizado o teste para avaliar essa distribuição de elementos entre as regiões. Como existem 22 elementos no conjunto de dados, então o resultado da Equação 4.1 é $\lceil 22/4 \rceil = 6$. Aplicando a Equação 4.2, tem-se como resultado para o par s_1 e s_2 o valor 1,5. Como esse valor é o primeiro a ser testado, ele é então armazenado como o melhor valor obtido até o momento. Considere agora o processamento do par de pivôs s_5 e s_8, ilustrado na Figura 4.1(c). Esse par resulta na seguinte distribuição dos elementos nas regiões: três elementos na região I, quatro elementos na região II, sete elementos na região III e seis elementos na região IV. Aplicando a Equação 4.2 nesse par de pivôs, o valor 1,75 é obtido. Portanto, a escolha do par de pivôs s_1 e s_2 é mais adequada por distribuir melhor (i.e. mais uniformemente entre as regiões) os dados entre as regiões em relação à escolha dos pivôs s_5 e s_8, visto que o resultado da Equação 4.2 da escolha de pivôs foi menor. Nesse caso, nenhuma alteração é feita ao melhor valor da Equação 4.2 obtida até o momento. A análise da distribuição dos elementos entre as regiões de acordo com a escolha do par de pivôs é ilustrada na

Figura 4.1(d). O processo de escolha de pivôs é repetido para todos os pares de pivôs possíveis, até que aqueles que distribuam melhor os dados sobre cada uma das regiões sejam encontrados.

Por se tratar da primeira chamada à função de *bulk-loading*, o nó raiz da estrutura é criado contendo os elementos s_1 e s_2 como pares de pivôs. Na sequência, o algoritmo realiza chamadas recursivas para cada uma das quatro regiões, usando como base os elementos associados a elas, exceto os pares de pivôs já escolhidos. Ou seja, o algoritmo *GreedyBL(Elementos,NumeroExpansoes,NoPai)* é invocado para cada uma das regiões com os seguintes parâmetros: (i) para a região I são fornecidos quatro elementos de entrada, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (ii) para a região II são fornecidos cinco elementos de entrada, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (iii) para a região III são fornecidos quatro elementos de entrada, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (iv) para a região IV são fornecidos sete elementos de entrada, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado.

O mesmo processo de escolha de pivôs ilustrado anteriormente é repetido recursivamente para gerar a estrutura completa. Note que, no entanto, os nós gerados pelos pivôs escolhidos nos próximos passos são nós internos e que a recursão ocorre enquanto existirem mais de dois elementos associados por região.

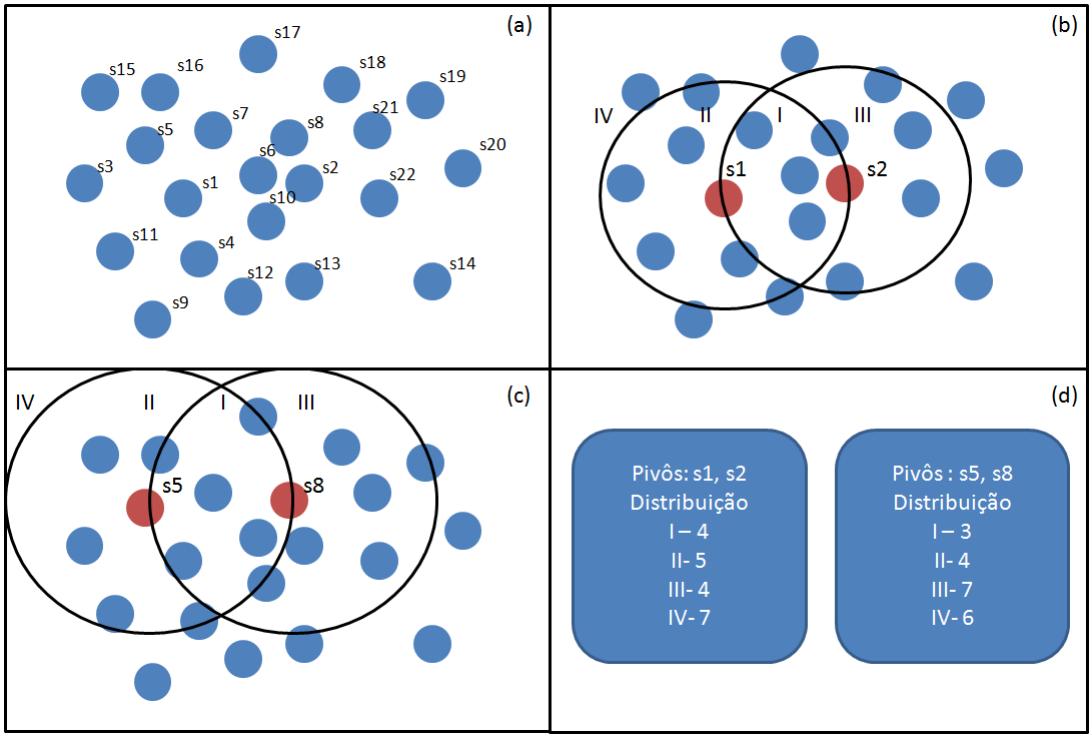

Figura 4.1: Exemplo de funcionamento do algoritmo de *bulk-loading* guloso. (a) Conjunto inicial de dados. (b) Distribuição do conjunto de dados para o par de pivôs s1 e s2. (c) Distribuição do conjunto de dados para o par de pivôs s5 e s8. (d) Exemplo da lista de distribuição dos dados entre as regiões para cada par de pivôs escolhido.

4.2 Algoritmo de Bulk-Loading Baseado em Amostragem

O custo cúbico do algoritmo *GreedyBL* motivou o desenvolvimento do algoritmo *SampleBL*, o qual introduz dois grandes diferenciais com relação ao algoritmo *GreedyBL*. O primeiro deles refere-se à etapa de amostragem, a qual é caracterizada por gerar amostras do conjunto de dados para serem testadas como possíveis pares de pivôs. O segundo diferencial refere-se à etapa de particionamento, a qual é direcionada especificamente à V-Onion-tree e é caracterizada por realizar o particionamento de cada nó dessa estrutura levando em consideração todos os elementos que podem ser nela inseridos.

A Figura 4.2(a) ilustra as etapas do algoritmo *GreedyBL*, enquanto que a Figura 4.2(b) contextualiza o algoritmo *SampleBL* em relação a essas etapas. No retângulo pontilhado, é mostrado que, ao invés de se realizar a seleção de todos os pares de pivôs, o algoritmo *SampleBL* seleciona apenas os pares de pivôs escolhidos na amostra. Além disso, é mostrado também que o algoritmo *SampleBL* inclui um teste adicional de particionamento para a V-Onion-tree.

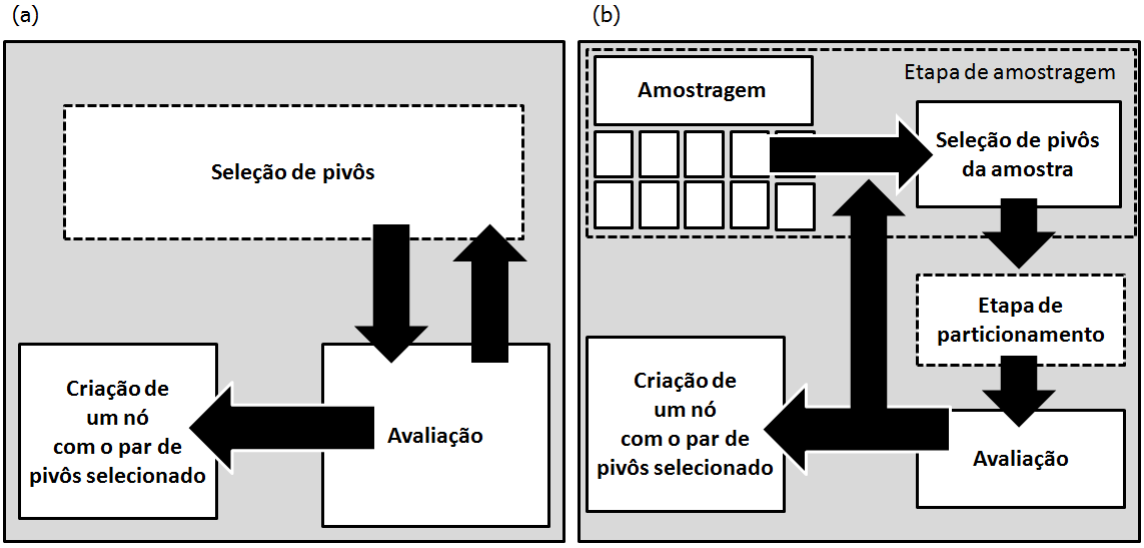

Figura 4.2: Figura 4.4 (a) Etapas do algoritmo GreedyBL. (b) Etapas do algoritmo SampleBL. Os retângulos tracejados mostram a diferença entre os algoritmos, ou seja, aonde o uso de amostras de dados (i.e., etapa de amostragem) e método de particionamento são usados pelo algoritmo SampleBL.

Para facilitar o entendimento do algoritmo *SampleBL*, esse algoritmo é apresentado da seguinte forma. Primeiramente, são descritas as características da etapa de amostragem. A seção 4.2.1 descreve os princípios básicos nos quais o algoritmo *SampleBL* é baseado. A seção 4.2.2 detalha o algoritmo *Sampling*, núcleo do algoritmo *SampleBL*. A seção 4.2.3 introduz várias melhorias que podem ser aplicadas ao algoritmo *Sampling* para aumentar o seu desempenho. A seção 4.2.4 introduz alternativas ao particionamento da V-Onion-tree. Por fim, a seção 4.2.5 incorpora todas essas melhorias ao algoritmo *Sampling*, detalhando a versão completa do algoritmo *SampleBL* proposto.

4.2.1 Princípios básicos da etapa de amostragem

O algoritmo *SampleBL* é um algoritmo de *bulk-loading* baseado em amostragem, que possui como núcleo o algoritmo *Sampling*, responsável pela etapa de amostragem. O princípio básico usado para a seleção das amostras é o seguinte. Existem partes do conjunto de dados que, quando analisadas, são automaticamente descartadas, desde que não irão selecionar pares de pivôs que garantirão resultados mínimos para a Equação 4.2. Existem duas situações nas quais isso ocorre. A primeira delas é caracterizada quando são avaliados pivôs de porções extremas do conjunto de dados, resultando na maioria dos elementos associados à região I da Onion-tree. A Figura 4.3(a) ilustra essa situação, na qual os pares de pivôs s_{20} e s_{12} dividem o espaço métrico do seguinte modo: dezesseis elementos na região I, dois elementos na região II, quatro elementos na região III e nenhum elemento na região IV. A segunda situação ocorre quando os pivôs avaliados estão muito próximos, resultando na maioria dos elementos associados à região IV (ou exterior) da Onion-tree. A Figura 4.3(b) ilustra a situação em que os pares de pivôs s_4 e s_5 dividem o espaço métrico do seguinte modo: dois elementos na região I, nenhum elemento na região II, nenhum elemento na região III e vinte elemento na região IV. Nessas duas situações, a estrutura gerada será desbalanceada, quando considerada a região que possui a maior quantidade de elementos.

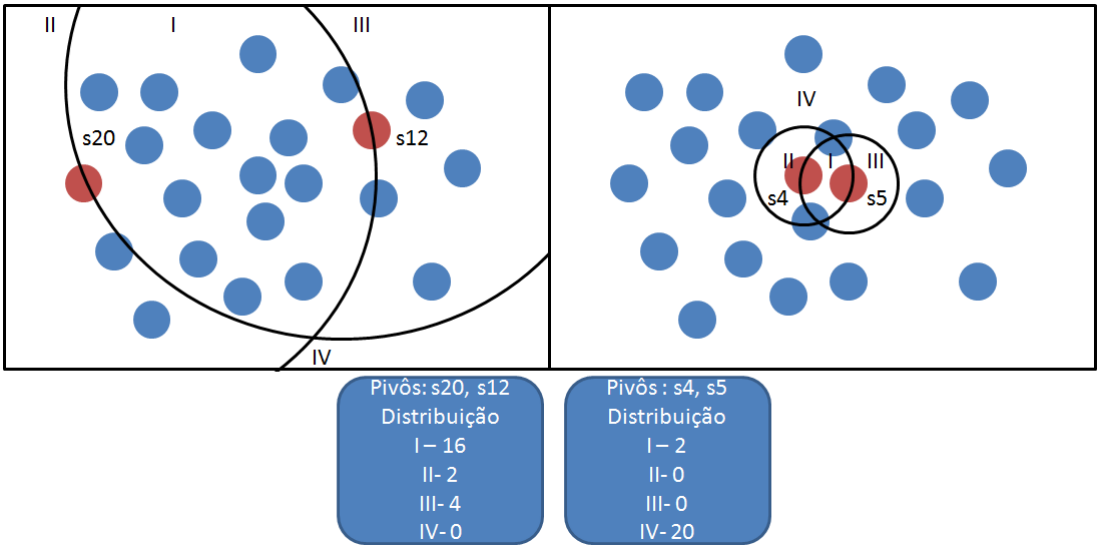

Figura 4.3: Exemplo de escolha de pivôs que não minimizam a Equação 4.2. (a) Escolha de pivôs muito distantes. (b) Escolha de pivôs muito próximos.

A etapa de amostragem tem o objetivo de cortar porções do conjunto de dados para evitar comparações desnecessárias de pivôs. Pares de pivôs muito distantes do conjunto de dados são descartados pelo algoritmo, assim como pares de pivôs muito próximos. Intuitivamente, o algoritmo funciona do seguinte modo. Inicialmente, o elemento medóide do conjunto de dados é encontrado, como mostrado na Figura 4.4(a). Isso é feito a partir da busca pelo elemento cuja soma das distâncias a todos os outros elementos do conjunto de dados é a menor possível. Na sequência, são consideradas as distâncias do medóide a cada um dos demais elementos. Essas distâncias são ordenadas e, então, determina-se o valor da mediana das distâncias, o qual é usado como o raio para realizar uma busca por amostras do conjunto de dados

(Figura 4.3)(b). A próxima etapa consiste de um iterativo aumento na parte superior e de uma iterativa redução na parte inferior do raio calculado até que seja formado um anel ao redor dos elementos, os quais são usados como amostras (Figura 4.4(c)). Como a quantidade de amostras deve ser muito pequena em relação ao conjunto de elementos de entrada, menos elementos são usados para comparação e, portanto, espera-se que o tempo de construção da Onion-tree por meio do algoritmo *SampleBL* tenha um ganho considerável em comparação ao tempo de construção do algoritmo *GreedyBL*.

O aumento do raio seque a Equação 4.3. Em conjuntos de dados com alta dimensionalidade e grande quantidade de elementos, existe alta probabilidade de encontrar elementos próximos, mesmo a uma pequena distância. Assim, a Equação 4.3 leva em consideração a quantidade de elementos de um conjunto de dados e a sua dimensionalidade para, ao assumir valores entre 0 e 1, selecionar elementos do conjunto de dados. Nessa equação, o termo *QtdElementos* se refere à quantidade de elementos do conjunto de dados e *Dimensionalidade* à sua dimensionalidade.

$$Incremento = QtdElementos/(Dimensionalidade + QtdElementos) \tag{4.3}$$

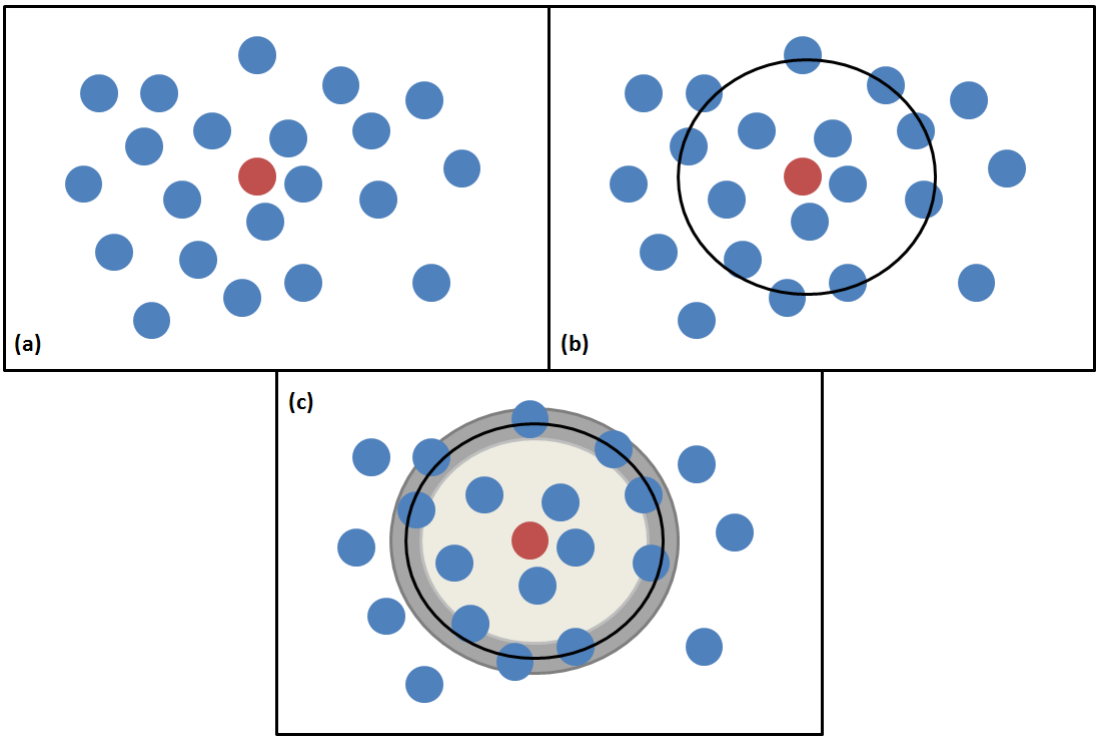

Figura 4.4: Exemplo do algoritmo de amostragem. (a) Escolhendo o medóide. (b) Calculando a mediana. (c) Determinando o anel de busca por pares de pivôs.

4.2.2 Algoritmo para a etapa de amostragem

Esta seção descreve o algoritmo de amostragem *Sampling*, núcleo do algoritmo *SampleBL*. O algoritmo *Sampling* (Algoritmo 3), responsável pela etapa de amostragem, funciona como descrito a seguir. Ele recebe como entrada os elementos do conjunto de dados, o tamanho da amostra a ser gerada e o valor

que o raio é alterado a cada iteração para que o anel de busca seja gerado. Primeiramente, o algoritmo determina o medóide do conjunto de dados, verificando dentre todos os elementos do conjunto de entrada, qual é aquele que produz o menor valor da soma das distâncias aos demais elementos (linhas 1 a 9).

Na sequência, o algoritmo calcula a distância mediana do centróide aos demais elementos. Essas distâncias são calculadas, armazenadas no de distâncias chamado *radius* e ordenadas (linha 10). A distância mediana é determinada pelo valor central do vetor de distâncias se o número de elementos do vetor for ímpar ou pela média da soma dos dois elementos centrais se o número de elementos do vetor for par.

Por fim, a cria-se o conjunto de amostras (linhas 14 a 23). Isso é feito por meio de iterações em que um anel é criado ao redor da distância mediana, ao incrementar a parte superior do anel e decrementar a parte inferior. Enquanto o número de amostras é inferior ao esperado, as porções inferior da cobertuda do anel tem o seu valor reduzido e a porção superior da cobertuda do anel tem o seu valor aumentado para encontrar mais amostras na próxima iteração.

Algorithm 3: `Sampling` (*Elementos*, *TamanhoAmostra*, *IncrementoRaio*)

Input : *Elementos* {elementos do conjunto de dados}, *TamanhoAmostra* {tamanho da amostra gerada}, *IncrementoRaio* {tamanho do incremento do raio para gerar o anel}

Output: *Amostra* {amostras para serem testados como pivôs}

```
1  distanciaMin ← MAXINT;
2  posicaoMedoide ← -1;
3  for (i = 0 ; i < tamanho(Elementos) ; i++) do
4      distancia ← distanceCalculations(Elements[i]);
5      if distancia < distanciaMin then
6          distanciaMin ← distancia;
7          posicaoMedoide ← i;
8      end
9  end
10 radius ← getMedian(Elementos, posicaoMedoide);
11 contador ← 0;
12 maxRaio ← raio;
13 minRaio ← raio;
14 while contador < TamanhoAmostra do
15     for (i = 0 ; i < sizeOf(Elementos) ; i++) do
16         if distancia(Elementos,posicaoMedoide,i) <= maxRaio OR
           distancia(Elementos,posicaoMedoide,i) >= minRaio then
17             adicionarAmostra(amostra[], Elementos, i );
18             contador++;
19         end
20     end
21     maxRaio ← maxRaio + IncrementoRaio;
22     minRaio ← minRaio - IncrementoRaio;
23 end
24 return sample[]
```

4.2.3 Melhorando o desempenho da etapa de amostragem

Esta seção introduz duas melhorias ao algoritmo *Sampling*, visando melhorar o seu desempenho. Todas essas melhorias referem-se à parte do algoritmo voltado à seleção de pivôs. São enfocadas as seguintes melhorias: filtragem de elementos irrelevantes, descrita na seção 4.2.3.1, e método aproximado de escolher o elemento medóide, detalhada na seção 4.2.3.2.

4.2.3.1 Filtrando elementos similares

Ao final da fase de amostragem, os elementos do conjunto retornado podem conter mais do que um elemento que se fossem reduzidos a apenas um, não causariam diferença significativa no resultado final. Tais elementos são denominados nesta dissertação de elementos similares. Eles são caracterizados por

possuírem entre si valor de distância menor do que δ (ilustrado na Figura 4.5). Esse valor é determinado pela Equação 4.3.

Os elementos similares, quando testados como possíveis pares de pivôs, possuem as seguintes características:

- Quando os elementos similares são comparados com o restante dos elementos do conjunto de dados como possíveis pares de pivôs, a troca entre elementos similares não causa grande mudança na estrutura resultante, pois a divisão dos elementos pelas regiões definidas no espaço métrico pouco se altera. A Figura 4.6(a) ilustra a troca do par de pivos s_{10}, s_4 pelo par de pivos s_{10}, s_5, ou seja, o elemento s_4 é trocado pelo elemento similar s_5. Nota-se que a distribuição do número de elementos no espaço métrico é muito próxima.

- Devido à sua grande proximidade, os elementos similares, quando testados como pares de pivôs em potencial entre si, resultam em má divisão do espaço métrico, concentrando a maioria dos demais elementos na região externa. Por exemplo, a escolha dos elementos similares s_4 e s_5 como pares de pivôs na Figura 4.6(b) gera uma distribuição do espaço métrico na qual a região IV contém a grande maioria dos elementos, o que gerará uma estrutura muito desbalanceada.

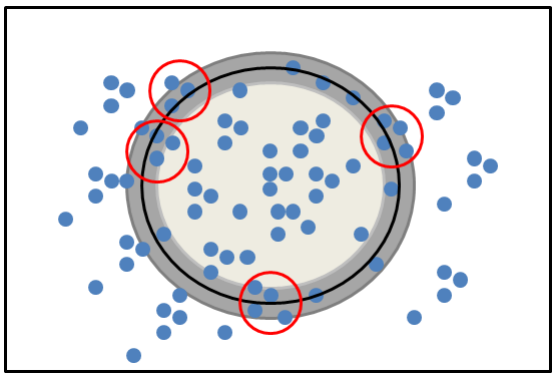

Figura 4.5: Exemplo de elementos similares

Como os elementos similares introduzem aproximadamente a mesma contribuição como pivôs quando testados, eles podem ser reduzidos a somente um membro do conjunto, que pode ser escolhido aleatoriamente. Desse modo, uma melhoria que pode ser aplicada ao Algoritmo 3 refere-se ao fato de que, imediatamente depois da seleção de amostras, é interessante que se inicie uma fase de filtragem dessas amostras. A etapa de filtragem corresponde à remoção dos *n-1* elementos de cada conjunto de elementos similares e uso do elemento restante como representante do grupo de elementos similares. Essa etapa reduz ainda mais o número de amostras que irão ser checadas pelo algoritmo *SampleBL*, o que tende a aumentar ainda mais o seu desempenho.

O Algoritmo 4, referente à fase de filtragem das amostras, funciona do seguinte modo. Para cada par *i*, *j* de elementos selecionados como amostras (linhas 1 e 2), verifica-se primeiramente se o elemento *i* já não foi removido em alguma iteração anterior do algoritmo e, em caso afirmativo o algoritmo continua sua execução para o próximo elemento (linhas 3 a 5). Caso o elemento não tenha sido removido, é

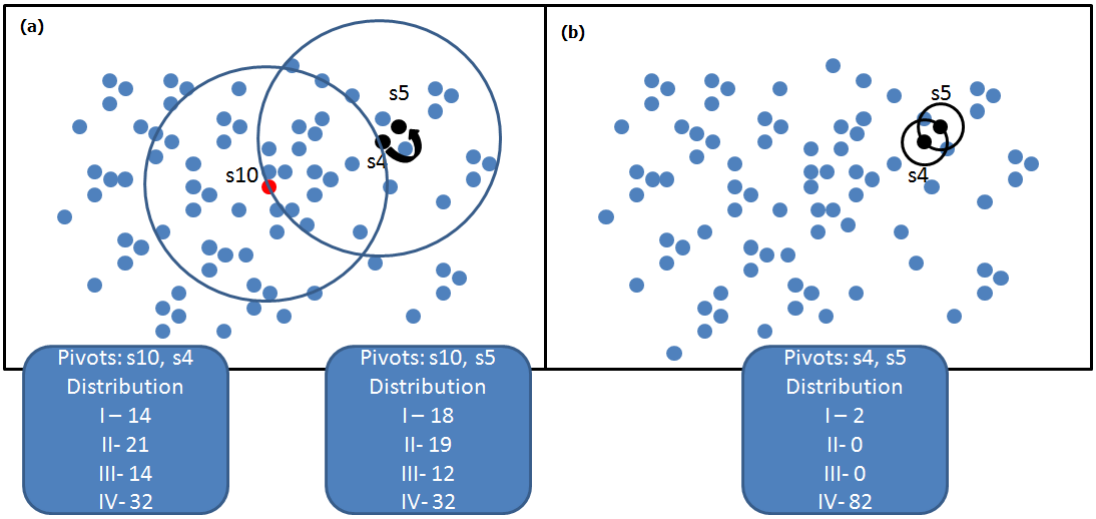

Figura 4.6: Exemplo da distribuição dos dados no espaço métrico. (a) Distribuição considerando os pares de pivôs s10,s4 e s10, s5. (b) Distribuição considerando o par de pivôs s4 e s5.

realizado o teste de fato, verificando se o elemento *j* está dentro de um raio determinado como δ (delta) que tem como centro o elemento *i*. Nesse caso, o elemento *j* é removido do conjunto das amostras a serem testadas pelo algoritmo (linhas 6 a 8).

Algorithm 4: `Filtering` (*Elementos*, *Delta*)

Input : *Elementos* {elementos a serem filtrados},
Delta {raio que ira considerar os elementos similares}
Output: *AmostraFiltrada* {amostras que retornaram do filtro}

```
1  for (i = 0 ; i < sizeOf(Elementos) ; i++) do
2      for (j = 0 ; j < sizeOf(Elementos) ; j++) do
3          if amostraRemovida(Elementos,i) then
4              continue;
5          end
6          if distancia(Elementos,i,j) <= Delta then
7              removerAmostra(Elementos, j );
8          end
9      end
10 end
11 return AmostraFiltrada;
```

4.2.3.2 Escolhendo um Medóide Aproximado

O método para selecionar o medóide aproximado do conjunto de dados descrito nas linhas 1 a 9 do algoritmo 3 possui complexidade quadrática. Isso pode resultar em um tempo de execução alto quando o algoritmo executar sobre bases de dados com muitos elementos e alta dimensionalidade. Em especial, esse algoritmo realiza mais processamento para encontrar o medóide dos primeiros níveis da Onion-tree, nos quais a quantidade de elementos a serem analisadas é maior do que nos níveis inferiores. Portanto, outra melhoria desenvolvida nesta dissertação consiste em encontrar o medóide aproximado do conjunto de dados, a qual possui complexidade n*log(n).

Intuitivamente, a estratégia usada para determinar o medóide aproximado funciona do seguinte modo. Inicialmente, um elemento *F* é escolhido aleatoriamente no conjunto de dados. Ademais, escolhem-se dois outros elementos, chamados nesta dissertação de elementos extremos. O primeiro deles, denominado *F1*, é determinado como sendo o elemento mais distante de *F*. Já o segundo, denominado *F2*, é determinado como sendo o elemento mais distante de *F1*. Um exemplo de escolha dos elementos é ilustrada na Figura 4.7(a). Note que essa escolha possui complexidade O(n).

Na sequência, são calculadas as distâncias de cada elemento extremo a cada um dos demais elementos do conjunto de dados. As distâncias medianas, denominadas *M1* e *M2*, para cada elemento extremo são determinadas pela ordenação das distâncias calculadas anteriormente. Essa ordenação possui complexidade n*log(n).

Finalmente, a geração do medóide aproximado é iniciada em um processo iterativo no qual verifica-se a região de intersecção determinada pelas bolas dos pontos extremos *F1* e *F2*, cujos raios são as medianas *M1* e *M2*. Podem ocorrer duas situações. Na primeira delas, a região de interseção contém elementos, os quais são considerados como candidatos a medóides aproximados. Na segunda situação, a região de interseção não tem elementos, ou seja, é vazia. Nesse caso, os raios são incrementados de um valor delta até que existam candidatos. O valor de delta é definido pela Equação 4.3. Em ambas as situações, o elemento escolhido como medóide aproximado é aquele que tem a menor soma das distâncias aos

outros elementos do conjunto de dados. Desde que seja retornado um conjunto pequeno de amostras, esse processamento pode ser feito em tempo considerado constante.

O Algoritmo 5 detalha a escolha do medóide aproximado. Seus parâmetros de entrada são: os elementos do conjunto de dados, o tamanho da amostra a ser gerada e o valor *delta* que define o incremento do raio definido pelas medianas. Nas linhas 1 e 2, um elemento aleatório é selecionado do conjunto de dados. Na linha 3, um elemento *F1* é determinado como sendo o elemento mais distante ao selecionado anteriormente e, na linha 4, um elemento *F2* é determinando como sendo o mais ditante ao elemento *F1*. Nas linhas 5 e 6, são calculadas as medianas das distâncias de *F1* e *F2*, respectivamente, aos demais elementos do conjunto de dados. Na sequência, inicia-se um processo que irá se encerrar quando o número de elementos selecionados atingir a quantidade de elementos de amostragem indicada (linhas 8 a 17). Se o elemento testado estiver interno à intersecção determinada pelas mediadas calculadas, ele será adicionado ao conjunto resposta como possível candidato a medóide (linhas 9 a 12). Se os elementos foram processados e ainda não chegaram à quantidade de candidatos necessária, então os valores das medianas são incrementados de um fator definido como δ (delta) (linhas 15 e 16). Os candidatos selecionados deverão ser, no algoritmo de *bulk-loading* principal (*SampleBL*), testados para verificar qual deles possui a menor soma das distâncias aos demais elementos do conjunto de dados. O algoritmo é ilustrado na Figura 4.7(b).

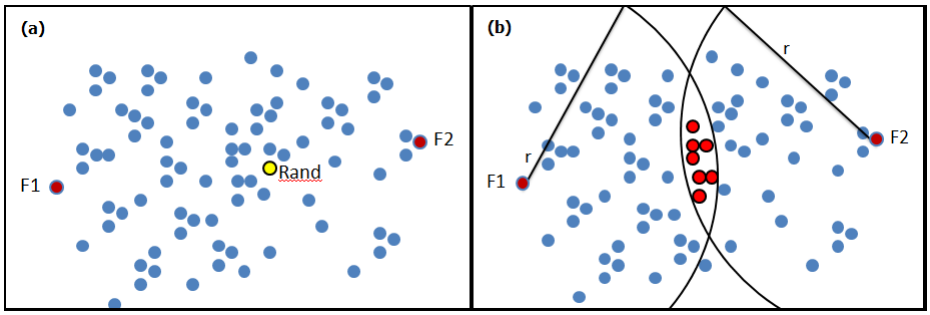

Figura 4.7: Passos para determinar o medóide aproximado. (a) Escolha dos elementos extremos. (b) Candidados para serem testados como medóides aproximados.

Algorithm 5: `Aproximated Medoid` (*Elementos*)

Input : *Elementos* {elementos do conjunto de dados}, *TamanhoAmostra* {quantidade de elementos que serão testados como medóides}, *Delta* {incremento no calculo da mediana}

Output: *Candidatos* {elementos candidatos a medóide}

```
1  size ← tamanho(Elementos);
2  rand ← random(0,size);
3  f1 ← elementoDistante(Elementos, rand);
4  f2 ← elementoDistante(Elementos, f1);
5  m1 ← calculaMediana(f1,Elementos);
6  m2 ← calculaMediana(f2,Elementos);
7  contador ← 0;
8  while contador < TamanhoAmostra do
9      for (i = 0 ; i < sizeOf(Elementos) ; i++) do
10         if distancia(f1,Elementos,i) <= m1 AND distancia(f2,Elementos,i) <= m2 then
11             adicionar(Elementos,i);
12             contador++;
13         end
14     end
15     m1 += Delta;
16     m2 += Delta;
17 end
18 return Candidatos;
```

4.2.4 Método de Particionamento da V-Onion-tree

As melhorias descritas anteriormente aplicam-se a ambas as versões da Onion-tree, ou seja, à V-Onion-tree e à F-Onion-tree. Entretanto, a V-Onion-tree introduzida em [Carélo et al., 2009] possui uma limitação relacionada ao seu modo de particionamento que possibilita melhorias diretamente relacionadas a essa versão sejam feitas no algoritmo de *bulk-loading*. A seguir é descrita a limitação da V-Onion-tree (seção 4.2.4.1), e introduzidas as melhorias propostas (seção 4.2.4.2).

4.2.4.1 Limitação do particionamento da V-Onion-tree

O modo de particionamento da V-Onion-tree utiliza como método para determinar a quantidade do número de partições de um nó a razão entre o raio do nó pai e o raio do nó filho. No entanto, existem situações em que a quantidade de expansões é elevada, inclusive gerando um número de expansões tal que a quantidade de elementos que podem ser armazenados nesse único nó é maior do que a quantidade de elementos da árvore toda.

Suponha que exista um nó pai cujo raio seja 1 e que exista um nó filho cujo raio tenha valor 0,001. Em se tratando de uma V-Onion-tree, existe a necessidade de expansão desse nó, visto que o critério para realizar uma expansão na V-Onion-tree é analisar que o raio do nó filho é menor que a metade do raio do nó pai. A quantidade de expansões a ser aplicada é a divisão do raio do nó pai pelo raio do nó filho, o que resulta em 1000 expansões, e equivale a 3004 regiões a serem definidas para esse nó (o número de regiões é determinado pela fórmula 3xNúmero de Expansões + 4). Um nó com 3004 regiões armazena, portanto, 6008 elementos. Por exemplo, suponha que está sendo criada uma Onion-tree para o conjunto de dados real Brazilian Cities, usado nos experimentos do capítulo 5, o qual possui 5507 elementos. No pior caso, o conjunto todo de dados poderia ser armazenado em um único nó e ainda haveriam regiões não ocupadas.

A situação descrita anteriormente ilustra como o procedimento de expansão da V-Onion-tree, introduzido em [Carélo et al., 2009], pode gerar quantidades de expansões desnecessárias em relação à quantidade de dados a ser inserida. No pior caso, ou seja, caso um único nó armazene todos os elementos a serem inseridos, a estrutura gerada seria semelhante à uma lista ligada, fazendo com que qualquer consulta seja respondida no pior caso com complexidade O(n).

4.2.4.2 Melhorando o Método de Particionamento da V-Onion-tree

O algoritmo de *bulk-loading* para a V-Onion-tree possui duas vantagens em relação ao processo de expansão: conhecer com antecedência a quantidade e os elementos que serão inseridos na estrutura. Desse modo, são introduzidas nesta dissertação duas melhorias baseadas nessas vantagens: expansão baseada em quantidade e expansão baseada em média. O objetivo da expansão baseada em quantidade é manter um limite superior de expansões que cada nó pode apresentar. O limite é baseado na quantidade de elementos a serem inseridos naquele nível da árvore. Se em um dado nível da estrutura são inseridos n elementos, então o número de regiões que devem existir é $NumeroRegioes = n/2$. Desse modo, o número de expansões máximo, denominado *ExpMax*, que esse nó pode ter é dado pela Equação 4.4.

$$ExpMax = (NumeroRegioes - 4)/3 \tag{4.4}$$

Outro fator a ser considerado no procedimento de expansão é que nem todos os elementos que são dados de entrada a um nó irão de fato ser armazenados nele. Muitos desses elementos serão novamente utilizados nas demais chamadas recursivas do algoritmo de *bulk-loading*. Desse modo, outra melhoria desenvolvida neste trabalho consiste em expandir a V-Onion-tree com o método denominado expansão baseada em média. O método funciona como explicado a seguir. Para o par de pivôs selecionado para o nó atual, calcula-se a distância média dos pivôs s_1 e s_2 aos demais elementos, denominadas como *m1* e *m2*, respectivamente. Como esse processo é acompanhado da seleção de pares de pivôs de um dado nó, as

distâncias necessárias para o processo já foram calculadas e o processo não demanda cálculos de distância adicionais. O próximo passo é calcular o valor médio das distâncias, dado por *(m1+m2)/2*. Finalmente, o número de expansões é determinado como sendo tal que o raio da maior bola determinada pelos pivôs s_1 e s_2 consiga ser maior do que o valor dado pelo valor médio das distâncias. Com isso, espera-se que o número de expansões seja o suficiente para cobrir parte dos elementos que foram recursivamente destinados a esse nó, além de também respeitar a quantidade máxima de expansões determinada por *ExpMax*.

Enquanto a expansão baseada em quantidade determina a quantidade máxima possível de expansões que um nó pode possuir, a expansão baseada em média assume que a grande parte dos elementos destinados ao nó é recursivamente encaminhada aos nós dos níveis inferiores. Assim, ela normalmente diminui o número de expansões calculadas pelo algoritmo original da V-Onion-tree, que não considera a quantidade de dados a ser inserida. A expansão baseada em média normalmente gera menos expansões do que a expansão baseada em quantidade.

Nas melhorias propostas nesta dissertação, as expansões baseada em quantidade e baseada em média são combinadas, de forma que a expansão baseada em média atue como limite inferior para o número de expansões (definido no algoritmo como a variável *nroMaxRegioes*), enquanto que a expansão baseada em quantidade atue como limite superior para esse valor (definido no algoritmo como a variável *nroMinRegioes*).

O Algoritmo 6 detalha o algoritmo de particionamento da V-Onion-tree, que pode ser utilizado por qualquer algoritmo de *bulk-loading* proposto nesta dissertação de mestrado. O algoritmo possui como parâmetros de entrada os dois representantes do nó, o raio do nó pai, raio do nó atual e também a forma de construção a ser aplicada para a V-Onion-tree: tradicional ou baseada em expansões. Enquanto a construção tradicional contrói a V-Onion-tree sem as melhorias propostas nesta dissertação, a construção baseada em expansões contrói a V-Onion-tree usando as expansões baseada em quantidade e em média.

Inicialmente, há o cálculo da quantidade máxima de expansões que um nó pode possuir com base na quantidade de elementos que a ele é associada (linhas 1 a 3), ou seja, é feito o uso da expansão baseada em quantidade. Se a forma de construção da V-Onion-tree é a tradicional, o número de expansões calculado é dado pela razão entre o raio do nó pai e o raio do nó filho (linhas 4 a 6). Se a forma de construção é a baseada em expansões, então as distâncias médias de cada um dos elementos do nó aos demais elementos são calculadas (linhas 8 a 10). Finalmente, um procedimento iterativo se inicia incrementando o número de expansões até que o raio máximo da bola gerada seja maior do que o valor da média da distâncias determinado anteriormente. Isso é feito de forma que as expansões nunca tenham um valor maior do que o número máximo de expansões atribuído ao nó (linhas 12 a 15).

Algorithm 6: `Paticioning Method` ($Elementos, S1, S2, RaioPai, RaioNo, TipoExpansao$)

Input : $Elementos$ {elementos do conjunto de dados},
$S1$ {representante um do nó},
$S2$ {representante dois do nó},
$RaioPai$ {valor do raio do no pai},
$RaioNo$ {valor do raio do no atual} ,
$TipoExpansao$ {determina qual o modo de expansao da V-Onion: tradicional ou baseado em quantidade e media combinados}

Output: $Expansoes$ {numero de expansoes calculado para o no}

```
1  size ← tamanho(Elementos);
2  nroMaxRegioes ← size/2;
3  nroMaxExpansoes ← (nroMaxRegioes - 4)/3;
4  if TipoExpansao = tradicional then
5  |   Expansoes ← RaioPai/RaioNo;
6  end
7  else if TipoExpansao = baseadoEmExpansoes then
8  |   m1 ← mediaDistancias(S1,Elementos);
9  |   m2 ← mediaDistancias(S2,Elementos);
10 |   med ← (m1 + m2)/2;
11 |   Expansoes ← 0;
12 |   while raioTotal(Expansoes,RaioNo) < med AND Expansoes < nroMaxExpansoes do
13 |   |   Expansoes++;
14 |   end
15 |   return Expansoes;
16 end
```

4.2.5 Algoritmo SampleBL

O Algoritmo 7 detalha o algoritmo *SampleBL*, o qual engloba as etapas de amostragem (Algoritmo 3), filtro (Algoritmo 4), medóide aproximado (Algoritmo 5) e também o novo método de particionamento da V-Onion-tree (Algoritmo 6). Além disso quando comparado com o algoritmo *GreedyBL*, o algoritmo *SampleBL* inclui outra modificação a fim de aumentar o seu desempenho no instante da construção da Onion-tree. O primeiro laço do algoritmo *GreedyBL* (linha 2 do Algoritmo 1) foi eliminado, assumindo que os dados a serem verificados são resultantes da etapa de amostragem ao invés do conjunto inteiro de dados. Para isso, um elemento que é retornado da etapa de amostragem pode ser selecionado aleatoriamente do conjunto de amostras com complexidade O(1). Desse modo, o algoritmo reduz a sua complexidade em O(n), onde *n* é o tamanho da amostra a ser coletada. Isso resulta em ganho de desempenho significativo durante o processamento do algoritmo para a construção da Onion-tree.

O Algoritmo 7 recebe como entrada: os elementos a serem inseridos na Onion-tree; o número de expansões, o qual define se uma F-Onion-tree ou uma V-Onion-tree será criada e; a referência ao nó pai do nó atual. Primeiramente, o algoritmo determina o conjunto de elementos que são testados como medóides do conjunto de dados (linha 1). Para a escolha do medóide propriamente dito, é analisado qual o elemento a partir do conjunto gerado que minimiza a soma das distâncias aos demais elementos do

conjunto de entrada (linha 2). Em seguida, são geradas e filtradas as amostras a serem testadas como pares de pivôs (linhas 3 e 4). Na sequência, o algoritmo analisa qual par de pivôs da amostra coletada divide melhor os elementos do conjunto de dados, da seguinte forma. O primeiro pivô é determinado aleatoriamente do conjunto da amostra dos dados (linha 5). Nas linhas 6 a 19, os demais elementos da amostra são testados em conjunto com o primeiro pivô selecionado, até que o melhor par de pivôs seja identificado. Durante a escolha dos pivôs, ocorre a chamada ao método de particionamento, que pode considerar também as expansões baseadas em média e em quantidade desenvolvidas nesta dissertação, além do método de particionamento tradicional da V-Onion-tree.

O par de pivôs escolhidos (linha 20) forma a raiz da árvore (linhas 21 a 23) ou é inserido em um nó interno ou folha da estrutura (linhas 24 a 26), dependendo se o nó pai é, respectivamente, nulo ou não nulo. Finalmente, para cada conjunto de dados associado a uma dada região *r* determinada pelo par de pivôs escolhido, é feita a verificação da sua quantidade de elementos (linha 28). Se a quantidade é superior a dois elementos, o algoritmo *SampleBL* é invocado recursivamente para a região *r* utilizando o respectivo conjunto de dados como entrada (linhas 29 a 31), além do nó atual como nó pai e o número de expansões definido. Caso contrário, os elementos são inseridos em um nó folha da estrutura (linhas 32 a 34).

Algorithm 7: `SampleBL` (*Elementos*,*NumeroExpansoes*,*NoPai*)

Input : *Elementos* {elementos do conjunto de dados},
NumeroExpansoes {número de expansões da Onion-tree},
NoPai {referência ao nó pai do nó atual}

Output: *Onion − tree* {estrutura Onion-tree final}

```
1  medoid ← approximatedMedoid(Elementos,sizeMedoid);
2  bestMedoid ← minimizeMedoid(medoid,Elementos);
3  sample ← sampling(Elementos,sizeSample);
4  filteredSample ← filtering(sample);
5  pivo1 ← random(filteredSample);
6  for (i = 0 ; i < tamanho(filteredSample); i++) do
7      pivo2 ← selecionarPivos(filteredSample[], i);
8      if NumeroExpansoes = -1 then
9          NumeroExpansoes = PatitioningMethod(pivos[i], pivos[j],NoPai, tipoExpansao);
10     end
11     pivos.pivo1 ← pivo1;
12     pivos.pivo2 ← pivo2;
13     elementosRegiao ← verificarRegioes(pivos, Elementos - pivos, NumeroExpansoes);
14     valorObjetivo = funcaoObjetivo(elementoPorRegiao);
15     if valorObjetivo < minObjetivo then
16         minObjetivo ← valorObjetivo;
17         armazena(pivos, elementoPorRegiao);
18     end
19 end
20 melhoresPivos ← pivosArmazenados();
21 if NoPai = null then
22     inserirRaiz(melhoresPivos);
23 end
24 else
25     inserirNo(melhoresPivos, NoPai);
26 end
27 for (r = 0 ; r < numeroDeRegioes(NumeroExpansoes) ; r++) do
28     entrada ← elementosRegiao(elementosRegiao,r);
29     if tamanho(entrada) > 2 then
30         SampleBL(entrada[r], NumeroExpansoes, noAtual);
31     end
32     else
33         inserirNo(entrada, r);
34     end
35 end
```

O algoritmo *SampleBL* possui como núcleo o algoritmo *Sampling*, que retorna uma quantidade c de elementos amostras para serem testados como possíveis pares de pivôs, o que reduz a complexidade

do algoritmo *SampleBL* em relação ao algoritmo *GreedyBL*. Assim, a complexidade de algoritmo *SampleBL*, no que se refere ao teste dos pares de pivôs, é reduzida para $O(c \times c)$, além de *O(n)* no que se refere à verificação de qual região cada elemento será associado. Como c é constante, a complexidade final nesse trecho do algoritmo é reduzida para *O(n)*, em contraste com $O(n^3)$ do algoritmo *GreedyBL*.

4.3 Algoritmo de Bulk-Loading Baseado na Altura da Estrutura Final

O terceiro algoritmo de *bulk-loading* desenvolvido para a Onion-tree, chamado de *HeightBL*, utiliza como premissa o fato de que a escolha de pares de pivôs de um determinado nível pode ser feita a partir de uma análise comparativa da altura máxima que uma subárvore pode possuir de acordo com a quantidade de elementos de entrada. Diferentemente dos algoritmos *GreedyBL* (Seção 4.1) e *SampleBL* (Seção 4.2), o algoritmo *HeightBL* não é baseado no fato de que a distribuição dos elementos entre as regiões da Onion-tree precisa ser uniforme, ou seja, o algoritmo *HeightBL* não requer que a Onion-tree gerada seja balanceada. Em oposição, ele visa encontrar o par de pivôs que garanta que a altura para a árvore final tenha aproximadamente o valor de altura estimado antes da execução do algoritmo.

A seção 4.3.1 discute os princípios nos quais o algoritmo *HeightBL* se baseia. A seção 4.3.2 detalha esse algoritmo, a seção 4.3.4 descreve observações sobre o algoritmo *HeightBL* e a seção 4.3.3 descreve um exemplo da sua execução.

4.3.1 Altura Ideal de uma Onion-tree

Considere *AlturaEstimada* a altura estimada para uma Onion-tree com seus nós totalmente preenchidos, dado uma quantidade n de elementos do conjunto de dados de entrada. O algoritmo *HeightBL* considera que a Onion-tree resultante pode possuir altura aproximadamente igual ao valor de *AlturaEstimada*. A Equação 4.5 define essa altura estimada, usando a noção que o número total de elementos n que podem ser armazenados em uma Onion-tree preenchida é dada pela soma do número de elementos que podem ser armazenados a cada nível h do índice, i.e. $n = \sum_{h=0}^{h=H} 2 * R^h$, no qual R é o número de regiões, e 2 representa que existem dois pivôs por nó. Essa equação é obtidaa partir desse somatório, considerando que ele pode ser visto como uma soma dos $H+1$ primeiros elementos de uma progressão geométrica de razão R e primeiro elemento 2, e isolando h.

$$AlturaEstimada = \lceil \log_R^{(\frac{n*(R-1)}{2}+1)} - 1 \rceil \tag{4.5}$$

O Algoritmo 8 implementa a função que determina a altura que uma Onion-tree pode possuir a partir do número de elementos de entrada e também do número de regiões que um determinado nó possui. Seu funcionamento se resume a retornar a altura estimada que será utilizada como parâmetro de entrada para o algoritmo *HeightBL*

Algorithm 8: `Height Calculation` ($NumeroElementos$, $NumeroRegioes$)

Input : $NumeroElementos$ {quantidade de elementos a ser inserida em um dado nível da Onion-tree},
$NumeroRegies$ {quantidade de regiões do nó da Onion-tree}
Output: $AlturaEstimada$ {altura estimada para a Onion-tree}

1 return $\lceil \log_{NumeroRegies^2}(\frac{NumeroElementos*(NumeroRegies-1)}{NumeroRegies^2}+1) - 1 \rceil$;

Intuitivamente, o funcionamento do algoritmo *HeightBL* baseado em alturas é do seguinte modo. Primeiramente, ele estima a altura da Onion-tree final, chamada de *AlturaEstimada*. Em seguida, a cada nível é escolhido o par de pivôs que distribui os elementos entre as regiões determinadas de modo que a altura estimada de cada uma das subárvores não supere o valor de *AlturaEstimada*. A altura estimada para a subárvore é calculada por meio do Algoritmo 8, a partir da soma entre o nível atual da estrutura e a altura estimada para cada uma das subárvores que serão geradas a partir dos pivôs escolhidos. Assim que o primeiro par de pivôs é encontrado, o processamento é recursivamente aplicado a cada uma das subárvores por ele definidas, utilizando como dados de entrada os elementos associados a cada uma das regiões determinadas pelo par de pivôs selecionados. Assim, a escolha dos pares de pivôs leva em consideração a quantidade de elementos em cada região, sem fixar uma forma em que os dados devem estar distribuídos. Note que o algoritmo encerra a sua execução ao encontrar o primeiro par de pivos que obtém a melhor altura para uma dada subárvore.

4.3.2 Algoritmo HeightBL

O Algoritmo 9 detalha o algoritmo *HeightBL*, o qual tem como parâmetros de entrada os elementos do conjunto de dados, altura estimada para a Onion-tree a ser gerada, número de expansões da estrutura e uma referência ao nó pai. A altura inicial da Onion-tree é calculada usando-se o Algoritmo 8, e deve ser obtida antes da primeira execução do Algoritmo 9.

Inicialmente, o algoritmo seleciona as amostras para serem testadas como pares de pivôs. Isso envolve a escolha do medóide e do conjunto de amostras, além da filtragem do conjunto de amostragem (linhas 1 a 4). Ou seja, o algoritmo *HeightBL* usa também técnicas definidas para o algoritmo *SampleBL*.

Em seguida, o primeiro pivô é escolhido aleatoriamente do conjunto de amostragem (linha 5) e um laço se inicia percorrendo as amostras filtradas até que se encontre o par de pivôs para o nível ou se percorra todos os pivôs da amostra (linhas 6 a 8).

Em detalhes, após a escolha do segundo pivô da amostra (linha 9), verifica-se quantas expansões o par de pivôs irá aplicar ao nó (linhas 10 a 12) e ocorre a verificação se a estrutura é uma V-Onion-tree (o parâmetro NumeroExpansoes vale -1) ou uma F-Onion-tree (o parâmetro NumeroExpansoes tem valor maior do que 0). Nessa etapa também são determinados os elementos associados às suas respectivas regiões (linhas 9 a 15).

O próximo passo consiste em testar as alturas que serão determinadas pelo par de pivôs escolhido (linhas 16 a 28). Para cada região r presente no nó, é calculada qual a altura que sua subárvore terá com base na quantidade de elementos associados a ela (linha 18). Em seguida, é feita a verificação se a altura final da estrutura estimada é menor do que a altura calculada para a estrutura a paritir do nó atual. Em caso afirmativo, esses pares de pivôs são armazenados apenas se forem o par que possui a menor diferença de altura para a altura ideal (linhas 17 a 23). Se todas as subárvores possuem quantidades de

elementos tal que a altura final da estrutura seja inferior ou igual à altura inicial estimada, então o par de pivôs para o nível atual é escolhido e o laço é interrompido (linhas 26 a 27).

Na sequência, ocorre a inserção dos pivôs na estrutura criando um nó raiz (linhas 31 a 33) ou um nó interno (linhas 34 a 36), dependendo se o nó pai possui valor, respectivamente, nulo ou não nulo. Em seguida, ocorre a chamada recursiva para cada uma das regiões determinadas pelos pares de pivôs utilizando como dados de entrada os elementos associados a cada uma delas e eventualmente inserindo na estrutura os conjuntos de elementos cujos tamanhos são menores ou iguais a dois (linhas 37 a 44).

4.3.3 Exemplo de Execução

Nesta seção é ilustrado um exemplo de execução do Algoritmo *HeightBL*. Para facilitar a visualização dessa execução, ela é mostrada usando como base uma F-Onion-tree sem expansões. Uma F-Onion-tree sem expansões divide o espaço métrico em apenas 4 regiões, representadas por I, II, III e IV. Além disso, considere que o conjunto de elementos a ser inserido na estrutura seja $S = s_1, ..., s_{30}$, de tamanho n = 30.

Um passo executado antes do algoritmo de *HeightBL* é determinar qual seria a altura final de uma Onion-tree que pudesse armazenar essa quantidade de elementos. Para isso, o Algoritmo 8 é invocado, *HeightCalculation(NumeroElementos)*, com o parâmetro *NumeroElementos* com valor 30. Esse algoritmo faz uso da Equação 4.5 para os seguintes valores de h: 0, 1 e 2. A partir desse cálculo, a altura estimada inicialmente é $HEst$ = 3. Esse processamento é ilustrado na Figura 4.8 (a), onde n significa a quantidade de elementos por nível e h significa a altura correspondente na estrutura.

A chamada inicial para o algoritmo *HeightBL* consiste na invocação da função *HeightBL(Elementos,AlturaEstimada,NumeroExpansoes,NoPai)* com os seguintes parâmetros, na sequência: (i) conjunto S com 30 elementos; (ii) altura estimada igual a 3; (iii) número de expansões igual a 0; (iv) nó pai com valor nulo.

Primeiramente, ocorre o teste dos pares de pivôs providos pelo algoritmo de amostragem, ilustrado na Figura 4.8(b). O objetivo inicial é, portanto, encontrar pares de pivôs para o nó raiz tal que a quantidade de elementos distribuídos entre as quatro regiões determinadas resultem em uma Onion-tree com a altura final menor ou igual à determinada em $HEst$. Considere que o par de pivôs escolhidos seja s_1 e $s2$, e que esse par divida os elementos entre as regiões de modo que nove elementos sejam associados às regiões I e II e dez elementos sejam associados à região III, além da região IV não conter elementos. A escolha desse par de pivôs leva a uma Onion-tree com altura estimada h = 3 devido à quantidade de elementos associada a cada região. Note que a distribuição dos elementos não necessariamente precisa ser uniforme.

Na sequência, o algoritmo realiza chamadas recursivas para cada uma das quatro regiões, usando como base os elementos associados a elas, exceto os pares de pivôs já escolhidos. Ou seja, o algoritmo *HeightBL(Elementos,AlturaEstimada,NumeroExpansoes,NoPai)* é invocado para cada uma das regiões com os seguintes parâmetros: (i) para a região I são fornecidos nove elementos de entrada, indicação da altura estimada como sendo 3, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (ii) para a região II são fornecidos nove elementos de entrada, indicação da altura estimada como sendo 3, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (iii) para a região III são fornecidos dez elementos de entrada, indicação da altura estimada como sendo 3, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado; (iv) para a região IV não são fornecidos elementos

de entrada, indicação da altura estimada como sendo 3, número de expansões 0 e como nó pai o nó raiz que acabou de ser criado.

Em cada uma dessas chamadas recursivas, o mesmo princípio da verificação da altura final da estrutura é aplicado. No entanto, é levado também em consideração o nível atual da árvore, que passa a ter seu valor incrementado no cálculo da altura final. Considerando cada um dos nós criados a partir das chamadas recursivas, note que eles estão na altura $h = 2$ da estrutura. Assim, o par de pivôs a ser escolhido para cada um desses nós deve somente ser aquele que aumente a estrura em um nível, visto que a altura inicial estimada para a estrutura é 3. O processamento do algoritmo *HeightBL* é repetido por meio de chamadas recursivas em cada nó criado da mesma forma que detalhado anteriormente.

Por fim, como aos nós da altura $h = 3$ da estrutura são fornecidos dois ou menos elementos, esses próprios elementos são gerados como nós folhas, encerrando o algoritmo *HeightBL*.

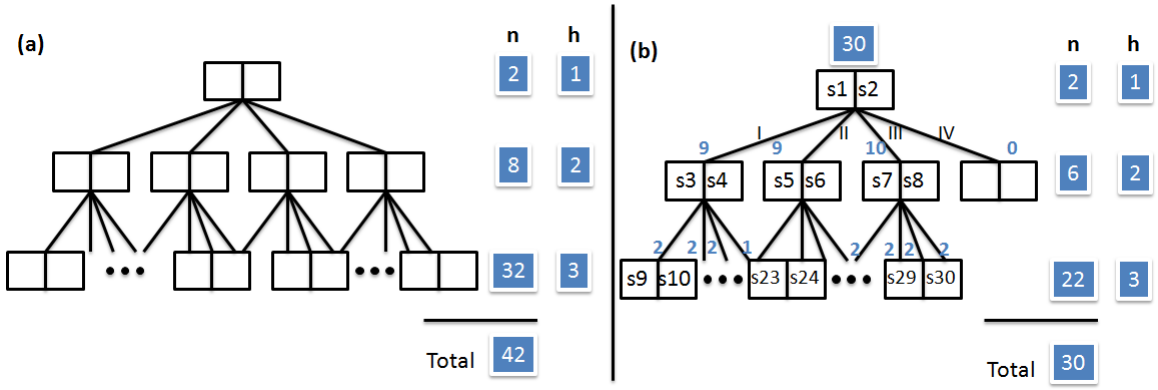

Figura 4.8: Execução do algoritmo de *bulk-loading* baseado em alturas. (a) Estimativa da altura final da Onion-tree;(b) Estrutura criada pela execução do algoritmo de *bulk-loading* baseado em alturas.

4.3.4 Observações Sobre o Algoritmo HeightBL

Vale destacar algumas observações sobre o Algoritmo 9. A primeira delas é que o processo utilizado pode não ser exato. Ou seja, é feita uma estimativa da altura inicial da Onion-tree, mas pode ser que, de acordo com a escolha do par de pivôs, a altura final atingida não seja exatamente a altura estimada, pois o processo estima a altura baseado no nó atual e não em todos os nós filhos da estrutura. Ou seja, o processo de determinar a altura é local ao nó, e não global a todos os nós filhos da estrutura. No entanto, possuir um limite a ser respeitado para a altura da Onion-tree garante que a escolha de pivôs tende a produzir uma estrutura com essa altura ao fim do processamento do algoritmo.

A segunda observação refere-se ao fato de que o Algoritmo 9 pode ser aplicado com mais precisão à F-Onion-tree do que à V-Onion-tree, visto que ele depende da quantidade de regiões que a Onion-tree irá possuir para estimar a altura resultante da estrutura. A abordagem utilizada para a V-Onion-tree foi calcular a altura da estrutura com sobras. Ou seja, a altura da V-Onion-tree é determinada como se fosse a pior altura que uma F-Onion-tree poderia alcançar com a ocupação de todos os seus nós, situação que ocorre quando a Onion-tree possui apenas quatro regiões. Portanto, a altura da V-Onion-tree pode ser, frequentemente, inferior à altura estimada inicialmente para a estrutura, visto que quanto mais regiões um nó possui, mais elementos ele pode armazenar. Como consequência, a altura estimada para uma

mesma quantidade de elementos será menor em relação a uma F-Onion tree com apenas quatro regiões em todos os nós.

Finalmente, vislumbra-se que a abordagem baseada em altura empregada pelo algoritmo *HeightBL* apresente um ganho de processamento na construção da estrutura com relação à abordagem baseada em amostras usada no algoritmo *SampleBL*. A principal razão para essa expectativa se deve ao fato de que, no algoritmo *HeightBL*, não é necessário avaliar todos os pivôs do conjunto de amostragem em um dado nó para que se identifique o par de pivôs que satisfaça a diferença de alturas necessária. Ou seja, a etapa de escolha de pivôs do algoritmo *HeightBL* se encerra ao encontrar o primeiro que satisfaça à condição das alturas, sem a necessidade de verificar e avaliar qual par de pivôs é o melhor.

Algorithm 9: `HeightBL` (*Elementos*,*AlturaEstimada*,*NumeroExpansoes*,*NoPai*)

```
Input  : Elementos {elementos do conjunto de dados},
         AlturaEstimada {altura estimada para a Onion-tree completa},
         NumeroExpansoes {número de expansões da Onion-tree},
         NoPai {referência ao nó pai do nó atual}
Output: Onion−tree {estrutura Onion-tree final}
1  medoid ← approximatedMedoid(Elementos,sizeMedoid);
2  bestMedoid ← minimizeMedoid(medoid,Elementos);
3  sample ← sampling(Elementos,sizeSample);
4  filteredSample ← filtering(sample);
5  pivo1 ← random(filteredSample);
6  i ← 0;
7  achouPivos ← falso;
8  while i < tamanho(filteredSample) AND achouPivos = falso do
9      pivo2 ← selecionarPivos(filteredSample, i);
10     if NumeroExpansoes = -1 then
11         NumeroExpansoes = PatitioningMethod(pivos[i], pivos[j],NoPai, tipoExpansao);
12     end
13     pivos[0] ← pivo1;
14     pivos[1] ← pivo2;
15     elementosRegiao ← verificarRegioes(pivos, Elementos - pivos, NumeroExpansoes);
16     testeAltura ← verdadeiro;
17     for (r = 0 ; r < numeroRegioes(NumeroExpansoes); r++) do
18         altura ←
           HeightCalculation(tamanho(elementosRegiao[r][]),numeroRegioes(NumeroExpansoes));
19         if AlturaEstimada <= nivelAtual + altura then
20             testeAltura ← verdadeiro;
21             armazenarPivosMinimos(pivo1,pivo2,altura);
22         end
23     end
24     if testeAltura = verdadeiro then
25         achouPivos ← verdadeiro;
26         armazenarPivos(pivo1,pivo2,altura);
27     end
28     i++;
29 end
30 melhoresPivos[] ← pivosArmazenados();
31 if NoPai = null then
32     inserirRaiz(melhoresPivos);
33 end
34 else
35     inserirNo(melhoresPivos, NoPai);
36 end
37 for (r = 0 ; r < numeroDeRegioes(NumeroExpansoes) ; r++) do
38     entrada ← elementosRegiao(elementosRegiao,r);
39     if tamanho(entrada) > 2 then
40         HeightBL(entrada[r], AlturaEstimada, NumeroExpansoes, NoAtual);
41     end
42     else
43         inserirNo(entrada, r);
44     end
45 end
```

4.4 Considerações Finais

Este capítulo apresentou três algoritmos propostos para o *bulk-loading* do MAM Onion-tree, denominados *GreedyBL*, *SampleBL* e *HeightBL*, os quais utilizam as seguintes abordagens principais: (i) gulosa; (ii) amostragem; e (iii) estimativa da altura do índice. No próximo capítulo são descritos experimentos que validam os algoritmos propostos e que demonstram as vantagens que eles possuem em relação ao algoritmo de inserção um-a-um da Onion-tree.

Capítulo 5

Experimentos e Resultados

Nesse capítulo são detalhados os experimentos realizados para validação dos algoritmos de *bulk-loading* propostos nesta dissertação de mestrado. Nos testes de desempenho, foram utilizados três conjuntos de dados com diferentes dimensionalidades (i.e. 32 a 117 dimensões) e quantidades de elementos (i.e. 2.536 a 102.240 elementos). A Tabela 5.1 descreve os conjuntos de dados utilizados nos testes de desempenho, indicando seus nomes, números de elementos, dimensionalidade e descrição. Esses conjuntos de dados também foram utilizados no artigo que propõe a Onion-tree [Carélo et al., 2009].

Os algoritmos propostos foram comparados com o algoritmo de inserção um-a-um da Onion-tree devido à ausência de algoritmos de *bulk-loading* para essa estrutura na literatura. Os algoritmos *GreedyBL*, *SampleBL* e *HeightBL* foram implementados usando a linguagem de programação C++. Já o algoritmo de inserção da Onion-tree utilizado foi o mesmo que o usado nas implementações descritas em [Carélo et al., 2009]. Nos testes realizados, foram consideradas as duas versões da Onion-tree: a

Tabela 5.1: Conjunto de dados utilizados nos experimentos

Conjunto de Dados	Número de Elementos	Dimensionalidade	Descrição
Color Histograms	68.025	32	Histogramas de imagens do repositório KDD da Universidade da Califórnia (kdd.ics.uci.edu)
Ozone	2.536	73	Séries temporais de 1998 a 2004 para detecção do nível de ozônio (archive.ics.uci.edu/ml/datasets/Ozone+Level+Detection)
KDD Cup 2008	102.240	117	Conjunto de dados contendo imagens de câncer (www.kddcup2008.com)

F-Onion-tree e a V-Onion-tree. Para a F-Onion-tree, foram utilizados os seguintes valores de F: 7 para Color Histograms, 7 para Ozone e 11 para KDD Cup 2008. Esses valores são os valores de expansão que garantem o melhor desempenho do índice para esses conjuntos de dados, de acordo com os resultados descritos no artigo que propõe a Onion-tree [Carélo et al., 2009]. A métrica utilizada para os conjuntos de dados Color Histograms e KDD Cup 2008 foi L_2. Para o conjunto de dados Ozone foi utilizada a métrica *dynamic time warping*, a fim de investigar o impacto causado por uma métrica que exige mais processamento do que a métrica L_2.

Os experimentos foram executados em um computador com processador Intel Core i7 2.67 Ghz, 12 GB de memória RAM e 1 TB de espaço em disco. Para cada um dos conjuntos de dados utilizados, foram coletadas as seguintes medidas: tempo de construção do índice em segundos, número de cálculos de distância para construir o índice, tamanho do índice em kilobytes, tempo total gasto em segundos para a realização de consultas por abrangência e aos k-vizinhos mais próximos, e números de cálculos de distância para esses dois tipos de consulta.

Especificamente com relação ao processamento das consultas por abrangência e aos k-vizinhos mais próximos, para cada conjunto de dados, foram escolhidos aleatoriamente 500 elementos para serem os centros das consultas. Desses 500 elementos, 250 foram removidos do conjunto original antes da execução dos algoritmos propostos, de forma que apenas metade do conjunto de dados foi indexado. Os resultados descritos nos testes de desempenho referem-se ao número médio de cálculos de distância e ao tempo médio de processamento para executar as 500 consultas. As consultas foram executadas variando o raio de modo a recuperar de 1% a 10% da quantidade total de elementos nas consultas por abrangência. Nas consultas aos k-vizinhos mais próximos, o valor de k variou de 2 a 20.

Para facilitar a visualização e a análise dos testes realizados, os resultados obtidos são descritos considerando a seguinte forma de apresentação. Para o algoritmo *GreedyBL*, são detalhados apenas os resultados relativos à F-Onion-tree, considerando esse algoritmo proposto e a inserção um-a-um. Para os algoritmos *SampleBL* e *HeightBL*, primeiramente são detalhados os resultados para a F-Onion-tree, considerando esses algoritmos propostos e o algoritmo de inserção um-a-um. Na sequência, são descritos os resultados para a V-Onion-tree, considerando esses algoritmos propostos e o algoritmo de inserção um-a-um. As análises realizadas levam em consideração essa forma de apresentação.

O capítulo é organizado como segue. A seção 5.1 descreve os resultados obtidos para o algoritmo *GreedyBL*, enquanto que as seções 5.2 a 5.4 detalham os resultados relacionados aos algoritmos *SampleBL* e *HeightBL*. A seção 5.2 descreve o tempo gasto e o número de cálculos de distância para a construção dos índices, a seção 5.3 enfoca a comparação do tamanho dos índices, e a seção 5.4 descreve o comportamento dos algoritmos no processamento das consultas por similaridade. Por fim, a seção 5.5 finaliza o capítulo resumindo as considerações finais.

5.1 Análise de Desempenho para o Algoritmo GreedyBL

Na seção 4.1.2 foi mostrado que a complexidade teórica do algoritmo *GreedyBL* é $O(n^3)$. Isso faz com que a construção do índice seja muito demorada, cúbica, de acordo com o número de elementos de entrada. Essa seção mostra alguns resultados de desempenho para o algoritmo, com o objetivo de comprovar experimentalmente a sua complexidade teórica.

Para os testes foi utilizado o menor conjunto de dados, Ozone, com a métrica (i.e. função de distância) mais cara, *dynamic time warping*, e foi considerada a versão F-Onion-tree. Os resultados

obtidos, mostrados nas Figuras 5.2 e 5.1, demonstraram que o número de cálculos de distância e o tempo gasto para a construção do índice pelo algoritmo proposto apresentaram valores muito maiores do que a inserção um-a-um. Esses resultados comprovam a complexidade teórica do algoritmo, e evidenciam o fato de que o tempo gasto e o número de cálculos de distância requeridos para a construção providos pelo algoritmo *GreedyBL* são proibitivos, desde que a Onion-tree é um índice voltado para a memória primária.

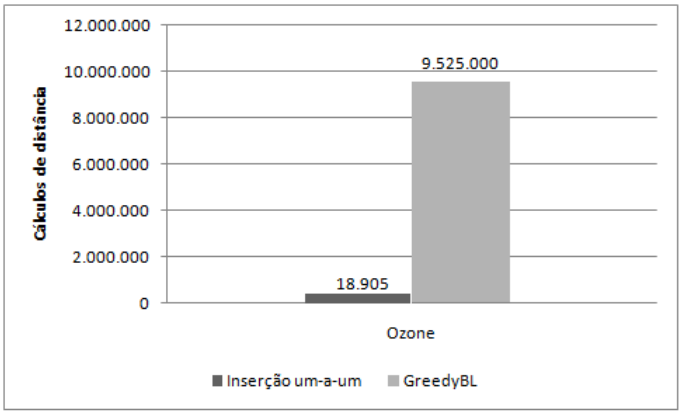

Figura 5.1: Número de cálculos de distância para construir o índice: inserção um-a-um e algoritmo *GreedyBL*.

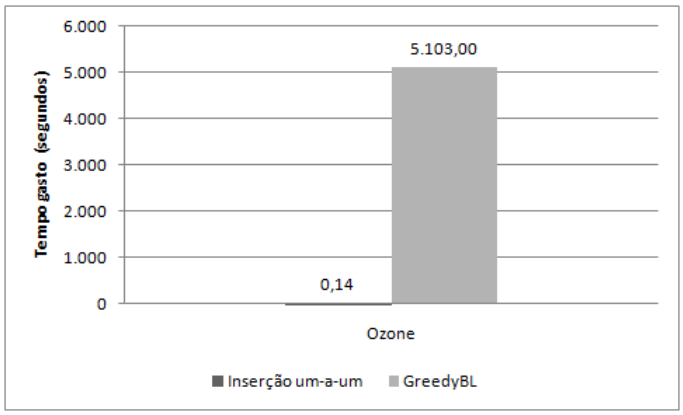

Figura 5.2: Tempo gasto em segundos para construir o índice: inserção um-a-um e algoritmo *GreedyBL*.

Em contrapartida, o algoritmo *GreedyBL* garantiu melhores resultados do que a inserção um-a-um quando considerados o tamanho do índice e as consultas por abrangência e aos k-vizinhos mais próximos. A Figura 5.3 ilustra o tamanho do índice gerado. Os resultados mostraram que, apesar do tempo elevado para construir o índice, o fato do algoritmo *GreedyBL* testar todos os pivôs possíveis garante a geração de uma estrutura que ocupa muito menos espaço de armazenamento do que a estrutura gerada pelo algoritmo de inserção um-a-um.

A Figura 5.4 mostra os resultados de desempenho no processamento de consultas por abrangência. Os seguintes resultados foram obtidos:

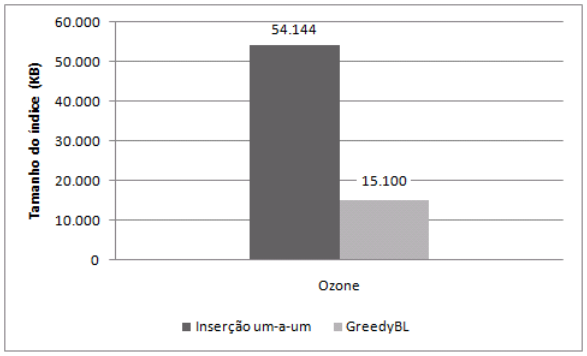

Figura 5.3: Tamanho em kilobytes: inserção um-a-um e algoritmo *GreedyBL*.

- À medida que o raio da consulta por abrangência aumentou, aumentou também o ganho de desempenho do algoritmo *GreedyBL* com relação ao algoritmo de inserção um-a-um para a medida de número de cálculos de distância (Figura 5.4a). O ganho de desempenho do algoritmo proposto foi sempre muito expressivo, variando de 89% a 98%. A diferença de desempenho entre os algoritmos foi de até 2 ordens de grandeza.

- Com relação ao tempo gasto no processamento de consultas por abrangência (Figura 5.4b), o algoritmo *GreedyBL* sempre proveu melhores resultados de desempenho do que a inserção um-a-um, sem uma indicação de aumento ou de diminuição de ganho de desempenho à medida do aumento do raio. O ganho de desempenho variou de 82% a 96%. A diferença de desempenho entre os algoritmos sempre foi de 1 ordem de grandeza.

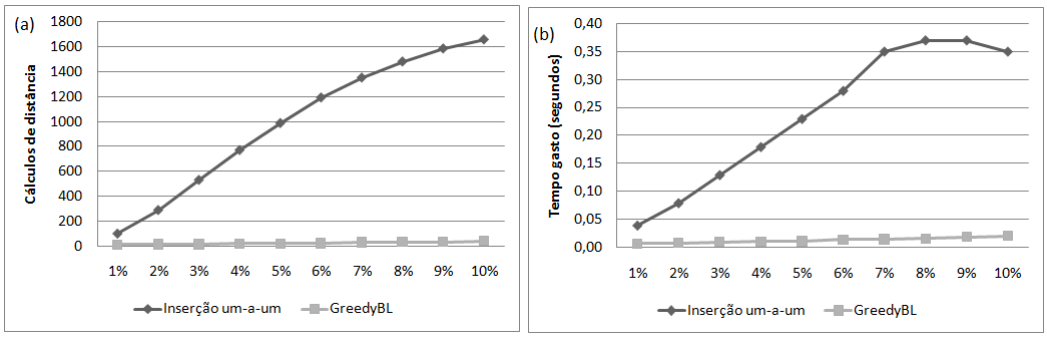

Figura 5.4: Processamento de consultas por abrangência sobre o conjunto de dados Ozone: inserção um-a-um e algoritmo *GreedyBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

A Figura 5.5 mostra os resultados de desempenho no processamento de consultas aos k-vizinhos mais próximos. Os seguintes resultados foram obtidos:

- Considerando o número de cálculos de distância e as consultas aos k-vizinhos mais próximos (Figura 5.5a), inicialmente o ganho de desempenho do algoritmo *GreedyBL* aumentou conforme o valor de k aumentou, variando de 66% para $k = 2$ a 74% para $k = 6$ em relação à inserção um-a-um. Em seguida, o ganho de desempenho do algoritmo proposto frente à inserção um-a-um diminuiu

conforme o valor de k aumentou, chegando a 60% para $k = 20$. Os resultados de desempenho dos algoritmos foram da mesma ordem de grandeza.

- Com relação ao tempo gasto no processamento de consultas aos k-vizinhos mais próximos (Figura 5.5b), o ganho de desempenho do algoritmo *GreedyBL* variou de 39% a 56% com relação à inserção um-a-um. A partir de $k = 6$, o ganho de desempenho do algoritmo proposto diminuiu conforme o valor de k aumentou. A diferença de desempenho entre os algoritmos foi de até 1 ordem de grandeza.

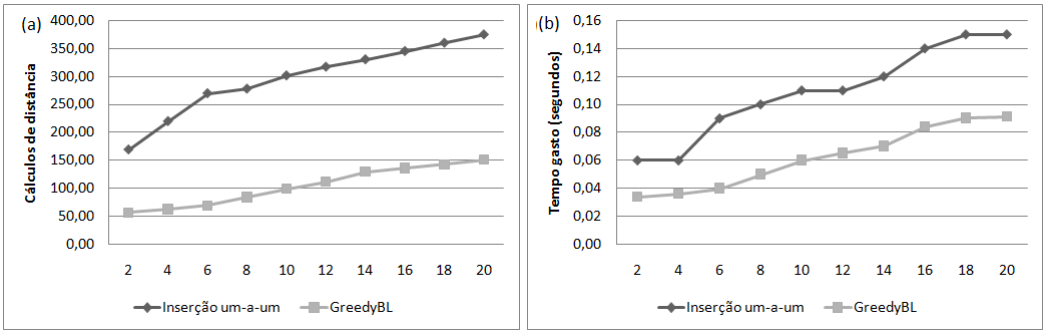

Figura 5.5: Processamento de consultas aos k-vizinhos mais próximos sobre o conjunto de dados Ozone: inserção um-a-um e algoritmo *GreedyBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

Os ganhos de desempenho providos pelo algoritmo *GreedyBL* no processamento das consultas por similaridade são justificados pelo fato de que o teste exaustivo para encontrar o melhor par de pivôs a ser escolhido a cada nó gera uma estrutura que garante uma boa divisão do espaço métrico.

Apesar dos resultados providos pelo algoritmo *GreedyBL* relacionados ao tamanho do índice e às consultas por abrangência e aos k-vizinhos mais próximos serem muito positivos (os melhores dentre todos os algoritmos propostos nesta dissertação), o tempo de construção inviabiliza o seu uso. Entretanto, a proposta desse algoritmo foi de suma importância para a proposta dos algoritmos *SampleBL* e *HeightBL*.

5.2 Tempo Gasto e Cálculos de Distância na Construção do Índice pelos Algoritmos SampleBL e HeightBL

A seção 5.2.1 descreve os resultados considerando a F-Onion-tree, enquanto que a seção 5.2.2 descreve os resultados considerando a V-Onion-tree.

5.2.1 Construção da F-Onion-tree

Na Figura 5.6 é ilustrado o número de cálculos de distância para a construção do índice para os diferentes conjuntos de dados da Tabela 5.1, considerando a F-Onion-tree. A construção por meio do algoritmo de inserção um-a-um requereu um número menor de cálculos de distância do que os algoritmos *SampleBL* e *HeightBL*, desde que os algoritmos propostos realizam cálculos de distância entre todos os elementos de

um nível da estrutura no instante da construção do índice. Apesar dos algoritmos propostos requererem maior número de acessos a disco, isso é compensado tanto em redução do tamanho do índice (seção 5.3) quanto em redução no número de cálculos de distância e no tempo gasto para processar as consultas por abrangência e aos k-vizinhos mais próximos (seção 5.4).

Considerando o algoritmo *SampleBL* e a inserção um-a-um, em seu melhor caso, para o conjunto Color Histograms, o algoritmo proposto requereu 38% mais cálculos de distância e, em seu pior caso, para o conjunto KDD Cup 2008, o algoritmo proposto requereu 47% mais cálculos de distância. Por outro lado, considerando o algoritmo *HeightBL* e a inserção um-a-um, em seu melhor caso, no conjunto Color Histograms, o algoritmo proposto requereu 33% mais cálculos de distância e, em seu pior caso, para o conjunto KDD Cup 2008, o algoritmo proposto fez aproximadamente 43% mais cálculos de distância.

Comparando-se os algoritmos *SampleBL* e *HeightBL*, pode-se verificar que o algoritmo *HeightBL* requereu menos cálculos de distância para a construção do índice do que o algoritmo *SampleBL* para todos os conjuntos de dados. O ganho de desempenho do algoritmo *HeightBL* variou de 6% a 8%. O algoritmo *HeightBL* calcula menos distâncias porque finaliza o teste de pivôs assim que o primeiro par de pivôs que satisfaz à restrição das alturas é encontrado, enquanto que o algoritmo *SampleBL* testa todos os pares de elementos da amostra coletada até que se encontre o par de elementos que melhor divide os demais elementos entre as regiões.

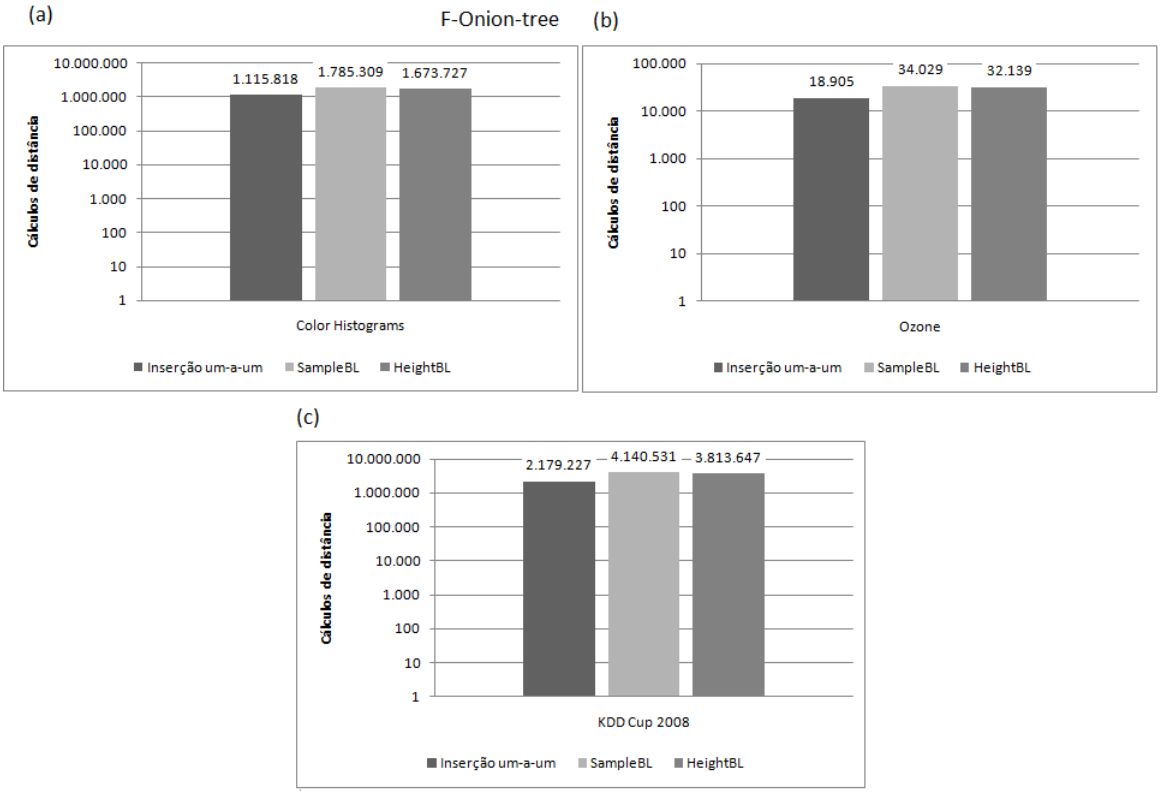

Figura 5.6: Número de cálculos de distância para construir a F-Onion-tree: inserção um-a-um e algoritmos *SampleBL* e *HeightBL*. (a) Conjunto de dados Color Histograms. (b) Conjunto de dados Ozone. (c) Conjunto de dados KDD Cup 2008.

Na Figura 5.7 é ilustrado o tempo gasto para a construção do índice para os diferentes conjuntos de dados da Tabela 5.1, considerando a F-Onion-tree. De forma similar aos resultados discutidos com relação ao número de cálculos de distância para a construção do índice, a construção por meio do algoritmo de inserção um-a-um é mais rápida do que os algoritmos *SampleBL* e *HeightBL*. A complexidade da inserção um-a-um é menor em relação aos algoritmos *SampleBL* e *HeightBL* devido ao fato de não existir a necessidade de se testar vários elementos entre si no instante da inserção. Novamente, apesar dos algoritmos propostos gastarem mais tempo para a construção do índice, isso é compensado tanto em redução do tamanho do índice (seção 5.3) quanto em redução no número de cálculos de distância e no tempo gasto para processar as consultas por abrangência e aos k-vizinhos mais próximos (seção 5.4).

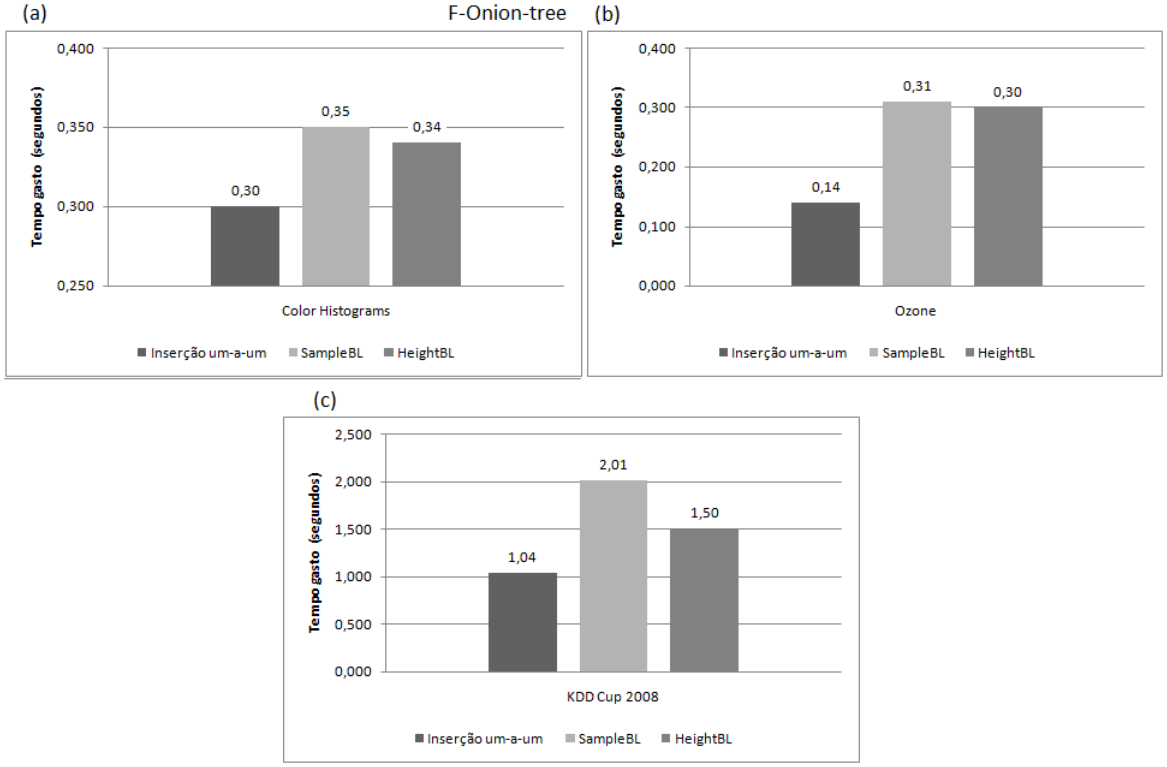

Figura 5.7: Tempo gasto em segundos para construir a F-Onion-tree: inserção um-a-um e algoritmos *SampleBL* e *HeightBL*. (a) Conjunto de dados Color Histograms. (b) Conjunto de dados Ozone. (c) Conjunto de dados KDD Cup 2008.

Comparando o algoritmo *SampleBL* com a inserção um-a-um, a perda de desempenho do algoritmo proposto foi de 14%, 55% e 48% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente. Já a perda de desempenho do algoritmo *HeightBL* com relação à inserção um-a-um foi de 12%, 53% e 31% para os mesmos conjuntos de dados, respectivamente. O pior caso dos algoritmos propostos refere-se à indexação do conjunto Ozone, o qual utiliza uma função de distância custosa.

Comparando-se os algoritmos *SampleBL* e *HeightBL*, pode-se verificar que o algoritmo *HeightBL* proveu melhores resultados de desempenho quanto ao tempo gasto para construir a F-Onion-tree para todos os conjuntos de dados. A perda de desempenho do algoritmo *SampleBL* com relação ao algoritmo *HeightBL* foi de 3%, 3% e 25% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente. O algoritmo *HeightBL* é em geral mais rápido e calcula menos distâncias porque

finaliza o teste de pivôs assim que o primeiro par de pivôs que satisfaz à restrição das alturas é encontrado, enquanto que o algoritmo *SampleBL* testa todos os pares de elementos da amostra coletada até que se encontre o par de elementos que melhor divide os demais elementos entre as regiões. Além disso, a diferença de tempo entre os algoritmos para o conjunto KDD Cup 2008 se deve à razão de que esse conjunto possui grande volume de dados e dimensionalidade.

5.2.2 Construção da V-Onion-tree

Na Figura 5.8 é ilustrado o número de cálculos de distância para a construção do índice para os diferentes conjuntos de dados da Tabela 5.1, considerando a V-Onion-tree. Pode-se verificar que o algoritmo de inserção um-a-um realizou menos cálculos de distância em relação aos algoritmos *SampleBL* e *HeightBL*. Além disso, o algoritmo *HeightBL* requereu menos cálculos de distância para construção do índice do que o algoritmo *SampleBL*. Esses resultados para a V-Onion-tree seguiram, portanto, o mesmo padrão dos resultados obtidos para a F-Onion-tree em termos do número de cálculos de distância (seção 5.2.1).

Considerando o algoritmo *SampleBL* e a inserção um-a-um, em seu melhor caso, no conjunto Color Histograms, o algoritmo *SampleBL* gastou 29% mais cálculos de distância para a construção da estrutura em relação ao algoritmo de inserção um-a-um. Em seu pior caso, para o conjunto KDD Cup 2008, o algoritmo proposto fez aproximadamente 55% mais cálculos de distância do que a inserção um-a-um. Por outro lado, considerando o algoritmo *HeightBL*, em seu melhor caso, no conjunto KDD Cup 2008, o algoritmo proposto gastou 20% mais cálculos de distância para a construção da estrutura em relação ao algoritmo de inserção um-a-um. Em seu pior caso, para o conjunto Ozone, o algoritmo proposto fez aproximadamente 29% mais cálculos de distância do que a inserção um-a-um.

Comparando-se os algoritmos *SampleBL* e *HeightBL*, os ganhos de desempenho do algoritmo *HeightBL* foram de 7%, 31%, e 43% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente. Assim como os resultados apresentados em relação a número de cálculos de distância para a construção da F-Onion-tree, o algoritmo *HeightBL* exige menos cálculos de distância para construir o índice em relação ao algoritmo *SampleBL* devido ao fato de encerrar o teste de pivôs assim que encontra o primeiro par que satisfaça à condição de altura estabelecida.

Na Figura 5.9 é ilustrado o tempo gasto para a construção do índice para os diferentes conjuntos de dados da Tabela 5.1, considerando a V-Onion-tree. Os resultados mostrados nessa figura são similares aos resultados descritos na seção 5.2.1 quanto ao tempo gasto, considerando que:

- A construção por meio do algoritmo de inserção um-a-um é mais rápida.
- Os resultados são compensados tanto em redução do tamanho do índice (seção 5.3) quanto em redução no número de cálculos de distância e no tempo gasto para processar as consultas por abrangência e aos k-vizinhos mais próximos (seção 5.4).
- O algoritmo *HeightBL* proveu melhores resultados de desempenho do que o algoritmo *SampleBL*.

Comparado com a inserção um-a-um, o algoritmo *SampleBL* apresentou seu melhor caso em relação ao conjunto KDD Cup 2008, para o qual gastou 82% a mais de tempo, e o pior caso foi em relação ao conjunto Ozone, para o qual foi 87% mais lento para construir o índice. Considerando o algoritmo

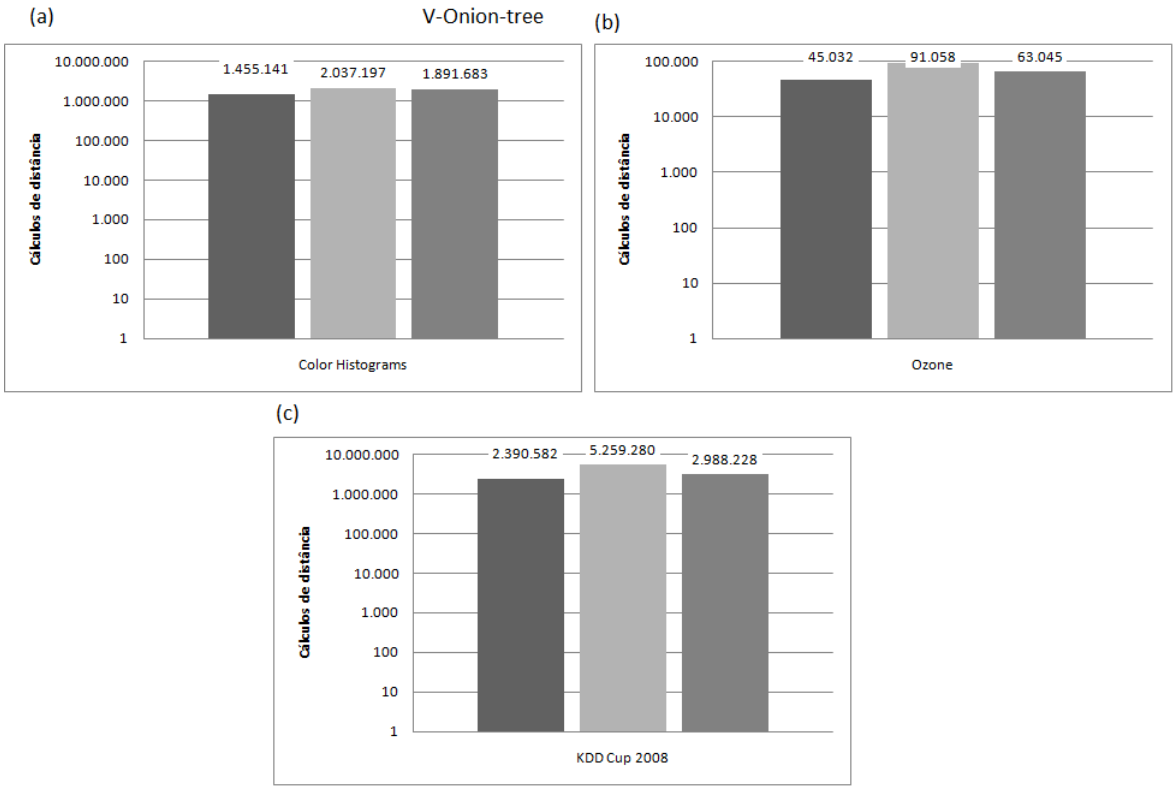

Figura 5.8: Número de cálculos de distância para construir a V-Onion-tree: inserção um-a-um e algoritmos *SampleBL* e *HeightBL*.

HeightBL, seu melhor caso foi em relação ao conjunto KDD Cup 2008, para o qual foi 25% mais lento, e o pior caso foi em relação ao conjunto Color Histograms, para o qual foi 83% mais lento.

Comparando-se os algoritmos *SampleBL* e *HeightBL*, os resultados são similares aos discutidos em relação ao número de cálculos de distância para construir o índice. Como o algoritmo *HeightBL* realiza menos cálculos de distância para construção do índice, consequentemente o tempo gasto nesse processo é menor. Assim, os ganhos de desempenho do algoritmo *HeightBL* em relação ao algoritmo *SampleBL* foram de 5%, 77%, e 75% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente.

Como pode ser notado, a construção de uma V-Onion-tree influenciou mais fortemente os algoritmos propostos do que a construção de uma F-Onion-tree. Isso está relacionado ao fato de que, para se construir a V-Onion-tree, existe um gasto adicional de tempo devido ao método de particionamento da estrutura descrito na seção 4.2.4. Como esse método de particionamento envolve cálculos adicionais para determinar o número de expansões em cada nível, consequentemente o tempo total dos algoritmos propostos para construir a V-Onion-tree é maior em relação à F-Onion-tree. Em especial, mesmo considerando a inserção um-a-um, o tempo gasto para se construir uma V-Onion-tree é sempre maior do que o tempo gasto para se construir uma F-Onion-tree, corroborando os resultados descritos em [Carélo et al., 2009].

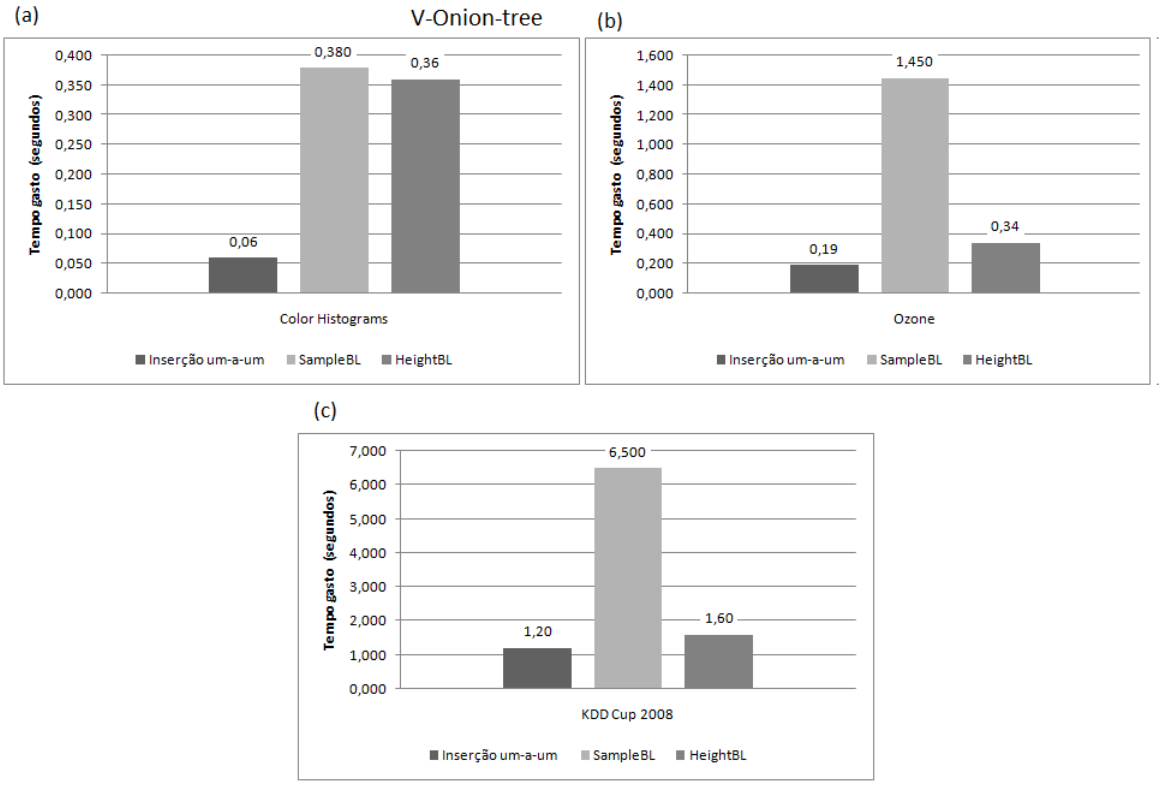

Figura 5.9: Tempo gasto em segundos para construir a V-Onion-tree: inserção um-a-um e algoritmos *SampleBL* e *HeightBL*.

5.3 Tamanho do Índice Gerado pelos Algoritmos SampleBL e HeightBL

A seção 5.3.1 descreve os resultados considerando a F-Onion-tree, enquanto que a seção 5.3.2 descreve os resultados considerando a V-Onion-tree.

5.3.1 Tamanho da F-Onion-tree

Na Figura 5.10 são ilustrados os tamanhos dos índices construídos para os diferentes conjuntos de dados da Tabela 5.1, considerando uma F-Onion-tree. O tamanho dos índices gerados pelos algoritmos *SampleBL* e *HeightBL* foram inferiores aos tamanhos dos índices gerados pela inserção um-a-um, resultando em uma economia de espaço significativa. A economia de espaço se deve ao fato de que, ao tentar buscar estruturas com melhor divisão dos elementos entre as regiões determinadas no espaço métrico, os algoritmos evitam o gasto de espaço desnecessário causado pelo desbalanceamento do índice. À medida em que o índice vai se desbalanceando devido à má distribuição dos elementos entre as regiões, novos ponteiros para regiões vazias vão sendo adicionados à estrutura, o que ocasiona espaço desnecessário armazenado no índice. Assim, além de apenas adicionar ponteiros quando de fato existe um nó a ser adicionado, os algoritmos de *bulk-loading* propostos diminuem o gasto do espaço armazenado com a melhor divisão dos elementos entre as regiões da estrutura.

Com relação à inserção um-a-um, o algoritmo *SampleBL* proveu uma economia de espaço de de armazenamento de 54%, 55% e 57% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente. Já o algoritmo *HeightBL*, quando comparado com a inserção um-a-um, gerou

uma economia de espaço de armazenamento de 53%, 70% e 71% para os conjuntos analisados, respectivamente.

Comparando os algoritmos *SampleBL* e *HeightBL*, nota-se que os índices gerados para o conjunto Color Histograms possuem aproximadamente o mesmo tamanho. Por outro lado, os índices gerados pelo algoritmo *HeightBL* para os conjuntos de dados Ozone e KDD Cup 2008 são, respectivamente, 33% e 34% menores em relação aos índices gerados pelo algoritmo *SampleBL*. Ou seja, à medida em que a dimensionalidade aumenta, o algoritmo *HeightBL* gera índices mais compactos em relação ao algoritmo *SampleBL* devido à maior rigidez no controle da altura da estrutura final. Como o algoritmo *HeightBL* aproxima a altura do índice final à altura estimada, o tamanho da estrutura também é limitado.

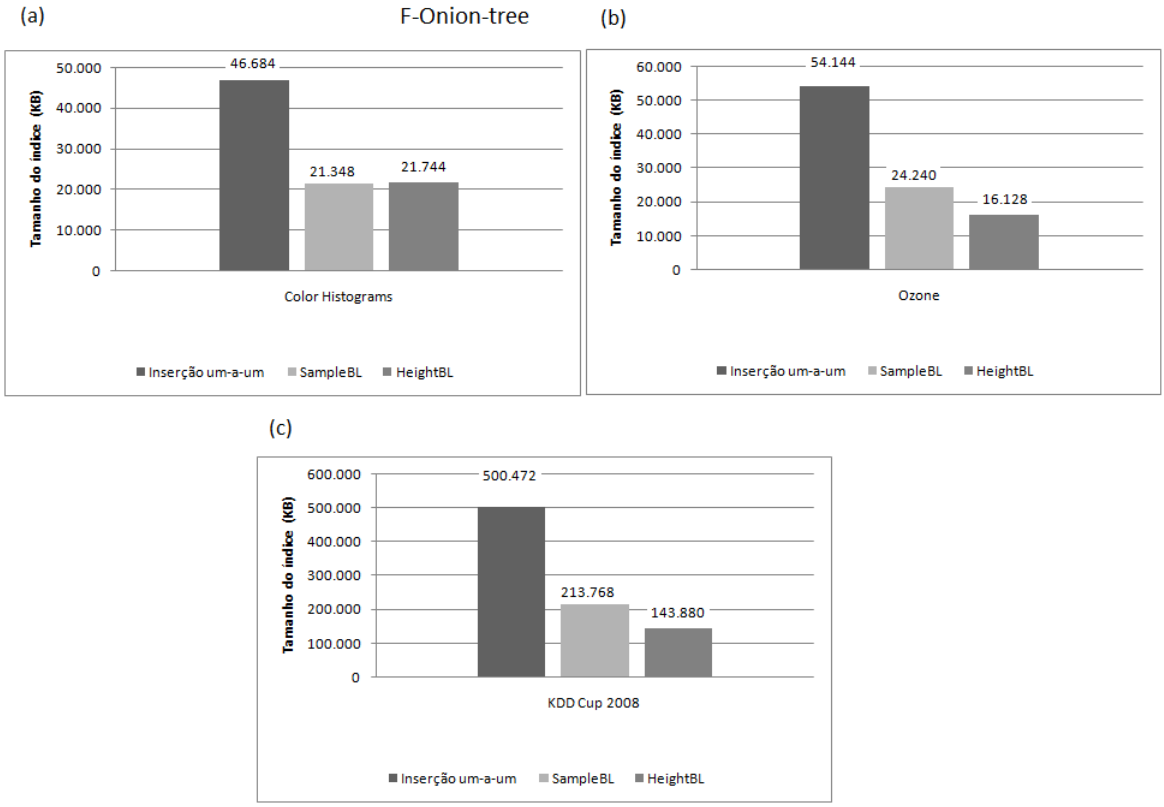

Figura 5.10: Tamanho em kilobytes para a F-Onion-tree: inserção um-a-um e algoritmos *GreedyBL* e *HeightBL*.

5.3.2 Tamanho da V-Onion-tree

Na Figura 5.11 são ilustrados os tamanhos dos índices construídos para os diferentes conjuntos de dados da Tabela 5.1, considerando uma V-Onion-tree. Os resultados mostrados nessa figura são similares aos resultados descritos na seção 5.3.1, considerando que:

- O tamanho dos índices gerados pelos algoritmos *SampleBL* e *HeightBL* foram inferiores aos tamanhos dos índices gerados pela inserção um-a-um, resultando em uma economia de espaço significativa. De acordo com os resultados obtidos, o algoritmo *HeightBL* proveu uma economia

de espaço de armazenamento de 87%, 33% e 9% para os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente. Já o algoritmo *HeightBL* proveu uma economia de espaço de 88%, 72% e 27% para esses mesmos conjuntos de dados, respectivamente.

- Comparando os algoritmos *SampleBL* e *HeightBL*, nota-se que, assim como para a F-Onion-tree, o algoritmo *HeightBL* obteve melhores resultados em termos de tamanho do índice do que o algoritmo *SampleBL*. Para os conjuntos de dados Color Histograms, Ozone e KDD Cup 2008, o algoritmo *HeightBL* gerou índices que foram 2%, 58% e 20%, respectivamente, menores em relação aos índices gerados pelo algoritmo *SampleBL*.

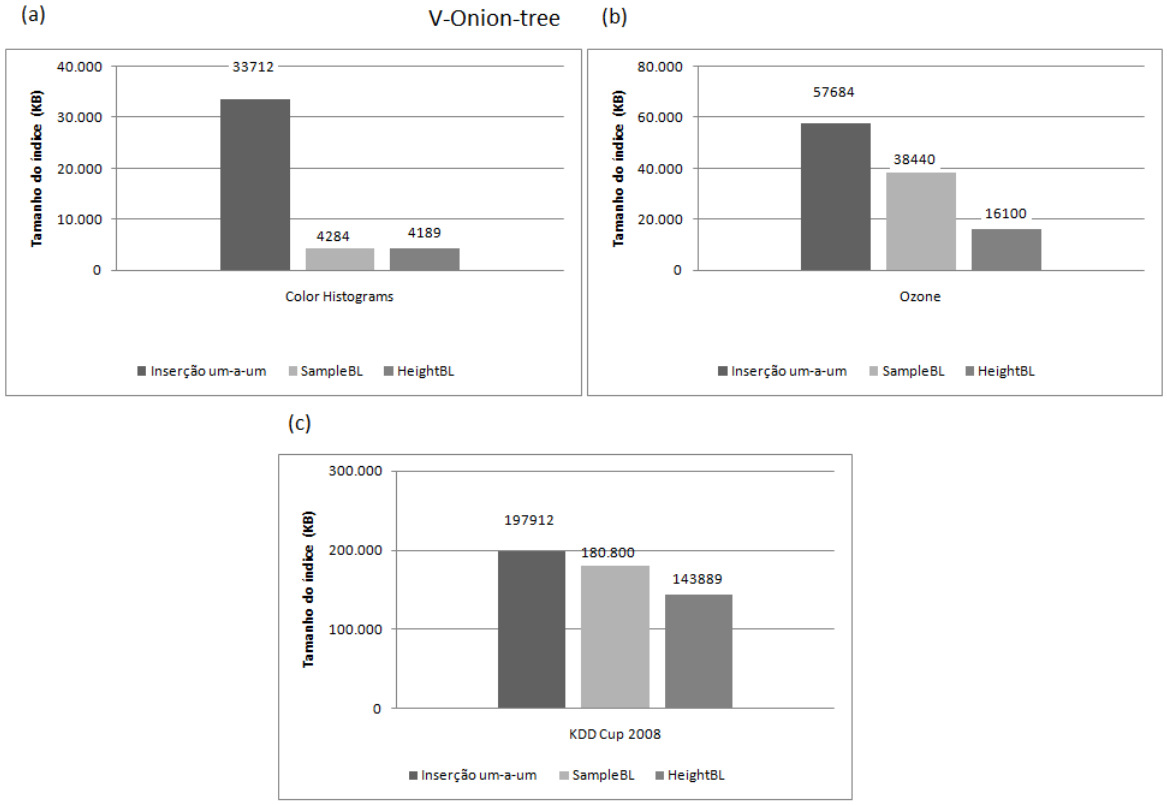

Figura 5.11: Tamanho em kilobytes para a V-Onion-tree: inserção um-a-um e algoritmos *GreedyBL* e *HeightBL*.

5.4 Processamento de Consultas Usando os Índices Gerados pelos Algoritmos SampleBL e HeightBL

A seção 5.4.1 resume os resultados de desempenho no processamento de consultas considerando a F-Onion-tree, enquanto que a seção 5.4.2 resume os resultados considerando a V-Onion-tree. Os dados detalhados usados como base para essas duas seções são apresentados na seção 5.4.3

5.4.1 Processamento de Consultas usando a F-Onion-tree

As tabelas 5.2 e 5.3 resumem o ganho mínimo e máximo em cálculos de distância e os valores mínimo e máximo de redução no tempo de processamento das consultas por similaridade que os algoritmos *SampleBL* e *HeightBL* proveram em relação à inserção um-a-um. Como pode ser observado, os algoritmos de *bulk-loading* propostos geraram índices que proveram melhor desempenho no processamento de consultas por similaridade do que os índices produzidos pela inserção um-a-um, para todos os conjuntos de dados. Isso está relacionado ao fato de que o índice produzido pelo algoritmo *SampleBL* garante uma distribuição dos dados entre as regiões da F-Onion-tree aproximadamente balanceada, enquanto que o índice produzido pelo algoritmo *HeightBL* garante um maior controle da altura da F-Onion-tree final. Tanto a estrutura aproximadamente balanceada quanto a garantia de controle da altura garantem melhor divisão do espaço métrico, de forma que as consultas por abrangência e aos k-vizinhos mais próximos requeiram um número menor de cálculos de distância e gastem menos tempo para serem processadas. Por outro lado, no instante do processamento de consultas, a inserção um-a-um realiza mais chamadas recursivas até que os nós folhas sejam analisados ou os elementos que fornecem a resposta para a consulta sejam encontrados.

Em detalhes, comparando cada algoritmo proposto individualmente com a inserção um-a-um, destacam-se os seguintes resultados:

- O algoritmo *SampleBL*, em consultas por abrangência, proveu redução do número de cálculos de distância de 32% a 97% e redução do tempo de consulta de 17% até 89%.

- O algoritmo *SampleBL*, em consultas aos k-vizinhos mais próximos, proveu redução do número de cálculos de distância de 25% a 70% e redução do tempo de consulta de 17% até 53%.

- O algoritmo *HeightBL*, em consultas por abrangência, proveu redução do número de cálculos de distância de 16% a 99% e redução do tempo de consulta de 10% até 99%.

- O algoritmo *HeightBL*, em consultas aos k-vizinhos mais próximos, proveu redução do número de cálculos de distância de 13% a 52% e redução do tempo de consulta de 9% até 30%.

Tabela 5.2: Ganhos em números de cálculos de distância dos algoritmos de *bulk-loading* propostos em cada conjunto de dados para consultas por abrangência e aos k-vizinhos mais próximos, considerando a F-Onion-tree.

Consulta por Abrangência	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	32%	41%	28%	39%
Ozone	52%	97%	28%	96%
KDD Cup 2008	33%	93%	16%	99%

K-vizinhos mais próximos	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	25%	35%	13%	30%
Ozone	59%	70%	17%	52%
KDD Cup 2008	50%	58%	37%	40%

Tabela 5.3: Ganhos no tempo gasto pelos algoritmos de *bulk-loading* propostos em cada conjunto de dados para consultas por abrangência e aos *k*-vizinhos mais próximos, considerando a F-Onion-tree.

Consulta por Abrangência	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	17%	20%	10%	20%
Ozone	23%	89%	11%	77%
KDD Cup 2008	24%	83%	16%	99%

K-vizinhos mais próximos	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	24%	38%	9%	17%
Ozone	17%	53%	13%	30%
KDD Cup 2008	24%	35%	12%	15%

Esses resultados provam a aplicabilidade dos algoritmos *SampleBL* e *HeightBL*, que proveram bom desempenho no processamento de consultas mesmo em conjuntos de dados que apresentam grande volume de dados e alta dimensionalidade, superando o algoritmo de inserção um-a-um da F-Onion-tree.

A Tabela 5.4 detalha os ganhos de desempenho do algoritmo *SampleBL* em relação ao algoritmo *HeightBL*. Nesse sentido, um resultado positivo nessa tabela indica que o algoritmo *SampleBL* proveu melhores resultados de desempenho, enquanto que um resultado negativo indica que o algoritmo *HeightBL* proveu melhores resultados de desempenho. Pode-se perceber que o algoritmo *SampleBL* superou o algoritmo *HeightBL* na maioria dos casos. O único caso em que o algoritmo *SampleBL* foi superado pelo algoritmo *HeightBL* foi para o conjunto KDD Cup 2008, em que o algoritmo *SampleBL* obteve resultados inferiores para os raios de 1% a 5%. Em contrapartida, para os raios variando de 6% a 10%, o algoritmo *SampleBL* proveu melhores resultados de desempenho, os quais aumentaram conforme o raio também aumentou. A diferença de desempenho entre os dois algoritmos propostos se deve ao fato de o algoritmo *HeightBL* suspende a etapa de análise de pivôs ao encontrar o primeiro par de pivôs que satisfaça à altura estimada, enquanto que o algoritmo *SampleBL* percorre todos os pivôs da amostra até que o par de pivôs adequado seja encontrado. Em detalhes:

- Para a medida número de cálculos de distância, os ganhos de desempenho providos pelo algoritmo *SampleBL* com relação ao algoritmo *HeightBL* variaram da perda de 8% ao ganho de 64%.
- Para a medida tempo gasto, os ganhos de desempenho providos pelo algoritmo *SampleBL* com relação ao algoritmo *HeightBL* variaram da perda de 15% ao ganho de 67%.

Tabela 5.4: Comparação demonstrando os ganhos do algoritmo *SampleBL* em relação ao algoritmo *HeightBL* para a F-Onion-tree: (a) Número de cálculos de distância. (b) Tempo gasto.

(a) Cálculos de distância

Consulta por Abrangência	Mínima	Máxima
Color histograms	2%	15%
Ozone	5%	64%
KDD Cup 2008	-8%	7%

K-vizinhos mais próximos	Mínima	Máxima
Color histograms	6%	22%
Ozone	15%	57%
KDD Cup 2008	17%	30%

(b) Tempo gasto

Consulta por Abrangência	Mínima	Máxima
Color histograms	0%	11%
Ozone	13%	67%
KDD Cup 2008	-15%	33%

K-vizinhos mais próximos	Mínima	Máxima
Color histograms	13%	31%
Ozone	0%	42%
KDD Cup 2008	13%	25%

5.4.2 Processamento de Consultas usando a V-Onion-tree

As Tabelas 5.5 e 5.6 resumem o ganho mínimo e máximo em cálculos de distância e os valores mínimo e máximo de redução no tempo de processamento das consultas por similaridade que os algoritmos *SampleBL* e *HeightBL* proveram em relação à inserção um-a-um. Da mesma forma que discutido na seção 5.4.1, os algoritmos de *bulk-loading* propostos geraram índices que proveram melhor desempenho no processamento de consultas por similaridade do que os índices produzidos pela inserção um-a-um, para todos os conjuntos de dados.

Comparando cada algoritmo proposto individualmente com a inserção um-a-um, destacam-se os seguintes resultados:

- O algoritmo *SampleBL*, em consultas por abrangência, proveu redução do número de cálculos de distância de 40% a 95% e redução do tempo de consulta de 13% até 91%.
- O algoritmo *SampleBL*, em consultas aos *k*-vizinhos mais próximos, proveu redução do número de cálculos de distância de 37% a 86% e redução do tempo de consulta de 23% até 63%.
- O algoritmo *HeightBL*, em consultas por abrangência, proveu redução do número de cálculos de distância de 16% a 99% e redução do tempo de consulta de 9% até 99%.
- O algoritmo *HeightBL*, em consultas aos *k*-vizinhos mais próximos, proveu redução do número de cálculos de distância de 16% a 54% e redução do tempo de consulta de 10% até 60%.

Tabela 5.5: Ganhos em números de cálculos de distância dos algoritmos de *bulk-loading* propostos em cada conjunto de dados para consultas por abrangência e aos *k*-vizinhos mais próximos, considerando a V-Onion-tree.

Consulta por Abrangência	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	40%	82%	36%	77%
Ozone	41%	92%	40%	87%
KDD Cup 2008	61%	95%	16%	99%

K-vizinhos mais próximos	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	78%	86%	29%	49%
Ozone	37%	56%	16%	54%
KDD Cup 2008	64%	72%	37%	50%

Tabela 5.6: Ganhos no tempo gasto pelos algoritmos de *bulk-loading* propostos em cada conjunto de dados para consultas por abrangência e aos *k*-vizinhos mais próximos, considerando a V-Onion-tree.

Consulta por Abrangência	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	26%	55%	21%	50%
Ozone	13%	77%	9%	69%
KDD Cup 2008	27%	91%	16%	99%

K-vizinhos mais próximos	SampleBL (mínimo)	SampleBL (máximo)	HeightBL (mínimo)	HeightBL (máximo)
Color histograms	23%	33%	10%	16%
Ozone	33%	63%	29%	60%
KDD Cup 2008	33%	48%	30%	44%

Portanto, os resultados obtidos demonstram a aplicabilidade dos algoritmos *SampleBL* e *HeightBL* para realizar o *bulk-loading* da V-Onion-tree mesmo em conjuntos de dados com grande volume de dados e alta dimensionalidade, superando sempre o algoritmo de inserção um-a-um da V-Onion-tree.

A Tabela 5.7 detalha os ganhos de desempenho do algoritmo *SampleBL* em relação ao algoritmo *HeightBL*. Nesse sentido, um resultado positivo nessa tabela indica que o algoritmo *SampleBL* proveu melhores resultados de desempenho, enquanto que um resultado negativo indica que o algoritmo *HeightBL* proveu melhores resultados de desempenho. Assim como para a F-Onion-tree, o algoritmo *SampleBL* também, no geral, superou o desempenho do algoritmo *HeightBL* no que se refere ao processamento de consultas quando considerada a V-Onion-tree. O único caso em que o algoritmo *SampleBL* foi superado ocorreu para o conjunto Ozone, em que a perda foi de 4% em números de cálculos de distância para consultas aos k-vizinhos mais próximos. No entanto, essa perda ocorreu apenas para valores de k iguais a 2 e 4, sendo que para os demais valores, ou seja, para k variando de 6 a 20, o algoritmo *SampleBL* voltou novamente a superar os resultados do algoritmo *HeightBL*. Em detalhes:

- Para a medida número de cálculos de distância, os ganhos de desempenho providos pelo algoritmo *SampleBL* com relação ao algoritmo *HeightBL* variaram da perda de 4% ao ganho de 93%.
- Para a medida tempo gasto, os ganhos de desempenho providos pelo algoritmo *SampleBL* com relação ao algoritmo *HeightBL* variaram de 0% a 91%.

Tabela 5.7: Comparação demonstrando os ganhos do algoritmo *SampleBL* em relação ao algoritmo *HeightBL* para a V-Onion-tree: (a) Número de cálculos de distância. (b) Tempo gasto.

(a) Cálculos de distância

Consulta por Abrangência	Mínima	Máxima
Color histograms	3%	23%
Ozone	0%	55%
KDD Cup 2008	51%	93%

K-vizinhos mais próximos	Mínima	Máxima
Color histograms	64%	81%
Ozone	-4%	32%
KDD Cup 2008	27%	55%

(b) Tempo gasto

Consulta por Abrangência	Mínima	Máxima
Color histograms	0%	18%
Ozone	0%	50%
KDD Cup 2008	9%	91%

K-vizinhos mais próximos	Mínima	Máxima
Color histograms	12%	23%
Ozone	3%	14%
KDD Cup 2008	1%	14%

Por fim, comparando os resultados de desempenho providos pelos algoritmos de *bulk-loading* propostos para a F-Onion-tree e para a V-Onion-tree, destaca-se o seguinte. Os algoritmos *SampleBL* e *HeightBL*, em geral, apresentaram resultados similares para as duas estruturas. O principal motivo para isso é que, apesar do maior tempo de construção da V-Onion-tree, o novo método de particionamento desenvolvido para essa estrutura evitou que fossem aplicadas expansões desnecessárias aos nós do índice e, com isso, o processamento de consultas sobre a V-Onion-tree foi favorecido por evitar que regiões fossem desnecessariamente visitadas.

5.4.3 Análise Detalhada do Processamento de Consultas

As Figuras 5.12 a 5.14 mostram o número de cálculos de distância e o tempo gasto em segundos para processar as consultas por abrangência sobre os conjuntos Color Histograms, Ozone e KDD Cup 2008, respectivamente, usando a F-Onion-tree. Já as Figuras 5.15 a 5.17 mostram o número de cálculos de

distância e o tempo gasto em segundos para a F-Onion-tree no processamento de consultas aos *k*-vizinhos mais próximos para os mesmos conjuntos de dados.

Em detalhes, o algoritmo *SampleBL* para F-Onion-tree, quando comparado com a inserção um-a-um, teve o seguinte comportamento:

- Para o conjunto Color Histograms, observa-se que para consultas por abrangência o ganho de desempenho no número de cálculos de distância diminuiu de 41% para 31% à medida em que o raio aumentou, especialmente a partir do raio com valor 7%, enquanto que ganho de desempenho no tempo gasto foi aproximadamente constante de 20%. Em consultas aos *k*-vizinhos mais próximos, o ganho de desempenho no número de cálculos de distância variou de 30% a 35% à medida em que o valor de *k* aumentou. Já o ganho de desempenho no tempo gasto aumentou de 35% a 38% à medida que o valor de *k* variou de 2 a 8, e depois diminuiu para os valores restantes de *k* até atingir aproximadamente 25% de ganho.

- Para o conjunto Ozone, em consultas por abrangência observa-se que os ganhos de desempenho no número de cálculos de distância e do tempo gasto diminuíram à medida em que o valor do raio aumentou, variando de 97% a 52% para a medida cálculos de distância e de 88% a 23% para a medida tempo. Em consultas aos *k*-vizinhos mais próximos, os ganhos de desempenho no número de cálculos de distância (variação de 59% a 70%) e de tempo gasto (variação de 17% a 44%) aumentaram conforme o valor de *k* também aumentou, para valores de *k* variando de 2 a 6. Para os demais valores de *k*, os ganhos permaneceram aproximadamente constantes, e foram de 65% e 45%, respectivamente, para número de cálculos de distância e tempo gasto.

- Para o conjunto KDD Cup 2008, em consultas por abrangência, os ganhos de desempenho no número de cálculos de distância e do tempo gasto diminuíram à medida em que o valor do raio aumento, variando de 93% a 33% para a medida cálculos de distância e de 83% a 24% para a medida tempo. Em consultas aos *k*-vizinhos mais próximos, o mesmo comportamento foi observado, ou seja, à medida em que o valor de *k* aumentou, o ganho de desempenho do algoritmo *SampleBL* em relação à inserção um-a-um diminuiu, sendo a variação no número de cálculo de distância de 58% a 50% e no tempo gasto de 35% a 24%.

Já o algoritmo *HeightBL* para a F-Onion-tree apresentou o seguinte comportamento em relação à inserção um-a-um:

- Para o conjunto Color Histograms, em consultas por abrangência, o ganho de desempenho em números de cálculos de distância pouco variou (ou seja, de 32% a 29%) à medida em que o raio aumentou. Para o tempo gasto, os ganhos de desempenho variaram de 10% a 20% para raios de 1% a 7% e de 15% a 13% para raios de 8% a 10%. Em consultas aos *k*-vizinhos mais próximos, os ganhos de desempenho no número de cálculos de distância diminuíram de 20% a 14% à medida em que o valor de *k* aumentou. Já os ganhos de desempenho no tempo gasto variaram de 17% a 9%, sem porém apresentar um padrão.

- Para o conjunto Ozone, em consultas por abrangência, o ganho de desempenho no número de cálculos de distância diminuiu de 96% para 28% à medida em que o raio aumentou e, do mesmo modo, o ganho de desempenho no tempo gasto diminuiu de 75% a 11%. Em consultas aos

k-vizinhos mais próximos, à medida em que o valor de *k* aumentou, o ganho em número de cálculos de distância diminuiu de 52% para 17%. Já o ganho de desempenho no tempo gasto variou de 17% até 30% para valores de *k* variando de 2 a 8, e depois reduziu até a 13% à medida em que o valor de *k* aumentou.

- Para o conjunto KDD Cup 2008, em consultas por abrangência, à medida em que o raio aumentou, o ganho de desempenho no número de cálculos de distância diminuiu de 99% para 19%. O mesmo padrão também foi obervado para o ganho de desempenho no tempo gasto, o qual variou de 99% a 16%. Em consultas aos *k*-vizinhos mais próximos, à medida em que o valor de *k* aumentou, houve apenas uma pequena variação do ganho de desempenho no número de cálculos de distância, que foi de aproximadamente 40% para todos os valores de *k*. O mesmo padrão também foi observado para os ganhos no tempo gasto, os quais foram de aproximadamente 15%.

As Figuras 5.18 a 5.20 mostram o número de cálculos de distância e o tempo gasto em segundos para processar as consultas por abrangência para a V-Onion-tree. Por fim, as Figuras 5.21 a 5.23 mostram o número de cálculos de distância e o tempo gasto em segundos para processar as consultas aos *k*-vizinhos mais próximos.

Em detalhes, o algoritmo *SampleBL* para V-Onion-tree, quando comparado com a inserção um-a-um, teve o seguinte comportamento:

- Para o conjunto Color Histograms, em consultas por abrangência, à medida em que o raio aumentou, o ganho de desempenho no número de cálculos de distância reduziu de 82% para 40%, assim como o ganho de desempenho no tempo gasto reduziu de 50% para 26%. Em consulta aos *k*-vizinhos mais próximos, à medida em que o valor de *k* aumentou, os ganhos de desempenho diminuíram de 86% para 78% no número de cálculos de distância e de 33% para 23% no tempo gasto.

- Para o conjunto Ozone, em consultas por abrangência, à medida em que o raio aumentou, o ganho de desempenho no número de cálculos de distância diminuiu de 92% a 41%. Já o ganho de desempenho no tempo gasto aumentou de 67% para 77% quando o raio variou de 1% a 3%, e reduziu até 13% à medida em que o raio variou de 4% a 10%. Em consultas aos *k*-vizinhos mais próximos, à medida em que o valor de *k* aumentou, os ganhos de desempenho no número de cálculos de distância e tempo gasto diminuíram, variando de 52% a 37% e de 60% a 33%, respectivamente.

- Para o conjunto KDD Cup 2008, em consultas por abrangência, à medida que o raio aumentou, os ganhos de desempenho no número de cálculos de distância variaram de 95% a 61% e os ganhos de desempenho no tempo gasto variaram de 82% a 27%. Em consultas aos *k*-vizinhos mais próximos, à medida em que o valor de *k* aumentou, os ganhos de desempenho no número de cálculos de distância variaram de 72% a 64% e os ganhos de desempenho no tempo gasto variaram de 48% a 33%.

O algoritmo *HeightBL* para a V-Onion-tree apresentou o seguinte comportamento em relação à inserção um-a-um:

- Para o conjunto Color Histograms, em consultas por abrangência, à medida em que o raio aumentou, os ganhos de desempenho no número de cálculos de distância reduziram de 77% para 36%. Já os ganhos de desempenho no tempo gasto aumentaram de 40% para 50% para os raios de 1% a 3%, e reduziram até 21% para os valores de raio restantes. Em consultas aos k-vizinhos mais próximos, os ganhos de desempenho no número de cálculos de distância aumentou de 29% para 49% para valores de k de 2 a 12, e reduziu até 36% para os valores restantes de k. Já os ganhos de desempenho no tempo gasto foi variaram de 10% a 15% à medida em que k aumentou.

- Para o conjunto Ozone, em consultas por abrangência, à medida em que o raio aumentou, os ganhos de desempenho diminuíram de 87% a 40% no número de cálculos de distância e de 69% a 9% no tempo gasto. Em consultas aos k-vizinhos mais próximos, à medida em que o valor de k aumentou, os ganhos de desempenho diminuíram de 54% para 16% no número de cálculos de distância e de 56% para 29% no tempo gasto.

- Para o conjunto KDD Cup 2008, em consultas por abrangência, à medida em que o raio aumentou, os ganhos de desempenho diminuíram de 99% para 9% no número de cálculos de distância e de 99% para 16% no tempo gasto. Em consulta aos k-vizinhos mais próximos, à medida em que o valor de k aumentou, os ganhos de desempenho no número de cálculos de distância também aumentaram, variando de 38% a 50%, enquanto que os ganhos de desempenho no tempo gasto foi aproximadamente constante de 35%.

5.5 Considerações Finais

Neste capítulo foram investigadas as vantagens introduzidas pelos algoritmos de *bulk-loading* propostos. Nos testes realizados, foram considerados conjuntos de dados com diferentes dimensionalidades e volume de dados. Foram detalhados e discutidos resultados relacionados ao tamanho do índice, ao tempo gasto e o número de cálculos de distância para a construção do índice, e o tempo gasto e o número de cálculos de distância para o processamento de consultas por abrangência e aos k-vizinhos mais próximos. Nos testes, foram consideradas as duas versões da Onion-tree: a F-Onion-tree e a V-Onion-tree. Ademais, os resultados providos pelos algoritmos propostos foram comparados com os resultados providos pela inserção um-a-um.

Os testes de desempenho descritos na seção 5.1 comprovaram a complexidade teórica do algoritmo *GreedyBL*, o qual não pode ser usado na prática porque sua complexidade é elevada (cúbica) e, portanto, seu tempo de execuçãoé proibitivo para um MAM baseado em memória primária. Entretanto, esse algoritmo foi usado como base para a proposta dos demais algoritmos *SampleBL* e *HeightBL*.

Por outro lado, os testes de desempenho descritos nas seções 5.3 e 5.4 demonstraram diversas vantagens introduzidas pelo uso dos algoritmos *SampleBL* e *HeightBL*. Comparados com a inserção um-a-um, os algoritmos *SampleBL* e *HeightBL* proveram uma economia de espaço de armazenamento no tamanho do índice que variou de 9% a 88%; garantiram uma melhora de desempenho no processamento de consultas por abrangência que variou de 16% a 99% em números de cálculos de distância e de 9% até 99% no tempo gasto; e garantiram uma melhora de desempenho no processamento de consultas aos k-vizinhos mais próximos que variou de 13% até 86% em números de cálculos de distância e de 9% até 63% no tempo de processamento dessas consultas.

Os testes de desempenho descritos nas seções 5.3 e 5.4 também demostraram que, comparando o algoritmo *SampleBL* com o algoritmo *HeightBL*, o algoritmo *HeightBL* proveu, em geral, ganho de desempenho com relação ao tamanho dos índices. O algoritmo *HeightBL* gerou índices que garantiram uma redução de 2% a 58% com relação aos tamanhos dos índices gerados pelo algoritmo *SampleBL*. Em contrapartida, o algoritmo *SampleBL* proveu, em geral, melhores resultados de desempenho no processamento das consultas. Esses resultados variaram da perda de 8% ao ganho de 64% para cálculos de distância e da perda de 15% ao ganho de 67% no tempo gasto.

Comparando os resultados de desempenho da F-Onion-tree com os resultados de desempenho da V-Onion-tree, Os algoritmos *SampleBL* e *HeightBL* apresentaram ganhos aproximadamente iguais de desempenho em ambas estruturas. Em especial, o método de particionamento para V-Onion-tree descrito nesta dissertação evita que expansões desnecessárias sejam criadas para o índice, o que gera ganhos no processamento de consultas em relação à V-Onion-tree descrita em [Carélo et al., 2009], que apresentava resultados inferiores à F-Onion-tree.

Por fim, quanto aos resultados de desempenho relacionados à construção dos índices descritos na seção 5.2, os algoritmos *SampleBL* e *HeightBL* apresentaram resultados inferiores aos resultados providos pela inserção um-a-um, a qual foi de 12% a 87% mais rápida em relação aos algoritmos de *bulk-loading* propostos. Entretanto, esses resultados são compensados com relação à redução da quantidade de espaço de armazenamento e também pela redução no número de cálculos de distância e no tempo gasto para processar consultas por similaridade e aos k-vizinhos mais próximos.

No próximo capítulo são discutidas as conclusões desta dissertação de mestrado.

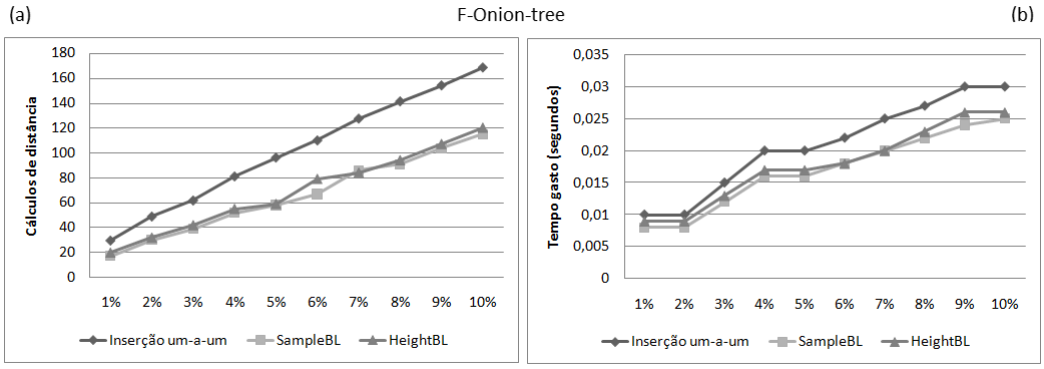

Figura 5.12: Processamento de consultas por abrangência sobre o conjunto de dados Color Histograms usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

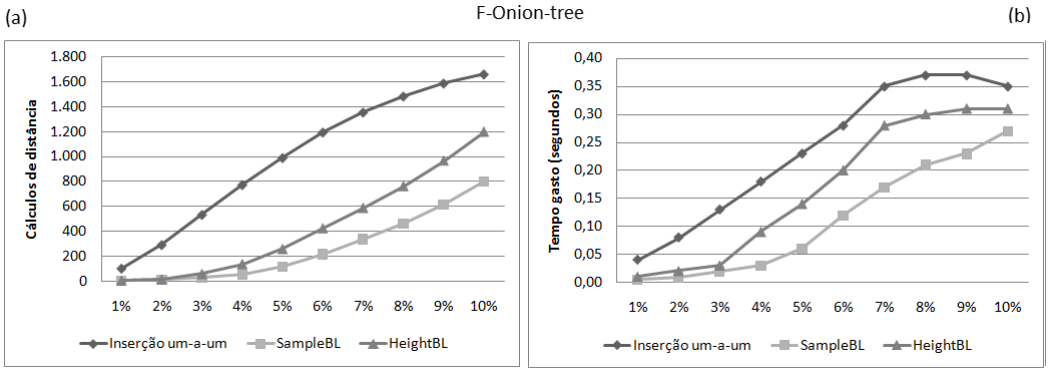

Figura 5.13: Processamento de consultas por abrangência sobre o conjunto de dados Ozone usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

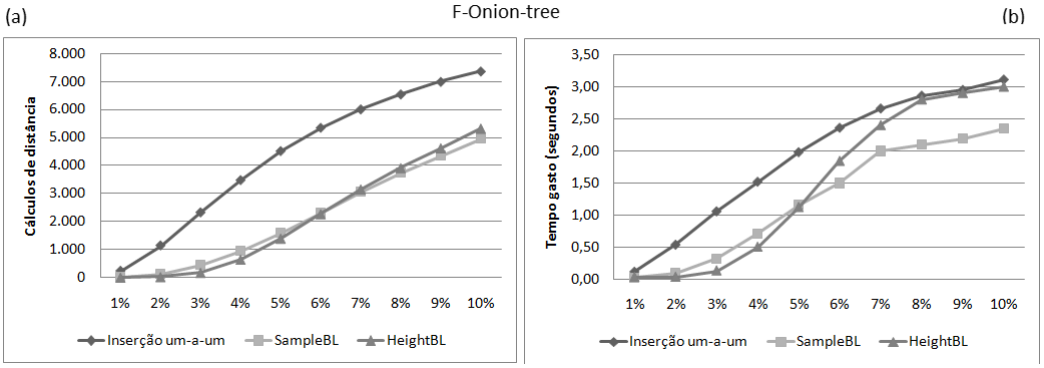

Figura 5.14: Processamento de consultas por abrangência sobre o conjunto de dados KDD Cup 2008 usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

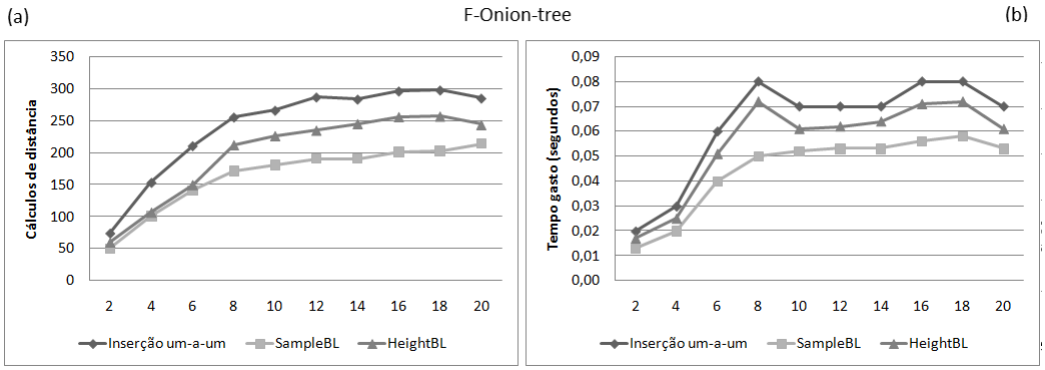

Figura 5.15: Processamento de consultas aos *k*-vizinhos mais próximos sobre o conjunto de dados Color Histograms usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

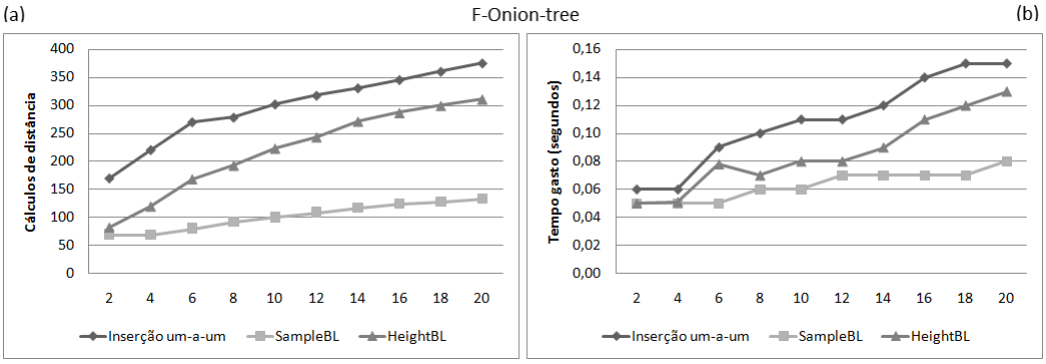

Figura 5.16: Processamento de consultas aos *k*-vizinhos mais próximos sobre o conjunto de dados Ozone usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

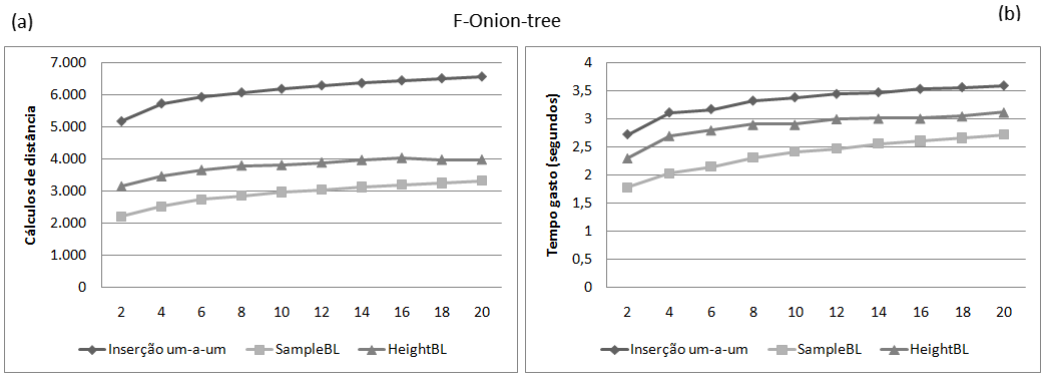

Figura 5.17: Processamento de consultas aos *k*-vizinhos mais próximos sobre o conjunto de dados KDD Cup 2008 usando a F-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

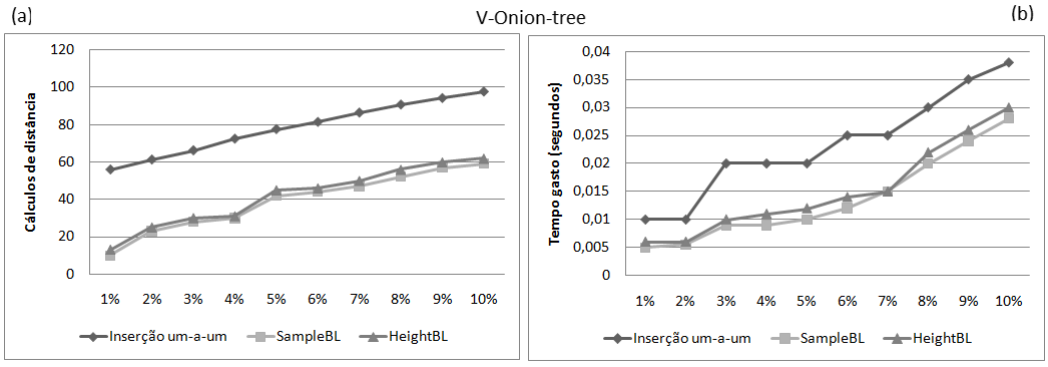

Figura 5.18: Processamento de consultas por abrangência sobre o conjunto de dados Color Histograms usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

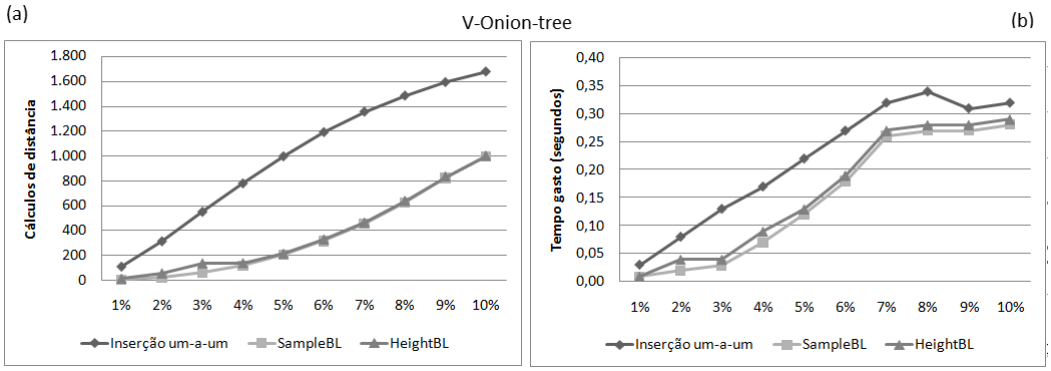

Figura 5.19: Processamento de consultas por abrangência sobre o conjunto de dados Ozone usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

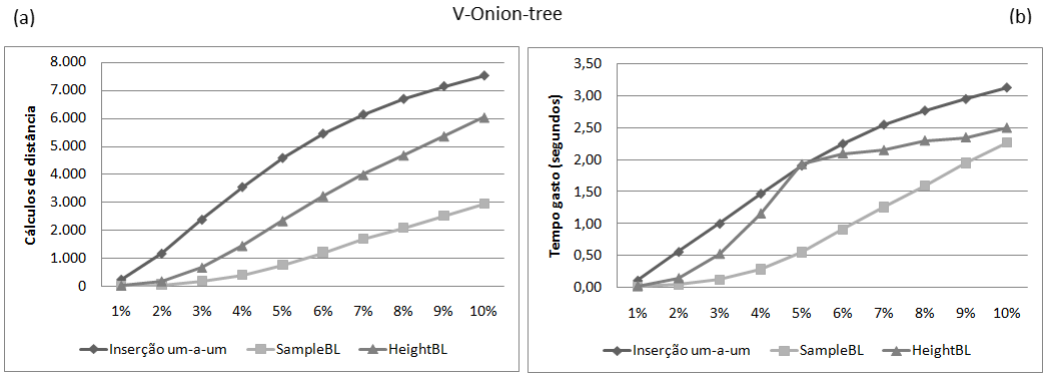

Figura 5.20: Processamento de consultas por abrangência sobre o conjunto de dados KDD Cup 2008 usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

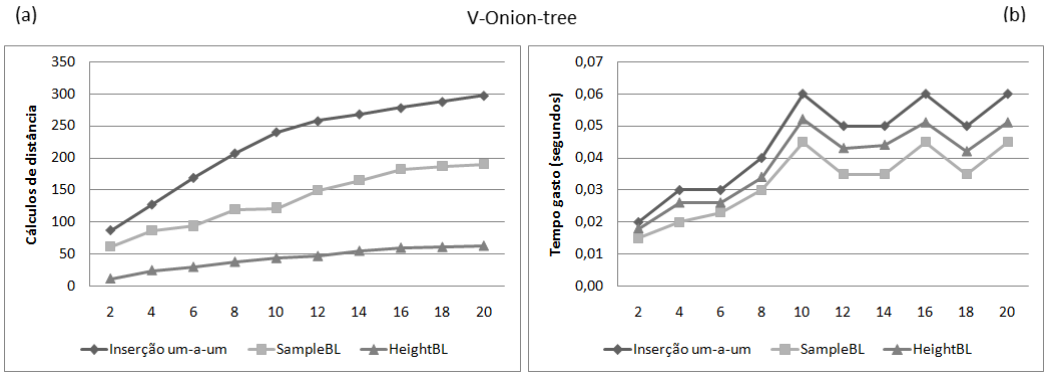

Figura 5.21: Processamento de consultas aos *k*-vizinhos mais próximos sobre o conjunto de dados Color Histograms usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

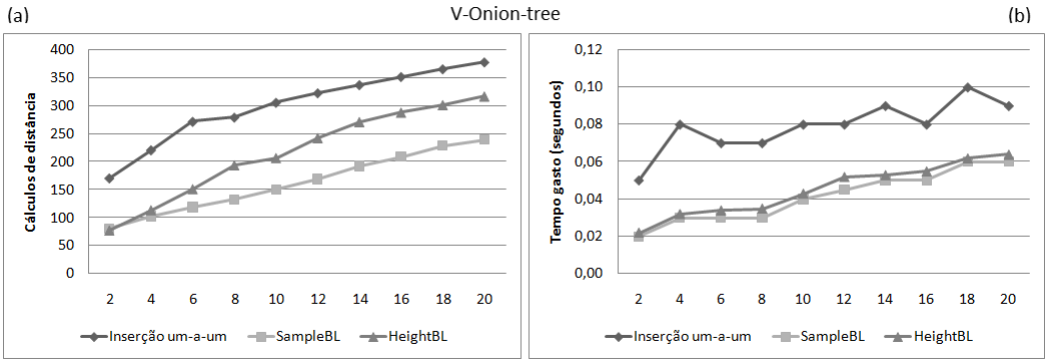

Figura 5.22: Processamento de consultas aos *k*-vizinhos mais próximos sobre o conjunto de dados Ozone usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

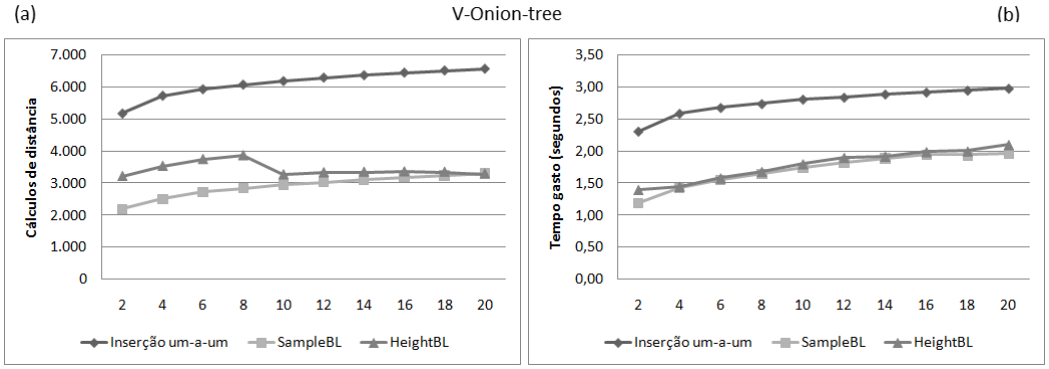

Figura 5.23: Processamento de consultas aos k-vizinhos mais próximos sobre o conjunto de dados KDD Cup 2008 usando a V-Onion-tree: inserção um-a-um, algoritmo *SampleBL* e algoritmo *HeightBL*. (a) Número de cálculos de distância. (b) Tempo gasto em segundos.

Capítulo 6

Conclusões

Nesta dissertação de mestrado foram propostos três algoritmos de *bulk-loading* para o MAM Onion-tree, que é o MAM baseado em memória primária mais eficiente para pesquisa por similaridade disponível na literatura. Em especial, os algoritmos propostos são os primeiros algoritmos de *bulk-loading* para esse MAM. Os experimentos mostraram que os algoritmos desenvolvidos superam o algoritmo de inserção um-a-um da Onion-tree nos critérios tamanho do índice, número de cálculos de distância e tempo gasto no processamento de consultas por abrangência e aos k-vizinhos mais próximos.

O primeiro algoritmo proposto, chamado *GreedyBL*, realiza a construção da estrutura no sentido *top-down*, verificando qual o melhor par de pivôs a ser escolhido para cada nó da estrutura. O algoritmo utiliza como premissa a construção de uma estrutura balanceada e, em cada nível, testa e avalia todos os pares de pivôs quanto à divisão uniforme dos elementos entre as regiões do espaço métrico. O algoritmo *GreedyBL* é baseado na abordagem gulosa e, devido à sua alta complexidade, requereu o maior tempo para a construção do índice quando comparado com os demais algoritmos propostos. Por outro lado, apresentou os melhores resultados no processamento das consultas, desde que gerou estruturas que garantiram uma boa divisão do espaço métrico.

O segundo algoritmo proposto, chamado *SampleBL*, introduz dois grandes diferenciais com relação ao algoritmo *GreedyBL*. O primeiro deles refere-se à etapa de amostragem, a qual é caracterizada por gerar amostras do conjunto de dados para serem testadas como possíveis pares de pivôs. O segundo diferencial refere-se à etapa de particionamento, a qual é direcionada especificamente à V-Onion-tree e é caracterizada por realizar o particionamento de cada nó dessa estrutura levando em consideração todos os elementos que podem ser nela inseridos. Nos testes de desempenho realizados, a redução no tempo de criação da estrutura é significativa, quando comparada à criação provida pelo algoritmo *GreedyBL*. Adicionalmente, o algoritmo *SampleBL* apresentou bons resultados relacionados ao desempenho do processamento de consultas, os quais foram muito próximos aos resultados providos pelo algoritmo *GreedyBL*.

O terceiro algoritmo proposto, chamado *HeightBL*, utiliza como premissa o fato de que a escolha de pares de pivôs de um determinado nível pode ser feita a partir de uma análise comparativa da altura máxima que uma subárvore pode possuir de acordo com a quantidade de elementos de entrada. Esse

algoritmo visa encontrar o par de pivôs que garante a melhor altura para a árvore. É o algoritmo de *bulk-loading* proposto que garante o melhor tempo de execução na construção do índice.

Detalhando os resultados obtidos nos testes de desempenho realizados para os algoritmos *SampleBL* e *HeightBL*, considerando o tamanho do índice gerado, a economia de espaço provida pelos algoritmos propostos foi de 9% a 88% quando comparado com o algoritmo de inserção um-a-um da Onion-tree. Em consultas por abrangência, a melhora de desempenho foi de 16% até 99% em números de cálculos de distância e de 9% até 99% no tempo de processamento dessas consultas. Em consultas aos k-vizinhos mais próximos, a melhora de desempenho foi de 13% até 86% em números de cálculos de distância e de 9% até 63% no tempo de processamento dessas consultas. Em comparação à inserção um-a-um, os algoritmos propostos apresentam resultados inferiores apenas no tempo e número de cálculos de distância para construir o índice, mas esse ponto é compensado na redução do tamanho do índice e na redução do tempo de e no número de cálculo de distâncias durante o processamento de consultas.

6.1 Contribuições

As principais contribuições desta dissertação de mestrado são:

- Desenvolvimento do algoritmo *GreedyBL*, o qual serve de base para a criação dos demais algoritmos propostos.
- Desenvolvimento do algoritmo *SampleBL*, que introduz a etapa de amostragem e um novo método de particionamento da V-Onion-tree.
- Desenvolvimento do algoritmo *HeightBL*, que introduz a abordagem de escolha de pares de pivôs por meio da estimativa da altura final da Onion-tree.
- Estudo comparativo entre o algoritmo de inserção um-a-um provido pela Onion-tree e os algoritmos de *bulk-loading* desenvolvidos, mostrando que os algoritmos de *bulk-loading* apresentaram vantagens no tempo de processamento de consultas, número de cálculos de distâncias e também no tamanho do índice gerado.

Como resultado desta dissertação de mestrado, foi submetido um artigo para o evento científico *17th East-European Conference on Advances in Databases and Information Systems (ADBIS 2013)* (classificação CAPES QUALIS Ciência da Computação B1), o qual encontra-se em fase de julgamento. Esse artigo contém os resultados dos algoritmos *HeightBL*. Também está sendo preparado um artigo estendido para ser submetido para a revista *Information Systems*, o qual contém todos os demais resultados descritos nesta dissertação.

6.2 Trabalhos Futuros

As propostas de trabalhos futuros são:

- Identificar os casos em que o algoritmo de *bulk-loading* para a V-Onion pode ser melhorado, implementar essas melhorias e aprimorar o seu algoritmo de particionamento da estrutura.

- Criar e implementar algoritmos de *bulk-loading* que considerem técnicas de clusterização para determinar os pivôs a serem testados.
- Desenvolvimento do algoritmo de *bulk-loading* da Onion-tree baseado em amostragem mas que realize a construção da estrutura no sentido *bottom-up*.
- Investigar o impacto de outras estratégias para encontrar o elemento medóide no algoritmo *SampleBL*.

Referências Bibliográficas

[Arge, 1995] Arge, L. (1995). The buffer tree: A new technique for optimal i/o-algorithms. In *University of Aarhus*, pages 334–345. Springer-Verlag.

[Arge et al., 1999] Arge, L., Hinrichs, K., Vahrenhold, J., and Vitter, J. S. (1999). Efficient bulk operations on dynamic r-trees. In *Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, ALENEX '99, pages 328–348, London, UK. Springer-Verlag.

[Bayer and McCreight, 1972] Bayer, R. and McCreight, E. M. (1972). Organization and maintenance of large ordered indices. *Acta Inf.*, pages 173–189.

[Bercken and Seeger, 2001] Bercken, J. V. d. and Seeger, B. (2001). An evaluation of generic bulk loading techniques. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 461–470, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Bercken et al., 1997] Bercken, J. V. d., Seeger, B., and Widmayer, P. (1997). A generic approach to bulk loading multidimensional index structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 406–415, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Bozkaya and Ozsoyoglu, 1997] Bozkaya, T. and Ozsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 357–368.

[Bueno, 2009] Bueno, R. (2009). *Tratamento do tempo e dinamicidade em dados representados em espaços métricos*. Phd thesis, Universidade de São Paulo.

[Burkhard and Keller, 1973] Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM (CACM)*, 16(4):230–236.

[Carelo, 2009] Carelo, C. C. M. (2009). Perseus: uma nova técnica para tratar árvores de sufixo persistentes. Master's thesis, University of São Paulo (USP).

[Carélo et al., 2011] Carélo, C. C. M., Pola, I. R. V., Ciferri, R. R., Traina, A. J. M., Jr, C. T., and Ciferri, C. D. A. (2011). Slicing the metric space to provide quick indexing of complex data in the main memory. *Information Systems*, 36(1):79–98.

[Carélo et al., 2009] Carélo, C. C. M., Pola, I. R. V., Ciferri, R. R., Traina, A. J. M., Traina-Jr., C., and Ciferri, C. D. A. (2009). The Onion-tree: quick indexing of complex data in the main memory. In *Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems*, pages 235–252.

[Chávez et al., 2001] Chávez, E., Navarro, G., Baeza-Yates, R. A., and Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys (CSUR)*, 33(3):273–321.

[Ciaccia and Patella, 1998] Ciaccia, P. and Patella, M. (1998). Bulk loading the m-tree. In *In Proceedings of the 9th Australasian Database Conference (ADC'98*, pages 15–26.

[Ciaccia et al., 1997] Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 426–435, Athens, Greece.

[Ciferri, 2002] Ciferri, R. R. (2002). *Distribuição dos dados em ambientes de data warehousing: o sistema WebD2W e algoritmos voltados à fragmentação horizontal dos Dados*. Phd thesis, Universidade Federal de Pernambuco.

[Comer, 1979] Comer, D. (1979). The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137.

[Gaede and Günther, 1998] Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231.

[Gallager et al., 1983] Gallager, R. G., Humblet, P. A., and Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77.

[García R et al., 1998] García R, Y. J., López, M. A., and Leutenegger, S. T. (1998). A greedy algorithm for bulk loading r-trees. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, GIS '98, pages 163–164, New York, NY, USA. ACM.

[Guttman, 1984] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, USA.

[Henrich et al., 1989] Henrich, A., werner Six, H., Hagen, F., and Hagen, F. (1989). The lsd tree: spatial access to multidimensional point and non-point objects. pages 45–53.

[Hjaltason and Samet, 2003] Hjaltason, G. R. and Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 28(4):517–580.

[Lee and Lee, 2003] Lee, T. and Lee, S. (2003). Omt: Overlap minimizing top-down bulk loading algorithm for r-tree. In Eder, J. and Welzer, T., editors, *The 15th Conference on Advanced Information Systems Engineering (CAiSE 03), Klagenfurt/Velden, Austria, 16-20 June, 2003, CAiSE Forum, Short Paper Proceedings, Information Systems for a Connected Society*, volume 74 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Lin et al., 1994] Lin, K., Jagadish, H. V., and Faloutsos, C. (1994). The tv-tree – an index structure for high-dimensional data. *VLDB Journal*, 3:517–542.

[Marrach, 2011] Marrach, D. G. R. (2011). *Investigando A Remoção De Elementos No Método De Acesso Métrico Indexação Onion-Tree*. Qualificação de mestrado, Universidade de São Carlos, UFSCAR.

[Pola, 2005] Pola, I. (2005). *Indexação em domínios métricos generalizáveis*. Mater's thesis, Universidade de São Paulo.

[Pola, 2010] Pola, I. (2010). *Explorando conceitos da teoria de espaços métricos em consultas por similaridade sobre dados complexos*. Phd thesis, Universidade de São Paulo.

[Pola et al., 2007] Pola, I. R. V., Traina-Jr, C., and Traina, A. J. M. (2007). The MM-tree: A memory-based metric tree without overlap between nodes. In *Proceedings of the 12th East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 157–171.

[Smeulders et al., 2000] Smeulders, A. W. M., Worring, M., Santini, S., Gupta, A., and Jain, R. (2000). Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1349–1380.

[Talavera, 1999] Talavera, L. (1999). Feature selection as a preprocessing step for hierarchical clustering. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 389–397, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Traina-Jr et al., 2000] Traina-Jr, C., Traina, A. J. M., Seeger, B., and Faloutsos, C. (2000). Slim-trees: high performance metric trees minimizing overlap between nodes. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT)*, pages 51–65, Konstanz, Germany.

[Uhlmann, 1991] Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

[Vespa et al., 2010] Vespa, T. G., Jr, C. T., and Traina, A. J. (2010). Efficient bulk-loading on dynamic metric access methods. *Information Systems*, 35(5):557 – 569. Twenty-second Brazilian Symposium on Databases (SBBD 2007).

[Wilson and Martinez, 1997] Wilson, D. R. and Martinez, T. R. (1997). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research (JAIR)*, 6:1–34.

[Yianilos, 1993] Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321.