Operadores binários para consulta de similaridade em banco de dados multimídia

Enzo Seraphim

Data de Depósito:

Assinatura: _____

Operadores binários para consulta de similaridade em banco de dados multimídia¹

Enzo Seraphim

Orientador: *Prof. Dr. Caetano Traina Júnior*

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, para obtenção do título de Doutor em Ciências de Computação e Matemática Computacional.

USP - SÃO CARLOS
Outubro de 2005

¹Este trabalho conta com o apoio financeiro da FAPESP - Processo 00/12026-4

À minha esposa Thatyana.
À minha mãe Malvina.
Em memória do meu pai Irineu.

“A mente que se abre a uma nova idéia
jamais voltará ao seu tamanho original.”
Albert Einstein

Agradecimentos

Ao Senhor Deus que a mim concedeu toda graça e inspiração para a realização desse trabalho.

Ao meu orientador, Prof. Dr. Caetano Traina Jr pelo apoio, confiança e orientação concedida, sem a qual seria impossível a realização desse trabalho. Obrigado por me fazer compreender o verdadeiro papel de um pesquisador dentro da comunidade e certo que ainda realizaremos muitos outros trabalhos juntos.

A meu pai, agradeço por ter me passado toda sua sabedoria e experiência de vida durante nossa convivência.

A minha mãe que um dia quando garoto, indaguei-a sobre o que gostaria que eu fosse. Ela respondeu com toda simplicidade: “Seja um doutorzinho”.

A minha esposa Thatyana, agradeço seu amor, dedicação e compreensão de todos os momentos bons e ruins vividos juntos durante todo esse trabalho. “Você é a mulher da minha vida”.

À Profa. Dra. Agma Juci Machado Traina pelo conhecimento e ajuda oferecida nestes anos de trabalho no Grupo de Base de Dados e Imagens (GBDI).

Aos membros do GBDI, obrigado pela colaboração, amizade durante este doutorado.

Um agradecimento especial aos casais Humberto e Camila, Walter e Cláudia que proporcionaram momentos divertidíssimos nas partidas de bozó.

A meus tios e tias, padrinho e madrinha, irmãos Cláudio e Marcos, e às irmãs Elza, Léo, Soraia e Silvia, e demais familiares. Muito obrigado pelo carinho e amor dedicado a minha pessoa.

Aos meus amigos que torceram por mim, mesmo não estando perto: Edílson, Guilherme, Fabiano, Ericson e Luis. Vocês sempre estarão na minha memória.

A todas as pessoas que, direta ou indiretamente, contribuíram para que eu chegasse até aqui.

À FAPESP, pelo apoio financeiro.

Sumário

Resumo	vii
<i>Abstract</i>	viii
1 Introdução	1
1.1 Motivação	2
1.2 Objetivo	2
1.3 Organização do Trabalho	4
2 Consultas por Similaridade	5
2.1 Modelo Espaço Métrico	6
2.2 Modelo de Espaço Vetorial	7
2.3 Tipos de Consulta por Similaridade	8
2.4 Método de Acesso Espacial	10
2.4.1 <i>R-Tree</i>	10
2.5 Métodos de Acesso Métrico	12
2.5.1 Técnicas de Burkhard e Keller	12
2.5.2 <i>GH-Tree</i>	14
2.5.3 <i>VP-Tree</i>	15
2.5.4 <i>MVP-Tree</i>	16
2.5.5 <i>M-Tree</i>	17
2.5.6 <i>Slim-Tree</i>	19
2.5.7 <i>Omni-Family</i>	20
2.5.8 Estrutura de Particionamento de Espaço (<i>VA-File</i>)	23
2.5.9 Estrutura <i>D-Index</i>	24
2.6 Conclusões	25

3	Junção	27
3.1	Tipos de Junção	27
3.2	Implementações de Junção	30
3.2.1	Junção <i>Nested-Loops</i>	30
3.2.2	Junção <i>Sort-Merge</i>	31
3.2.3	Métodos de Junção <i>Hash</i>	32
3.3	Junção Espacial	35
3.3.1	Junção Espacial para <i>R-Trees</i>	36
3.4	Junção por Similaridade	40
3.4.1	Junção por Abrangência	41
3.4.2	Junção por Vizinhos mais Próximos	41
3.4.3	Junção por Proximidade	42
3.5	Conclusão	47
4	Coordenadas Métricas	48
4.1	Coordenadas Métricas	49
4.2	Ilustração do conceito	50
4.2.1	Extensibilidade	52
4.3	Métodos de Acessos Métricos	54
4.3.1	Partição Local	54
4.3.2	Consulta por Abrangência	58
4.3.3	Consulta aos Vizinhos mais Próximos	59
4.3.4	Partição Local+	60
4.4	Experimentos	62
4.4.1	Conjunto de Proteínas	62
4.4.2	Conjunto de Palavras	73
4.4.3	Conjunto de Pontos Geográficos	82
4.5	Escalabilidade	92
4.6	Conclusões	95
5	Junção por Similaridade	97
5.1	Junção por Similaridade Sequencial	97
5.1.1	Junção por Abrangência	98
5.1.2	Junção por Vizinhos mais Próximos	98

5.1.3	Junção por Proximidade	99
5.2	Junção por Similaridade em Região de Cobertura	100
5.2.1	Junção por Abrangência	100
5.2.2	Junção por Vizinhos mais Próximos	104
5.2.3	Junção por Proximidade	105
5.3	Junção por Similaridade por Particionamento	105
5.3.1	Junção por Abrangência	105
5.3.2	Junção por Vizinhos mais Próximos	108
5.3.3	Junção por Proximidade	109
5.4	Técnicas usando Árvore AVL para Lista Global de Resposta	110
5.5	Experimentos	112
5.6	Conclusão	122
6	Conclusões	123
6.1	Resultados	123
6.2	Trabalhos Futuros	125
	Referências Bibliográficas	127

Lista de Figuras

2.1	Representação dos pontos situados à distância r a partir do objeto s_q , considerando as diferentes métricas da família L_p e exemplos de consultas por abrangência utilizando essas métricas.	8
2.2	Exemplos de consultas por similaridade	9
2.3	Representação Estrutural de uma R -Tree	11
2.4	Decomposição do espaço na GH -Tree [1]	14
2.5	Particionamento de um uma VP -Tree	15
2.6	Descarte de Objetos com o uso da desigualdade triangular	19
2.7	Representação de uma $Slim$ -tree com seus nós índices e folhas	21
2.8	Particionamento ao Meio por Exclusão	25
3.1	Dois Conjuntos de Retângulos e sua Projeção no Eixo X [2]	37
3.2	Exemplo para o Teste de Interseção Ordenada [2]	39
3.3	Ordenação espacial usando Ordenação-Z [2]	40
3.4	Diferença entre as Operações de Junção por Similaridade [3]	40
4.1	Particionamento com 2 mapeadores	50
4.2	Particionamento com 3 Mapeadores	51
4.3	Objeto Mapeado para a região 0, 1, 2	52
4.4	Extensibilidade das Coordenadas Métricas	52
4.5	$Bucket$ Índice para Partição Local	54
4.6	$Bucket$ Folha para Partição Local	56
4.7	Regiões qualificadas na Consulta por Abrangência	58
4.8	Coordenada que encontra-se o objeto de consulta q	60
4.9	$Bucket$ Folha para Partição Local+	61
4.10	Matrix MPAM	63
4.11	Gráficos da Média de Números de Cálculos de Distância - Base UNIPROT	66

4.12	Gráficos da Média de Números de Acessos a Disco - Base UNIPROT	68
4.13	Gráficos da Média de Tempo de Execução - Base UNIPROT	71
4.14	Gráficos da Média do Número de Cálculos de Distância - Base <i>OpenOffice</i>	76
4.15	Gráficos da Média de Números de Acessos a Disco - Base <i>OpenOffice</i>	78
4.16	Gráficos da Média de Tempo de Execução - Base <i>OpenOffice</i>	80
4.17	Gráficos da Média de Número de Cálculos de Distância - Base de GEONet	85
4.18	Gráficos da Média do Número de Acessos a Disco - Base GEONet	88
4.19	Gráficos da Média de Tempo de Execução - Base GEONet	91
4.20	Gráficos de Escalabilidade variando o Tamanho de Blocos (1ª coluna) e variando o Número de Objetos (2ª coluna)	93
5.1	Regiões que se sobrepõem entre duas <i>Slim-Tree</i> : (a) Regiões sem raio de consulta e (b) Regiões com raio de consulta	101
5.2	Poda através da Desigualdade Triangular	102
5.3	Regiões que se sobrepõem entre duas estruturas Partição Local: (a) Regiões sem raio de consulta e (b) Regiões com o raio de consulta aumentando o número de cobertura de cada nó	108
5.4	Gráficos de acessos a disco e de cálculos de distância para junção por abrangência entre Argentina e Brasil	115
5.5	Gráficos de tempo de execução para junção por abrangência entre Argentina e Brasil	116
5.6	Gráficos de acessos a disco e de cálculos de distância para junção por abrangência entre Equador e Peru	117
5.7	Gráficos de tempo de execução para junção por abrangência entre Equador e Peru . .	118
5.8	Gráficos do Número de Acessos a Disco, Número de Cálculos de Distância e o Tempo de Execução para junção aos vizinhos mais próximos entre Equador e Brasil	119
5.9	Gráficos do Número de Acessos a Disco, Número de Cálculos de Distância e o Tempo de Execução para junção por proximidade entre Peru e Brasil	120

Lista de Tabelas

4.1	Tamanho do <i>Bucket</i> Índice sem Objetos Mapeadores	55
4.2	Informações sobre as Estruturas do Conjunto de Proteínas	64
4.3	Informações sobre as Estruturas do Conjunto de Palavras	74
4.4	Informações sobre as Estruturas do Conjunto Pontos Geográficos	82
4.5	Informações sobre as Estruturas do conjunto GEONet	92
4.6	Informações sobre as Estruturas do conjunto GEONet	95
5.1	Características das Estruturas usadas na Junção da Base GEONet	113

SERAPHIM, E. *Operadores binários para consulta de similaridade em banco de dados multimídia*, São Carlos, 2005. 133 p. Tese de Doutorado - Instituto de Ciências Matemáticas e de Computação - ICMC, USP.

RESUMO

Os atuais gerenciadores de banco de dados não são adequados para manipulação de dados complexos; e entre eles destacamos os dados multimídia que, para agilizar as consultas usam a operação de igualdade sobre as estruturas de indexação. Operações de igualdade são pouco utilizadas em operações que envolvem dados complexos, uma vez que, a existência de dois elementos extremamente iguais é rara. Uma classe de operadores que se adequa melhor para manipulação desses dados são os operadores por similaridade. Exemplo de operadores de seleção por similaridade são a consulta por abrangência (*range queries*) e consulta aos vizinhos mais próximos. Exemplificando, o operador de seleção aos vizinhos mais próximos responde a consultas como, “selecione as cinco proteínas mais parecidas pelo alinhamento da proteína Sparc (responsável pelo câncer de pele)”. Existem muitos trabalhos desenvolvidos no sentido de prover operadores de seleção por similaridade envolvendo estruturas baseadas em árvores. Entretanto, poucos estudos têm sido realizados envolvendo a utilização de operadores diferentes da seleção, por exemplo, a junção. Um operador de junção compara pares de objetos de elementos pertencentes ao domínio dos dados, ao passo que um operador de seleção recebe uma constante para a comparação dos elementos. Podemos ter assim, três operadores de junção por similaridade: operadores de junção por abrangência, por vizinhos mais próximos e sobre os pares de vizinhos mais próximos. Exemplificando, uma consulta utilizando junção por abrangência responde a consultas do tipo: “Selecione as proteínas contidas no vírus da Hepatite B que diferem em até duas unidades de alinhamento das contidas no vírus da Hepatite C”. Este trabalho apresenta um novo método de acesso métrico em extrema quantidade de dados bem como, formas de implementação das formas de junção em estruturas métricas.

Abstract

The present databases managers are not adequated for complex data manipulation and among them we point out the multimedia data that to speed up the query use the equality operation on the index structure. Equality operations are not much used in operations that involve complex data sence the presence of two elements extremely equal is rare. An operator class that better manipulates these data are the similarity operators. Selection operators by similarity are the range query and the nearest neighbors query. For example, the selection operator to the nearest neighbors answers the queries like: “select five proteins more similar by the alignment of the Sparc protein (responsible for the skin cancer)”. There are many works developed to provide similarity by selection operators envvolving structures based in trees. However, few studies have been done involving the use of different operators selection, for example, the join. A join operator compares pairs of objects of the elements belonging to the domain of the data, whereas a selection operator receives a constant to make the comparison of the elements. We can have three similarity join operators: the range join operators, the nearest neighbor and the closest neighbors pair. For instance, a query using the range join answers these kind of queries: “Select the proteins restrained in the Hepatitis B virus that differ up to two unities from the alignment of the protein found in the Heapatitis C virus”. This work presents a new metric access method with an extreme amount of data as well as implementations forms of the join in metric structures.

CAPÍTULO 1 Introdução

Os atuais gerenciadores de bancos de dados foram desenvolvidos inicialmente para manipular domínios de dados numéricos e/ou pequenas seqüências de caracteres (palavras) aproveitando a relação de ordem total que existe entre os elementos do domínio para executar as operações de consulta e atualização dos dados. A estrutura de indexação que tem sido usualmente adotada pelos gerenciadores de banco de dados para recuperar os dados é a *B-Tree* [4].

O surgimento de novas formas de captura de informação e o aumento da capacidade de armazenamento possibilitaram que se armazenem grandes quantidades de outros tipos de dados tais como, séries temporais, seqüências de proteínas, imagens, áudio, textos longos, entre outros. Esses dados são chamados de dados complexos, pois não têm uma relação de ordem entre seus elementos. A recuperação desses dados não é atendida de maneira adequada pelas estruturas de indexação tradicionais.

Para dar suporte às novas operações sobre esses dados, sugeriram novas estruturas de indexação, como por exemplo, as estruturas temporais, onde existem relações de tempo entre seus elementos, tais como: *Time Index* [5], *NR+-Tree* [6], *B+-Trees* para Dados Temporais [7], *GR-Tree* [8], entre outras.

Um outro tipo de dados é o que apresenta a noção de dimensão, onde os elementos podem ser projetados em um espaço de dimensões menores ou, podem ter seus valores em nova dimensão restrita pelos valores em outra dimensão (definindo hiper-volumes) e também existe a noção de distância entre os elementos. Neste caso, todos os elementos devem possuir a mesma dimensão. As estruturas que indexam dados desse domínio são amplamente descritas na literatura, e são chamadas de Métodos de Acessos Espaciais (*SAM - Spatial Access Methods*) [9]. O desenvolvimento dessas estruturas iniciou-se com o trabalho pioneiro sobre as *R-Trees* descrito em [10] e, prosseguiu com numerosas variações e adaptações [11, 12, 13, 14].

1.1 Motivação

Existem conjuntos de dados em que a única relação existente entre os elementos é a similaridade. A similaridade é definida através de uma função de distância, ou função de "dissimilaridade", $d(O_i, O_j)$, que retorna zero se ambos os objetos O_i e O_j forem idênticos ou um valor positivo que aumenta quanto maior for a distância (ou dissimilaridade) entre os objetos.

Um fato que justifica a escolha das estruturas métricas para o desenvolvimento desse projeto é que as operações de comparação por igualdade têm pouca utilidade nas operações envolvendo dados complexos, uma vez que a existência de dois elementos exatamente iguais é rara [15]. Nesses casos, o grau de similaridade entre os dados é o fator mais importante. Além disso, o conjunto dos dados complexos não respeitam a relação de ordem total.

Existem basicamente dois operadores de seleção por similaridade: consulta por abrangência (*range queries*) e consulta por vizinhos mais próximos (*k-nearest neighbors queries*) [16]. Uma consulta por abrangência recebe como parâmetros um elemento do domínio de dados (objeto da consulta) e um grau de dissimilaridade, e o resultado são todos os elementos da base de dados que diferem do objeto da consulta até no máximo o grau de dissimilaridade indicado. Um exemplo de consulta por abrangência em uma base de dados de seqüências genéticas é “*selecione as proteínas que diferem da proteína Sparc (responsável pelo câncer de pele) por até duas unidades de alinhamento*”.

Uma consulta por vizinhos mais próximos recebe como parâmetros um elemento do domínio de dados (o objeto da consulta) e uma quantidade K , e obtém como resultado a quantidade K de elementos da base de dados que sejam os mais próximos do objeto da consulta. Um exemplo de consulta por vizinhos mais próximos em uma base de dados de seqüências genéticas é “*selecione as cinco proteínas mais semelhantes à proteína Sparc pelo alinhamento*”. Pode-se considerar que ambos os operadores são equivalentes ao operador de seleção em domínios de dados que dão suporte a relações de ordem total. Ambas as operações recebem um objeto de consulta, e retornam um conjunto de objetos como resposta [17, 18, 19].

1.2 Objetivo

Para suportar as consultas por similaridade surgiram várias estruturas de dados métricas sendo que, algumas são baseadas em árvores implementadas em memória primária como *GH-Tree* [1], *VP-Tree* [20] e *MVP-Tree* [21, 22]. Outras árvores são implementadas em memória secundária e baseadas em regiões de cobertura como *M-Tree* [23] e *Slim-Tree* [24]. Outras estruturas implementadas em memória secundária são baseadas em particionamento disjunto de regiões como *SH* (*Similarity Hashing*)

[25].

Essas estruturas foram desenvolvidas no sentido de prover operadores de seleção por similaridade em gerenciadores para dar suporte a dados multimídia. No entanto, poucos estudos têm sido voltados a consultas por abrangência e vizinhos mais próximos em operadores diferentes da seleção, como por exemplo a junção.

Como explicado acima, um operador de junção difere de um operador de seleção, pois o operador de seleção recebe uma constante para a comparação, e um operador de junção compara pares de objetos pertencentes a um mesmo domínio de dados, um de cada conjunto R e S . Existem três operadores de junção por similaridade: operadores de junção por abrangência, operadores de junção por vizinhos mais próximos e operadores de junção por proximidade.

A junção por abrangência tem como resposta todos os pares de objetos, um de cada conjunto, que estão distantes entre si por no máximo uma distância máxima de busca ε .

A junção de vizinhos mais próximos tem como resposta para cada objeto do conjunto R , uma quantidade de k objetos mais próximos do conjunto S . Em caso de empate de distâncias na maior distância obtida, pode-se selecionar apenas o número necessário de objetos que satisfazem a regra, ou criar uma lista de empates.

A junção por proximidade (K -CN (K -Closest Neighbors Join)) tem como resposta os k pares de objetos mais próximos entre os objetos do conjunto R para os objetos do conjunto S . Esse também pode criar uma lista em caso de empates de distâncias.

O objetivo deste trabalho é implementar esses operadores de junção nas estruturas métricas baseadas em região de cobertura, como *M-Tree* e *Slim-Tree*.

Um problema que vem sendo tratado nessas estruturas é a sobreposição entre regiões de um mesmo nível. Essa sobreposição prejudica os algoritmos de consulta e trabalhos como *Slim-Tree* [24] tentam minimizar essa sobreposição em regiões de um mesmo nível.

Esse trabalho também explorou o problema da sobreposição de regiões de um mesmo nível, definindo uma nova estrutura baseada em particionamento disjunto de regiões, chamado de coordenada métrica.

As implementações das operações de junção possibilitam verificar a eficiência desses algoritmos de junção nas estruturas baseadas em regiões de cobertura e nas estruturas baseadas em particionamento disjunto de regiões.

1.3 Organização do Trabalho

No capítulo 2 são apresentadas algumas consultas por similaridade que avaliam o grau de dissimilaridade entre os objetos e técnicas de indexação para melhorar o desempenho das consultas por similaridade. O capítulo 3 apresenta a operação de junção, os tipos de junção e as suas possíveis implementações. Um novo conceito de particionamento extensível do espaço métrico, chamado de coordenadas métricas é apresentado no capítulo 4. O capítulo 5 apresenta a junção por similaridade e seus algoritmos desenvolvidos no trabalho. As conclusões e trabalhos futuros são apresentados no capítulo 6.

CAPÍTULO 2 Consultas por Similaridade

Os Sistemas de Gerenciamento de Bases de Dados (SGBDs) tradicionais realizam buscas sobre **dados convencionais** (números e textos curtos) por meio de relações de igualdade e de ordem existentes nesses dados. Entretanto, quando se utilizam SGBDs para armazenar **dados complexos**, como por exemplo dados multimídia (imagens, sons, etc), as buscas por igualdade ou por intervalo não se aplicam, ou simplesmente têm pouca utilidade. Exames de tomografia são um bom exemplo disso. Dificilmente a tomografia do cérebro de um paciente é idêntica à de outro paciente, ou até mesmo duas tomografias do próprio paciente tomadas com alguns instantes de separação entre elas. Neste caso, o grau de similaridade entre os dados complexos é o fator mais importante [26].

O tipo de busca denominado **consulta por similaridade** consiste em procurar em um conjunto por objetos que, segundo algum critério de similaridade, sejam mais “parecidos” ou mais “distintos” com/de um determinado objeto. Em outras palavras, consultas por similaridade comparam todos os objetos do conjunto com um objeto escolhido, selecionando apenas os elementos que atendam a um certo critério de similaridade. Por exemplo, procurando dar um diagnóstico mais preciso sobre o tumor no cérebro de um paciente, e até mesmo tentando encontrar subsídios para decidir como encaminhar o tratamento, um médico estaria interessado em encontrar as tomografias de cérebro que apresentem um tumor semelhante àquele em estudo.

Para se determinar a similaridade entre os objetos, eles podem ser comparados diretamente ou a partir de vetores de características (cor, forma e textura por exemplo) extraídos dos mesmos. Nos dois casos, uma função de distância $d()$ é definida para calcular o grau de similaridade, retornando um valor numérico que quantifica o quão similares dois objetos são. Por definição, esses valores são sempre maiores ou iguais a zero, tendo valores próximos de zero para objetos muito similares (zero para objetos iguais) e valores maiores para objetos menos similares. Essa função é chamada de **função de dissimilaridade** ou de **função de distância** e deve sempre ser definida por um especialista no domínio em questão. Na realidade, a função de distância indica a dissimilaridade entre dois objetos,

e convém colocar que a similaridade é medida pelo seu inverso.

Existem vários tipos de consultas por similaridade, sendo a consulta por abrangência e a consulta aos k -vizinhos mais próximos as mais comuns. Pesquisadores possuem grande interesse na elaboração de algoritmos de busca para executar consultas por similaridade. Dois principais modelos relacionados com o domínio dos dados onde as consultas serão efetuadas têm sido propostos e estudados [9, 27, 28]: modelo de espaço métrico e modelo de espaço vetorial.

A seção 2.1 apresenta uma discussão sobre espaço métrico [20, 21, 23, 27, 29] e na seção 2.2 é apresentada uma discussão sobre o espaço vetorial. A seção 2.3 apresenta os tipos de consultas por similaridade. Nas seções 2.4 e 2.5 são apresentados os principais métodos de acesso espacial e métrico.

2.1 Modelo Espaço Métrico

Para alguns domínios a extração de vetores de características de dimensão definida pode ser uma tarefa muito complicada, ou até inviável, como no caso em que os tipos de características variam para cada objeto. Isso ocorre, por exemplo, com impressões digitais, pois cada impressão tem um conjunto próprio de características: deltas, curvatura e espaçamento dos sulcos, sumidouros, entre outras.

Essa restrição ocorre com frequência quando a noção de similaridade é complexa e altamente dependente do domínio [30]. No entanto, se ainda é possível avaliar a similaridade entre os objetos a partir de uma **função de distância métrica** então os dados podem ser considerados estar em um domínio métrico.

Seja $\mathbb{S}$ um domínio de objetos. O subconjunto finito $S \subseteq \mathbb{S}$ representa o conjunto de objetos no qual as consultas são efetuadas. A função $d()$:

$$d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$$

corresponde à medida de “distância”, também chamada de “similaridade”, entre dois objetos. Isso significa que quanto menor a distância entre dois objetos, mais próximos ou semelhantes (similares) eles são. Se uma função de distância possuir as seguintes propriedades:

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

onde $\{s_1, s_2, s_3\} \in \mathbb{S}$, o espaço $\mathbb{M}$ definido pelo par $\langle \mathbb{S}, d() \rangle$ é chamado de **Espaço Métrico** e a função de distância é chamada **Função de Distância Métrica**, ou simplesmente **Métrica**. Note-se

que nenhuma informação geométrica, como num domínio espacial, pode ser usada em um espaço métrico.

As funções métricas mais utilizadas são as métricas de **Minkowski** - (L_p) [31] (detalhadas na Seção 2.2), utilizadas em domínios multidimensionais, e de **Levenshtein** - (L_{Edit}) [32], utilizada para o cálculo de similaridade entre cadeias de caracteres. A métrica $L_{Edit}(s_1, s_2)$ retorna o número mínimo de operações de edições (inserções, remoções e substituições de caracteres) necessárias para transformar a cadeia de caracteres s_1 na cadeia s_2 .

As propriedades dos espaços métricos, principalmente a desigualdade triangular, permitem a elaboração de técnicas de indexação capazes de responder a consultas por similaridade de modo eficiente. (ver Seção 2.3).

Uma característica dos espaços métricos é a possibilidade de englobar os espaços vetoriais, bem como espaços n -dimensionais (conjuntos de imagens, palavras, sons ou dados genômicos), desde que haja uma métrica adequada. Um espaço é n -dimensional quando o domínio dos objetos não pode ser identificado por um conjunto de coordenadas em eixos ortogonais numa dimensão finita.

2.2 Modelo de Espaço Vetorial

Se os objetos do domínio $\mathbb{S}$ correspondem a vetores de valores numéricos então o espaço é chamado **Espaço Vetorial com Dimensão Finita**, ou simplesmente **Espaço Vetorial**. Os objetos de um espaço vetorial de dimensão n (ou n -dimensional) são representados por n coordenadas de valores reais $(x_1, \dots, x_k)$. Várias métricas podem ser aplicadas para comparar os objetos de $\mathbb{S}$, sendo as mais conhecidas as da família L_p (ou Minkowski), definidas por:

$$L_p((x_0, \dots, x_{e-1}), (y_0, \dots, y_{e-1})) = \left(\sum_{i=0}^{e-1} |x_i - y_i|^p \right)^{1/p} \quad (2.1)$$

A Figura 2.1(a) ilustra, para diferentes casos da família L_p , o conjunto de pontos que estão à mesma distância r , a partir de um centro s_q . No exemplo da Figura 2.1(a), a métrica L_1 , também conhecida como **Bloco** ou **Manhattan**, corresponde ao somatório do módulo das diferenças entre as coordenadas. Nesse caso, o conjunto de pontos de mesma distância r forma um losango com os diâmetros paralelos aos eixos das coordenadas. A métrica L_2 , mais conhecida como distância **Euclidiana**, corresponde à função usual para distância entre vetores. O conjunto de pontos de mesma distância ao centro s_q para a métrica L_2 forma uma circunferência. Calculando-se o limite da métrica 2.1 quando p tende ao infinito, obtém-se a métrica L_∞ , também conhecida como **Chebychev**, dada pela Equação 2.2, na qual o conjunto de pontos com mesma distância formam um quadrado com

Figura 2.1: (a) Representação dos pontos situados à distância r a partir do objeto s_q , considerando as diferentes métricas da família L_p . (b) Exemplos de consultas por abrangência em um conjunto de objetos: $\sigma_{(Rq(s_q, r))}$ sobre $\langle S, L_1 \rangle = \{s_6, s_9, s_{10}, s_{14}\}$; $\sigma_{(Rq(s_q, r))}$ sobre $\langle S, L_2 \rangle = \{s_4, s_6, s_9, s_{10}, s_{14}\}$; e $\sigma_{(Rq(s_q, r))}$ sobre $\langle S, L_\infty \rangle = \{s_1, s_2, s_4, s_6, s_9, s_{10}, s_{14}\}$.

os lados paralelos aos eixos das coordenadas.

$$L_\infty((x_1, \dots, x_k), (y_1, \dots, y_k)) = \max_{i=1}^k |x_i - y_i| \quad (2.2)$$

A Figura 2.1(b) representa um conjunto de objetos (s_i), um objeto de consulta s_q , o raio de consulta r e três métricas L_1 , L_2 e L_∞ para uma consulta que deseja obter quais são os objetos a uma distância $\leq r$ do objeto de referência s_q . O conjunto de objetos $\{s_6, s_9, s_{10}, s_{14}\}$ contido no losango de consulta da Figura 2.1(b) faz parte do conjunto resposta quando a métrica L_1 é utilizada; os objetos $\{s_4, s_6, s_9, s_{10}, s_{14}\}$ contidos na circunferência de consulta da Figura 2.1(b) fazem parte do conjunto resposta considerando a métrica L_2 ; e, os objetos $\{s_1, s_2, s_4, s_6, s_9, s_{10}, s_{14}\}$ contidos no quadrado de consulta da Figura 2.1(b) fazem parte do conjunto resposta, quando a métrica L_∞ é utilizada.

As três métricas citadas anteriormente foram escolhidas por serem muito utilizadas em consultas por similaridade.

2.3 Tipos de Consulta por Similaridade

As consultas que avaliam o grau de dissimilaridade entre os objetos são chamadas de **consultas por similaridade** e envolvem: uma função de dissimilaridade; um objeto de busca, também considerado como o **centro da consulta**, a partir do qual se deseja encontrar os mais semelhantes; e um parâmetro que depende do tipo de consulta por similaridade em questão. Os dois tipos mais comuns são a **consulta por abrangência** e **consulta aos vizinhos mais próximos**.

Considere-se um conjunto de objetos $S = \{s_1, s_2, s_3, \dots, s_n\}$ pertencentes a um domínio $\mathbb{S}$ e uma

Figura 2.2: Exemplos de consultas por similaridade: (a) consulta por abrangência, (b) 5-vizinhos mais próximos

métrica $d()$. Os tipos mais comuns de consultas por similaridade são:

- **Consulta por Abrangência (Range Query - RQ):** a consulta $\sigma_{RQ(o_q, r_q)} S$ visa recuperar objetos situados a uma distância máxima r_q (raio de busca - *query radius*) do objeto de referência o_q (objeto de busca - *query object*), onde $o_q \in \mathbb{S}$ (Figura 2.2(a)). Formalmente, pretende-se encontrar o subconjunto $A \subseteq S$ que atenda:

$$RQ(o_q, r_q) = A = \{a | a \in S, d(o_q, a) \leq r_q\}$$

Um exemplo de consulta RQ seria: “Selecione as proteínas que difiram da proteína ζ por até 5 bases purínicas”, no qual s_q é a proteína ζ , o universo $\mathbb{S}$ é o conjunto de proteínas, o subconjunto $S \subseteq \mathbb{S}$ é uma base de dados contendo as proteínas conhecidas, o raio de busca $r = 5$ bases purínicas e a métrica $d() \equiv L_{Edit}$. Essa consulta é representada por $\sigma_{(RQ(\zeta, 5))} S$. Note-se que nessa representação, a métrica utilizada não está explícita. Isso ocorre porque se considera o conjunto de dados S amostrado no domínio do espaço métrico onde a métrica está definida. Assim, a indicação do conjunto de dados S já inclui a indicação da métrica a ser utilizada.

- **Consulta aos K -Vizinhos mais Próximos (K -Nearest Neighbor Query - k -NNQ):** essa consulta visa recuperar os k objetos mais próximos do objeto de referência o_q , $o_q \in \mathbb{S}$ (Figura 2.2(b)). Em caso de empate, muito comum quando a função d retorna valores discretos, a resposta pode conter mais do que apenas k elementos. Formalmente, pretende-se encontrar o subconjunto $A \subseteq S$ que atenda a:

$$kNN(o_q) = A = \{a | a \in S, \forall s_i \in S - A, d(o_q, a) \leq d(o_q, s_i), |A| = k\}$$

Os dois tipos de consultas de seleção por similaridade apresentados podem ser facilmente executados a partir da inspeção seqüencial de todos os objetos de um conjunto fornecido. Ou seja,

considerando o conjunto de dados S , o objeto de busca o_q e o raio de busca r_q para responder a consulta $RQ(o_q, r_q)$, cada objeto s_i de S é comparado com o objeto de busca e, se $d(o_q, s) \leq r_q$, então s_i é inserido no conjunto resposta R .

Se a cardinalidade do conjunto S for muito alta ou se a métrica utilizada envolver algoritmos muito custosos, o tempo para a obtenção da resposta pode se tornar inaceitável, sendo interessante o uso de alguma técnica para agilizar tal processo. Pode ser necessário também, armazenar os objetos em disco, dependendo de sua complexidade e tamanho.

O desempenho de consultas por similaridade em ambientes altamente dinâmicos pode ser afetado por dois processos principais: a quantidade de acessos a disco e a quantidade de comparações entre objetos efetuadas através de cálculos de distância [27].

Diversas técnicas de indexação têm sido propostas para melhorar o desempenho de consultas por similaridade [9, 27, 33] e podem ser classificadas em Métodos de Acesso Espaciais, voltados para o modelo em que os objetos são representados por vetores em um espaço e -dimensional; e Métodos de Acesso Métricos, voltados para o modelo em que apenas a distância entre os objetos é levada em consideração. Em geral, a estrutura dos métodos de acesso dinâmicos voltada para o armazenamento secundário é baseada em estruturas hierárquicas que organizam os dados em agrupamentos (*clusters*) [34].

Métodos de Acesso Métrico como *MVP-Tree* [20], *M-Tree* [23], *Slim-Tree* [24, 35], entre outras apresentadas nas seções 2.5, podem melhorar o desempenho de consultas por similaridade sobre dados complexos.

2.4 Método de Acesso Espacial

O método de acesso espacial mais encontrado na literatura é a *R-Tree* [10], escolhido por ser o mais referenciado e utilizado para a indexação de dados vetoriais. Outro ponto importante é que a *R-Tree* é sempre comparada com os outros métodos e pode ser vista como uma referência para demonstrar e validar as melhorias propostas. Dentre os vários métodos de acessos existentes, o método *R-Tree* é apresentado a seguir.

2.4.1 *R-Tree*

A *R-Tree* [10] é um método de acesso espacial para indexar dados vetoriais não necessariamente pontuais, ou seja, que possuem extensão (área, volume, hiper-volume). A organização da estrutura interna é semelhante à da *B⁺-Tree* e atende às premissas feitas por Korn [16]. A Figura 2.3 contém a

Figura 2.3: Exemplo da representação estrutural de uma *R-Tree* [10]

representação estrutural de uma *R-Tree*.

A *R-Tree* é uma estrutura de indexação dinâmica, portanto, os objetos podem ser inseridos e removidos da árvore sem reconstruir a árvore completamente [36]. Cada nó na *R-Tree* corresponde a uma página de disco que armazena um conjunto de objetos geométricos k -dimensional representando-os pelo limite mínimo de retângulos k -dimensional. Cada nó da *R-Tree* corresponde ao MBR (*Minimum Bounding Rectilinear Rectangle*) que contém seus filhos. Uma MBR é caracterizada por pontos “min” e “max” dos super retângulos com faces paralelas aos eixos coordenados. Usando a MBR em vez da representação geométrica do objeto, sua complexidade representacional é reduzida para dois pontos, onde as características mais importantes do objeto (posição e extensão) são mantidas. Conseqüentemente, o MBR é uma aproximação amplamente empregada [37].

As folhas da árvore *R-Tree* contêm ponteiros para os objetos da base de dados, em vez de ponteiros para os nós filhos.

Deve ser notado que os retângulos que cercam nós diferentes podem ser sobrepostos. Além disso, um retângulo pode estar incluído (no sentido geométrico) em muitos nós, mas pode estar associado a somente um deles. Isto significa que uma busca espacial pode demandar a visita a muitos nós, antes de confirmar a existência ou não de um dado retângulo [38].

As regras obedecidas pela *R-Tree* são as seguintes: cada folha contém pares da forma (R, O) , tal que R é o retângulo mínimo que contém espacialmente o objeto determinado pelo identificador O . Todos outros nós contêm pares da forma (R, P) , onde P é um ponteiro para um filho do nó e R é o retângulo mínimo que contém espacialmente todos os retângulos contidos em seus filhos.

A *R-Tree* de classe (m, M) tem a característica que todo nó, exceto a raiz contém entre m e M pares, onde $m < \lfloor M/2 \rfloor$. A raiz contém no mínimo dois pares, senão ela é uma folha.

A inserção é feita da seguinte maneira: considerando inicialmente uma árvore onde só existe o nó raiz, os objetos são inseridos até que a raiz fique totalmente cheia. Nesse momento, a inserção de um novo objeto faz com que ela seja dividida em dois e os objetos são re-arranjados de forma

que o hiper-volume dos dois novos nós seja mínimo. Caso o hiper-volume dos dois novos nós não esteja contido dentro do hiper-volume do nó acima, esse último deve ser ajustado para que contenha o hiper-volume dos novos nós.

A R^* -Tree [13] e a R^+ -Tree [12, 39] são variações posteriores que visam melhorar o desempenho da R -Tree [9]. As principais melhorias apresentadas pela R^* -Tree são: o algoritmo de inserção passa a reinserir alguns objetos antes de particionar um nó; e o algoritmo de particionamento de nós que procura minimizar o perímetro e maximizar a ocupação dos novos nós. O particionamento proposto pela R -Tree procura apenas, minimizar o volume dos nós. Porém, a sobreposição dos nós em qualquer variante da R^* -Tree aumenta drasticamente à medida que a dimensão aumenta [33, 40]. Procurando evitar a sobreposição, a R^+ -Tree quebra os objetos e armazena seus pedaços em nós diferentes. Como consequência, o particionamento de um nó interno precisa ser propagado tanto para seu nó pai quanto para sua subárvore. Por sua vez, ter a subárvore particionada pode reduzir a taxa de ocupação da árvore como um todo [9].

Outra variante da R -Tree foi apresentada em [41]. A X -Tree, como é chamada, define um super-nó para tratar a questão de sobreposição. Quando o grau de sobreposição é alto, os nós são concatenados em um super-nó de tamanho variado (geralmente um múltiplo do tamanho original). Essa abordagem é interessante apenas quando é possível acessar páginas de disco de tamanhos diferentes, diminuindo o número de acessos a disco. No entanto, se o tamanho da página for fixo, como é o caso da maioria dos SGBD, o número de acessos aumenta e o desempenho passa a ser próximo ao da R -Tree.

2.5 Métodos de Acesso Métrico

A consulta por similaridade baseia-se apenas no grau de dissimilaridade fornecido pela comparação entre “pares de objetos”.

A seguir serão apresentados vários métodos de acesso métrico que fornecem suporte para responder buscas por similaridade, isto é, buscas cujo propósito é recuperar objetos que são similares a um objeto de busca, e onde a dissimilaridade entre objetos é medida por uma função de distância métrica específica d .

2.5.1 Técnicas de Burkhard e Keller

A indexação de dados complexos com base na distância entre pares de objetos foi proposta pela primeira vez por Burkhard e Keller em [29]. Três técnicas foram apresentadas para indexar cadeias de *bits*, nas quais o valor retornado pela métrica é restrito a números inteiros não-negativos. O processo

de indexação particiona recursivamente o conjunto de dados, sendo materializado como uma árvore. As técnicas são apresentadas a seguir.

- Na primeira técnica, cada nó possui uma entrada e um número pré-definido de filhos maior ou igual a dois. Um elemento qualquer do conjunto é escolhido para ser colocado como raiz em cada subárvore. Os demais elementos são organizados de acordo com sua distância para o objeto da raiz, ocupando uma mesma subárvore aqueles que estão à mesma faixa de distância. Como a distância é discreta, o agrupamento é sempre possível. Esse procedimento é repetido recursivamente para cada subárvore, até que só reste um objeto, ou até uma profundidade pré-definida.
- Na segunda técnica, o conjunto de dados é dividido em um número fixo de subconjuntos e, para cada um, são definidos um objeto representante e um raio máximo. Os demais objetos são distribuídos de forma que sua distância ao representante seja menor ou igual ao respectivo raio. Esse procedimento é repetido recursivamente. Os representantes são agrupados em nós.
- A terceira técnica, corresponde a particionar o conjunto de objetos em *cliques* (um clique é um conjunto que define um grafo completo maximal, ou seja, que nenhum outro nó pode ser inserido sem que o mesmo deixe de ser completo), cada qual com um valor diferente para c (distância máxima entre os objetos), de modo a garantir que todos os objetos de S estejam em pelo menos um desses *cliques*. Em seguida, um elemento arbitrário de cada *clique* é escolhido como seu representante. A definição não impede que um mesmo objeto apareça em *cliques* diferentes.

Em [42], Shasha e Wang sugerem armazenar uma matriz de distâncias entre todos os pares de objetos, aumentando ainda mais a exclusão de nós durante as consultas. Os autores comentam que essa abordagem é indicada principalmente quando o cálculo da distância é muito custoso. Porém, ela torna-se inviável se o conjunto de dados for muito grande, pois seria necessário armazenar a matriz triangular superior inteira, envolvendo uma complexidade da ordem de $O((n^2 - n)/2)$, onde $n = |S|$.

Um método de acesso métrico é a *Slim-Tree* [24], em que os pontos fortes são: a medida do *fat-factor* que avalia o grau de sobreposição entre os nós de um índice, permitindo indicar se a árvore é adequada para o processamento de consultas; e o algoritmo de pós-otimização *slim-down*, que reduz o número de acessos a disco durante as consultas. Os métodos de acessos métricos *Slim-Tree*, *M-Tree* entre outros e a técnica *Omni* serão discutidos nas próximas subseções.

Figura 2.4: Decomposição do espaço na *GH-Tree* [1]

2.5.2 *GH-Tree*

A *GH-Tree* (*Generalized Hyperplane Tree*) [1], foi um dos primeiros trabalhos a sugerir a construção de um MAM. Esta árvore divide o espaço métrico em hiperplanos generalizados. A definição desses hiperplanos é a seguinte:

Definição: Sejam dois pontos P_1 e P_2 , $P_1 \neq P_2$. Um hiperplano generalizado é o conjunto de pontos Q que satisfazem $d(Q_i, P_1) = d(Q_i, P_2)$. Um ponto x é dito pertencer à partição de P_1 se $d(x, P_1) < d(x, P_2)$.

Cada nó define dois ramos que representarão os subespaços particionados pelo hiperplano. A mesma política é aplicada em cada subpartição, construída na abordagem *top down*. A Figura 2.4 mostra a representação da partição em um espaço euclidiano bidimensional. Nota-se que P_1 e P_2 delimitam dois subespaços, divididos pelo hiperplano generalizado. Assim, todos os objetos q tal que $d(q, P_1) \leq d(q, P_2)$ pertencem à região de P_1 , caso contrário pertencem à região de P_2 .

A *GH-Tree* é construída escolhendo dois objetos aleatoriamente como representativos e dividindo os objetos restantes associando-os aos representantes mais próximos. Em seguida, este processo é aplicado em ambos subespaços, recursivamente. Esta estrutura dá suporte à dinamicidade, ou seja, inserções individuais, seguindo a mesma política de escolha acima. A dinamicidade vem do fato de que a estrutura não controla o balanceamento, então a inserção de um novo objeto não quebra nenhuma propriedade da estrutura.

Uma das desvantagens da *GH-Tree* é que somente consultas pontuais podem ser feitas, uma vez que somente pontos podem ser comparados às partições. Uma outra desvantagem é que precisam ser feitos dois cálculos de distância por nó durante consultas e a cardinalidade dos nós é limitada a dois, o que inviabiliza seu uso para armazenamento em disco.

Figura 2.5: Particionamento de uma *VP-Tree*: (a) representação no espaço; e (b) simplificação da árvore gerada

2.5.3 *VP-Tree*

Métodos de acessos métricos semelhantes à primeira técnica de Burkhard e Keller foram propostos, sendo que a idéia básica é particionar o conjunto de dados em dois subconjuntos de acordo com um representante e um raio máximo. Geralmente, o valor do raio máximo corresponde à mediana das distâncias entre o representante e os demais objetos. Todos os objetos a uma distância do representante menor ou igual ao raio são colocados na subárvore esquerda. Os demais são colocados na subárvore direita.

A *VP-Tree* (*Vantage Point Tree*) foi implementada por Yianilos [20] com melhorias e variações que visam criar uma estrutura para responder a consultas por abrangência e aos vizinhos mais próximos, com o objetivo de minimizar cálculos de distância.

A *VP-Tree* divide o espaço em cortes esféricos, a partir de um ponto escolhido, chamado de ponto de vantagem. A política de construção a torna em princípio uma estrutura estática, onde novas inserções não são possíveis. A construção da árvore é feita da seguinte maneira: a partir de um conjunto de objetos, é escolhido um objeto para atuar como ponto de vantagem e as distâncias dos objetos restantes são calculadas para este objeto. Encontra-se um raio que seja a mediana das distâncias de cada objeto ao ponto de vantagem, que divide o espaço de maneira que metade dos objetos fique dentro do raio de cobertura e a outra metade fora do raio. O mesmo princípio é realizado em ambas as partes recursivamente, formando uma hierarquia, como mostra a Figura 2.5.

A *VP-Tree* tem sua construção e pesquisa análoga à ordenação e busca binária em uma dimensão, tanto por causa de sua estratégia quanto por sua complexidade. Algumas variações foram sugeridas para melhorar o desempenho da estrutura ou mesmo para permitir dinamicidade.

Uma das variações sugeridas em [20] foi a *VP^s-Tree* que mantém em cada nó um histórico das

distâncias dos objetos representativos ancestrais até a raiz, sendo que, as distâncias são calculadas durante a construção da árvore. Cada nó da árvore mantém, além de seu identificador, uma lista de limites de distâncias superior e inferior, denotando o subespaço visto por cada ancestral. Esta melhoria diminui o número de nós visitados durante as consultas por exemplo, aos vizinhos mais próximos, pois ajudam a escolher a ordem adequada de visita aos ramos, o que pode resultar em uma consulta mais rápida. No entanto, esta técnica sofre se a dimensionalidade dos objetos for muito alta.

Outra variação da *VP-Tree* é a *VP^{sb}-Tree* [20], onde os nós folhas formam *buckets* armazenando mais objetos, economizando assim mais espaço, o que resulta em menos acesso a disco para as consultas.

2.5.4 *MVP-Tree*

A *MVP-Tree* (*Multiple Vantage Points Tree*) foi apresentada em [21, 22] com o objetivo de otimizar a utilização dos pontos de vantagem da *VP-Tree*. O espaço métrico na *MVP-Tree* é particionado em cortes esféricos a partir dos pontos de vantagens. Esta estrutura cria partições a partir de dois objetos representantes por nó e mantém informações extras nos nós folhas para filtragem de cálculos de distância, utilizando a desigualdade triangular.

A estrutura da *MVP-Tree* surgiu de algumas alterações de como utilizar os objetos representativos e seus cortes. Duas motivações estimularam seu desenvolvimento: a primeira vem do fato de que é possível particionar uma região sem que o objeto representativo esteja contido na mesma região, o que significa que um mesmo objeto representativo pode particionar diferentes regiões associadas a nós em um mesmo nível. Isto se aplica ao objeto representativo adicional em cada nó, quando comparada à *VP-Tree*.

A segunda motivação recai na possibilidade de manter informações extras nos nós folhas, ou seja, manter para cada objeto armazenado no nó folha, as distâncias aos objetos representativos no caminho da raiz até o nó folha onde cada objeto reside. Tais distâncias podem ser aplicadas posteriormente em consultas através da desigualdade triangular para evitar cálculos de distâncias desnecessários.

A construção da *MVP-Tree* leva em consideração o número de partições m criadas para cada ponto de vantagem; o número máximo de k objetos nos nós folhas; o número de distâncias p pré-calculadas a serem armazenadas para cada objeto nos nós folhas. Uma vez definido o parâmetro m , o grau de um nó intermediário é definido por m^2 . Em uma *MVP-Tree* binária, o primeiro ponto de vantagem divide o espaço em duas partes, e o segundo divide cada uma dessas partes em outras duas, obtendo um grau $2^2 = 4$ regiões disjuntas. Em uma *MVP-Tree* de ordem n , o primeiro ponto de vantagem divide o espaço em m partes e o segundo divide cada uma destas partes em outras m partes, ou seja,

$$m \times m = m^2.$$

Cada nó interno de uma *MVP-Tree* binária é formado pelos atributos que seguem: O_{p1} é o primeiro ponto de vantagem, r_1 é o seu raio de cobertura, O_{p2} é o segundo ponto de vantagem e r_{21}, r_{22} são seus raios de cobertura. Cada nó folha armazena dois pontos de vantagem, k objetos $P_1, P_2, \dots, P_k$, suas distâncias $D_1[i]$ e $D_2[i]$ aos pontos de vantagem, e uma lista das distâncias aos p ancestrais para cada objeto P_i .

Durante a construção da *MVP-Tree* binária, o primeiro ponto de vantagem O_{p1} é escolhido arbitrariamente e o segundo ponto de vantagem O_{p2} é escolhido como sendo o objeto mais distante de O_{p1} . A construção de uma *MVP-Tree* genérica tem ordem de complexidade $O(n \log_m n)$. Apesar de ser um método estático, é eficiente, pois supera as árvores *VP-Tree* [20] e *M-Tree* [23] (apresentada na subseção 2.5.5) em consultas por similaridade.

2.5.5 *M-Tree*

A *M-Tree* foi proposta para organizar e buscar grandes conjuntos de dados em um espaço métrico [23].

Os objetos são armazenados em nós de tamanho fixo de dois tipos: nós folhas e nós índices. Os nós folhas da *M-Tree* armazenam todos objetos indexados, ao passo que os nós internos armazenam os objetos representantes. Para cada rota do objeto O_r (objeto representante) existe um ponteiro associado ($ptr(T(O_r))$), que referencia a raiz da subárvore, $T(O_r)$, chamado de cobertura da árvore de O_r . Todos os objetos na cobertura da árvore de O_r estão dentro da distância $r(O_r)$ de O_r , $r(O_r > 0)$, que é chamado de raio de cobertura de O_r e forma uma parte da entrada O_r em cada nó da *M-Tree*. Finalmente, um objeto representante O_r está associado com uma distância de $P(O_r)$, seu objeto pai, que é o objeto representante que referencia o nó onde a entrada O_r está armazenada. Esta distância não é definida para entrada na raiz da *M-Tree*.

Os algoritmos para construir a *M-Tree* especificam como os objetos são inseridos e removidos e como são gerenciadas as regiões de cobertura. O algoritmo de inserção desce recursivamente a árvore para localizar o nó folha mais apropriado para acomodar o novo objeto. Se o nó folha onde o novo objeto será inserido estiver cheio, provocará uma quebra.

O raciocínio básico usado para determinar o nó folha mais apropriado é descer em cada nível da árvore, ao longo da subárvore, $T(O_r)$, em que o objeto O_n a ser inserido pertença a região de cobertura, definida pelo raio $r(O_r)$ do objeto representante O_r , ou seja, $d(O_r, O_n) \leq r(O_r)$. Se existem múltiplas subárvores com esta propriedade, será escolhido o nível em que o objeto O_r está mais próximo do objeto O_n .

Se não existe nenhum objeto representante para $d(O_r, O_n) \leq r(O_r)$, a escolha é minimizar o aumento do raio de cobertura, $d(O_r, O_n) - r(O_r)$. Isto está relacionado à heurística que sugere minimizar o volume total coberto pelos objetos representativos no nó atual.

Como qualquer árvore balanceada dinâmica, a *M-Tree* cresce de maneira *bottom-up*. A quebra de um nó N provoca a alocação de um novo nó N' no mesmo nível de N , particionando as entradas entre estes dois nós, e promovendo para o nó pai N_p , dois objetos representativos para referenciar os dois nós o que pode ocasionar mais quebras. Quando a raiz é quebrada, uma nova raiz é criada e a *M-Tree* cresce um nível [23].

Quando se insere um objeto, calculam-se e armazenam-se as distâncias entre esse objeto e seus representativos. Estas distâncias são usadas durante as consultas para descartar sub-níveis através da propriedade de desigualdade triangular do espaço métrico, sem que seja necessário, um novo cálculo de distância. Espera-se que a quantidade de cálculos de distância em uma consulta seja drasticamente reduzida, melhorando o desempenho do método [24].

O descarte por desigualdade triangular é feito da seguinte maneira: considerando o espaço métrico $M = (S, d)$, o conjunto $S \subseteq \mathbf{S}$, o objeto de busca $o_q \in S$, o raio de busca r_q e um representativo o_{rep} , o objeto $s \in S$ poderá ser descartado caso uma das duas condições a seguir seja satisfeita (sendo que a distância $d(o_{rep}, s)$ foi previamente obtida e armazenada):

$$d(o_{rep}, s) < d(o_{rep}, o_q) - r_q$$

$$d(o_{rep}, s) > d(o_{rep}, o_q) + r_q$$

A Figura 2.6 mostra o_q como objeto de consulta, o_{rep} é o objeto representativo, r_q é o raio. $s_1, s_2, s_3 \in S_4$ são elementos do conjunto S . Os objetos s_2 e s_3 são descartados, pois $d(o_{rep}, s_2) < d(o_{rep}, o_q) - r_q$ e $d(o_{rep}, s_3) > d(o_{rep}, o_q) + r_q$. Porém, nada pode ser presumido em relação a s_1 e s_4 , forçando o cálculo de distância entre eles e o objeto de busca. O resultado é que s_4 encontra-se dentro da região de busca. O cálculo de distância final é necessário para garantir que não ocorram os objetos qualificados erroneamente.

A garantia de que não existem descartes falsos, ou seja, de que os objetos descartados efetivamente não fazem parte do conjunto de resposta, é embasada na propriedade da desigualdade triangular, sendo que pode ser comprovada em [29]. O espaço vetorial foi utilizado no exemplo da Figura 2.6 apenas para facilitar o entendimento do processo de descarte de objetos no espaço métrico. Porém, como as equações apresentadas levam em consideração apenas a distância entre os objetos, pode-se garantir que esse processo aplica-se para qualquer tipo de dado complexo, desde que a função de distância seja métrica.

Durante as consultas, o raio de cobertura e as distâncias pré-calculadas são utilizados para podar

Figura 2.6: Descarte de objetos com o uso da desigualdade triangular.

subárvores e entradas nas folhas através da desigualdade triangular, reduzindo o número de cálculos de distância. Só se calcula a distância entre objetos quando a poda é possível. Percorre-se uma subárvore se a região que define estiver sobreposta pela região de busca. Ao se atingir uma folha, os objetos que não podem ser descartados são comparados (por cálculos de distância) diretamente com a região de busca e os que estiverem dentro são inseridos no conjunto resposta. O algoritmo para consultas aos k -vizinhos mais próximos é semelhante ao da *R-Tree*, ou seja, em uma consulta por abrangência na qual o raio diminui à medida que novos vizinhos são encontrados.

2.5.6 *Slim-Tree*

A idéia desse método de acesso métrico é semelhante ao da *M-Tree*, e consiste em selecionar um ou mais objetos e colocá-los como representantes do conjunto.

A diferença está na inclusão de um novo algoritmo de quebra na *Slim-Tree*, chamado *Minimum Spanning Tree (MST)*, que é baseado na árvore de caminho mínimo de um grafo, sendo resumido da seguinte forma: os objetos de um nó são considerados vértices de um grafo e as distâncias entre eles são os pesos das arestas que os conectam. O MST do grafo é construído e a aresta de maior peso é removida, gerando os dois agrupamentos correspondentes ao particionamento do nó. Como representante de cada agrupamento é escolhido o objeto mais parecido com todos os outros ao mesmo tempo, ou seja, o centróide. Conforme mostram seus autores, apesar de bem menos custoso que o algoritmo *Minimum-Maximum (min-Max)*, o desempenho das consultas de índices construídos com o MST é igualável ao de um criado pela *M-Tree*.

Outra diferença é que a *Slim-Tree* apresenta algoritmos para pós-otimização do índice que procura minimizar o grau de sobreposição entre os nós da árvore. Com esse processo, chamado pelos autores

de *slim-down* (emagrecimento), o número de acessos a disco é drasticamente reduzido durante as consultas por similaridade. O algoritmo *slim-down* é executado quando a árvore possui um alto grau de sobreposição, o que é indicado pelo *fat-factor*. A *Slim-Tree* foi o primeiro método de acesso métrico que permitiu quantificar a sobreposição entre nós da árvore. Descrições mais completas da estrutura interna da *Slim-Tree* podem ser encontradas em [24, 35].

A Figura 2.7 mostra um exemplo de uma *Slim-tree* com seus nós índices e folhas indexando 13 elementos. No exemplo, a *Slim-tree* contém 2 níveis de índice. No primeiro nível existem dois objetos anta e tatu, que serão representantes para o nó inferior na árvore. O segundo nível da árvore com o *objeto_{representante}* anta, indexa os objetos anta e cabra. O segundo nível da árvore com o *objeto_{representante}* tatu indexa os objetos leão, tatu e urso.

Na Figura 2.7 os valores anta, bode, boi, etc, são usados apenas como ilustração, e a função de distância utilizada não é *Ledit*.

2.5.7 Omni-Family

A técnica Omni [43] propõe o uso de alguns representantes globais, chamados focos, para podar cálculos de distância durante as consultas por similaridade, melhorando o desempenho em relação à abordagem de usar o representante local do nó, existente nos métodos *M-Tree* e *Slim-Tree*. A idéia principal da Omni é escolher um conjunto de focos, com h elementos, e representar cada objeto armazenado através de suas distâncias (pré-calculadas) para os focos. Os objetos são armazenados em um arquivo de acesso randômico (AAR), e acessados diretamente através de um identificador que é gerado pelo AAR no momento da inserção.

Os principais componentes da técnica Omni são:

- **Base de Focus Omni:** é o conjunto $F = \{f_0, f_1, \dots, f_{h-1} \text{ mod } f_g, f_j \in S, f_g \neq f_j\}$, onde cada f_g é um foco usado para representar os objetos de S e h é o número total de focos na base-Omni, isto é, $h = |F|$.
- **Coordenadas Omni:** considerando um objeto qualquer $s \in S$ e a base Omni F , as coordenadas Omni $C(s)$ do objeto s correspondem ao vetor de distâncias entre este objeto e cada um dos focos $f_g \in F$, $0 \leq g < h$. Isto é, $C(s) = (d(f_0, s), d(f_1, s), \dots, d(f_{h-1}, s))$, onde $h = |F|$.

A técnica Omni pode ser combinada com a busca sequencial, com *B-Trees* e com o método de acesso espacial *R-Tree*, gerando três novos métodos de acesso métrico: *Omni-Sequential*, *OmniB-Tree* e *OmniR-Tree*, os quais são descritos a seguir.

Figura 2.7: Representação de uma *Slim-tree* com seus nós índices e folhas

Omni-Sequential Armazena as coordenadas *Omni* em um arquivo seqüencial, onde cada entrada é composta pelo vetor de coordenadas e pelo código interno. O AAR que gerencia os objetos é representado por O-AAR e o arquivo de acesso seqüencial (AAS), que gerencia as coordenadas *Omni*, por CO-AAS.

A execução de uma consulta por abrangência $RQ(o_q, r_q)$ começa com uma etapa de filtragem sobre as coordenadas *Omni* armazenadas no arquivo CO-AAS. O primeiro passo é gerar as coordenadas *Omni* do objeto de busca, $C(o_q)$, calculando as distâncias $d(o_q, f_g)$ entre o_q e cada foco f_g . Em seguida, o arquivo CO-AAS é percorrido seqüencialmente e as coordenadas *Omni* $C(s_i)$ indexadas são comparadas com as coordenadas $C(o_q)$. Durante a comparação das coordenadas, se algum foco f_g com $0 \leq g < h$, for válido para a relação $|d(o_q, f_g) - d(s_i, f_g)| > r_q$, o objeto s_i pode ser descartado sem precisar compará-lo diretamente com o objeto de busca. Vale ressaltar que $d(s_i, f_g)$ já estava calculada e armazenada no CO-AAS como parte de $C(s_i)$.

Uma consulta $k - NNQ(o_q, k)$ aos k -vizinhos mais próximos é semelhante à consulta por abrangência, com a diferença de que o raio de busca é atualizado sempre que um novo vizinho é encontrado. O conjunto de candidatos corresponde a uma lista encadeada dos k -vizinhos encontrados ao longo do processo, e mantida constantemente ordenada pela distância entre o_q e os candidatos vizinhos. O valor do raio de busca é inicialmente ajustado para o infinito.

Para os dois tipos de consulta por similaridade, um objeto só é recuperado do O-AAR e comparado com o_q quando não pode ser descartado de nenhum foco.

OmniB-Tree O método de acesso auxiliar usado para armazenar as coordenadas *Omni* pode ser substituído por um conjunto de *B-Trees*. Esse método permite a indexação de conjuntos de dados métricos usando recursos que já existem em sistemas de gerenciamento de base de dados comerciais [44].

O método *OmniB-Tree* utiliza uma floresta de *B-Trees* para indexar as coordenadas *Omni*. O arquivo que armazena os objetos é representado por O-AAR e o conjunto de *B-Trees* por CO-BF.

Na consulta por abrangência, obtém-se o conjunto de candidatos da etapa de filtragem através da interseção dos candidatos gerados individualmente através de cada foco. Um vetor A de contadores, onde cada posição corresponde a um objeto armazenado ($|A| = |S|$) é usado para acelerar a operação de interseção. Cada posição de A é acessada pelo código interno armazenado com as distâncias. Este vetor é usado para contabilizar os focos que não puderam descartar os objetos durante as consultas. A etapa de refinamento consiste em percorrer o vetor A e, para cada posição que tenha valor h (número total de focos), o respectivo objeto é comparado diretamente com o objeto de busca.

O algoritmo de consulta aos k -vizinhos mais próximos requer uma abordagem diferente da tra-

dicional, que é executar uma consulta por abrangência com reduções sucessivas do raio. Em geral, ajusta-se o valor inicial do raio para o infinito. Porém, a consulta por abrangência na *OmniB-Tree* posicionará cada árvore exatamente na primeira chave da primeira folha. Um problema é a separação dos elementos do vetor das coordenadas *Omni*, fazendo com que os objetos sejam contabilizados como candidatos em instantes diferentes, a depender de cada foco. Na solução adotada a busca é iniciada em cada árvore exatamente onde os respectivos componentes do vetor de coordenadas *Omni* do objeto de busca seriam inseridos, prosseguindo com sucessivos passos para frente e para trás em cada árvore, alternando entre elas após estes dois passos.

OmniR-Tree Este método de acesso métrico utiliza uma *R-Tree* para indexar as coordenadas *Omni*. Esse método mostra praticidade e a validade de se usar um método de acesso espacial para indexar as coordenadas *Omni*, requerendo poucas mudanças nos algoritmos de manipulação dos índices originais (inserção, remoção e consultas), proporcionando uma melhoria no desempenho do método de acesso métrico gerado com a técnica *Omni*.

Os algoritmos originais da *R-Tree* para inserção, particionamento dos nós, consulta por abrangência, entre outros, são os mesmos utilizados pela *OmniR-Tree*. Consultas aos k -vizinhos mais próximos podem ser efetuadas diretamente com o algoritmo original de consultas por abrangência da *R-Tree*.

O algoritmo para consultas aos k -vizinhos mais próximos combina as etapas de filtragem e refinamento. O processo é visto como uma consulta por abrangência com raio decrescente no índice armazenado na *R-Tree*. Sempre que um objeto não consegue ser descartado pela coordenada *Omni*, deve-se comparar esse objeto diretamente com o objeto de busca. Se o resultado indicar que este objeto está mais próximo que o último vizinho, ele é inserido na lista de vizinhos mais próximos e o raio é atualizado. Caso contrário, o objeto é descartado.

2.5.8 Estrutura de Particionamento de Espaço (*VA-File*)

O método *VA-File* (*Vector Approximation File*) [36, 45] reduz a quantidade de dados que deve ser lida durante buscas por similaridade. Este método não usa uma estrutura em árvore, mas em vez disso, armazena uma aproximação do vetor de características de cada objeto em um arquivo seqüencial e realiza uma varredura seqüencial do vetor de aproximações. O vetor de aproximações é uma representação comprimida dos vetores de características originais e são tipicamente 25% do tamanho do vetor de características.

Para obter o vetor de aproximações, o espaço de dados é dividido em 2^b vezes ao longo de cada dimensão, onde b denota o número de *bits* usado por dimensão. Isto divide o espaço de dados em

$2^{b \times D}$ células hiper-retangulares onde D é a dimensionalidade do espaço de dados e cada objeto é aproximado pela *string* de *bits* da célula.

Para cada busca o *VA-File* é lido sequencialmente e um limite inferior e superior da distância atual entre o vetor de características de cada objeto e o objeto de busca é computado. A distância do limite inferior é computado do objeto de busca para o ponto mais perto na célula do objeto, e o limite superior é computado como a distância do objeto de busca para o ponto mais distante na célula do objeto. Objetos com limites inferiores maiores do que o raio de busca ou o limite superior menor do que o objeto atual são podados do conjunto de candidatos porque sua distância atual para objeto de busca deve ser grande para o objeto estar no conjunto de resposta. Isto permite que muitos objetos candidatos sejam descartados sem ter que recuperar o vetor de característica completo do objeto. Para garantir a resposta correta, é realizado um passo de refinamento, onde a distância atual entre os objetos não podados e o objeto de busca é computado. Se b escolhido é grande, este método requer a recuperação de um número pequeno de vetores de características no passo de refinamento.

Um problema com o *VA-File* é que b tem que ser escolhido antes que o *VA-File* seja construído. Se b é mal escolhido, ele pode limitar o desempenho do *VA-File*. Se b é muito pequeno, um grande número de vetores de características terá que ser recuperado do arquivo de dados original. Isto é caro, por que cada vetor de características deve ser acessado aleatoriamente porque não existe nenhuma ordem no arquivo de dados original. O *VA-File* deve também computar um grande número de cálculos de distância em alta dimensionalidade. O número de cálculos de distância é linear com a cardinalidade do conjunto de dados.

O *VA-File* mostrou exibir desempenho linear em relação à cardinalidade e a dimensionalidade do conjunto de dados. É importante notar que o *VA-File* pode somente indexar dados espaciais. Em espaços de alta dimensionalidade, o *VA-File* mostrou-se mais eficiente do que a *R*-Tree* e a *X-Tree*.

2.5.9 Estrutura *D-Index*

Em [25], foi apresentado o primeiro trabalho usando uma estrutura que usa uma função de mapeamento local para realizar a consulta por abrangência, chamada *Similarity Hashing*. Essa estrutura foi melhorada para realizar consulta aos vizinhos mais próximos e rebatizada para *D-Index* [46].

Essa estrutura particiona o espaço em regiões disjuntas, ou seja, não existe sobreposição entre os nós de um mesmo nível. Para armazenar os objetos, a estrutura de mapeamento local usa uma função mapeadora chamada ρ -split, onde ρ é um número real limitado como $0 \leq \rho < d^+$ e d^+ é uma distância máxima. Essa função mapeadora particiona o espaço métrico em regiões centradas em um objeto chamado ponto de vantagem ou pivô x_v .

Figura 2.8: Particionamento ao Meio por Exclusão

A Figura 2.8 mostra a adição de um novo arco X_2 para dividir ao meio a região entre os arcos X_1 e X_0 através da distância média d_m . Assim, um objeto qualquer pode pertencer a 3 regiões:

- 0 se $d(x, x_v) \leq d_m - \rho$
- 1 se $d(x, x_v) > d_m + \rho$
- — demais

A idéia dessa estrutura é criar vários níveis onde cada nível usa uma função de particionamento. O espaço métrico é particionado enquanto houver espaço no *bucket*. Quando não houver espaço para o particionamento, cria-se um nível com uma outra função de particionamento que subdivide esse espaço.

Dessa forma, o primeiro nível usa uma função *p-split* para separar os objetos do conjunto. Para qualquer outro nível, os objetos são mapeados pela exclusão do *bucket* do nível anterior. Finalmente, a exclusão do *bucket* do último nível forma o *bucket* de exclusão de toda a estrutura *D-Index*.

2.6 Conclusões

Neste capítulo, foram discutidos os métodos de acessos espaciais e métricos. Sobre os métodos de acessos espaciais foi apresentada a estrutura *R-Tree* que é a mais citada na literatura. Essa estrutura permite realizar junção por similaridade, que será apresentada no capítulo 3.

Os métodos de acessos métricos oferecem um suporte mais eficiente para responder consultas por similaridade, sendo que, foram apresentados os seguintes métodos: *GH-Tree*, *VP-Tree*, *MVP-Tree*, *M-Tree*, *Slim-Tree*, *Omni-Family*, *VA-File* e a estrutura *D-Index*. Vários conceitos apresentados foram usados na criação de um novo método de acesso métrico chamado de coordenadas métricas que é uma contribuição inovadora deste trabalho e será apresentado no capítulo 4.

No próximo capítulo serão apresentados vários tipos de junções, e as suas respectivas implementações.

CAPÍTULO 3 Junção

A operação de junção é uma das operações fundamentais de busca em sistemas de gerenciamento de banco de dados relacional. Ela possibilita a recuperação de dados em duas relações diferentes baseada no produto cartesiano de duas relações [47]. A operação de junção tem sido estudada e discutida na literatura porque ela é uma operação que consome mais tempo e tem acesso intensivo aos dados em um processamento de consulta relacional.

A operação de junção é usada para combinar tuplas relacionadas de duas relações em tuplas simples que são armazenadas na relação resultado. O relacionamento desejado entre tuplas ou alguns atributos nas tuplas é especificado em termos da condição de junção. Na forma mais simples, a junção de R e S é escrita como: $R \bowtie_{r(a)\theta s(b)} S$, onde $r(a)\theta s(b)$ define a condição de junção. O operador θ define a condição que se deve manter verdadeira entre os atributos $r(a)$ e $s(b)$ de R e S , respectivamente. Esta junção é chamada de *theta-join*. Nas operações sobre conjuntos de dados que atendem a propriedade de relação de ordem total, o operador teta pode ser um dos seguintes: $=, \neq, <, >, \leq, \geq$.

A operação de junção é equivalente ao produto cartesiano seguida por uma operação de seleção [48], onde uma operação de seleção é implícita na condição de junção, ou seja:

$$R \bowtie_{r(a)\theta s(b)} S \equiv \sigma_{r(a)\theta s(b)}[R \times S].$$

O resultado de unir duas relações é um subconjunto do produto Cartesiano de duas relações. O resultado da junção das relações R e S com n e m atributos é a relação Q com $(n + m)$ atributos. A relação Q tem uma tupla para cada par de tuplas de R e S que satisfaz a condição de junção. O resultado da relação Q pode então ser definida como:

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge t(a)\theta t(b)\}$$

3.1 Tipos de Junção

Vários tipos de junção têm sido definidos, como por exemplo:

Equijoin O operador *theta* mais usado é o operador de igualdade; e nestes casos a junção é chamada de *equijoin*. Para todos outros operadores *theta* a junção é chamada de uma *nonequijoin*. A relação resultante Q é definida por:

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge t(a) = b(a)\}$$

A relação resultante contém tuplas t compostas de duas partes, r e s , onde r deve ser uma tupla da relação R e s deve ser uma tupla da relação S . Em cada tupla t , os valores dos atributos da junção $t(a)$, pertencentes a r , são idênticos em todos aspectos aos valores dos atributos da junção $t(b)$, pertencentes a s .

Natural Join A junção natural pode ser definida como a *equijoin* sobre os atributos com o mesmo nome em ambas as relações, seguido por projeção para remover um conjunto dos atributos da junção. Essa junção natural pode ser escrita como:

$$R * S = \pi_{(a_i b_j - s(a))} [R \bowtie_{r(a)=s(a)} S],$$

onde a_i e b_i , são os atributos em relação a R e S , respectivamente. A relação resultante Q é dada por:

$$Q = \{t \mid t = (rs - s(a)) \wedge r \in R \wedge s \in S \wedge r(a) = s(a)\}$$

Semijoin Na execução de uma operação de junção convencional, a relação resultante tem todos os atributos de ambas as relações de entrada. Algumas vezes, é requerido que somente os atributos de uma das relações esteja presente na relação de saída. A operação *semijoin* é projetada para realizar tal junção. Ela também tem sido definida como uma operação que seleciona o conjunto das tuplas de uma relação que se relaciona com uma ou mais tuplas da outra relação. O relacionamento é definido pela condição de junção. Ela é equivalente à junção de duas relações seguidas por uma operação de projeção que resulta nos atributos da segunda relação. A própria junção inicial pode ser realizada por qualquer uma das técnicas de junção.

A operação *semijoin* é escrita como:

$$R \ltimes_{r(a)\theta s(b)} S = \pi_{a_i} [R \bowtie_{r(a)\theta s(b)} S],$$

e a relação resultante Q é dada por:

$$Q = \{t \mid t = r \wedge t(a)\theta s(b) \wedge r \in R \wedge s \in S\}.$$

Diferente da maioria das operações de junção, a operação *semijoin* não é comutativa, isto é,

$$(R \ltimes S) \neq (S \ltimes R).$$

Uma expressão alternativa para a operação de junção *semijoin* é:

$$R \ltimes S = \pi_{a_i}[R \bowtie (\pi_{s(b)}S)]$$

Nenhuma condição de junção é especificada nesta expressão por que ela representa o caso geral. Embora o efeito seja o mesmo em ambos os casos, esta versão da *semijoin* reduz o tamanho da segunda relação participante na operação de junção. *Semijoin* pode ser usada para reduzir a carga de processamento de junção regular e para evitar a criação de grandes relações intermediárias. Ela é especialmente útil em processamento distribuído, para diminuir a quantidade de dados que trafegam entre unidades de processamento.

Outerjoin A operação *outerjoin* é uma extensão da operação de junção. Ela é também chamada de junção externa. Três tipos de *outerjoin* são definidas: a *left-outerjoin*, a *right-outerjoin* e a *full-outerjoin* [48]. Os símbolos de junção correspondente são: $=\bowtie$, $\bowtie=$ e $=\bowtie=$. Tanto a junção *left-outerjoin* quanto a *right-outerjoin* são coletivamente referidas como junção direcional. Deve-se notar que somente a junção *full-outerjoin* é comutativa. Embora as junções *left/right-outerjoin* não sejam comutativas, elas são apresentadas como segue: $(R =\bowtie S) = (S \bowtie= R)$. A relação resultante Q pode ser escrita como:

$$Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge r(a)\theta s(b)\} \cup \{t \mid t = rn \wedge r \in R\} \cup \{t \mid t = ns \wedge s \in S\},$$

onde, n representa o enchimento das tuplas com valores nulos.

Self-join O *self-join* pode ser considerado um caso espacial do *theta-join*. A única diferença é que a relação de entrada é unida com ela mesma. A relação de saída Q é dada por:

$$Q = \{t \mid r \in R \wedge s \in S \wedge R = S \wedge t = rs \wedge r(a)\theta s(b)\}$$

Composition A composição de duas relações é equivalente a junção de duas relações, e formalmente isto é descrito como:

$$R \otimes S = \pi_{(a_i b_j - r(a) - s(b))}[R \bowtie S],$$

onde $\otimes$ representa o operador de composição e $\bowtie$ sem a condição de junção representa qualquer junção entre as relações R e S .

A composição natural de R e S é definida como a composição baseada na junção natural entre R e S . Ela é escrita como:

$$R \cdot S = \pi_{(a_i b_j - r(a) - s(a))}[R * S]$$

A relação resultante Q , da composição natural de R e S , é dada por:

$$Q = \{t \mid t = (rs - r(a) - s(a)) \wedge r \in R \wedge s \in S \wedge r(a) = s(a)\}$$

Division A operação de divisão não é um membro do conjunto de operações de álgebra relacional. O operador de divisão permite uma tupla ser recuperada de uma relação se ela está relacionada a todas as tuplas em outra relação baseada em alguma condição pré definida. As relações R e S têm atributos a_i e b_j tal que $b_j \subseteq a_i$. Então, $c_k = a_i - b_j$. Isto é, c_k é o conjunto de atributos na relação R que não está na relação S . Então, R dividido por S é escrito como:

$$R \div S = \pi_{c_k}(R) - \pi_{c_k}((\pi_{c_k}(R) \times S) - R)$$

Em outras palavras, a relação resultante Q é o subconjunto da relação $\pi_{c_k}(R)$ tal que o produto Cartesiano das relações Q e S esteja contido em R .

3.2 Implementações de Junção

As técnicas e métodos usados para implementar junções são apresentados e discutidos nas próximas subseções.

3.2.1 Junção *Nested-Loops*

Este é o método de junção mais simples, e segue da definição da operação de junção. Uma das relações sendo unidas é designada como a relação mais interna, e a outra é a mais externa. Para cada tupla da relação mais externa, todas as tuplas da relação mais interna são lidas e comparadas com a tupla da relação mais externa. Quando a condição de junção é satisfeita, as duas tuplas são concatenadas e colocadas no *buffer* de saída.

O Algoritmo 3.1 mostra como realizar $R \bowtie_{r(a)\theta s(b)} S$.

Algoritmo 3.1 *Nested-Loops* Join [47]

```

1: para cada tupla  $s$  faça
2:   para cada tupla  $r$  faça
3:     se  $r(a)\theta s(b)$  então
4:       concatena  $r$  e  $s$ 
5:       coloca na relação  $Q$ 
6:   fim-se
7: fim-para
8: fim-para

```

Note que para melhorar o desempenho do algoritmo, a relação com a cardinalidade mais alta (neste caso em R) deve ser escolhida para ser a relação mais interna.

O algoritmo de junção *nested-loops* é implementado como uma junção *nested-block*; isto é, tuplas são recuperadas em unidades de blocos. Esta implementação pode ser descrita da seguinte forma: a relação mais interna é lida em um bloco uma vez. O número de blocos de memória principal disponível, determina o número de blocos lidos da relação mais externa. Então todas as tuplas do

bloco da relação mais interna são unidas com todas as tuplas do bloco da relação mais externa. Este processo é repetido com todos os blocos da relação mais interna antes que o próximo conjunto do bloco da relação mais externa seja lido.

3.2.2 Junção *Sort-Merge*

A junção *sort-merge* é executada em dois estágios. Primeiro, ambas relações são ordenadas pelos atributos da condição de junção. Então, ambas as relações são examinadas pela ordem dos atributos da junção, e as tuplas que satisfizerem a condição de junção são unidas para formar a relação resultado. Quando as tuplas da primeira relação combinam com uma tupla da segunda relação, as tuplas são concatenadas e colocadas na relação de saída.

O algoritmo exato para realizar o *sort-merge join* depende de se os atributos da junção são ou não atributos sem chave e do operador *theta*. Em todos os casos, no entanto, é necessário que as duas relações estejam fisicamente ordenadas por seus respectivos atributos de junção. O Algoritmo 3.2 mostra como realizar *equijoins*.

Algoritmo 3.2 *Sort-Merge Join to Equijoin*[47]

```

1: Passo1: Ordenar o processo
2: ordenar R em r(a);
3: ordenar S em s(b);
4: Passo 2: Unir o processo
5: ler a primeira tupla de R;
6: ler a primeira tupla de S;
7: para (cada tupla r) faça
8:   enquanto ( $s(b) < r(a)$ ) faça
9:     ler a proxima tupla de S;
10:   se ( $r(a) = s(b)$ ) então
11:     juncao de r e s
12:     coloca na relacao saida Q;
13:   fim-se
14: fim-enquanto
15: fim-para

```

O processo *merge* varia ligeiramente dependendo se os atributos da junção são chaves primárias, secundárias ou sem chave. Se os atributos da junção não são atributos chave primária, várias tuplas com o mesmo valor de atributo pode existir. Isto necessita de vários passos sobre o mesmo conjunto de tuplas da relação interna. O processo é descrito abaixo.

Suponha-se que existem duas tuplas, $r1$ e $r2$, em R que tem um dado valor x do atributo de junção $r(a)$ e três tuplas, $s1$, $s2$ e $s3$, em S que têm o mesmo valor de x do atributo de junção $s(b)$. Se o algoritmo de junção acima é usado onde $r2$ é a tupla corrente em R , a tupla corrente em S seria a tupla $s3$. Agora a relação resultante deve também incluir a junção de $r2$ com $s1$, $s2$ e $s3$. Para alcançar isto, o algoritmo acima deve ser modificado para lembrar o último valor $r(a)$ e o ponto em S onde ele iniciou o último loop interno. Quando encontra um valor $r(a)$ duplicado, retrocede-se a recursão

ao ponto inicial anterior em S . Este retrocesso pode ser especialmente caro em termos de I/O se o conjunto de tuplas não se adequa na memória principal disponível e as tuplas têm que ser recuperadas do armazenamento secundário para cada passo.

O algoritmo *sort-merge join* pode também ser usado para implementar a *outerjoin* completa. O Algoritmo 3.3 mostra como realizar a *outerjoin* completa $R \bowtie_{r(a)=s(b)} S$ usando o método *sort-merge*.

Algoritmo 3.3 *Outerjoin* usando o método *sort-merge* [47]

```

1: ordenar  $R$  em  $r(a)$ ;
2: ordenar  $S$  em  $s(b)$ ;
3: ler a primeira tupla de  $R$ ;
4: ler a primeira tupla de  $S$ ;
5: para cada tupla  $r$  faça
6:   enquanto  $s(b) < r(a)$  faça
7:     escrever  $s$  em  $Q$ 
8:     atributos de  $R$  com valores nulos
9:     ler a proxima tupla de  $S$ ;
10:  se  $r(a) = s(b)$  então
11:    juncao de  $r$  e  $s$ 
12:    coloca na relacao saida  $Q$ ;
13:  fim-se
14: fim-enquanto
15: fim-para

```

3.2.3 Métodos de Junção *Hash*

O sucesso da junção *sort-merge* está no fato de que ela reduz o número de comparações entre tuplas. Uma tupla da primeira relação não é comparada com aquelas tuplas na segunda relação em que não é possível fazer a junção. Os métodos de junção *hash* alcançam o mesmo efeito de outra maneira. Eles tentam isolar as tuplas da primeira relação em que pode ser feita a junção com uma dada tupla da segunda relação. Então, tuplas da segunda relação são comparadas com um conjunto limitado de tuplas da primeira relação.

Um grande número de métodos de junção usando *hashing* tem sido proposto e/ou implementado [47]. Alguns serão discutidos aqui.

Método de Junção *Hash* Simples Com este método, os valores dos atributos da junção na primeira relação serão usados pela função *hash*. Estes valores da chave *hash* apontam para a tabela *hash* em que cada entrada pode conter tuplas ou *ids* das tuplas [49]. Pode ser útil armazenar os valores das chaves também. Dependendo da eficiência da função *hash*, uma ou mais entradas podem ser mapeadas para o mesmo *bucket*. Então para cada tupla da outra relação participante na junção, os valores dos atributos da junção são mapeados usando a mesma função *hash*.

O Algoritmo 3.4 mostra como realizar $R \bowtie_{r(a)\theta s(b)} S$.

Algoritmo 3.4 Junção *Hash* Simples [47]

```

1: para cada tupla  $s$  em  $S$  faça
2:   hash juncao dos atributos  $s(b)$ 
3:   coloca as tuplas na tabela hash baseadas nos valores hash;
4: fim-para
5: para cada tupla  $r$  faça
6:   hash juncao dos atributos  $r(a)$ 
7:   se  $r$  hashes para um bucket no vazio da tabela hash para  $S$  então
8:     se  $r\theta$  - combina qualquer  $s$  no bucket então
9:       concatena  $r$  e  $s$ 
10:      coloca na relacao  $Q$ ;
11:   fim-se
12: fim-se
13: fim-para

```

No algoritmo 3.4, a mesma função *hash* deve ser usada para ambas as relações.

O método de junção *hash* pode também ser usado para implementar as junções *right/left outer-joins*. O algoritmo 3.5 mostra como realizar a *right-outerjoin*, $R = \bowtie_{r(a)\theta(b)} S$, usando a técnica de junção *hash* simples.

Algoritmo 3.5 Junção *Right Outerjoin* usando a Junção de *Hash* Simples [47]

```

1: para (cada tupla  $s$  em  $S$ ) faça
2:   hash juncao dos atributos  $s(b)$ 
3:   coloca tuplas na tabela hash baseada nos valores hash;
4: fim-para
5: para (cada tupla  $r$ ) faça
6:   hash juncao dos atributos  $r(a)$ 
7:   se ( $r$  hashes para uma entrada nao vazia da tabela hash para  $S$ ) então
8:     se (ajusta  $s$  encontrado) então
9:       concatena  $r$  e  $s$ 
10:      coloca na relacao  $Q$ 
11:   senão
12:     pad atributos de  $S$  com valores nulos;
13:     escrever  $r$  em  $Q$ 
14:   fim-se
15: fim-se
16: fim-para

```

Junção *Hash* Particionada A junção *hash* particionada tenta otimizar a execução da junção particionando o problema em partes. É usado *hash* para decompor o problema de junção em vários subproblemas que são mais fáceis para resolver. A abordagem *dividir-e-conquistar* tem sido usada por apresentar várias vantagens; não só a eficiência do processo como um todo é melhorada, mas também o particionamento faz com que a paralelização do algoritmo seja mais fácil permitindo que pares independentes sejam processados, o que resulta na melhora de desempenho.

Uma função *hash* é usada para particionar as tuplas em cada relação em um número fixo de conjuntos sem junção. As tuplas no primeiro conjunto de uma relação podem combinar somente com as tuplas no primeiro conjunto na segunda relação. Assim, o processamento de diferentes pares de partições correspondentes nos dois conjuntos são independentes de outros conjuntos. Como resultado,

esses pares de partições podem ser processados em paralelo. Tuplas nos pares de partição de ambas as relações podem ser unidas por uma outra junção *hash* ou qualquer outro método de junção.

Junção de *Hash* Particionada Simples Na junção *hash* particionada simples nem todas as partições são criadas no início da operação; em vez disso, cada par de partição é criado e usado antes que a próxima seja gerada.

O número de partições e a área dos valores dos atributos em cada partição são percorridos. Antes das relações serem percorridas, a área dos valores *hash* para a posição atual é escolhida. Idealmente, a área deve ser tal que todas as partições sejam de tamanhos iguais. As áreas são determinadas dividindo a área total dos valores dos atributos em subáreas de tamanhos iguais. O número de subáreas é igual ao número de partições desejadas. Isto depende de fatores como a quantidade de memória disponível e o número de processadores. A relação externa é percorrida e os valores dos atributos são mapeados. Se o valor mapeado cai na área de valores mapeados atuais, a tupla é inserida na tabela *hash* na partição atual; por outro lado, ela é escrita em um arquivo temporário. A partição da tabela *hash* está completa quando toda a relação tiver sido percorrida.

A relação interna é percorrida e os atributos da junção são mapeados usando a mesma função *hash*. Se o valor mapeado cai na área atual, o processo de junção é iniciado; por outro lado, a tupla é escrita em um arquivo temporário.

O algoritmo 3.6 mostra como realizar $R \bowtie_{r(a)\theta s(b)} S$.

Algoritmo 3.6 Junção de *Hash* Particionada Simples [47]

```

1: repita
2:   escolher uma area hash para particao atual;
3:   para (todo s) faça
4:     hash valores dos atributos;
5:     se (value falls in current range) então
6:       make entry in partition hash table
7:     senão
8:       escrever a tupla em temp-S;
9:     fim-se
10:  fim-para
11:  para todo r faça
12:    hash nos atributos da juncao;
13:    se value falls in current range então
14:      se match found in hash table então
15:        concatena as tuplas r e s
16:        coloca na relacao saida Q
17:      senão
18:        discard tuple
19:      fim-se
20:    senão
21:      escrever a tupla em temp-R;
22:    fim-se
23:  fim-para
24:  R := temp-S
25:  S := temp-R
26: até que (| S | = null);

```

Temp- R e temp- S são relações temporárias usadas para evitar o processamento de todas as tuplas em cada passo no algoritmo. A relação menor S é particionada primeiro para reduzir o tempo de execução.

Junção *Hash Loops* Esta é uma variação da técnica de junção *nested-loops*. Na primeira fase, a relação externa é dividida em um número fixo de partes. Próximo passo, cada partição é lida na memória uma vez. Uma tabela *hash* é construída para cada partição quando ela é lida da memória principal. As tuplas na relação interna são lidas. Os atributos da junção interna são mapeados, e os valores mapeados são usados para investigar a tabela *hash* para que a partição atual possa combinar as tuplas. Assim, a relação interna é lida uma vez para cada partição, e a tabela *hash* é usada para acelerar o processo de encontrar uma combinação.

O algoritmo 3.7 apresenta como realizar $R \bowtie_{r(a)\theta s(b)} S$.

Algoritmo 3.7 Junção de *Hash Loops* [47]

```

1: particiona  $R$  usando a função de divisão hash;
2: para cada partição  $R_i$  faça
3:   ler a partição em memória;
4:   criar a tabela hash para a partição;
5: fim-para
6: para cada tupla  $s$  faça
7:   hash valor do atributo;
8:   se  $s$  hashes para uma entrada não vazia da tabela hash para  $R_i$  então
9:     iniciar a junção de  $s$  e  $r$ ;
10:  fim-se
11:  coloca o resultado em  $Q$ ;
12: fim-para

```

3.3 Junção Espacial

Durante as últimas décadas, sistemas de base de dados espaciais têm tornado-se mais e mais importantes para indústria, administração pública, ciência e negócios. Vários sistemas de base de dados espaciais, projetados para organizar dados espaciais de um sistema de informação geográfica, têm sido desenvolvidos para aplicações tais como cartografia, ciência ambiental e geografia [2, 50].

Um objeto espacial consiste de (no mínimo) um atributo espacial que descreve a geometria do objeto. Estes atributos contêm dados de duas ou três dimensões de um tipo comum como por exemplo, pontos, linhas, retângulos, polígonos e até tipos mais complexos compostos de tipos simples [50].

O processamento eficiente de uma junção espacial é importante, uma vez que seu tempo de execução é superlinear no número de objetos espaciais das relações participantes, e o número de objetos pode ser grande [2, 51].

A junção espacial é definida sobre duas ou mais relações. Ela computa um subconjunto do produto

cartesiano e combina objetos espaciais destas relações de acordo com seus atributos geométricos, isto é, estes atributos têm que atender a um predicado espacial.

A junção espacial é uma das operações mais frequentes em um sistema de bases de dados espaciais. O desempenho de um sistema de bases de dados espaciais é provável ser mais afetado por uma operação de junção espacial do que por uma operação de seleção espacial, porque uma junção espacial tem que acessar os objetos várias vezes. Seu tempo de execução é geralmente superlinear no número de objetos.

Apesar da importância da junção espacial, a maioria das pesquisas tem sido voltada para junção natural e *equijoins*. Quase todos os métodos projetados para processamento de junção- θ relacional não podem ser usados para junções espaciais sem modificações. Junções baseadas em *hash* (algumas apresentadas na subseção 3.2.3) não são apropriadas para realizar junções espaciais já que os objetos espaciais, os quais devem atender ao predicado da junção, não estão em um *bucket* comum no *hash*. Junções *sort-merge* (descrita na subseção 3.2.2) podem ser aplicadas para junções espaciais, mas estas requerem uma ordem dimensional dos objetos bi-dimensionais. Assim, somente a abordagem da junção *nested-loops* (apresentada na subseção 3.2.1) pode ser usada sem quaisquer modificações para ambas as junções, espacial e relacional.

3.3.1 Junção Espacial para *R-Trees*

A primeira abordagem apresentada em [2], para realizar uma junção espacial com *R*-Tree* é usar os retângulos MBR que redirecionam para as sub-árvores internas. Se os retângulos das duas entradas E_R e E_S não têm uma interseção comum, não existirá nenhum par de retângulos de dados intersectando ($rect_R, rect_S$) onde $rect_R$ está na subárvore de E_R e $rect_S$ está na subárvore de E_S . Caso contrário, deve-se navegar em um nível abaixo e continuar verificando quais retângulos se intersectam. O Algoritmo 3.8 apresenta a abordagem descrita acima.

Algoritmo 3.8 JuncaoEspacial01($R, S: R_Node$)

```

1: // altura de R é igual a altura de S
2: para cada  $E_S \in S$  faça
3:   para cada  $E_R \in R$  com  $E_R.rect \cap E_S.rect \neq \emptyset$  faça
4:     se R é FOLHA então
5:       // S também é folha
6:       adiciona o par  $E_R$  e  $E_S$  na resposta
7:     senão
8:        $R' \leftarrow$  bloco referenciado em  $E_R.ref$ 
9:        $S' \leftarrow$  bloco referenciado em  $E_S.ref$ 
10:      JuncaoEspacial01( $R', S'$ )
11:   fim-se
12: fim-para
13: fim-para

```

Aqui é assumido que R e S são do tipo nó na *R*-Tree* (R_Node), sendo que cada nó possui uma

Figura 3.1: Dois Conjuntos de Retângulos e sua Projeção no Eixo X [2]

coleção de entradas. Cada entrada E consiste de um ponteiro ref e um retângulo $rect$.

O algoritmo percorre recursivamente ambas as árvores de modo *top-down* verificando cada entrada de um nó com todas as entradas do outro nó.

O Algoritmo 3.9 usa a técnica de restrição de espaço de busca para melhorar o Algoritmo 3.8. Essa técnica consiste em usar o retângulo de interseção entre os nós do nível superior para restringir os espaços.

Algoritmo 3.9 *JuncaoEspacial02*(R, S : R_Node ; $rect$: *Retangulo*) [2]

```

1: para cada  $E_S \in S$  com  $E_S.rect \cap rect \neq \emptyset$  faça
2:   para cada  $E_R \in R$  com  $(E_R.rect \cap rect \neq \emptyset)$  e  $(E_R.rect \cap E_S.rect \neq \emptyset)$  faça
3:     se  $R$  é FOLHA então
4:       //  $S$  também é folha
5:       adiciona o par  $E_R$  e  $E_S$  na resposta
6:   senão
7:      $R' \leftarrow$  bloco referenciado em  $E_R.ref$ 
8:      $S' \leftarrow$  bloco referenciado em  $E_S.ref$ 
9:      $rect \leftarrow$  retângulo de intersecao das MBRs do nos  $R'$  e  $S'$ 
10:    JuncaoEspacial02( $R', S', rect$ )
11:   fim-se
12: fim-para
13: fim-para

```

No Algoritmo 3.9, $rect$ é o retângulo de interseção MBR dos nós R e S e é usado para restringir o espaço para todas as entradas de R e S .

A segunda abordagem apresentada em [2] para o melhoramento da junção espacial é ordenar as entradas em um nó da R^* -Tree de acordo com a localização espacial dos retângulos correspondentes. O problema acontece pois retângulos bi-dimensionais não podem ser ordenados, isto é, mapeados para uma sequência 1-dimensional, sem perda de localização. A solução adotada foi utilizar a interseção da seguinte maneira: considere uma sequência $Rseq = \langle r_1, \dots, r_n \rangle$ de n retângulos, sendo que um r_i qualquer é composto por seu canto esquerdo inferior $(r_i.x1, r_i.y1)$ e seu canto direito superior $(r_i.xu, r_i.yu)$. Foi usado $\pi_X(r_i)$ e $\pi_Y(r_i)$ que referem-se a projeção de r_i nos eixos X e Y . A sequência $Rseq = \langle r_1, \dots, r_n \rangle$ é ordenada em relação ao eixo X , se $r_i.x1 \leq r_{i+1}.x1$, $1 \leq i < n$. Por exemplo, a sequência ordenada de 6 retângulos mostrada na figura 3.1, é $\langle r_1, r_4, r_2, r_5, r_3, r_6 \rangle$

Uma outra abordagem chamada *Plane Sweep* (varredura de plano) é uma técnica comum para interseções [52]. A idéia básica é mover uma linha, também chamada de linha de varredura na per-

pendicular para um dos eixos, isto é, o eixo X da esquerda para direita. Dado duas seqüências de retângulos $Rseq$ e $Sseq$, explora-se a técnica de varredura de plano sem o *overhead* de construir qualquer estrutura de dados dinâmica. Dado $Rseq = \langle r_1, \dots, r_n \rangle$ e $Sseq = \langle s_1, \dots, s_m \rangle$ seqüências de retângulos. Primeiro os retângulos de $Rseq$ são ordenados por $r_i.x1 \leq r_{i+1}.x1, 1 \leq i < n$, e os retângulos de $Sseq$ são ordenados por $s_j.x1 \leq s_{j+1}.x1, 1 \leq j < m$, como descrito acima. Então move-se a linha de varredura para o retângulo, chamado t , em $Rseq \cup Sseq$ com o valor $x1$ inferior. Se o retângulo está em $Rseq$, em seqüência através de $Sseq$ iniciando de seu primeiro retângulo até um retângulo s_h , em $Sseq$ ser encontrado o valor $x1$ que é maior do que $t.xu$. Agora, conhecido o intervalo $\pi_X(t)$ que intersecta o intervalo $\pi_X(s_j)$ para todos j com $1 \leq j < h$. Se o intervalo π_Y também intersecta o intervalo $\pi_Y(s_j)$, então, o retângulo t também intersecta o retângulo s_j . Se o retângulo t está em $Sseq$, $Rseq$ intersecta analogamente. Depois disso, o retângulo t é marcado para ser processado. Então, a linha de varredura é movida para o próximo retângulo desmarcado em $Rseq \cup Sseq$ com o valor $x1$ inferior e o mesmo passo como foi descrito acima é repetido para todos os retângulos não marcados. Quando a última entrada R ou S forem processadas, todas interseções são computadas. A descrição formal do algoritmo é mostrada nos Algoritmos 3.10 e 3.11.

Algoritmo 3.10 TesteIntersecaoOrd($Rseq, Sseq: SeqRetang$; $Saida: ParesSeqRetang$) [2]

```

1: // Rseq e Sseq estao ordenados; || Rseq || numero de retangulos em Rseq
2: Saida:=; i = 1; j = 1;
3: enquanto (i ≤ || Rseq ||) ∧ (j ≤ || Sseq ||) faça
4:   se (ri.x1 < sj.x1) então
5:     // t = ri
6:     LoopInterno(ri, j, Sseq, Saida);
7:     i = i + 1;
8:   senão
9:     // t = sj
10:    LoopInterno(sj, i, Rseq, Saida);
11:    j = j + 1;
12: fim-se
13: fim-enquanto

```

Algoritmo 3.11 LoopInterno (t:Retangulo; desmarc:int; Sseq:SeqRetang; Saida:ParesSeqRetang) [2]

```

1: K = desmarc;
2: enquanto (k ≤ || Sseq ||) ∧ (sk.x1 ≤ t.xu) faça
3:   // intersecao X
4:   se (t.y1 < sk.yu) ∧ (t.yu > sk.y1) então
5:     // intersecao Y
6:     Append(Saida(t, sk))
7:   fim-se
8:   k = k + 1;
9: fim-enquanto

```

Um exemplo de como o algoritmo procede é ilustrado na Figura 3.2. A linha de varredura pára nos retângulos r_1, s_1, r_2, s_2 e r_3 . Para cada parada, os pares de retângulos que são testados para interseção são mostrados no lado direito da figura 3.2.

Figura 3.2: Exemplo para o Teste de Interseção Ordenada [2]

O algoritmo *plane-sweep* (varredura de plano) determina as interseções de dois conjuntos de retângulos. Baseado na ordenação espacial, este algoritmo cria uma sequência de interseção de pares de retângulos. Esta sequência pode também ser usada para determinar o escalonamento de leitura da junção espacial. Para preservar a localidade espacial no *buffer*, esta abordagem é usada sem qualquer custo extra. Esta abordagem é chamada de ordenação por varredura de plano local, "*local plane-sweep order*", e corresponde ao Algoritmo 3.12.

Algoritmo 3.12 *JuncaoEspacial03* ($R, S: R_Node$; $rect: Retang$) [2]

```

1:  $R = \{E_1 \mid (E_1 \in R) \wedge (E_1.rect \cap rect \neq \emptyset)\}$ 
2:  $S = \{E_1 \mid (E_1 \in S) \wedge (E_1.rect \cap rect \neq \emptyset)\}$ 
3: OrdenaRetang ( $R$ );
4: OrdenaRetang ( $S$ );
5: TesteIntersecaoOrd ( $R, S, Seq$ );
6: para  $i = 1$  ate  $\| Seq \|$  faça
7:    $(E_R, E_S) = Seq[i]$ 
8:   se  $R$  e  $folha$  então
9:      $// S$  também e  $folha$ 
10:    adiciona o par  $E_R$  e  $E_S$  na resposta
11:   senão
12:     $R' \leftarrow$  bloco referenciado em  $E_R.ref$ 
13:     $S' \leftarrow$  bloco referenciado em  $E_S.ref$ 
14:     $rect \leftarrow$  retângulo de intersecao das MBRs do nos  $R'$  e  $S'$ 
15:    JuncaoEspacial03 ( $R', S', rect$ )
16:   fim-se
17: fim-para

```

Existem duas outras variações de ordenação baseadas em varreduras de planos (*plane-sweep*) que são:

- **Ordenação por Varredura de Planos Locais com Pinos:** Primeiro, o algoritmo é igual ao de planos locais que determina um par de entradas (E_R, E_S) . Após processar as sub-árvores $E_R.ref$ e $E_S.ref$ é computado o grau do retângulo de ambas entradas. O grau de um retângulo de uma entrada E é dado pelo número de interseção entre os retângulos $E.rect$ e os retângulos que pertencem às entradas da outra árvore não processadas até agora. Segundo, fixa-se a página no *buffer* com o retângulo que tem o grau máximo. Terceiro, a junção é realizada na página fixada com todas as outras páginas. Então determina-se o próximo par de entradas usando novamente a ordenação por varredura de planos locais.

- **Ordenação em Z Local:** a idéia básica é realizar a ordenação através dos centros dos retângulos representado por pontos bidimensionais. Uma técnica para ordenar pontos bidimensionais é baseada em curvas de preenchimento de espaço, por exemplo curvas de Peano, também chamada de ordenação-Z. Essa técnica consiste em decompor o espaço em células de tamanho igual e realizar a ordenação nesse conjunto de células.

Figura 3.3: Ordenação espacial usando Ordenação-Z [2]

Um exemplo dessa ordenação é ilustrado na Figura 3.3. No diagrama mais à esquerda, todas as interseções são computadas entre os retângulos $(r_1, \dots, r_3)$ e $(s_1, \dots, s_3)$. Isto resulta no diagrama central, que representa os centros dos conjuntos de retângulos $(I, \dots, V)$ por pontos na grade. A ordem das células da grade é determinada pela ordenação-Z que está ilustrada no diagrama mais à direita da Figura 3.3.

3.4 Junção por Similaridade

As operações de junção por similaridade combinam dois conjuntos de objetos complexos em que o resultado contém todos os pares de objetos similares. Os tipos de junção por similaridade são: por abrangência, por vizinhos mais próximos e por proximidade (do inglês *Closest Neighbors Join*).

Figura 3.4: Diferença entre as Operações de Junção por Similaridade [3]

A Figura 3.4 mostra as diferenças entre as operações de junção por similaridade. Como mostrado na Figura 3.4(a), a junção por abrangência $R \bowtie_{\epsilon} S$ de dois conjuntos R e S , é o conjunto de pares onde a distância dos objetos não excede o parâmetro ϵ . A Figura 3.4(b), mostra a junção por proximidade $R \bowtie_{K-CN} S$ que recupera os K -CN pares de R e S com distância mínima. A Figura 3.4(c), mostra a junção por vizinhos mais próximos $R \bowtie_{K-nn} S$ que recupera para cada objeto do conjunto R todos os

K -nn objetos mais próximos do conjuntos S . Pontos de S podem aparecer uma ou mais vezes se o objeto estiver entre K -nn mais próximos de vários pontos em R .

A junção por similaridade tem sido um mecanismo importante para auxiliar buscas por similaridade e *data mining* [53, 54].

A seguir serão apresentadas as implementações das junções por similaridade usando a estrutura espacial *R-Tree*.

3.4.1 Junção por Abrangência

Formalmente, uma operação de junção por abrangência (*Range Join*) pode ser definida como:

$$R \bowtie_{\varepsilon} S := \{(r_i, s_j) \in R \times S : dist(r_i, s_j) \leq \varepsilon\}$$

Assim, a junção por abrangência recebe como parâmetros dois conjuntos de dados R , S , ambos pertencentes a um mesmo domínio, e uma distância máxima de busca ε , sendo que as respostas para essa consulta são pares de objetos, um de cada subconjunto, que estão distantes entre si por no máximo um valor igual ao raio dado.

Bernhard em [55] cita que o algoritmo apresentado na subseção 3.3.1 pode ser adaptado para fazer consulta por abrangência usando a *R-Tree*.

Algumas estruturas baseadas em *grids* que melhoram a junção por abrangência são EGO (*Epsilon Grid Order*) apresentadas em [55] e *Grid Index* apresentado em [56].

3.4.2 Junção por Vizinhos mais Próximos

Formalmente a junção por vizinhos mais próximos (k -nn (*k-Nearest Neighbor Join*)) é definida como:

$R \bowtie_{K-nn} S$ é o menor subconjunto de $R \times S$ que contém para cada ponto de R no mínimo k pontos de S e para os quais a seguinte condição se mantém:

$$\forall (r, s) \in R \bowtie_{K-nn} S, \forall (r, s') \in R \times S \setminus R \bowtie_{K-nn} S : dist(r, s) < dist(r, s')$$

Assim, a junção de vizinhos mais próximos recebe como parâmetro dois conjuntos de dados R e S , ambos pertencentes a um mesmo domínio, e uma quantidade de k qualquer, sendo que, as respostas para essa consulta são os k pares de objetos mais próximos do conjunto R para cada objeto do conjunto S . Em caso de empate de distâncias na maior distância obtida, pode-se selecionar apenas o número necessário de objetos que satisfazem a regra, ou criar uma lista de empates.

Algumas implementações de junção por vizinhos mais próximos usando a *R-Tree* são: o algoritmo *Z-Order* [2], *GORDER* (*G-ordering KNN*) que substitui a contagem das interseções na *R-Tree* pela análise de componentes principais (PCA - *Principal Component Analysis*) apresentado em [57]; *Top-K* apresentado em [58]. Uma extensão feita na *R-Tree* originou a estrutura *MUX* (*Multipage Index*) apresentada em [3, 53, 54, 59].

3.4.3 Junção por Proximidade

A operação de busca por proximidade (*K-CN (K-Closest Neighbors Join)*) [37, 38], recupera os k pares de $R \times S$ que têm a menor distância entre os elementos de cada par. Formalmente, uma junção por proximidade pode ser definida por: $R \bowtie_{K-CN} S$ é o menor subconjunto de $R \times S$ que contém no mínimo k pares de pontos e para o qual a seguinte condição se mantém:

$$\forall (r, s) \in R \bowtie_{K-CN} S, \forall (r', s') \in R \times S \setminus R \bowtie_{K-CN} S : dist(r, s) < dist(r', s')$$

Alguns algoritmos para busca de pares mais próximos [38], serão apresentados em seguida. Os algoritmos apresentados utilizam a *R-Tree*.

Algoritmo Simples A introdução mais simples do problema de busca por proximidade é seguir uma simples solução recursiva para o sub-problema 1-CP e para duas *R-Trees* de mesma altura. Tal algoritmo consiste dos seguintes passos:

- **Passo1:** começar das raízes das duas *R-Trees* e ajustar a distância mínima encontrada, T , para ∞ ;
- **Passo2:** se acessar um par de nós internos, processar os descendentes recursivamente para todos possíveis pares de MBR;
- **Passo3:** se acessar dois níveis, calcular a distância de cada possível par de pontos. Se esta distância é menor que T , atualizar T .

O 1-CP desejado é que um corresponde ao valor final de T .

Algoritmo Completo O melhoramento do algoritmo anterior é fazer uso da parte esquerda da desigualdade 1 e podar alguns caminhos nas duas árvores que não são adequados para conduzir para a melhor solução. Isto é, processa os descendentes somente para aqueles pares de MBR que satisfazem esta propriedade. O passo 2 do algoritmo anterior agora seria:

- **Passo2:** se acessar um par de nós internos, calcular $MINMINDIST$ para cada possível par de MBR. Processar os descendentes recursivamente somente para aqueles pares que têm $MINMINDIST \leq T$.

Algoritmo Recursivo Simples Um melhoramento adicional é tentar minimizar o valor de T assim que possível. Isto pode ser feito utilizando a desigualdade 2, isto é, quando um par de nós internos é visitado, examinar se a desigualdade 2 aplicada para todos pares de MBR, pôde dar um

valor de T menor. Como a desigualdade 2 mantém no mínimo um par de pontos, este melhoramento é produzido para o problema 1-CP. O passo 2 agora seria:

- **Passo2:** se acessar um par de nós internos, calcular o mínimo de MINMAXDIST para todos possíveis pares de MBRs. Se este mínimo é menor que T , atualizar T . Calcular MINMINDIST para cada possível par de MBR. Processar os descendentes recursivamente somente para aqueles pares que têm $MINMINDIST \leq T$.

Algoritmo Recursivo de Distâncias Ordenadas Pares de MBR que têm MINMINDIST menor são mais adequados para conter o 1-CP e conduzir para um T menor. Uma heurística que visa melhorar os algoritmos ainda mais quando dois nós internos são acessados, é ordenar os pares de MBR de acordo com a ordem ascendente de MINMINDIST e obedecer esta ordem processando os descendentes recursivamente. Esta ordem de processamento é exigida para melhorar a poda dos caminhos. O passo 2 do algoritmo anterior seria:

- **Passo2:** se acessar pares de nós internos, calcular o mínimo de MINMAXDIST para todos possíveis pares de MBR. Se este mínimo é menor que T , atualizar T . Calcular MINMINDIST para cada possível par de MBR e ordenar estes pares em ordem ascendente de MINMINDIST. Seguindo esta ordem, processar os descendentes recursivamente somente para aqueles pares que tem $MINMINDIST \leq T$.

Algoritmo Heap Este algoritmo não é recursivo. A fim de superar a recursão e manter a rotina de processamentos dos ascendentes enquanto acessa as duas árvores, é usado um heap. Este heap mantém pares de MBR de acordo com seus MINMINDIST. O par com menor valor consiste no topo do heap. Este par é o próximo candidato a ser visitado. O algoritmo é como segue:

- **Passo1:** iniciar das raízes das duas *R-Trees*, ajustar T para ∞ e inicializar o heap;
- **Passo2:** se acessar um par de nós internos, calcular o mínimo de MINMAXDIST para todos possíveis pares de MBR. Se este mínimo é menor do que T , atualizar T . Calcular MINMINDIST para cada possível par de MBR. Inserir no heap aqueles pares que têm $MINMINDIST \leq T$;
- **Passo3:** se acessar dois níveis, calcular a distância de cada possível par de pontos. Se esta distância é menor que T , atualizar T ;
- **Passo4:** se o heap está vazio então interromper, e

- **Passo5:** pegar o par no topo do heap. Se este tem $MINMINDIST > T$, então interromper. Senão repetir o algoritmo do passo 2 para este par.

O 1-CP é o par que têm distância T . Note que dois ou mais pares que tem o mesmo valor de $MINMINDIST$ podem também aparecer no algoritmo. Se este valor é o mínimo, então mais do que um par apareceria próximo ao topo do heap. As rotinas de tratamento do heap são responsáveis por especificar o par que finalmente ocupa o topo.

Tratamento de Empates Nos algoritmos onde empates de valores de $MINMINDIST$ podem aparecer, é possível alcançar um melhoramento adicional escolhendo o próximo par no caso de um empate usando alguma heurística (e não seguindo a ordem gerada pela ordenação ou pelo algoritmo de tratamento do heap). A lista seguinte apresenta cinco critérios que são adequados para melhorar o desempenho. Assim, no caso de um empate de dois ou mais pares, escolhe-se o par que tem:

1. como um de seus elementos a maior MBR (a área da MBR é representada como uma porcentagem da área da raiz apropriada);
2. o menor $MINMAXDIST$ entre seus dois elementos;
3. a maior soma das áreas de seus dois elementos;
4. a menor diferença de áreas entre a MBR;
5. a maior área de interseção entre seus dois elementos.

Tratamento de Diferentes Alturas Quando as duas *R-Trees* armazenam os dois conjuntos de pontos que tem alturas diferentes os algoritmos são levemente mais complicados. Em algoritmos recursivos, existem duas abordagens para tratamento de diferentes alturas.

- A primeira abordagem é chamada de *fix-at-root*. A idéia é, quando o algoritmo é chamado com um par de nós internos em diferentes níveis, o processamento aos ascendentes termina na árvore de nó de nível mais baixo, enquanto que o processamento na outra árvore continua até um par de nós no mesmo nível seja atingido. Então o processamento continua em ambas as *sub-trees*, e
- A segunda abordagem é chamada de *fix-at-leaves* e trabalha de maneira oposta. Processa os descendentes recursivamente como habitual. Quando o algoritmo é chamado com um nó folha de um lado e um nó interno de outro, o processamentos dos descendentes pára na árvore do nó folha, enquanto o processamento nas outras árvores continua.

Por exemplo, para o algoritmo Simples, a aplicação da estratégia *fix-at-leaves* resulta no seguinte passo extra:

- **Passo2.1:** se acessa um par com um folha e um nó interno, calcula o mínimo de MINMAXDIST para todos possíveis pares de MBR que consistem da MBR deste folha e a MBR contida no nó interno. Se este mínimo é menor do que T , atualiza T . Calcula também MINMINDIST para todos estes pares. Para cada par que tem $MINMINDIST \leq T$, faz uma chamada recursiva com parâmetros dos nós correspondente às MBR do par (note que um parâmetro será o nó folha e o outro parâmetro será um filho do nó interno).

Enquanto que a aplicação da estratégia *fix-at-root*, para o mesmo algoritmo resulta no seguinte passo extra:

- **Passo2.1:** se acessa dois nós residindo em diferentes níveis, calcula o mínimo de MINMAXDIST para todos possíveis pares de MBR que consistem de um MBR contido no nó de nível mais alto e a MBR do nó no nível mais baixo. Se este mínimo é menor do que T , atualiza T . Calcula também MINMINDIST para todos estes pares. Para cada par que tem $MINMINDIST \leq T$, faz uma chamada recursiva com parâmetros do nó correspondente à MBR do par (note que um parâmetro será o nó residente no nível mais baixo).

O algoritmo do *Heap* pode ser modificado para lidar com diferentes alturas pelas estratégias de *fix-at-root* e *fix-at-leaves*. O passo extra que precisa ser adicionado ao algoritmo *Heap* é semelhante a um dos passos apresentados acima, dependendo da estratégia. A única diferença é que a chamada recursiva é substituída pela inserção no heap.

Estendendo para os k Pares mais Próximos Para esclarecer o problema K-CP é necessário uma estrutura extra que mantém os k pares mais próximos. Esta estrutura é organizada com um *k-heap* e mantém pares de pontos conforme suas distâncias. Os pares de pontos com a maior distância sempre ficam no topo do heap.

- Inicialmente o *k-heap* está vazio.
- Os pares de pontos explorados no passo 3 são inseridos no *k-heap* até ele ficar cheio.
- Então, quando um novo par de pontos é explorado no passo 3 e se sua distância é menor do que o topo do *k-heap*, o topo é removido e seu par é inserido no *k-heap*.

As adições acima são necessárias e suficientes para os algoritmos Simples e Completo para resolver o problema K-CP e foram usados na implementação das versões K-CP destes algoritmos.

Estas adições não são suficientes para os algoritmos recursivo simples, recursivo de distâncias ordenadas e *heap*. Estes algoritmos fazem uso da desigualdade 2 que, em geral não mantém mais de um par de pontos. Isto significa que atualizar T baseado nos valores de MINMAXDIST deve ser descartado. Uma simples modificação que pode fazer esses três algoritmos adequados para resolver o problema K-CP é usar a distância do topo do k -*heap* como valor de T , logo após o k -*heap* ter k elementos. Enquanto o k -*heap* tem *slots* vazios, infinito seria usado como T .

O melhoramento da modificação acima (que foi usada na implementação das versões k-CP dos três algoritmos) é fazer uso da parte direita da desigualdade 1, enquanto poda caminhos desnecessários nas duas árvores. Isto é, entre um número de pares de MBR para encontrar um com MAXMAXDIST que pode atualizar o valor de T . Note que um par de MBR (M_P, M_Q) pertencente em R_P e R_Q , respectivamente, pode indicar no mínimo

$$m_P^{nível\ de\ M_P} \times m_Q^{nível\ de\ M_Q} \quad (3.1)$$

pares de pontos (m_P e m_Q são as saídas mínimas de R_P e R_Q , respectivamente). Isto leva à seguinte política quando o algoritmo trata um grupo de pares de MBR.

- MAXMAXDIST é calculado para cada par.
- Os pares são ordenados de acordo com MAXAXDIST (criando uma sequência de pares e um *array* de MAXMAXDIST).
- Usando a fórmula 3.1, o número de pontos X , que pode ser no mínimo produzido do *primeiro, primeiro+segundo, primeiro+segundo+...+iesimo* pares desta sequência são calculados até $X \geq K$.
- MAXMAXDIST[i] < T, então $T = \text{MAXMAXDIST}[i]$.

Os algoritmos recursivo simples e recursivo de distâncias ordenadas aplicam esta estratégia para atualizar T no passo2 (e passo2.1 em caso de alturas diferentes). O grupo de pares de MBR que pegam parte nos cálculos acima é o grupo de pares já usados neste passo.

O algoritmo *Heap* também aplica esta estratégia para atualizar T no passo2 (e passo2.1 em caso de alturas diferentes). De qualquer modo, o grupo de pares de MBR que pegam parte nos cálculos acima é o grupo de pares já usados neste passo mais os pares de MBR já inseridos no *heap*.

3.5 Conclusão

Neste capítulo, foram discutidos os tipos e implementações de junção. Sobre as junção por similaridade, a literatura cita a somente o uso da estrutura espacial *R-Tree* e variações.

Os tipos de junção por similaridade citados foram: por abrangência, por vizinhos mais próximos e por proximidade (do inglês *closest neighbors join*) [54].

No próximo capítulo será apresentada nova estrutura métrica baseada em particionamento do espaço métrico, seus algoritmos de consulta por abrangência e por vizinhos mais próximos e experimentos.

CAPÍTULO 4 Coordenadas Métricas

Muitas estruturas de indexação utilizam o conceito de região de cobertura para estabelecer seus níveis hierárquicos, sendo que as regiões de um mesmo nível podem apresentar sobreposição entre si. Esse conceito é aplicado tanto na construção de diversos métodos de acesso métrico quanto de métodos de acesso espacial.

Quando um algoritmo de consulta intercepta uma região de sobreposição, ele é obrigado a navegar por todas as regiões que compõem a sobreposição. Dessa forma, deve-se evitar que exista muita sobreposição para não prejudicar os algoritmos de consulta. No entanto, o algoritmo de quebra de nós da operação de inserção pode incrementar a cobertura de um nível mais baixo e processar o incremento até o nível raiz, uma vez que os níveis superiores mantêm a cobertura de todos seus níveis inferiores.

Esse incremento recursivo nas regiões de cobertura, originado pela quebra de nós da operação de inserção, aumenta a sobreposição entre regiões de um mesmo nível. Quando uma estrutura apresenta diversos níveis e muita sobreposição entre as regiões de cobertura de um mesmo nível, os algoritmos de consultas acabam realizando um número maior de acessos a disco e cálculos de distância do que se a estrutura apresentasse poucos níveis.

Para não gerar estruturas altas que aumentam o custo das consultas por similaridade, adota-se um tamanho de bloco maior em disco fazendo com que a estrutura cresça em largura e diminua em altura.

A altura máxima que não prejudica o desempenho das consultas depende muito da quantidade de registros, do tipo de dado e da distribuição de inserção dos dados, o que dificulta a estimativa do tamanho de bloco.

Portanto, estruturas que não apresentam sobreposição entre os níveis não influenciam seus algoritmos de consulta por similaridade com o crescimento da altura da estrutura.

Um conceito que não apresenta sobreposição entre elementos de um mesmo nível é o de particionamento disjunto de região. Esse conceito já é usado em algumas estruturas como a *D-Index*

[25, 46].

Neste capítulo dois novos métodos de acesso métrico que usam um novo conceito de particionamento disjunto de região, chamado de coordenada métrica.

4.1 Coordenadas Métricas

Este trabalho introduz um novo método para particionar o espaço métrico em regiões disjuntas empregando o conceito, também desenvolvido neste trabalho, chamado de coordenadas métricas.

Uma coordenada métrica é um vetor de valores inteiros $C = (c_i \in \mathbb{N} \mid c_i < n)$ sendo que $i \in [0, (n - 1)]$, onde n é a quantidade de mapeadores. Um mapeador é um objeto escolhido entre os objetos armazenados no nó. Inicialmente uma coordenada métrica usa dois mapeadores, sendo que a quantidade de mapeadores pode ser incrementada a qualquer momento.

A coordenada métrica é definida através da função f_{CM} que mapeia qualquer objeto O com um conjunto de objetos mapeadores $M = \{m_0, m_1, m_2, \dots, m_n\}$. Formalizando:

$$f_{CM} : O \times M \rightarrow C$$

Para realizar esse mapeamento, a função f_{CM} usa as distâncias entre pares de mapeadores. Essa distância é definida através de uma função de distância métrica.

As distâncias entre os mapeadores formam uma matriz $A = \{a_{ij} \in \mathbb{R} \mid a_{ij} = \text{dist}(m_i, m_j)\}$ sendo que $i \in [0, (n - 1)]$ e $j \in [0, (n - 1)]$, onde cada linha dessa matriz apresenta intervalos de distâncias para um mapeador base m_i . Por definição se um elemento $a_{i(j+1)} < a_{ij}$ deve-se inverter os valores dos elementos para manter uma ordem crescente de distâncias para o mapeador base m_i , ou seja, $a_{i(j+1)} \leftarrow a_{ij}$ e $a_{ij} \leftarrow a_{i(j+1)}$. As distâncias $a_{ij} \mid j < n - 1$ para o mapeador base m_i formam vários círculos com raios centrados no mapeador base m_i delimitando o espaço métrico em regiões disjuntas.

Cada coordenada c_i é definida através da distância entre um objeto e o objeto mapeador base m_i . O valor é determinado pelo índice j do intervalo de distâncias a_{ij} em que a distância entre um objeto O qualquer e um objeto mapeador base m_i satisfaz a expressão $a_{ij} \leq \text{dist}(O, m_i) < a_{i(j+1)}$.

Para ser armazenado, o vetor de inteiros das coordenadas métricas é convertido para um valor numérico através de:

$$\sum_{i=0}^{n-1} c_i \cdot n^{n-1}$$

onde c_i é a coordenada métrica e n é a quantidade de objetos mapeadores.

As coordenadas métricas permitem extensão ou incremento da quantidade de coordenadas através da adição de um novo objeto mapeador m_{novo} . Essa característica de extensibilidade será apresentada na subseção 4.2.1.

4.2 Ilustração do conceito

Os exemplos a seguir ilustram o conceito de coordenadas métricas desde a criação e a extensibilidade.

Inicialmente um *bucket* folha recebe objetos sem que seja necessário mapear para uma coordenada métrica, até que esse *bucket* folha fique cheio. Quando o *bucket* folha fica cheio, é necessário dividir os objetos armazenados nesse *bucket* entre dois *buckets*. Existem três formas de divisão do *bucket* folha: divisão com escolha de dois mapeadores, divisão de valores de coordenadas e divisão com incremento de mapeadores.

A *divisão com escolha de dois mapeadores* cria o esquema de coordenadas métricas através de dois objetos mapeadores que particionam o espaço métrico em 2^2 regiões.

Figura 4.1: Particionamento com 2 mapeadores

A Figura 4.1 mostra como exemplo um espaço bidimensional euclidiano que é inicialmente particionado através de dois mapeadores E_0 e E_1 , e uma distância r_0 entre esses mapeadores. Assim qualquer objeto O terá 2 coordenadas métricas que variam de 0 até 1, sendo $M = \{E_0, E_1\}$.

Para todo objeto, pertencente ao *bucket* folha cheio, deverá ser calculada a sua coordenada métrica através da função f_{CM} . A coordenada de cada objeto deve ser convertida para o valor numérico a ser armazenado, pela forma mencionada anteriormente. Assim, o objeto que tem a coordenada $\{0,1\}$ pertence à região 0 em relação ao primeiro mapeador e na região 1 em relação ao segundo mapeador, como pode ser visto na Figura 4.1.

Todos os objetos com suas respectivas coordenadas são divididos entre dois *buckets* folhas, sendo que um mesmo *bucket* deve receber todas coordenadas repetidas. A estrutura de índice deve guiar os valores de coordenadas para os respectivos *buckets*.

A *divisão de valores de coordenadas* ocorre quando um *bucket* folha cheio possuir coordenadas diferentes permitindo a divisão das coordenadas métricas entre o *bucket* cheio e um novo *bucket*.

A *divisão com incremento de mapeadores* ocorre quando todas as coordenadas de um *bucket* cheio possuem o mesmo valor, não sendo mais possível realizar a operação divisão de valores de

Figura 4.2: Particionamento com 3 Mapeadores

coordenadas. Nesse momento deve-se estender a coordenada métrica escolhendo um novo mapeador, sendo que três mapeadores particionam o espaço métrico em 3^3 regiões.

A Figura 4.2 mostra duas representações de particionamento do espaço métrico usando 3 mapeadores assumindo um espaço bidimensional euclidiano. No lado esquerdo da Figura 4.2, o mapeador E_0 particiona o espaço com duas circunferências concêntricas: uma com raio r_0 que é sua distância para o mapeador E_1 e outra com raio r_2 que é sua distância para o mapeador E_2 . Esse mesmo princípio é aplicado para os mapeadores E_1 e E_2 que também particionam o espaço com duas circunferências concêntricas, delimitando as 27 regiões disjuntas.

O lado direito da Figura 4.2 apresenta três semi-retas que indicam os possíveis valores para cada coordenada. Os segmentos das retas representam uma escala de distâncias ordenadas entre o objeto mapeador base do segmento e os demais objetos mapeadores.

Dessa maneira, qualquer objeto O terá 3 coordenadas métricas que variam de 0 até 2. O valor da primeira coordenada depende da distância do objeto O para o mapeador E_0 podendo ser:

- $0 \leftarrow \text{dist}(E_1, E_0) > \text{dist}(O, E_0)$;
- $1 \leftarrow \text{dist}(E_1, E_0) \leq \text{dist}(O, E_0) < \text{dist}(E_2, E_0)$ e
- $2 \leftarrow \text{dist}(E_2, E_0) \leq \text{dist}(O, E_0)$.

As demais coordenadas obedecem ao mesmo princípio, mudando a ordem para os mapeadores E_1 e E_2 .

Um exemplo pode ser visto na Figura 4.3, onde um objeto qualquer é mapeado para a coordenada métrica $\{0, 1, 2\}$. Essa coordenada indica que o objeto encontra-se na primeira região em relação

Figura 4.3: Objeto Mapeado para a região 0, 1, 2

Figura 4.4: Extensibilidade das Coordenadas Métricas. (a) Particionamento com 3 Mapeadores. (b) Extensão para 4 Mapeadores.

ao mapeador E_0 , na segunda região em relação ao mapeador E_1 e na terceira região em relação ao mapeador E_2 .

4.2.1 Extensibilidade

A Figura 4.4a mostra o espaço métrico particionado com 3 objetos mapeadores. Suponha-se agora que, um grande número de objetos mapeia para a região $\{2,1,0\}$ e por isso essa região precisa ser dividida. Assim, é escolhido um objeto mapeador E_3 da região $\{2,1,0\}$ dividindo essa coordenada como pode ser visto na Figura 4.4b, atualizando assim a matriz A.

O importante é fazer o ajuste nas coordenadas antigas, sendo que:

- todas as regiões com numeração maior do que a região dividida deverão ser incrementadas em um; isso pode ser feito sem nenhum novo cálculo de distância;
- todas as regiões com numeração menor do que a região dividida deverão permanecer com a

mesma numeração, o que também não causa nenhum cálculo de distância; e

- para todas as regiões com numeração igual à região dividida, deve-se calcular a distância para o novo objeto, para determinar se esse permanece com o mesmo valor ou se incrementa em um.

Além de ajustar as coordenadas antigas, deve-se calcular um novo dígito em relação ao mapeador E_3 . Por exemplo, um objeto X com coordenada antiga $\{1,1,1\}$ em que o novo mapeador E_3 foi escolhido na região $\{2,1,0\}$ será convertida para:

- primeiro dígito permanece 1 $\rightarrow$ pois essa coordenada é menor que a primeira coordenada do novo mapeador E_3 com valor dois;
- segundo dígito é 1 ou 2 $\rightarrow$ se a distância de E_1 (objeto base) para o objeto X for menor ou maior que o elemento a_{11} da matriz A, o dígito permanece 1; caso contrário o dígito torna-se 2; e
- terceiro dígito é incrementado para 2 $\rightarrow$ pois essa coordenada é maior que a terceira coordenada do novo mapeador E_3 com valor zero;
- quarto dígito $\rightarrow$ será realizado um cálculo de distância para determinar para qual região o objeto será mapeado.

Várias técnicas para a escolha do novo objeto mapeador podem ser adotadas, sendo que nesse trabalho utilizou-se as seguintes técnicas: randômica, variância e contagem.

A técnica randômica escolhe aleatoriamente um objeto mapeador entre os que estão armazenados no *bucket* folha cheio.

A técnica da variância escolhe o objeto mapeador que gera a maior variância de coordenadas métricas entre os objetos que estão armazenados no *bucket* folha cheio. Essa técnica tem custo quadrático em relação ao acesso de objetos ao *bucket* dividido, pois todos os objetos armazenados no *bucket* são eleitos um a um como mapeador para verificar aquele que gera a maior variância.

A técnica da contagem escolhe o objeto mapeador que gera a maior variedade de coordenadas métricas diferentes entre os objetos que estão armazenados no *bucket* folha cheio. Essa técnica também tem custo quadrático, pois todos os objetos armazenados no *bucket* são eleitos um a um como mapeador para verificar aquele que gera a maior variedade.

Problema das distâncias mínimas

Para que ocorra a extensibilidade das coordenadas métricas é imprescindível que o novo objeto mapeador escolhido gere novas coordenadas para os demais objetos do conjunto. No entanto, nem sempre

isso pode ser garantido.

Isso acontece quando a diferença entre quaisquer dois elementos seguidos da matriz de distâncias A possuir o menor valor de distância diferente de zero, ou seja, $a_{ij} - a_{i(j+1)} = MINDIST$. Assim não é possível mais dividir essa região e os objetos continuam sendo mapeados para a mesma região.

Esse problema pode acontecer em espaços métricos onde o domínio de valores possíveis para a função de distância é discreto.

Essa situação foi chamada de problema das distâncias mínimas e deve ser tratada pelos métodos de acesso métrico na seção 4.3.

4.3 Métodos de Acessos Métricos

Neste trabalho foram implementados dois métodos de acessos métrico, de inserção *top-down*, baseados em funções de mapeamento local: a **Partição Local** e uma extensão chamada de **Partição Local+**.

Os blocos dessas estruturas são chamados de *buckets* e esses são de dois tipos, *bucket* índice e *bucket* folha e serão detalhados para cada estrutura nas próximas seções.

4.3.1 Partição Local

Nos métodos de acesso métrico baseado em particionamento local, cada *bucket* na camada índice da estrutura, define uma nova função f_{CM} .

Uma coordenada métrica de um *bucket* índice redireciona para um *bucket* inferior. Se o *bucket* inferior é índice, ele define uma nova função f_{CM} . Se o *bucket* inferior é folha, ele apresenta os objetos indexados.

Figura 4.5: *Bucket* Índice para Partição Local

Para que um *bucket* índice possa definir a função f_{CM} , ele deve armazenar os n mapeadores, as distâncias entre os mapeadores e n^n identificadores para os blocos internos pois, cada coordenada direciona para um *bucket* sendo que, mais de uma coordenada pode redirecionar para o mesmo *bucket*. A Figura 4.5) mostra a organização de um *bucket* índice que é composto por três partes:

- *header*: contém informações do tipo do *bucket* (folha ou índice); próximo bloco vizinho que

está no mesmo nível hierárquico que é atribuído no momento de quebra do bloco; a quantidade de objetos mapeadores no *bucket*;

- *entries*: contém um vetor de marcas de início (*off-set*) para as informações de cada mapeador. Essa técnica foi adotada para evitar a reescrita das informações já existentes no *bucket* pois, a adição de um novo mapeador implica em aumentar a quantidade de distância entre os mapeadores e a quantidade de coordenadas métricas; e
- informações dos mapeadores: cada seção marcada pelos marcadores (*off-set*) com exceção do primeiro contém um vetor de identificadores de blocos para os *buckets* internos, um vetor com as distâncias para os objetos anteriores, e o objeto. O vetor de distâncias entre os mapeadores evita cálculos de distâncias já computados previamente.

Quando um *bucket* índice contém 2 mapeadores e um novo mapeador é adicionado, deve-se inserir também duas distâncias para os mapeadores anteriores e 23 identificadores de blocos que somados com os quatro identificadores anteriores totalizam 3^3 coordenadas.

A Tabela 4.1 mostra a quantidade de *bytes* (segunda coluna) usada para armazenar a quantidade de mapeadores da primeira coluna sem contabilizar o tamanho dos objetos mapeadores.

Tabela 4.1: Tamanho do *Bucket* Índice sem Objetos Mapeadores

Número de Mapeadores	Tamanho (<i>bytes</i>)
2	39
3	143
4	1.071
5	12.559

O incremento é ilimitado, mas o tamanho do *bucket* limita a quantidade de objetos mapeadores. Quando não existe espaço suficiente no *bucket* para o incremento de mapeadores, cria-se um novo nível, executando a operação de divisão com escolha de dois mapeadores, mencionada anteriormente.

A grande vantagem é que essa organização faz com que as estruturas sejam maiores em largura do que as estruturas baseadas em região de cobertura. Por exemplo, em blocos de 2048 *bytes* com 4 mapeadores, a estrutura pode ter até 4^4 ou 256 *sub-buckets*, pois 4 mapeadores permitem 256 coordenadas métricas. No entanto, um mesmo sub-bucket pode mapear mais de uma coordenada métrica podendo diminuir a quantidade de *sub-buckets*.

O método de inserção *top-down* faz com que as estruturas de particionamento fiquem mais altas em alguns ramos, comparada com as estruturas *bottom-up*, mas a maioria dos ramos apresenta poucos níveis por causa da largura da estrutura.

Figura 4.6: *Bucket* Folha para Partição Local

O *bucket* folha apresentado na Figura 4.6, armazena todos os objetos indexados pelo método de acesso métrico, e é organizado pelos seguintes campos:

- *header*: contém informações do tipo do *bucket* (folha ou índice); próximo bloco vizinho que está no mesmo nível hierárquico; a quantidade de objetos mapeadores do nível superior; a ocupação de objetos no *bucket*; o identificador de blocos para uma lista de *buckets* folhas que tratam o problema das distâncias mínimas;
- *entries*: contém um vetor de marcas de início (*off-set*) para cada objeto, bem como sua coordenada métrica; e
- objetos: contém todos os objetos armazenados no *bucket* folha;

A inserção na estrutura descrita no Algoritmo 4.1 mostra que, para cada *bucket* índice navegado, deve-se calcular a coordenada métrica para selecionar o *sub-bucket* que mapeia a coordenada. Quando o algoritmo chega em um *bucket* folha o objeto deve ser inserido.

Se um *bucket* folha não contém espaço para a inserção de um objeto, é realizado um dos 3 métodos de divisão: *divisão com escolha de dois mapeadores*; *divisão de valores de coordenadas* e *divisão com incremento de mapeadores*.

A tática de *divisão com escolha de dois mapeadores* é realizada quando o único *bucket* folha está cheio ou um *bucket* índice não suporta mais ser estendido, portanto, deve-se criar uma nova função que escolhe os dois objetos mapeadores. Essa tática possui 3 etapas:

1. Escolher dois objetos mapeadores que estão no *bucket* cheio, usando uma das técnicas: randômica, variância ou contagem;
2. Dividir as novas coordenadas métricas calculadas através dos dois mapeadores entre um *bucket* cheio e um novo *bucket*; e
3. Criar um *bucket* índice e ajustar os identificadores de blocos entre os dois *buckets*, conforme a coordenada métrica que cada *bucket* armazena.

Algoritmo 4.1 *Inserir(novoObj)*

```

1: bucket ← bucket raiz
2: enquanto bucket é ÍNDICE faça
3:   chave ← função mapeadora do bucket para novoObj
4:   bucket ← bloco que tem o identificador subBucket[chave]
5: fim-enquanto
6: se tem espaço no bucket então
7:   adiciona um novoObj e chave no bucket
8: senão
9:   se bucket raiz não é ÍNDICE então
10:    divisão com escolha de dois mapeadores
11:   senão
12:    se bucket pode ser dividido sem adicionar mapeador então
13:      divisão de valores de coordenadas
14:    senão
15:      se pode ser adicionado novo mapeador no bucket então
16:        divisão com incremento de mapeadores
17:      senão
18:        divisão com escolha de dois mapeadores
19:      fim-se
20:    fim-se
21:  fim-se
22: fim-se

```

A tática *divisão de valores de coordenadas* é realizada quando o *bucket* cheio mapeia mais de um valor de coordenada. Com ela é possível dividir o *bucket* folha sem que haja necessidade de incrementar o tamanho da coordenada métrica. Essa tática possui 2 etapas:

1. Dividir as coordenadas métricas entre um *bucket* cheio e um novo *bucket*; e
2. Ajustar os identificadores de blocos entre os dois *buckets*, conforme a coordenada métrica que cada *bucket* armazena.

A tática de *divisão com incremento de mapeadores* é realizada quando não é possível realizar a divisão de valores de coordenadas e há espaço no *bucket* índice para o incremento. Esse algoritmo possui 5 etapas:

1. Verificar se o *bucket* apresenta o problema das distâncias mínimas, ou seja, se não é possível escolher um novo mapeador que consiga gerar novos valores de coordenadas. Se apresentar esse problema, deve-se criar uma lista de *buckets* folhas que mapeiam as coordenadas métricas que colidem e as demais etapas do algoritmo não são realizadas;
2. Verificar se o *bucket* índice que antecede o *bucket* folha na hierarquia, tem capacidade para armazenar um novo mapeador;
 - (a) Se o *bucket* índice não tem capacidade para o novo mapeador, deve-se executar a tática de divisão com escolha de dois mapeadores e o algoritmo se encerra em seguida;

Algoritmo 4.2 *ConsultaPorAbrangencia(bucket, objConsulta, raio, inicio, termino)*

```

1: se bucket é ÍNDICE então
2:   início ← função mapeadora de início de faixa do bucket para o objConsulta e o raio
3:   término ← função mapeadora de término de faixa do bucket para o objConsulta e o raio
4:   para cada coordenada chave entre inicio e termino faça
5:     bucket ← bloco que tem identificador subBucket[chave] sem ler mais que uma vez o mesmo bloco
6:     ConsultaPorAbrangencia(bucket, objConsulta, raio, inicio, termino)
7:   fim-para
8: senão
9:   para cada bucket colisão ← bucket da lista de buckets FOLHA faça
10:    para cada objeto obj[i] na colisão faça
11:      se coordenada métrica de obj[i] está entre a faixa de inicio e termino então
12:        se distância(obj[i], objConsulta) ≤ raio então
13:          adiciona obj[i] na lista de resultado
14:        fim-se
15:      fim-se
16:    fim-para
17:  fim-para
18: fim-se

```

mesmo *bucket*. Quando o *bucket* é folha, são descartados os objetos que não estão dentro da faixa de início e término das coordenadas de consulta e, aqueles que estão dentro da faixa, calcula-se a distância para o objeto de consulta para verificar se fazem parte da resposta.

4.3.3 Consulta aos Vizinhos mais Próximos

Para uma consulta aos vizinhos mais próximos são dados um objeto q e uma quantidade de objetos k , e a estrutura retorna os k objetos mais próximos do objeto q . O algoritmo tem 2 etapas:

- criar uma primeira lista de resposta até completar a quantidade k para passar a ter um raio limite, e
- com o raio limite inicial, verificar nas regiões que se qualificaram se existem outros objetos com distância menor.

A Figura 4.8 ilustra como é feita a primeira etapa do algoritmo que cria uma lista de resposta com os k objetos mais próximos do objeto q que estão na coordenada $\{0,1,2\}$, onde está mapeado o objeto de consulta q . Se existirem no mínimo k objetos nessa região, consegue-se ter um raio inicial r_k . Caso não tenha k objetos, navega-se entre as regiões vizinhas até conseguir ter k objetos e assim ter um raio inicial r_k .

O Algoritmo 4.3 mostra a implementação da primeira etapa do algoritmo de consulta aos vizinhos mais próximos.

A segunda etapa do algoritmo de consulta aos vizinhos mais próximos é basicamente o algoritmo de consulta por abrangência modificado para que quando forem adicionados elementos na lista de

Figura 4.8: Coordenada que encontra-se o objeto de consulta q

Algoritmo 4.3 *ConsultaVizinhos(bucket, objConsulta, k, inicio, termino, raioMax)*

```

1: se bucket é ÍNDICE então
2:   inicio ← menor coordenada métrica para a função mapeadora do bucket
3:   termino ← maior coordenada métrica para a função mapeadora do bucket
4:   chave ← função mapeadora do bucket para objConsulta
5:   bucket ← bloco que tem identificador subBucket[chave]
6:   ConsultaVizinhos(bucket, objConsulta, k, inicio, termino, raioMax)
7:   ConsultaRaioVizinhos(bucket, objConsulta, k, inicio, termino, raioMax)
8: senão
9:   para cada bucket colisão ← bucket da lista de buckets FOLHA faça
10:    para cada objeto obj[i] na colisão faça
11:      se coordenada métrica de obj[i] está entre a faixa de inicio e termino então
12:        se distância(obj[i], objConsulta) ≤ raioMax então
13:          adiciona obj[i] na lista de resultado
14:          se quantidade de elementos em resultado ≤ k então
15:            remove em resultado os elementos excedentes ao k
16:            raioMax ← maior distância dos elementos de resultado
17:        fim-se
18:      fim-se
19:    fim-se
20:  fim-para
21: fim-para
22: fim-se

```

resposta, remova-se os elementos excedentes ao k e diminuindo o *raio* máximo para maior distância dos elementos remanescentes na lista de resposta.

O Algoritmo 4.4 mostra a implementação da segunda etapa do algoritmo de consulta aos vizinhos mais próximos.

4.3.4 Partição Local+

A estrutura de partição local+ foi gerada a partir das modificações feitas na estrutura de Partição Local para armazenar no *bucket* folha a distância para o primeiro mapeador para cada entrada no *bucket*.

Essa modificação foi realizada depois de observar que nos blocos folhas das estruturas métricas com regiões de cobertura, a distância para o representante evita a realização de muitos cálculos de

Algoritmo 4.4 *ConsultaRaioVizinhos(bucket, objConsulta, k, inicio, termino, raioMax)*

```

1: se bucket é ÍNDICE então
2:   início ← função mapeadora de início de faixa do bucket para o objConsulta e o raioMax
3:   término ← função mapeadora de término de faixa do bucket para o objConsulta e o raioMax
4:   para cada coordenada chave entre início e termino faça
5:     bucket ← bloco que tem identificador subBucket[chave] sem ler mais que uma vez o mesmo bloco
       ConsultaRaioVizinhos(bucket, objConsulta, k, início, termino, raioMax)
6:   fim-para
7: senão
8:   para cada bucket colisão ← bucket da lista de buckets FOLHA faça
9:     para cada objeto obj[i] na colisão faça
10:      se coordenada métrica de obj[i] está entre a faixa de início e termino então
11:        se distância(obj[i], objConsulta) ≤ raioMax então
12:          adiciona obj[i] na lista de resultado
13:        se quantidade de elementos em resultado ≤ k então
14:          remove em resultado os elementos excedentes ao k
15:        raioMax ← maior distância dos elementos de resultado
16:      fim-se
17:    fim-se
18:  fim-se
19: fim-para
20: fim-para
21: fim-se

```

distâncias eliminando objetos que não fazem parte da resposta através da desigualdade triangular.

Entretanto, em um nível folha das estruturas de particionamento não existe um representante, mas deve existir dois ou mais mapeadores. Dessa maneira, optou-se em armazenar apenas a distância do objeto ao primeiro mapeador. Para isso, o algoritmo de inserção foi adaptado para armazenar no *bucket* folha a distância de cada objeto para seu primeiro mapeador, como mostra a Figura 4.9.

Figura 4.9: *Bucket* Folha para Partição Local+

Os algoritmos de consulta por abrangência e consulta aos vizinhos mais próximos, além de verificar se a coordenada métrica do objeto no *bucket* folha está entre o início e término das coordenadas de resposta, devem também usar a distância para o primeiro mapeador para excluir objetos que não fazem parte da resposta através da desigualdade triangular.

4.4 Experimentos

Para avaliar a adequação dessas estruturas foram realizados experimentos com 3 conjuntos de dados distintos que são: pontos geográficos do projeto *GEOnet Names Server* [60], proteínas de diversos seres vivos do projeto *UNIPROT (Universal Protein Resource)* [61] e palavras de todas as línguas dos dicionários de ortografia do *software OpenOffice* [62].

A máquina utilizada para realizar esses experimentos possui a seguinte configuração: processador Intel de 2.08GHz com *cache L2* de 512K, 3Gb de memória RAM, HD 200Gb 7200rpm.

4.4.1 Conjunto de Proteínas

O conjunto de dados do projeto *UNIPROT (Universal Protein Resource)* contém 92.525 proteínas filtradas para ter no máximo 64 aminoácidos (média de 43 aminoácidos) de diversos seres vivos.

A função de distância criada realiza o alinhamento métrico entre duas proteínas. Esse alinhamento usa programação dinâmica para implementar a função de distância *Levenshtein* [32] tratando o problema da transposição [63], como pode ser visto no Algoritmo 4.5.

Algoritmo 4.5 *DistProteinas(prot1, prot2)*

```

1: dist[tamanho prot1 + 1][tamanho prot2 + 1]
2: para cada linha i da matriz dist faça
3:   para cada coluna j da matriz dist faça
4:     se é a primeira linha da matriz dist então
5:       dist[i][j] ← 8
6:     senão
7:       se é a primeira coluna da matriz dist então
8:         dist[i][j] ← 8
9:       senão
10:        inserir ← dist[i][j-1]+8
11:        remover ← dist[i-1][j]+8
12:        trocar ← dist[i-1][j-1]+getMPAM(prot1[i-1],prot2[j-1])
13:        se é a segunda linha ou é a segunda coluna da matriz dist então
14:          dist[i][j] ← menor valor entre inserir, remover e trocar
15:        senão
16:          transp ← dist[i-2][j-2]+getMPAM(prot1[i-2],prot[j-1])+getMPAM(prot1[i-1],prot2[j-2])+8
17:          dist[i][j] ← menor valor entre inserir, remover, trocar e transp
18:        fim-se
19:      fim-se
20:    fim-se
21:  fim-para
22: fim-para

```

Para realizar o alinhamento métrico, a função de distância *Levenshtein* é adaptada para usar uma matriz métrica de pesos entre os 20 aminoácidos, chamada MPAM [64], mostrada na Figura 4.10. Nessa implementação foi usado um peso constante para o deslocamento de partes da proteína chamado de “gap”. O valor do gap é a maior distância entre os aminoácidos da MPAM incrementado por

um, pois a penalidade é maior para deslocar parte da proteína do que para trocar um aminoácido por outro.

	A	R	W	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	0	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	2	5	4	2
R	2	0	2	2	4	2	2	2	2	3	3	2	2	4	2	2	2	4	4	3
N	2	2	0	2	4	2	2	2	2	3	3	2	2	4	2	2	2	5	4	2
D	2	2	2	0	4	2	2	2	2	3	3	2	3	4	2	2	2	6	4	2
C	3	4	4	4	0	4	4	3	4	3	4	4	4	4	3	3	3	7	3	3
Q	2	2	2	2	4	0	2	2	2	3	3	2	2	4	2	2	2	5	4	3
E	2	2	2	2	4	2	0	2	2	3	3	2	3	4	2	2	2	6	4	2
G	2	2	2	2	3	2	2	0	2	2	3	2	2	4	2	2	2	6	4	2
H	2	2	2	2	4	2	2	2	0	3	3	2	3	3	2	2	2	5	3	3
I	2	3	3	3	3	3	2	3	0	1	3	2	2	2	2	2	2	5	3	2
L	2	3	3	3	4	3	3	3	3	1	0	3	1	2	3	3	2	4	2	1
K	2	2	2	2	4	2	2	2	2	3	3	0	2	4	2	2	2	4	4	3
M	2	2	2	3	4	2	3	2	3	2	1	2	0	2	2	2	2	4	3	2
F	3	4	4	4	4	4	4	4	3	2	2	4	2	0	4	3	3	3	1	2
P	2	2	2	2	3	2	2	2	2	2	3	2	2	4	0	2	2	5	4	2
S	2	2	2	2	3	2	2	2	2	2	3	2	2	3	2	0	2	5	4	2
T	2	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	0	5	3	2
W	5	4	5	6	7	5	6	6	5	5	4	4	4	3	5	5	5	0	4	5
Y	4	4	4	4	3	4	4	4	3	3	2	4	3	1	4	4	3	4	0	3
V	2	3	2	2	3	3	2	2	3	2	1	3	2	2	2	2	2	5	3	0

Figura 4.10: Matriz MPAM [64]

A maior distância para essa função é o valor 448, que equivale a comparar uma proteína composta por 64 aminoácidos do tipo *C* com outra proteína de 64 aminoácidos do tipo *W*. Logo, o domínio dos valores possíveis para essa função de distância métrica com esse conjunto de dados serão os números inteiros entre 0 e 448.

A utilização desse conjunto de dados se fez necessária pois, ele possui uma função de distância custosa, onde um cálculo de distância é muito mais caro que um acesso a disco. Assim, espera-se verificar se as modificações que originaram a Partição Local+ garantem um bom desempenho mesmo ocupando mais espaço em disco e conseqüentemente, mais acessos a disco. Esse conjunto de dados não possui dimensão fixa e somente pode ser indexado por um método de acesso métrico.

Foram usadas 9 estruturas para a comparação, sendo: Arquivo Sequencial; *M-Tree* com quebra pelo algoritmo MINMAX; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador randômico (Rand.), com escolha de mapeador pela variância (Var.) e com escolha de mapeador pela contagem (Diff.); e Partição Local+ com escolha de mapeador randômico (Rand.), com escolha de mapeador pela variância (Var.) e com escolha de mapeador pela contagem (Diff.). Para cada uma das 8 estruturas mencionadas anteriormente variou o tamanho do bloco em 1024 *bytes*, 2048 *bytes* e 4096 *bytes* totalizando 27 estruturas.

A Tabela 4.2 mostra algumas características das estruturas geradas para indexar o conjunto de proteínas. Esse conjunto só pode ser armazenado por estruturas métricas, já que não possuem dimensão fixa.

Tabela 4.2: Informações sobre as Estruturas do Conjunto de Proteínas

Estrutura	Tamanho Bloco	Altura	Qtd. Nó Índice	Qtd. Nó Folha	Tam. Arq. (bytes)	% Obj. Colisão
Sequencial	1024	—	—	4.934	5.053.440	—
M-Tree	1024	06	2.314	11.999	14.657.536	—
Slim-Tree SlimDown	1024	06	2.314	8.682	11.260.928	—
Part.Local Rand.	1024	09	2.074	8.256	10.578.944	0,0000%
Part.Local Var.	1024	07	1.929	7.905	10.071.040	0,0000%
Part.Local Diff.	1024	07	1.830	7.983	10.049.536	0,0000%
Part.Local+ Rand.	1024	08	2.385	9.463	12.133.376	0,0000%
Part.Local+ Var.	1024	07	2.096	9.027	11.390.976	0,0000%
Part.Local+ Diff.	1024	07	2.049	9.125	11.443.200	0,0000%
Sequencial	2048	—	—	2.424	4.966.400	—
M-Tree	2048	05	460	6.348	13.944.832	—
Slim-Tree SlimDown	2048	05	460	4.289	9.728.000	—
Part.Local Rand.	2048	06	688	4.095	9.797.632	0,0000%
Part.Local Var.	2048	06	631	3.991	9.467.904	0,0000%
Part.Local Diff.	2048	05	614	3.990	9.431.040	0,0000%
Part.Local+ Rand.	2048	06	747	4.679	11.114.496	0,0000%
Part.Local+ Var.	2048	06	694	4.615	10.874.880	0,0000%
Part.Local+ Diff.	2048	06	665	4.538	10.657.792	0,0000%
Sequencial	4096	—	—	1.202	4.927.488	—
M-Tree	4096	04	84	2.913	12.279.808	—
Slim-Tree SlimDown	4096	04	84	2.164	9.211.904	—
Part.Local Rand.	4096	06	309	2.052	9.674.752	0,0000%
Part.Local Var.	4096	06	301	1.974	9.322.496	0,0000%
Part.Local Diff.	4096	05	296	1.954	9.220.096	0,0000%
Part.Local+ Rand.	4096	06	396	2.343	11.223.040	0,0000%
Part.Local+ Var.	4096	06	371	2.248	10.731.520	0,0000%
Part.Local+ Diff.	4096	05	346	2.248	10.629.120	0,0000%

A Tabela 4.2 possui os seguintes campos: o tamanho do bloco em *bytes*; a altura para as estruturas *M-Tree*, *Slim-Tree* e a altura máxima das estruturas Partição Local e Partição Local+, sendo que essa característica não se aplica para arquivo sequencial; a quantidade de nó índice sendo que, essa característica não se aplica para o arquivo sequencial; a quantidade de nós folhas; o tamanho do arquivo gerado e a porcentagem de objetos em relação ao total de registros que estão em regiões de colisão, sendo que essa característica só se aplica para as estruturas baseadas em coordenadas métricas.

A colisão indica que aconteceu o problema das distâncias mínimas em um *bucket* índice e assim, não é possível estender as coordenadas métricas. Esse problema deve ser evitado pois gera uma lista de *buckets* folhas e, se essa lista for grande prejudica as consultas.

A partir dos valores da Tabela 4.2, que mostra as estruturas geradas a partir dos dados do *UNI-PROT* usando uma função de distância de alinhamento métrico, com as características citadas acima, podemos concluir que:

- as estruturas baseadas em coordenadas métricas possuem mais níveis do que as estruturas baseadas em região de cobertura. No entanto, a quantidade adicional de níveis não foi expressiva. Essa diferença comprova que a escolha dos objetos mapeadores que compõem a função de mapeamento local foi boa em qualquer variação do tamanho do bloco;

- a estrutura Partição Local possui em média 7% menos blocos do tipo folha que a melhor estrutura de região de cobertura *Slim-Tree* e, conseqüentemente usa menos espaço em disco;
- a estrutura Partição Local+ possui em média 6% mais blocos do tipo folha que a melhor estrutura de região de cobertura *Slim-Tree* e, conseqüentemente usa mais espaço em disco;
- as estruturas que usaram o algoritmo de escolha de novo mapeador pela maior variância de regiões e pela maior contagem de regiões diferentes possuem pelo menos 3% menos nó folha comparado com as estruturas que usam o algoritmo randômico; e
- a colisão não aconteceu nesse conjunto de dados, pois a escolha do objeto mapeador foi sempre boa. Isto aconteceu porque a ordem dos registros no conjunto UNIPROT não apresenta um agrupamento de distâncias.

Análise do Número de Cálculos de Distância para Proteínas

A Figura 4.11 apresenta os gráficos da média de números de cálculos de distância dos resultados obtidos utilizando a base de proteínas UNIPROT. A primeira coluna mostra os gráficos obtidos utilizando consulta por abrangência e a segunda coluna apresenta os gráficos utilizando consulta aos vizinhos mais próximos.

Consulta por Abrangência

Comparando os gráficos da Figura 4.11 com os resultados da Tabela 4.2 nota-se que o tamanho do bloco influencia diretamente a altura da estrutura e que, quanto mais alta a estrutura maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.11 mostram o número médio de cálculos de distância realizados para responder a 500 consultas por abrangência em que o raio varia de 1 até 5 unidades de alinhamento, onde os objetos de consulta foram sorteados aleatoriamente do conjunto UNIPROT. Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O número de cálculos de distância é comparado com a quantidade de proteínas que compõem o conjunto de dados, que é de 92.575.

As estruturas que realizaram o menor número de cálculos de distância na consulta por abrangência no pior caso em que o raio vale 5 unidades de alinhamento foram:

- Partição Local+ para blocos de 1024 *bytes*, com escolha de mapeador pela variância, tendo realizado 22.205 cálculos ou 24% de proteínas da base;

Figura 4.11: Gráficos da média de números de cálculos de distância para 500 consultas por abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) da base UNIPROT

- Partição Local+ para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 17.699 cálculos ou 19% de proteínas da base; e
- Partição Local para blocos de 4096 *bytes*, com escolha de mapeador randômico, tendo realizado 19.794 cálculos ou 21% de proteínas da base.

As estruturas que realizaram o maior número de cálculos de distância na consulta por abrangência no pior caso em que o raio vale 5 unidades de alinhamento foram:

- *Slim-Tree* para blocos de 1024 *bytes*, tendo realizado 51.817 cálculos ou 56% de proteínas da base;
- *Slim-Tree* para blocos de 2048 *bytes*, tendo realizado 40.789 cálculos ou 44% de proteínas da base; e
- *Slim-Tree* para blocos de 4096 *bytes*, tendo realizado 36.760 cálculos ou 40% de proteínas da base.

Portanto, o algoritmo de consulta por abrangência com o raio valendo 5 unidades de alinhamento nas estruturas baseadas em regiões de coberturas comparado com a melhor estrutura baseada em coordenadas métricas, foi pior obtendo os seguintes valores: 133% mais cálculos de distância para blocos de 1024 *bytes*, 130% mais cálculos de distância para blocos de 2048 *bytes* e 85% mais cálculos de distância para blocos de 4096 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.11 mostram o número médio de cálculos de distância realizados para responder a 500 consultas aos vizinhos mais próximos com lista de empate em que o k varia de 1 até 5 unidades de alinhamento, onde os objetos de consulta foram sorteados aleatoriamente do conjunto UNIPROT. Novamente, as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O algoritmo de busca aos vizinhos mais próximos nas estruturas baseadas em região de cobertura realizou o número de cálculos de distância muito próximo ao da busca seqüencial, sendo que na média as duas estruturas para raio valendo 5 unidades de alinhamento realizaram:

- 83.339 cálculos ou 90% de proteínas da base para blocos de 1024 *bytes*;
- 77.870 cálculos ou 84% de proteínas da base para blocos de 2048 *bytes*; e
- 73.976 cálculos ou 80% de proteínas da base para blocos de 4096 *bytes*.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 5 unidades de alinhamento nas estruturas baseadas em regiões de coberturas, comparado com a melhor estrutura baseada em coordenadas métricas, foi pior obtendo os seguintes valores: 24% mais cálculos de distância para blocos de 1024 *bytes*, 42% mais cálculos de distância para blocos de 2048 *bytes* e 33% mais cálculos de distância para blocos de 4096 *bytes*.

Análise do Número de Acessos a Disco para Proteínas

A Figura 4.12 apresenta os gráficos da média de números de acessos a disco dos resultados obtidos utilizando a base de proteínas UNIPROT. A primeira coluna mostra os gráficos obtidos para realizar consulta por abrangência e a segunda coluna apresenta os gráficos para realizar consulta aos vizinhos mais próximos.

(a)

(b)

(c)

(d)

(e)

(f)

Figura 4.12: Gráficos da média de números de acessos a disco para 500 consultas por abrangência (1ª coluna) e de vizinhos mais próximos (2ª coluna) para a base UNIPROT

Consulta por Abrangência

Comparando os gráficos da Figura 4.12 com os resultados da Tabela 4.2 nota-se novamente que o tamanho do bloco influencia diretamente na altura da estrutura e que quanto mais alta é a estrutura, maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.12 mostram o número médio de acessos a disco realizados para responder a 500 consultas por abrangência em que o raio varia de 1 até 5 unidades de alinhamento, onde os objetos de consulta foram sorteados aleatoriamente do conjunto UNIPROT. Pôde-se observar que as estruturas baseadas em coordenadas métricas novamente ganham em qualquer situação das estruturas baseadas em região de cobertura.

As estruturas que realizaram o menor número de acessos a disco na consulta por abrangência no pior caso, que é quando o raio vale 5 unidades de alinhamento, foram:

- Partição Local para blocos de 1024 *bytes*, com escolha de mapeador pela diferença tendo realizado 2.733 acessos ou 55% dos blocos de um arquivo seqüencial;
- Partição Local para blocos de 2048 *bytes*, com escolha de mapeador pela variância tendo realizado 1.245 acessos ou 51% dos blocos de um arquivo seqüencial; e
- Partição Local para blocos de 4096 *bytes*, com escolha de mapeador pela variância tendo realizado 536 acessos ou 45% dos blocos de um arquivo seqüencial.

Por outro lado, as estruturas que realizaram o maior número de acessos a disco na consulta por abrangência no pior caso em que o raio estava valendo 5 unidades de alinhamento foram:

- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 11.772 acessos ou 2,38 vezes o número de blocos de um arquivo seqüencial;
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 5.613 acessos ou 2,31 vezes o número de blocos de um arquivo seqüencial; e
- *M-Tree* para blocos de 4096 *bytes*, tendo realizado 2.441 acessos ou 2,03 vezes o número de blocos de um arquivo seqüencial.

Consulta aos Vizinhos mais Próximos

O algoritmo de consulta aos vizinhos mais próximos com o k valendo 5 unidades de alinhamento foi pior nas estruturas baseadas em regiões de coberturas obtendo os seguintes valores: 3,3 vezes mais acessos a disco para blocos de 1024 *bytes*, 3,5 vezes mais acessos a disco para blocos de 2048 *bytes*

e 3,55 vezes mais acessos a disco para blocos de 4096 *bytes*, comparado com as estruturas baseadas em coordenada métrica.

As estruturas que realizaram o maior número de acessos a disco na consulta aos vizinhos mais próximos, em que o k vale 5 unidade de alinhamento, foram:

- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 13.728 acessos ou 2,78 vezes o número de blocos de um arquivo seqüencial;
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 6.538 acessos ou 2,69 vezes o número de blocos de um arquivo seqüencial; e
- *M-Tree* para blocos de 4096 *bytes*, tendo realizado 2.858 acessos ou 2,37 vezes o número de blocos de um arquivo seqüencial.

O algoritmo de consulta aos vizinhos mais próximos, com k valendo 5 unidades de alinhamento nas estruturas baseadas em regiões de coberturas comparado com a melhor estrutura baseada em coordenadas métricas, foi pior obtendo os seguintes valores: 93% mais acessos a disco para blocos de 1024 *bytes*, 101% mais acessos a disco para blocos de 2048 *bytes* e 90% mais acessos a disco para blocos de 4096 *bytes*.

Análise do Número de Tempo de Execução para Proteínas

Os gráficos apresentados na Figura 4.13 indicam os tempos de execuções referentes às operações de consulta por abrangência e consulta aos vizinhos mais próximos no conjunto de proteínas.

Pode-se observar na figura 4.13 que não houveram ganhos constantes de um mesmo método de escolha de mapeador variando o tamanho de bloco e o tipo de consulta. Por isso, recomenda-se o método de escolha de mapeador randômico que é mais rápido que os outros métodos (variância e contagem) para construir as estruturas.

Consulta por Abrangência

Os gráficos da primeira coluna da Figura 4.13 mostram o tempo médio em milisegundos (ms) para responder a 500 consultas por abrangência, em que o raio varia de 1 até 5 unidades de alinhamento, onde os objetos de consulta foram sorteados aleatoriamente do conjunto UNIPROT. Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham, em todas as situações das estruturas baseadas em região de cobertura, como nos casos de cálculos de distância e acessos a disco.

O algoritmo de consulta por abrangência com o raio valendo 5 unidades de alinhamento nas estruturas baseadas em regiões de coberturas, comparado com a melhor estrutura baseada em coordenadas

Figura 4.13: Gráficos da média de de tempo para 500 consultas por abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) para a base UNIPROT

métricas, foi pior, obtendo os seguintes valores: 2,66 vezes mais tempo de execução para blocos de 1024 *bytes*, 2,70 vezes mais tempo de execução para blocos de 2048 *bytes* e 1,66 vezes mais tempo de execução para blocos de 4096 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.13 mostram o tempo médio em milissegundos (ms) para responder a 500 consultas aos vizinhos mais próximos com lista de empate, em que o k varia de 1 até 5 unidades de alinhamento. Os objetos de consulta foram sorteados aleatoriamente do conjunto UNIPROT, e novamente as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 5 unidades de alinhamento nas estruturas baseadas em regiões de coberturas comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 33% mais tempo de execução para blocos de 1024 *bytes*, 42% mais tempo de execução para blocos de 2048 *bytes* e 31% mais tempo de execução para blocos de 4096 *bytes*.

Discussão dos Resultados

Os resultados mostraram que a organização das estruturas de particionamento realizam menor número de cálculos de distância nos blocos índices e folhas, e menor número de acessos a disco.

A justificativa para realizar um menor número de cálculos de distância nos blocos índices é que existe uma quantidade menor de mapeadores comparado com a quantidade de representantes nos níveis de índice das estruturas.

Por exemplo, nos blocos índices de tamanho de 2048 *bytes*, as estruturas baseadas em região de cobertura podem realizar até 40 cálculos de distância por nó índice, pois armazenam até 40 proteínas representantes com tamanho médio de 43 aminoácidos. No entanto, esse número é reduzido pelo uso da distância para seu representante (do nível acima) usando a desigualdade triangular.

No mesmo exemplo, para blocos índices de tamanho de 2048 *bytes*, as estruturas de particionamento sempre realizam seis cálculos de distância por *bucket* índice, pois armazenam no máximo 4 proteínas mapeadoras com tamanho médio de 43 aminoácidos. O número de cálculos é dado através das combinações possíveis entre as distâncias de 4 mapeadores A, B, C e D (AB, AC, AD, BC, BD e CD). Mesmo que as estruturas de particionamento sejam um pouco mais altas, o que deveria aumentar o número de cálculos de distância, a pouca quantidade de cálculos realizados no bloco índice compensa essa diferença de altura.

O uso da coordenada métrica de início e fim da consulta das estruturas baseadas em particionamento mostrou que evita mais cálculos de distância do que o uso da distância para o representante das estruturas de região de cobertura nos objetos dos *buckets* folhas que não fazem parte da resposta.

A justificativa para realizar um menor número de acessos a disco é que as estruturas baseadas em

particionamento também são maiores em largura do que as estruturas baseadas em região de cobertura.

Ainda usando o exemplo acima, para blocos índices de tamanho de 2048 *bytes*, as estruturas baseadas em região de cobertura possuem exatamente 40 sub-árvores, uma para cada representante. As estruturas baseadas em particionamento com 4 proteínas mapeadoras podem ter no máximo 4^4 ou 256 *sub-buckets*, pois um mesmo *sub-bucket* pode ser mapeado por mais de uma coordenada métrica. Mesmo sendo mais altas, em alguns ramos, as estruturas de particionamento são maiores em largura do que as estruturas baseadas em região de cobertura, o que deixa muitos ramos com uma quantidade menor de níveis do que as estruturas baseadas em região de cobertura.

O tempo mais baixo das consultas nas estruturas de particionamento é consequência da menor quantidade de cálculos de distância e acessos a disco.

Os métodos de escolha de mapeadores por variância e por contagem não garantiu ganhos constantes em todos os experimentos. Esses métodos tornam a inserção mais lenta e por isso recomenda-se utilizar o método de escolha randômica pois tornam a inserção mais rápida.

A maioria dos experimentos mostraram que as estruturas Partição Local+ realizam um menor número de cálculos de distância pois usam a distância para o primeiro mapeador, no *bucket* folha, para evitar cálculos de distância. Como essa função de distância é mais custosa do que o acesso a disco, as estruturas de Partição Local+ são mais rápidas do que as de Partição Local, mesmo realizando uma quantidade maior de acessos a disco.

4.4.2 Conjunto de Palavras

O conjunto de dados do *OpenOffice* contém 694.762 palavras filtradas com somente as que não possuíam acentuação no dicionário de ortografia de 61 línguas de todo o mundo. A palavra mais longa tem 50 letras, mas na média elas têm 9 letras.

Para o conjunto de palavras do *OpenOffice* foi usada a função de distância Levenshtein [32]. Essa função de distância métrica atribui peso 1 para a troca ou deslocamento entre qualquer letra. Logo, o domínio dos valores possíveis para essa função de distância métrica está com todos números inteiros entre 0 e 50, já que essa palavra não possui todas as letras do alfabeto.

A utilização desse conjunto de dados se fez necessária pois, ele possui uma função de distância menos custosa em relação ao conjunto de dados UNIPROT. Esse conjunto apresenta 7,5 vezes mais dados do que o conjunto de dados anterior e o domínio dos valores possíveis da função de distância é pequeno. Esse conjunto de dados não possui dimensão fixa e só pode ser indexado por um método de acesso métrico.

A Tabela 4.3 mostra as características das 15 estruturas de indexação construídas com esse conjunto de palavras. As estruturas usadas na comparação foram: *M-Tree* com quebra pelo algoritmo de

Tabela 4.3: Informações sobre as Estruturas do Conjunto de Palavras

Estrutura	Tamanho Bloco	Altura	Qtd. Nó Índice	Qtd. Nó Folha	Tam. Arq. (bytes)	% Obj. Colisão
Seqüencial	512	—	—	25.991	13.307.904	—
M-Tree	512	07	12.613	68.546	41.553.920	—
Slim-Tree SlimDown	512	07	12.613	58.008	36.158.464	—
Part.Local Rand.	512	11	12.918	47.760	31.067.648	0,0138%
Part.Local+ Rand.	512	13	18.251	66.648	43.468.800	0,0676%
Seqüencial	1024	—	—	12.776	13.083.648	—
M-Tree	1024	05	2.610	32.784	36.244.480	—
Slim-Tree SlimDown	1024	11	2.610	29.035	32.405.504	—
Part.Local Rand.	1024	11	6.406	23.553	30.679.040	0,0000%
Part.Local+ Rand.	1024	12	8.764	32.700	42.460.160	0,0000%
Seqüencial	2048	—	—	6.334	12.974.080	—
M-Tree	2048	04	578	15.559	33.050.624	—
Slim-Tree SlimDown	2048	04	578	14.107	30.691.328	—
Part.Local Rand.	2048	07	1.897	11.538	27.516.928	0,0356%
Part.Local+ Rand.	2048	09	2.695	16.066	38.424.576	0,0128%

MINMAX; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador randômico (Rand.), e Partição Local+ com escolha de mapeador randômico (Rand.). Para cada uma das 5 estruturas mencionadas anteriormente variou-se o tamanho do bloco em 512 *bytes*, 1024 *bytes* e 2048 *bytes*.

A Tabela 4.3 possui os seguintes campos: o tamanho do bloco em *bytes*; a altura para as estruturas *M-Tree*, *Slim-Tree* e a altura máxima das estruturas Partição Local e Partição Local+, sendo que essa característica não se aplica para arquivo seqüencial; a quantidade de nó índice, sendo que essa característica não se aplica para arquivo seqüencial; a quantidade de nó folha; o tamanho do arquivo gerado e a porcentagem de objetos em relação ao total de registros que estão em regiões de colisão, sendo que essa característica só se aplica para as estruturas baseadas em coordenadas métricas.

Não foram testadas para esse conjunto de dados, dentre as estruturas de particionamento, os métodos de escolha de mapeadores pela variância e pela contagem pois, não se manteve constante a melhora no desempenho das consultas. Outro motivo é que esses algoritmos tornam mais lenta a inserção de objetos para conjuntos com grande quantidade de dados como esse.

A Tabela 4.3 mostra as medidas obtidas das estruturas geradas do conjunto de palavras do *OpenOffice*, usando a função de distância Levenshtein com as características citadas acima. Pôde-se concluir que:

- as estruturas baseadas em coordenadas métricas chegam a ser mais altas mais de duas vezes quando comparadas com as estruturas baseadas em região de cobertura. Essa diferença mostra que a escolha dos objetos mapeadores que compõem a função de mapeamento local não foi boa para as diversas variações de tamanho de bloco como ocorreu com o conjunto de proteínas;
- a estrutura Partição Local possui em média 18% menos blocos do tipo folha que a melhor

estrutura de região de cobertura *Slim-Tree* e, conseqüentemente usa menos espaço em disco;

- a estrutura Partição Local+ possui em média 13% mais blocos do tipo folha que a melhor estrutura de região de cobertura *Slim-Tree* e, conseqüentemente usa mais espaço em disco; e
- a colisão foi bem pequena nesse conjunto de dados, pois a escolha do objeto mapeador apesar de não ser a ideal, foi razoável. Isto aconteceu porque a ordem dos registros no conjunto *OpenOffice* apresenta uma ordenação nas palavras, o que agrupa as distâncias.

Análise do Número de Cálculos de Distância para Conjunto de Palavras

A Figura 4.14 apresenta os gráficos da média de cálculos de distância obtidos utilizando a base de palavras do *OpenOffice*. A primeira coluna mostra os gráficos obtidos para realizar consultas por abrangência e a segunda coluna apresenta os gráficos para executar consultas aos vizinhos mais próximos.

Consulta por Abrangência

Comparando os gráficos da Figura 4.14 com os resultados da Tabela 4.3 nota-se que o tamanho do bloco em disco influencia diretamente a altura da estrutura e que, quanto mais alta a estrutura maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.14 mostram o número médio de cálculos de distância realizados para responder a 500 consultas por abrangência em que o raio varia de 1 até 5 unidades, onde os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*.

O algoritmo de consulta por abrangência com o raio valendo 5 unidades nas estruturas baseadas em regiões de cobertura comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 5% mais cálculos de distância para blocos de 512 *bytes*, 1% mais cálculos de distância para blocos de 1024 *bytes* e 18% mais cálculos de distância para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.11 mostram o número médio de cálculos de distância realizados para responder a 500 consultas aos vizinhos mais próximos com lista de empate, em que o k varia de 1 até 5 unidades, e os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*. As estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

Figura 4.14: Gráficos da média de números de cálculos de distância para 500 consultas por abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) da base *OpenOffice*

Nos experimentos realizados, o número de cálculos de distância foi comparado com a quantidade de palavras que compõem o conjunto de dados que é de 694.762.

As estruturas que realizaram o menor número de cálculos de distância na consulta aos vizinhos mais próximos no pior caso, em que o raio vale 5 unidades, foram:

- Partição Local+ para blocos de 512 *bytes*, com escolha de mapeador randômico, tendo realizado

276.269 cálculos ou 40% de palavras da base;

- Partição Local+ para blocos de 1024 *bytes*, com escolha de mapeador randômico, tendo realizado 294.898 cálculos ou 42% de palavras da base; e
- Partição Local+ para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 287.713 cálculos ou 41% de palavras da base.

As estruturas que realizaram o maior número de cálculos de distância na consulta aos vizinhos mais próximos no pior caso, em que o raio vale 5 unidades, foram:

- *Slim-Tree* para blocos de 512 *bytes*, tendo realizado 492.988 cálculos ou 71% de palavras da base;
- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 440.534 cálculos ou 63% de palavras da base; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 407.433 cálculos ou 59% de palavras da base.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 5 unidades nas estruturas baseadas em regiões de cobertura, comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 78% mais cálculos de distância para blocos 512 *bytes*, 49% mais cálculos de distância para blocos 1024 *bytes* e 41% mais cálculos de distância para blocos 2048 *bytes*.

Análise do Número de Acessos a Disco para Conjunto de Palavras

A Figura 4.15 apresenta os gráficos da média do número de acessos a disco utilizando a base de palavras do *OpenOffice*. A primeira coluna mostra os gráficos obtidos utilizando consulta por abrangência e a segunda coluna apresenta os gráficos utilizando consulta aos vizinhos mais próximos.

Consulta por Abrangência

Comparando os gráficos da Figura 4.15 com os resultados da Tabela 4.3 nota-se novamente que o tamanho do bloco influencia diretamente a altura da estrutura e que quanto mais alta a estrutura maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.15 mostram o número médio de acessos a disco realizados para responder a 500 consultas por abrangência com o raio variando de 1 até 5 unidades, onde os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*.

Figura 4.15: Gráficos da média de números de acessos a disco para 500 consultas por abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) para a base *OpenOffice*

O algoritmo de consulta por abrangência com o raio valendo 5 unidades na pior estrutura baseada em regiões de cobertura (*M-Tree*) comparado com a melhor estrutura baseada em coordenadas métricas, Partição Local, foi pior, obtendo os seguintes valores: 46% mais acessos a disco para blocos de 512 *bytes*, 33% mais acessos a disco para blocos de 1024 *bytes* e 48% mais acessos a disco para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.12 mostram o número médio de acessos a disco realizados para responder a 500 consultas aos vizinhos mais próximos com lista de empate, em que o k varia de 1 até 5 unidades. Os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*. Novamente as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

As estruturas que realizaram o menor número de acessos a disco na consulta aos vizinhos mais próximos no pior caso, em que o raio vale 5 unidades, foram:

- Partição Local para blocos de 512 *bytes*, com escolha de mapeador randômico tendo realizado 28.310 acessos ou 108% dos blocos de um arquivo seqüencial;
- Partição Local para blocos de 1024 *bytes*, com escolha de mapeador randômico tendo realizado 13.598 acessos ou 106% dos blocos de um arquivo seqüencial; e
- Partição Local para blocos de 2048 *bytes*, com escolha de mapeador randômico tendo realizado 6.486 acessos ou 102% dos blocos de um arquivo seqüencial.

As estruturas que realizaram o maior número de acessos a disco na consulta aos vizinhos mais próximos no pior caso, em que o raio vale 5 unidades, foram:

- *M-Tree* para blocos de 512 *bytes*, tendo realizado 72.652 acessos ou 2,79 vezes o número de blocos de um arquivo seqüencial;
- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 31.132 acessos ou 2,44 vezes o número de blocos de um arquivo seqüencial; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 13.791 acessos ou 2,18 vezes o número de blocos de um arquivo seqüencial.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 5 unidades nas estruturas baseadas em regiões de cobertura comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 1,56 vezes mais acessos a disco para blocos de 512 *bytes*, 1,29 vezes mais acessos a disco para blocos de 1024 *bytes* e 1,13 vezes mais acessos a disco para blocos de 2048 *bytes*.

Análise do Número de Tempo de Execução para o Conjunto de Palavras

Os gráficos da média de tempo de execução referentes às operações de consulta por abrangência e consulta aos vizinhos mais próximos no conjunto de palavras, são apresentados na Figura 4.16.

(a)

(b)

(c)

(d)

(e)

(f)

Figura 4.16: Gráficos da média de tempo para 500 consultas por abrangência (1ª coluna) e de vizinhos mais próximos (2ª coluna) para a base *OpenOffice*

Consulta por Abrangência

Os gráficos da primeira coluna da Figura 4.16 mostram o tempo médio em milisegundos (ms) para responder a 500 consultas por abrangência em que o raio varia de 1 até 5 unidades, onde os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*. Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura, como nos casos de cálculos de distância e acessos a disco.

O algoritmo de consulta por abrangência com o raio valendo 5 unidades nas estruturas baseadas em regiões de cobertura comparando com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 13% mais tempo de execução para blocos de 512 *bytes*, 12% mais tempo de execução para blocos de 1024 *bytes* e 32% mais tempo de execução para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.16 mostram o tempo médio em milisegundos (ms) para responder a 500 consultas aos vizinhos mais próximos com lista de empate, em que o k varia de 1 até 5 unidades, e os objetos de consulta foram sorteados aleatoriamente do conjunto *OpenOffice*. Novamente as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 5 unidades nas estruturas baseadas em regiões de cobertura, comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 93% mais tempo de execução para blocos de 512 *bytes*, 63% mais tempo de execução para blocos de 1024 *bytes* e 46% mais tempo de execução para blocos de 2048 *bytes*.

Discussão dos Resultados

Os resultados mostraram que a organização das estruturas de particionamento realizam menor número de cálculos de distância nos blocos índices e folhas, menor número de acessos a disco e tempo pelos mesmos motivos mencionados na discussão dos resultados do conjunto de dados de proteínas.

A maioria dos experimentos mostrou que as estruturas Partição Local+ realizam um menor número de cálculos de distância pois usam a distância para o primeiro mapeador, no *bucket* folha, para evitar cálculos de distância. Essa função de distância é menos custosa do que a de proteínas, pois o tamanho médio das palavras é menor do que o tamanho médio das proteínas. Dessa forma, o tempo das estruturas de Partição Local+ são próximos ao tempo da Partição Local, mesmo realizando uma

maior quantidade de acessos a disco.

4.4.3 Conjunto de Pontos Geográficos

O conjunto de dados do projeto *GEOnet Names Server* contém 3.976.221 pontos geográficos (latitude e longitude) filtrados pela unicidade do ponto geográfico. Os pontos correspondem ao centro de cidades e pontos geográficos de interesse em todo mundo.

A função de distância para o conjunto de pontos geográficos descrita na fórmula abaixo, retorna a distância entre dois pontos $P_1 = \langle lat_1, long_1 \rangle$ e $P_2 = \langle lat_2, long_2 \rangle$ na Terra em quilômetros, usando a lei dos cossenos na trigonometria esférica, onde o raio da terra é aproximadamente 6.370Km.

$$DistGeo = 6370 \times \arccos(\sin(lat_1) \times \sin(lat_2) + \cos(lat_1) \times \cos(lat_2) \times \cos(long_2 - long_1))$$

As variáveis lat_1 e $long_1$ é a latitude e a longitude do primeiro ponto geográfico representada em graus; e, lat_2 e $long_2$ é a latitude e a longitude respectivamente, do segundo ponto geográfico.

O domínio dos valores possíveis para essa função de distância métrica inclui todos os números fracionários entre 0 e a maior distância entre dois pontos da Terra que é pouco mais de 20.000 km.

A utilização desse conjunto de dados se fez necessária, pois ele possui uma função de distância barata. Esse conjunto apresenta 5,7 vezes mais registros em relação a conjunto de palavras e o domínio dos valores possíveis da função de distância é muito maior que os conjuntos anteriores. Esse conjunto de dados possui dimensão fixa e poderia ser indexado por um método de acesso espacial.

Tabela 4.4: Informações sobre as Estruturas do Conjunto Pontos Geográficos

Estrutura	Tamanho Bloco	Altura	Qtd. Nó Índice	Qtd. Nó Folha	Tam. Arq. (bytes)	% Obj. Colisão
Sequencial	512	—	—	128.266	65.672.704	—
M-Tree	512	8	43.875	292.721	172.337.664	—
Slim-Tree SlimDown	512	8	43.875	285.443	168.611.328	—
Part. Local Rand.	512	16	69.844	248.160	162.818.560	1,3371%
Part. Local+ Rand.	512	17	97.602	346.862	227.566.080	2,2140%
Sequencial	1024	—	—	63.115	64.630.784	—
M-Tree	1.024	6	9.643	144.633	157.979.648	—
Slim-Tree SlimDown	1.024	6	9.643	141.719	154.995.712	—
Part. Local Rand.	1.024	15	34.076	121.986	159.808.512	1,1960%
Part. Local+ Rand.	1.024	15	50.250	178.318	234.054.656	1,0620%
Sequencial	2048	—	—	31.309	64.122.880	—
M-Tree	2.048	5	2.244	70.827	149.651.456	—
Slim-Tree SlimDown	2.048	5	2.244	69.432	146.794.496	—
Part. Local Rand.	2.048	10	10.821	61.410	147.931.136	0,1676%
Part. Local+ Rand.	2.048	11	15.438	88.238	212.330.496	0,6240%

A Tabela 4.4 mostra características da construção das estruturas: Arquivo Sequencial, *M-Tree* com quebra pelo algoritmo MINMAX; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador randômico (Rand.), e Partição Local+ com escolha de mapeador randômico (Rand.). Para cada estrutura mencionada anteriormente, variou o tamanho do bloco em 512 bytes, 1024 bytes e 2048 bytes totalizando 15 estruturas.

A Tabela 4.4 possui os seguintes campos: o tamanho do bloco em *bytes*; a altura para as estruturas *M-Tree*, *Slim-Tree* e a altura máxima das estruturas Partição Local e Partição Local+, sendo que essa característica não se aplica para arquivo seqüencial; a quantidade de nós índice, sendo que essa característica não se aplica para arquivo seqüencial; a quantidade de nós folha; o tamanho do arquivo gerado e a porcentagem de objetos em relação ao total de registros que estão em regiões de colisão, sendo que essa característica só se aplica para as estruturas baseadas em coordenadas métricas.

Não foram testadas para esse conjunto de dados, dentre as estruturas de particionamento, os métodos de escolha de mapeadores pela variância e pela contagem pois, não se manteve constante a melhora no desempenho das consultas. Outro motivo é que esses algoritmos tornam mais lenta a inserção de objetos para conjuntos com grande quantidade de dados.

A partir dos valores da Tabela 4.4 que mostra as estruturas geradas com os dados do *GEOnet* usando uma função de distância com as características citadas acima, podemos concluir que:

- as estruturas baseadas em coordenadas métricas possuem até três vezes mais níveis do que as estruturas baseadas em região de cobertura. Essa diferença mostra que a escolha dos objetos mapeadores que compõem a função de mapeamento local foi ruim para todas as variações de tamanho de bloco, pois uma escolha boa deve manter a estrutura com a quantidade de níveis igual ou menor que as estruturas de região de cobertura;
- a estrutura Partição Local, possui em média 14% menos blocos do tipo folha que a melhor estrutura de região de cobertura *Slim-Tree*. Isso acontece porque essas estruturas armazenam para cada objeto, somente a coordenada métrica local, que é menor do que armazenar uma distância com precisão;
- a estrutura Partição Local+ possui em média 30% mais blocos do tipo folha que a melhor estrutura de região de cobertura *Slim-Tree*. Isso acontece porque essas estruturas armazenam para cada objeto a coordenada métrica local e a distância para o primeiro objeto mapeado. Como cada objeto é composto por apenas dois números de mesma precisão do número usado para mensagem e distância, o aumento proporcional ao consumo de memória é significativo (passa de 8 para 12 *bytes* por objeto); e
- a colisão foi até 2% da base, influenciadas pela escolha ruim dos objetos mapeadores. Isto aconteceu porque a ordem dos registros no conjunto *GEOnet* apresenta agrupamentos pelos estados e países, e portanto, prejudicam a escolha de mapeadores locais.

Análise do Número de Cálculos de Distância para o Conjunto de Pontos Geográficos

Os gráficos apresentados na Figura 4.17 mostram os resultados obtidos referentes aos testes realizados utilizando o conjunto de dados de pontos geográficos. Na primeira coluna estão os gráficos obtidos utilizando consulta por abrangência e a segunda coluna utilizando consulta aos vizinhos mais próximos.

Consulta por Abrangência

Comparando os gráficos da Figura 4.17 com os resultados da Tabela 4.4, nota-se que o tamanho do bloco influencia diretamente na altura da estrutura e que, quanto mais alta a estrutura maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.17 mostram o número médio de cálculos de distância realizados para responder a 500 consultas por abrangência, onde os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet, e que o raio foi: 4, 8, 12, 16, 20, 24, 28, 32, 36, 40 (km). Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O número de cálculos de distância será comparado com a quantidade de pontos geográficos que compõem o conjunto de dados que é 3.976.221.

As estruturas que realizaram o menor número de cálculos de distância na consulta por abrangência no pior caso, em que o raio vale 40 quilômetros, foram:

- Partição Local+ para blocos de 512 *bytes*, com escolha de mapeador randômico, tendo realizado 1031 cálculos ou 0,025% de pontos geográficos da base;
- Partição Local+ para blocos de 1024 *bytes*, com escolha de mapeador randômico, tendo realizado 1065 cálculos ou 0,026% de pontos geográficos da base; e
- Partição Local+ para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 998 cálculos ou 0,025% de pontos geográficos da base.

As estruturas que causam o maior número de cálculos de distância na consulta por abrangência no pior caso, em que o raio vale 40 quilômetros, foram:

- *M-Tree* para blocos de 512 *bytes*, tendo realizado 5.936 cálculos ou 0,149% de pontos geográficos da base;

Figura 4.17: Gráficos da Média de Números de Cálculos de Distância para 500 Consultas por Abrangência (1ª coluna) e de Vizinhos mais Próximos (2ª coluna) para a base GEONet

- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 2.721 cálculos ou 0,068% de pontos geográficos da base; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 2.104 cálculos ou 0,052% de pontos geográficos da base.

Como nos demais conjuntos, o algoritmo de consulta por abrangência com o raio valendo 40 km nas estruturas baseadas em regiões de cobertura comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 5,07 vezes mais cálculos de distância para blocos de 512 *bytes*, 1,65 vezes mais cálculos de distância para blocos de 1024 *bytes* e 1,18 vezes mais cálculos de distância para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.17 mostram o número médio de cálculos de distância realizados para responder a 500 consultas aos vizinhos mais próximos com lista de empate em que o k foi: 3, 6, 9, 12, 15, 18, 21 e 24, e os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet. Novamente, as estruturas baseadas em coordenadas métricas ganham em todas as situações das estruturas baseadas em região de cobertura.

As estruturas que realizaram o menor número de cálculos de distância na consulta aos vizinhos mais próximos, no pior caso em que o k vale 24, foram:

- Partição Local+ para blocos de 512 *bytes*, com escolha de mapeador randômico, tendo realizado 221 cálculos ou 0,005% de pontos geográficos da base;
- Partição Local+ para blocos de 1024 *bytes*, com escolha de mapeador randômico, tendo realizado 204 cálculos ou 0,005% de pontos geográficos da base; e
- Partição Local+ para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 252 cálculos ou 0,005% de pontos geográficos da base.

As estruturas que realizaram o maior número de cálculos de distância na consulta aos vizinhos mais próximos no pior caso, em que o k vale 24, foram:

- *M-Tree* para blocos de 512 *bytes*, tendo realizado 4.373 cálculos ou 0,11% de pontos geográficos da base;
- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 1.575 cálculos ou 0,04% de pontos geográficos da base; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 988 cálculos ou 0,03% de pontos geográficos da base.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 24 nas estruturas baseadas em regiões de cobertura, comparado com a melhor estrutura baseada em coordenadas métricas, foi pior,

obtendo os seguintes valores: 18,79 vezes mais cálculos de distância para blocos de 512 *bytes*, 6,72 vezes mais cálculos de distância para blocos de 1024 *bytes* e 2,92 vezes mais cálculos de distância para blocos de 2048 *bytes*.

Análise do Número de Acessos a Disco para o Conjunto de Pontos Geográficos

Os gráficos apresentados na Figura 4.18 mostram os resultados obtidos dos testes realizados, utilizando o conjunto de dados de pontos geográficos. A primeira coluna apresenta os gráficos obtidos utilizando consultas por abrangência e a segunda coluna mostra os gráficos utilizando consulta aos vizinhos mais próximos.

Consulta por Abrangência

Comparando os gráficos da Figura 4.18 com os resultados da Tabela 4.4 nota-se que o tamanho do bloco influencia diretamente na altura da estrutura e que, quanto mais alta a estrutura maior é o custo para realização da consulta.

Os gráficos da primeira coluna da Figura 4.18 mostram o número médio de acessos a disco realizados para responder a 500 consultas por abrangência, onde os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet, e que o raio foi: 4, 8, 12, 16, 20, 24, 28, 32, 36, 40 (km). Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

As estruturas que realizaram o menor número de acessos a disco na consulta por abrangência no pior caso, em que o raio vale 40 quilômetros, foram:

- Partição Local para blocos de 512 *bytes*, com escolha de mapeador randômico, tendo realizado 112 acessos a disco ou 0,087% dos blocos de um arquivo seqüencial;
- Partição Local para blocos de 1024 *bytes*, com escolha de mapeador randômico, tendo realizado 68 acessos a disco ou 0,107% dos blocos de um arquivo seqüencial; e
- Partição Local para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 35 acessos a disco ou 0,111% dos blocos de um arquivo seqüencial.

As estruturas que realizaram o maior número de acessos a disco na consulta por abrangência no pior caso, em que o raio vale 38 quilômetros, foram:

- *M-Tree* para blocos de 512 *bytes*, tendo realizado 1.507 acessos a disco ou 1,174% dos blocos de um arquivo seqüencial;

Figura 4.18: Gráficos da média de números de acessos a disco para 500 consultas por abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) para a base GEONet

- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 337 acessos a disco ou 0,533% dos blocos de um arquivo seqüencial; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 129 acessos a disco ou 0,412% dos blocos de um arquivo seqüencial.

O algoritmo de consulta por abrangência com o raio valendo 40 km nas estruturas baseadas em regiões de coberturas comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 13,23 vezes mais acessos a disco para blocos de 512 *bytes*, 4,12 vezes mais acessos a disco para blocos de 1024 *bytes* e 2,85 vezes mais acessos a disco para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.18 mostram o número médio de acessos a disco realizados para responder a 500 consultas aos vizinhos mais próximos com lista de empate em que o k foi: 3, 6, 9, 12, 15, 18, 21 e 24, e os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet. Novamente, as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

As estruturas que realizaram o menor número de acessos a disco na consulta aos vizinhos mais próximos no pior caso, em que o k vale 24, foram:

- Partição Local para blocos de 512 *bytes*, com escolha de mapeador randômico, tendo realizado 31 acessos a disco ou 0,02% dos blocos de um arquivo seqüencial;
- Partição Local para blocos de 1024 *bytes*, com escolha de mapeador randômico, tendo realizado 19 acessos a disco ou 0,03% dos blocos de um arquivo seqüencial; e
- Partição Local para blocos de 2048 *bytes*, com escolha de mapeador randômico, tendo realizado 13 acessos a disco ou 0,04% dos blocos de um arquivo seqüencial.

As estruturas que realizaram o maior número de acessos a disco na consulta aos vizinhos mais próximos no pior caso, em que o k vale 24, foram:

- *M-Tree* para blocos de 512 *bytes*, tendo realizado 1.373 acessos a disco ou 1,07% dos blocos de um arquivo seqüencial;
- *M-Tree* para blocos de 1024 *bytes*, tendo realizado 287 acessos a disco ou 0,45% dos blocos de um arquivo seqüencial; e
- *M-Tree* para blocos de 2048 *bytes*, tendo realizado 105 acessos a disco ou 0,34% dos blocos de um arquivo seqüencial.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 24, nas estruturas baseadas em regiões de cobertura comparado com a melhor estrutura baseada em coordenadas métricas, foi

pior, obtendo os seguintes valores: 43,29 vezes mais cálculos de distância para blocos de 512 *bytes*, 14,11 vezes mais acessos a disco para blocos de 1024 *bytes* e 7,08 vezes mais acessos a disco para blocos de 2048 *bytes*.

Análise do Número de Tempo de Execução para o Conjunto de Pontos Geográficos

Os gráficos apresentados na Figura 4.19 mostram os resultados de tempo de execução obtidos referentes aos dados do conjunto de dados de pontos geográficos. A primeira coluna apresenta os gráficos obtidos utilizando consulta por abrangência e a segunda coluna mostra os gráficos obtidos utilizando consulta aos vizinhos mais próximos.

Consulta por Abrangência

Os gráficos da primeira coluna da Figura 4.19 mostram o tempo médio em milisegundos (ms) gasto para responder a 500 consultas por abrangência em que o raio foi: 4, 8, 12, 16, 20, 24, 28, 32, 36, 40 (km), e os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet. Pôde-se observar que as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura, como nos casos de cálculos de distância e acessos a disco.

O algoritmo de consulta por abrangência com o raio valendo 40 km nas estruturas baseadas em regiões de coberturas comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 185% mais tempo de execução para blocos de 512 *bytes*, 36% mais tempo de execução para blocos de 1024 *bytes* e 39% mais tempo de execução para blocos de 2048 *bytes*.

Consulta aos Vizinhos mais Próximos

Os gráficos da segunda coluna da Figura 4.19 mostram o tempo médio em milisegundos (ms) para responder a 500 consultas dos vizinhos mais próximos com lista de empate em que o k foi: 3, 6, 9, 12, 15, 18, 21 e 24, e os objetos de consulta foram sorteados aleatoriamente do conjunto GEONet. Novamente as estruturas baseadas em coordenadas métricas ganham em qualquer situação das estruturas baseadas em região de cobertura.

O algoritmo de consulta aos vizinhos mais próximos com k valendo 24 nas estruturas baseadas em regiões de cobertura comparado com a melhor estrutura baseada em coordenadas métricas, foi pior, obtendo os seguintes valores: 177% mais tempo de execução para blocos de 512 *bytes*, 7% mais

Figura 4.19: Gráficos da média de tempo para 500 consultas de abrangência (1^a coluna) e de vizinhos mais próximos (2^a coluna) para a base GEONet

tempo de execução para blocos de 1024 *bytes* e 2% mais tempo de execução para blocos de 2048 *bytes*.

Discussão dos Resultados

Os resultados mostraram que a organização das estruturas de particionamento realizam menor número de cálculos de distância nos blocos índices e folhas, menor número de acessos a disco e tempo pelos mesmos motivos mencionados na discussão dos resultados do conjunto de dados de proteínas.

Todos os experimentos mostraram que as estruturas Partição Local+ realizam um menor número de cálculos de distância pois usam a distância para o primeiro mapeador, no *bucket* folha, para evitar cálculos de distância. Essa função de distância é menos custosa do que as anteriores, no entanto, o tempo das estruturas de Partição Local+ são próximos ao tempo da Partição Local, mesmo realizando uma maior quantidade de acessos a disco.

4.5 Escalabilidade

Neste experimento foi avaliada a escalabilidade das estruturas: *M-Tree* com quebra pelo algoritmo MINMAX; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador randômico (Rand.), e Partição Local+ com escolha de mapeador randômico (Rand.). Para cada estrutura mencionada variou o tamanho bloco de 512 *bytes*, 1024 *bytes*, 2048 *bytes* e 4096 *bytes* totalizando 16 estruturas.

O conjunto de dados escolhido foi o do projeto GEONet [60] pois, é o maior conjunto de dados contendo 3.976.221 pontos geográficos.

Os gráficos da primeira coluna da figura 4.20 mostram o tempo gasto em segundos (s) para criar as estruturas (gráfico 4.20(a)), o tempo médio em milisegundos (ms) para responder a 500 consultas dos 20 vizinhos mais próximos (gráfico 4.20(c)) e o tempo médio gasto em milisegundos (ms) para responder a 500 consultas por abrangência com raio de 20 km (gráfico 4.20(e)) variando o tamanho de blocos das estruturas.

Tabela 4.5: Informações sobre as Estruturas do conjunto GEONet

Estrutura	Tamanho de Blocos	Altura	Qtd. Nó Índice	Qtd. Nó Folha	% Obj. Colisão
M-Tree	4096	04	550	35.155	—
Slim-Tree SlimDown	4096	04	550	34.478	—
Part. Local Rand.	4096	09	5.387	30.835	0,035%
Part. Local+ Rand.	4096	09	7.717	44.222	0,582%

A Tabela 4.5 mostra características das estruturas com blocos de 4096 *bytes* pois, as informações sobre os blocos de 512, 1024 e 2058 *bytes* estão na tabela 4.4. As características das estruturas são: o tamanho de blocos; a altura para as estruturas *M-Tree*, *Slim-Tree* e a altura máxima das estruturas Partição Local e Partição Local+; a quantidade de nó índice; a quantidade de nó folha; e a porcen-

Figura 4.20: Gráficos de Escalabilidade variando o Tamanho de Blocos (1ª coluna) e variando o Número de Objetos (2ª coluna)

tagem de objetos em relação ao total de registros que estão em regiões de colisão, sendo que essa característica só se aplica para as estruturas baseadas em coordenadas métricas.

A partir dos gráficos da primeira coluna da Figura 4.20, pode-se concluir que:

- quanto maior o tamanho do bloco, mais tempo é gasto para construir as estruturas de região de cobertura, sendo que a estrutura *M-Tree* com blocos de 4096 *bytes* gastou 42 horas, 14 minutos

e 54 segundos e a estrutura de Partição Local Rand. gastou 55 minutos e 30 segundos em sua construção;

- quanto maior a altura, mais tempo é gasto para responder uma consulta aos vizinhos mais próximos nas estruturas de região de cobertura sendo que, a estrutura *M-Tree* com blocos de 4096 bytes gastou 7,08 milissegundos para responder a consulta dos 20 objetos mais próximos e a estrutura de Partição Local Rand. gastou 0,72 milissegundos para responder a consultas dos 20 objetos mais próximos; e
- quanto maior a altura, mais tempo é gasto para responder uma consulta por abrangência nas estruturas de região de cobertura sendo que, a estrutura *M-Tree* com blocos de 4096 bytes gastou 10,86 milissegundos para responder a consulta com raio de 20 km e a estrutura de Partição Local Rand. gastou 4,28 milissegundos para responder a consulta com raio de 20 km.

Os gráficos da segunda coluna da figura 4.20 mostram o tempo gasto em segundos (s) para criar as estruturas com bloco de 512 bytes (gráfico 4.20(b)), o tempo médio em milissegundos (ms) para responder a 500 consultas dos 20 vizinhos mais próximos (gráfico 4.20(d)) e o tempo médio gasto em milissegundos (ms) para responder a 500 consultas por abrangência com o raio de 20 km (gráfico 4.20(f)) variando o número de objetos indexados pelas estruturas.

A Tabela 4.6 mostra características da construção das estruturas com o bloco de tamanho de 512 bytes: *M-Tree* com quebra pelo algoritmo MINMAX; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador randômico (Rand.), e Partição Local+ com escolha de mapeador randômico (Rand.). Para cada estrutura mencionada anteriormente, foram criadas dez estruturas variando o número de objetos, com os seguintes valores: 397.622, 795.244, 1.192.866, 1.590.488, 1.988.110, 2.385.732, 2.783.354, 3.180.976, 3.578.598 e 3.976.221.

Optou-se em utilizar blocos de 512 bytes pois como pode ser visto no gráfico 4.20(a), nota-se que a diferença no tempo de construção das estruturas não é muito grande comparado com blocos de 1024 bytes, 2048 bytes e 4096 bytes.

A partir dos gráficos da segunda coluna da Figura 4.20 e da Tabela 4.6, pode-se concluir que:

- na média de 500 consultas dos 20 vizinhos mais próximos às estruturas de região de cobertura, quanto maior o número de objetos na base, maior é o tempo para responder a consultas, sendo que no pior caso em que o número de objetos é 3.276.221 a *M-Tree* gasta 16,82 vezes mais tempo que a estrutura de Partição Local; e

Tabela 4.6: Informações sobre as Estruturas do conjunto GEONet

Estrutura	Número de Objetos	Altura	Qtd. Nó Índice	Qtd. Nó Folha	% Obj. Colisão
M-Tree	397.622	6	4.384	29.219	—
Slim-Tree SlimDown	397.622	6	4.384	28.543	—
Part. Local Rand.	397.622	11	7.172	25.043	0,1478%
Part. Local+ Rand.	397.622	12	9.983	34.924	0,0759%
M-Tree	795.244	7	8.668	58.289	—
Slim-Tree SlimDown	795.244	7	8.668	56.991	—
Part. Local Rand.	795.244	11	14.285	50.002	0,2507%
Part. Local+ Rand.	795.244	13	19.861	69.745	0,5252%
M-Tree	1.192.866	7	12.995	87.239	—
Slim-Tree SlimDown	1.192.866	7	12.995	85.281	—
Part. Local Rand.	1.192.866	12	21.238	74.764	0,3603%
Part. Local+ Rand.	1.192.866	13	30.013	104.913	0,4484%
M-Tree	1.590.488	7	17.336	116.353	—
Slim-Tree SlimDown	1.590.488	7	17.336	113.759	—
Part. Local Rand.	1.590.488	12	28.325	99.698	0,4475%
Part. Local+ Rand.	1.590.488	14	39.768	139.713	0,8844%
M-Tree	1.988.110	7	21.718	145.469	—
Slim-Tree SlimDown	1.988.110	7	21.718	142.249	—
Part. Local Rand.	1.988.110	12	35.289	124.483	0,5444%
Part. Local+ Rand.	1.988.110	13	49.567	174.047	0,6036%
M-Tree	2.385.732	7	26.036	174.569	—
Slim-Tree SlimDown	2.385.732	7	26.036	170.752	—
Part. Local Rand.	2.385.732	12	42.311	149.333	0,6120%
Part. Local+ Rand.	2.385.732	15	59.626	209.157	0,6848%
M-Tree	2.783.354	7	30.364	203.579	—
Slim-Tree SlimDown	2.783.354	7	30.364	199.170	—
Part. Local Rand.	2.783.354	13	49.280	174.048	0,7012%
Part. Local+ Rand.	2.783.354	13	69.134	243.820	0,9135%
M-Tree	3.180.976	7	34.586	232.621	—
Slim-Tree SlimDown	3.180.976	7	34.586	227.550	—
Part. Local Rand.	3.180.976	13	56.232	198.695	0,7807%
Part. Local+ Rand.	3.180.976	14	78.867	278.326	0,9562%
M-Tree	3.578.598	7	39.012	261.759	—
Slim-Tree SlimDown	3.578.598	7	39.012	256.016	—
Part. Local Rand.	3.578.598	13	63.116	223.370	0,8837%
Part. Local+ Rand.	3.578.598	13	88.720	313.052	1,3302%
M-Tree	3.976.221	7	43.283	290.906	—
Slim-Tree SlimDown	3.976.221	7	43.283	284.521	—
Part. Local Rand.	3.976.221	13	70.012	248.134	0,9621%
Part. Local+ Rand.	3.976.221	14	98.204	347.596	1,3234%

- na média de 500 consultas por abrangência com o raio de 20 km, as estruturas de região de cobertura quanto maior o número de objetos na base, maior é o tempo para responder a consultas, sendo que no pior caso em que o número de objetos é 3.276.221 a *M-Tree* gasta 2,84 vezes mais tempo que a estrutura de Partição Local.

4.6 Conclusões

As estruturas de particionamento sempre ganham em tempo para responder consultas por abrangência e consultas aos vizinhos mais próximos das estruturas com sobreposição, sendo que esse ganho é maior à medida que aumenta a altura das estruturas de região de cobertura. A altura nas estruturas de região de cobertura é influenciada pelo tamanho do bloco, mas blocos muito grandes aumentam o tempo de construção das estruturas.

O tempo gasto na construção das estruturas de particionamento não é influenciado pela variação do tamanho do bloco. Entretanto, para a construção das estruturas de região de cobertura quanto maior é o tamanho do bloco, mais tempo é gasto.

A estrutura de partição+ perde em algumas situações nos acessos a disco para responder as consultas comparado com as estruturas de região de cobertura pois, possuem mais blocos folhas e conseqüentemente mais blocos índices. Essa diferença acontece porque os blocos folhas das estruturas de particionamento possuem um campo adicional que é responsável por armazenar as coordenadas métricas. No entanto, ela realiza menos cálculos de distâncias e assim possui um tempo inferior que as estruturas de região de cobertura.

As funções de distância (tal como a euclidiana usada no conjunto de dados GEONet) que têm custo pequeno e apresentam grande domínio de valores possíveis mostraram nos testes realizados que as estruturas de particionamento são mais rápidas, comparadas com as estruturas de região de cobertura em blocos de 512 *bytes*, 185% para responder as consultas por abrangência com o raio de 40 km e 177% para responder os 24 objetos mais próximos.

As funções de distância (conjunto de dados UNIPROT) que têm custo alto e apresentam domínio de valores possíveis pouco maior em relação ao conjunto de dados *OpenOffice*, mostraram nos testes realizados que as estruturas de particionamento são mais rápidas. Comparado com as estruturas de região de cobertura em blocos de 1024 *bytes*, 266% para responder as consultas por abrangência com o raio de 5 unidades de alinhamento e 33% para responder os 5 objetos mais próximos em alinhamento.

As funções de distância (conjunto de dados *OpenOffice*) que têm custo médio e apresentam pequeno domínio de valores possíveis mostraram nos testes realizados que as estruturas de particionamento são mais rápidas, comparado com as estruturas de região de cobertura em blocos de 512 *bytes*, 13% para responder as consultas por abrangência com o raio de 5 unidades e 93% para responder os 5 objetos mais próximos.

O próximo capítulo apresenta outra contribuição deste trabalho que é, a implementação da junção por abrangência, aos vizinhos mais próximos e por proximidade nas estruturas discutidas neste capítulo.

CAPÍTULO

5

Junção por Similaridade

As operações de junção por similaridade combinam objetos de dois subconjuntos que pertencem a um mesmo domínio D , entre os quais uma condição de similaridade deve ser satisfeita. As condições de similaridade podem ser as mesmas da seleção por similaridade: por abrangência e de vizinhos mais próximos. Além dessas, a literatura cita um outro operador de junção que se chama junção por proximidade (do inglês *closest neighbors join*) [54].

Neste capítulo serão apresentados as implementações destas condições de junção por similaridade em: arquivo seqüencial; estruturas métricas de região de cobertura; e estruturas métricas de particionamento. As implementações dessas condições de junção nas estruturas métricas (*Slim-Tree*, Partição Local e Partição Local+) são contribuições inovadoras deste trabalho.

5.1 Junção por Similaridade Seqüencial

A junção por similaridade em arquivo seqüencial utiliza duas listas de blocos em que os dados são armazenados seqüencialmente. Essa característica garante que as páginas tenham sempre a máxima quantidade de objetos, minimizando assim a quantidade de blocos usados.

Os algoritmos de junção para esse tipo de estrutura são caros, já que examinam todos os pares de objetos nas duas estruturas, direita e esquerda. Apesar disso, neste trabalho servirão de base para a comparação com os algoritmos otimizados. O princípio desses algoritmos é que, para cada bloco do arquivo seqüencial da direita deve-se navegar em todos os blocos do arquivo seqüencial da esquerda.

As implementações de qualquer condição de junção descritas nos Algoritmos 5.1, 5.2, 5.3, garantem que a quantidade de acessos a disco corresponde à multiplicação da quantidade de blocos de cada arquivo seqüencial, mais a quantidade de blocos da estrutura da esquerda. Além disso, a quantidade de cálculos de distância corresponde à multiplicação da quantidade de objetos de cada arquivo seqüencial mais a quantidade de objetos da estrutura da esquerda. Para tal, o consumo de memória varia de acordo com o algoritmo.

5.1.1 Junção por Abrangência

A junção por abrangência responde a seguinte consulta: “Selecione as proteínas contidas no vírus da Hepatite B que diferem em até duas unidades de alinhamento das que estão contidas no vírus da Hepatite C”. Essa operação é comutativa, ou seja, a ordem dos conjuntos entre o operador não altera a resposta considerando os pares encontrados desde que o raio permaneça o mesmo. No entanto, a inversão da ordem das estruturas envolvidas pode refletir em um custo computacional diferente.

O algoritmo em arquivos sequenciais da junção por abrangência é descrito no Algoritmo 5.1. Ele recebe dois arquivos sequenciais e a distância máxima entre os objetos. Esse algoritmo necessita apenas que as páginas sejam alocadas por vez na memória, consumindo menos memória que os demais, pois um par de objetos selecionado certamente faz parte da resposta, e não há necessidade de memória adicional. Como essa operação é comutativa, uma forma de otimizar esse algoritmo é usar no laço externo o arquivo sequencial com o menor número de objetos.

Algoritmo 5.1 *JuncaoAbrangSeq(esquerda, direita, raio)*

```

1: para cada bloco blocoEsq do arquivo esquerda faça
2:   para cada bloco blocoDir do arquivo direita faça
3:     para cada objeto objEsq[i] do blocoEsq faça
4:       para cada objeto objDir[j] do blocoDir faça
5:         se distancia(objEsq[i], objDir[j]) ≤ raio então
6:           adiciona par objEsq[i] e objDir[j] na lista de resultado
7:         fim-se
8:       fim-para
9:     fim-para
10:   fim-para
11: fim-para

```

5.1.2 Junção por Vizinhos mais Próximos

A junção por vizinhos mais próximos responde a seguinte consulta: “Selecione as três proteínas mais parecidas pelo alinhamento entre as proteínas do vírus da Hepatite C para cada proteína do vírus da Hepatite B”. Essa operação não é comutativa, ou seja, a ordem dos conjuntos entre o operador pode alterar a resposta.

O algoritmo em arquivos sequenciais da junção por vizinhos mais próximos, é descrito no Algoritmo 5.2. Esse algoritmo recebe dois arquivos sequenciais, uma quantidade de vizinhos qualquer e deseja-se manter os empates da última resposta.

O consumo de memória desse algoritmo é dado pela alocação de dois blocos (um de cada arquivo sequencial) e n listas com resultados locais, onde n é a quantidade de objetos do bloco da esquerda. Cada lista contém k quantidade de vizinhos mais próximos e são mantidas ordenadas aumentando assim o custo da execução.

Algoritmo 5.2 *JuncaoVizSeq(esquerda, direita, k, empate)*

```

1: para cada bloco blocoEsq do arquivo esquerda faça
2:   cria um vetor de listas de resultados resLocal com o numero de elementos do blocoEsq
3:   para cada bloco blocoDir do arquivo direita faça
4:     para cada objeto objEsq[i] do blocoEsq faça
5:       para cada objeto objDir[j] do blocoDir faça
6:         se numero de elementos em resLocal[i] < k então
7:           adiciona par objEsq[i] e objDir[j] na lista de resLocal[i]
8:         senão
9:           se  $\text{distancia}(\text{objEsq}[i], \text{objDir}[j]) \leq \text{maior distância da lista de } \text{resLocal}[i]$  então
10:            adiciona par objEsq[i] e objDir[j] na lista de resLocal[i]
11:            remove da lista resLocal[i] os elementos excedentes ao k verificando empate
12:          fim-se
13:        fim-se
14:      fim-para
15:    fim-para
16:  fim-para
17:  copia o vetor resLocal para a lista resultados
18: fim-para

```

As listas de resultados locais são necessárias para garantir que: a quantidade de acessos a disco corresponde à multiplicação da quantidade de blocos de cada arquivo seqüencial, mais a quantidade de blocos da estrutura da esquerda; e a quantidade de cálculos de distância, corresponde à multiplicação da quantidade de objetos de cada arquivo seqüencial mais a quantidade de objetos da estrutura da esquerda. Isto acontece porque a inclusão de um par de objetos não é garantia de que tal par faça realmente parte da resposta.

5.1.3 Junção por Proximidade

A junção por proximidade responde a seguinte consulta: “Selecione os cinco pares de proteínas mais parecidas pelo alinhamento entre as contidas no vírus da Hepatite C e das contidas no vírus da Hepatite B”. Essa operação também é comutativa, ou seja, a ordem dos conjuntos entre os operadores não altera a resposta, desde que se considere que (r,s) e (s,r) correspondem ao mesmo par.

A junção por proximidade aplica o princípio da busca aos vizinhos mais próximos no produto cartesiano. Primeiro, todos os pares são formados, e então, os melhores *k* são selecionados.

O algoritmo de junção por proximidade, descrito no Algoritmo 5.3, recebe dois arquivos seqüenciais, uma quantidade de pares qualquer e deseja-se manter os empates da última resposta. Esse algoritmo consome duas páginas em memória e uma lista com a quantidade de pares de objetos mais próximos. O problema é que essa lista é mantida ordenada, o que aumenta o custo da execução.

Algoritmo 5.3 *JuncaoProxSeq*(*esquerda*, *direita*, *k*, *empate*)

```

1: para cada bloco blocoEsq do arquivo esquerda faça
2:   para cada bloco blocoDir do arquivo direita faça
3:     para cada objeto objEsq[i] do blocoEsq faça
4:       para cada objeto objDir[j] do blocoDir faça
5:         se numero de elementos em resultado < k então
6:           adiciona par objEsq[i] e objDir[j] na lista de resultado
7:         senão
8:           se distancia(objEsq[i], objDir[j]) ≤ maior distância da lista de resultado então
9:             adiciona par objEsq[i] e objDir[j] na lista de resultado
10:            remove da lista resultado os elementos excedentes ao k verificando empate
11:          fim-se
12:        fim-se
13:      fim-para
14:    fim-para
15:  fim-para
16: fim-para

```

5.2 Junção por Similaridade em Região de Cobertura

Este tipo de implementação de junção é proposto neste trabalho e pode ser utilizado nas estruturas *Slim-Tree* e *M-Tree*, sem nenhuma alteração.

A seguir serão apresentados os algoritmos para as 3 condições de junção: por abrangência, por vizinhos mais próximos e por proximidade. Todas as implementações consideram que ambas as estruturas usadas na junção podem conter quantidade de níveis diferentes e por isso, os algoritmos são um pouco extensos.

5.2.1 Junção por Abrangência

Na implementação da junção por abrangência navega-se simultaneamente em cada nível hierárquico de duas regiões de cobertura de ambas as estruturas envolvidas na junção. Durante a navegação, o algoritmo de junção pode restringir os nós que serão navegados por ambos os lados (estrutura esquerda e direita). O algoritmo deve navegar em somente um dos lados quando chegar a um nó folha de uma das estruturas envolvidas. Isso acontece quando as estruturas envolvidas na junção não possuem a mesma quantidade de níveis.

A Figura 5.1(a) mostra como dois nós índices, cada um pertencente a uma *Slim-Tree*, cobrem o espaço métrico.

Todos os objetos representantes, com suas respectivas regiões de cobertura, da estrutura que está sendo usada do lado esquerdo da junção, são representados pela letra *a* e os da estrutura que está sendo usada do lado direito da junção são representados pela letra *b*.

As regiões de cobertura que têm os objetos representantes a_3 , a_4 e a_6 da estrutura do lado esquerdo, sobrepõem as regiões de cobertura que tem os objetos representantes b_1 , b_5 e b_7 da estrutura do lado

Figura 5.1: Regiões que se sobrepõem entre duas *Slim-Tree*: (a) Regiões sem raio de consulta e (b) Regiões com raio de consulta

direito da junção.

No entanto, quando é realizada uma operação de junção por abrangência, existe um raio que é passado para a consulta e todas essas regiões têm seu raio de cobertura incrementados pelo valor desse raio, como pode ser visto na Figura 5.1(b). Além das regiões sobrepostas descritas anteriormente, o raio da consulta sobrepõe a região de cobertura que tem o objeto representante b_6 da estrutura do lado direito da junção.

Portanto, serão visitadas as sub-nós da estrutura do lado esquerdo da junção que têm os representantes a_3 , a_4 e a_6 e os sub-nós da estrutura do lado direito da junção, que têm os representantes b_1 , b_5 , b_6 e b_7 .

A figura 5.2 mostra como é feita a poda na estrutura do lado direito usando a desigualdade triangular sem que seja necessário realizar cálculos de distâncias adicionais. Para podar o nó, a distância entre os representantes de cada nó da esquerda e da direita calculados previamente menos a cobertura do nó da esquerda menos o raio deve ser menor que a distância entre o representante da direita e o objeto i (que já está armazenado no nó) mais o raio de cobertura do objeto i .

Pode-se observar na figura 5.2 que a desigualdade triangular consegue podar sem realizar cálculos de distância os objetos *Dir1*, *Dir2* e *Dir4*. Os objetos *Dir5* e *Dir6* serão podados realizando cálculos de distância. Assim, somente a cobertura do objeto *Dir3* do lado direito da estrutura deve ser verificado se existem objetos para a resposta.

Usando o mesmo princípio, é realizada a poda do lado esquerdo da estrutura usando a cobertura do lado direito e o raio para cada distância entre o objeto do lado esquerdo e o representante da esquerda

Figura 5.2: Poda através da Desigualdade Triangular

e sua cobertura.

O Algoritmo 5.4 mostra, como é feita a implementação da junção por abrangência nas estruturas de região de cobertura com a estrutura de *buffer*.

Para minimizar o número de acessos a disco no Algoritmo 5.4 pode-se habilitar um *buffer* para a estrutura do lado direito da junção que mantém em memória primária todos sub-nós de um nó visitado. A quantidade de blocos que deve ser mantida em memória primária é dada pela altura da estrutura vezes a quantidade de sub-nós por nó mais 1 que é o nó raiz.

Para exemplificar, são utilizadas duas estruturas *Slim-Tree* com tamanho de bloco de 512 bytes, com três níveis sendo que cada nível referencia quatro sub-nós. O consumo de memória para manter o *buffer* é 8.192 bytes, ou seja, três blocos da altura da estrutura da direita mais treze blocos da estrutura de *buffer* da esquerda.

O Algoritmo 5.4 pode ser dividido em quatro etapas:

- se ambos os nós da esquerda e da direita são índices, deve-se verificar quais coberturas de objetos da esquerda e da direita com o raio de consulta se sobrepõem com a cobertura do representante da direita e da esquerda respectivamente. Utiliza-se a desigualdade triangular para podar os nós da esquerda e da direita;
- se o nó da esquerda é índice e o nó da direita é folha, deve-se verificar quais coberturas de objetos da esquerda com o raio de consulta se sobrepõem com a cobertura do representante da direita. Utiliza-se a desigualdade triangular para podar os nós da esquerda;
- se o nó da esquerda é folha e o nó da direita é índice, deve-se verificar quais coberturas de objetos da direita com o raio de consulta se sobrepõem com a cobertura do representante da esquerda. Utiliza-se a desigualdade triangular para podar os nós da direita; e

Algoritmo 5.4 *JuncaoAbrangCob*(noEsq, cobEsq, repEsq, noDir, cobDir, repDir, raio, distRep, ativaBuf)

```

1: se noEsq e noDir são INDICE então
2:   para cada objeto objEsq[i] do noEsq faça
3:     se (ativaBuf) então
4:       vetSubNoDir ← todos os sub-nos do noDir
5:     fim-se
6:     se distRep ≤ dist. repres. objEsq[i] + coberturaobjEsq[i] + cobEsq + raio então
7:       para cada objeto objDir[j] do noDir faça
8:         se distRep ≤ dist. repres. objDir[i] + coberturaobjDir[i] + cobDir + raio então
9:           dist ← distancia(objEsq[i], objDir[j])
10:          se dist ≤ coberturaobjEsq[i] + coberturaobjDir[j] + raio então
11:            noEsq' ← sub-no do objEsq[i]
12:            cobEsq ← cobertura objEsq[i]
13:            se ativaBuf então
14:              cobDir ← cobertura objDir[i]
15:              JuncaoAbrangCob(noEsq', cobEsq, objEsq[i], vetSubNoDir[j], cobDir, objDir[j], raio, dist, ativaBuf)
16:            senão
17:              noDir' ← sub-no do objDir[j]
18:              cobDir ← cobertura objDir[i]
19:              JuncaoAbrangCob(noEsq', cobEsq, objEsq[i], noDir', cobDir, objDir[i], raio, dist, ativaBuf)
20:          fim-se
21:        fim-se
22:      fim-se
23:    fim-para
24:  fim-se
25: fim-para
26: senão
27:   se noEsq é INDICE então
28:     para cada objeto objEsq[i] do noEsq faça
29:       se distRep ≤ dist. repres. objEsq[i] + coberturaobjEsq[i] + cobEsq + raio então
30:         dist ← distancia(objEsq[i], repDir)
31:         se dist ≤ cobertura objEsq[i] + raio então
32:           noEsq' ← sub-no do objEsq[i]
33:           cobEsq ← cobertura objEsq[i]
34:           JuncaoAbrangCob(noEsq', cobEsq, objEsq[i], noDir, cobDir, repDir, raio, dist, ativaBuf)
35:         fim-se
36:       fim-se
37:     fim-para
38:   senão
39:     se noDir é INDICE então
40:       se (ativaBuf) então
41:         vetSubNoDir ← todos os sub-nos do noDir
42:       fim-se
43:       para cada objeto objDir[j] do noDir faça
44:         se distRep ≤ dist. repres. objDir[i] + coberturaobjDir[i] + cobDir + raio então
45:           dist ← distancia(repEsq, objDir[j])
46:           se dist ≤ cobertura objDir[j] + raio então
47:             se ativaBuf então
48:               cobDir ← cobertura objDir[i]
49:               JuncaoAbrangCob(noEsq, cobEsq, repEsq, vetSubNoDir[j], cobDir, objDir[j], raio, dist, ativaBuf)
50:             senão
51:               noDir' ← sub-no do objDir[j]
52:               cobDir ← cobertura objDir[i]
53:               JuncaoAbrangCob(noEsq, cobEsq, repEsq, noDir', cobDir, objDir[j], raio, dist, ativaBuf)
54:             fim-se
55:           fim-se
56:         fim-se
57:       fim-para
58:     senão
59:       para cada objeto objEsq[i] do noEsq faça
60:         se distRep ≤ dist. repres. objEsq[i] + cobEsq + raio então
61:           para cada objeto objDir[j] do noDir faça
62:             se distRep ≤ dist. repres. objDir[i] + cobDir + raio então
63:               dist ← distancia(objEsq[i], objDir[j])
64:               se dist ≤ raio então
65:                 adiciona par objEsq[i] e objDir[j] na lista de resultado
66:               fim-se
67:             fim-se
68:           fim-para
69:         fim-se
70:       fim-para
71:     fim-se
72:   fim-se
73: fim-se

```

- se ambos os nós da esquerda e da direita são folhas, deve-se verificar quais distâncias entre os objetos são menores que o raio. Utiliza-se a desigualdade triangular para evitar cálculos de

distância entre objetos da esquerda e da direita.

5.2.2 Junção por Vizinhos mais Próximos

Na implementação da junção por vizinhos mais próximos navega-se em uma estrutura de cada vez, primeiro na estrutura do laço externo até chegar em um nó folha, e depois na estrutura do laço interno.

A junção por vizinhos mais próximos, mostrada no Algoritmo 5.5, deve descer somente do laço externo até chegar ao nó folha. A partir desse momento, inicia-se a descida do lado direito da estrutura realizando uma consulta por vizinhos mais próximos para cada nó da estrutura do laço externo. Quando o algoritmo completar a lista de respostas com k elementos, deve atualizar o raio máximo com o valor da maior distância de pares de objetos.

Algoritmo 5.5 *JuncaoVizCob*(noEsq, cobEsq, repEsq, noDir, cobDir, k, raioMax, distRep, ativaBuf, resLocal)

```

1: se noEsq é INDICE então
2:   para cada objeto objEsq[i] do noEsq faça
3:     noEsq' ← sub-no do objEsq[i]
4:     cobEsq ← cobertura objEsq[i]
5:     JuncaoVizCob(noEsq', cobEsq, objEsq[i], noDir, cobDir, k, raioMax, dist, ativaBuf, NULO)
6:   fim-para
7: senão
8:   se noDir é INDICE então
9:     se (ativaBuf) então
10:      vetSubNoDir ← todos os sub-nos do noDir
11:    fim-se
12:    cria um vetor de listas de resultados resLocal[] com o numero de elementos do blocoEsq
13:    para cada objeto objEsq[i] do noEsq faça
14:      para cada objeto objDir[j] do noDir faça
15:        se dist ≤ dist. repres. objDir[i] + coberturaobjDir[i] + cobDir + raioMax então
16:          dist ← distancia(repEsq, objDir[j])
17:          se dist ≤ cobertura objDir[j] + raioMax então
18:            se ativaBuf então
19:              cobDir ← cobertura objDir[i]
20:              JuncaoVizCob(noEsq, cobEsq, repEsq, vetSubNoDir[j], cobDir, k, raioMax, dist, ativaBuf, resLocal[i])
21:              // Consulta por abrangência que elimina os excedentes ao k e minimiza raioMax
22:              ConsultaAbranCob'(vetSubNoDir[j], cobDir, objEsq[i], k, raioMax, dist, resLocal[i])
23:            senão
24:              noDir' ← sub-no do objDir[j]
25:              cobDir ← cobertura objDir[i]
26:              JuncaoVizCob(noEsq, cobEsq, repEsq, noDir', cobDir, k, raioMax, dist, ativaBuf, resLocal[i])
27:              // Consulta por abrangência que elimina os excedentes ao k e minimiza raioMax
28:              ConsultaAbranCob'(noDir', cobDir, objEsq[i], k, raioMax, dist, resLocal[i])
29:          fim-se
30:        fim-se
31:      fim-se
32:    fim-para
33:  fim-para
34:  copia o vetor resLocal para a lista resultados
35: senão
36:  // noDir e noEsq são FOLHA
37:  para cada objeto objDir[j] do noDir faça
38:    se distRep ≤ dist. repres. objDir[i] + cobDir + raioMax então
39:      dist ← distancia(repEsq, objDir[j])
40:      se dist ≤ raioMax então
41:        adiciona par objEsq[i] e objDir[j] na lista de resLocal
42:        se numero de elementos em resultado ≤ k então
43:          remove em resultado os elementos excedentes ao k
44:          raioMax ← maior distância dos elementos de resLocal
45:      fim-se
46:    fim-se
47:  fim-se
48:  fim-para
49: fim-se
50: fim-se

```

Para minimizar o número de acessos a disco no Algoritmo 5.5 pode-se habilitar um *buffer* para a estrutura do lado direito da junção que mantém em memória primária todos sub-nós de um nó

visitado, como mencionado na seção anterior.

O consumo de memória sem o uso do *buffer* desse algoritmo é dado pela alocação de i blocos da estrutura da esquerda e j blocos da estrutura da direita, onde i e j são as alturas de cada estrutura respectivamente. Além disso, deve-se manter n listas com resultados locais, onde n é a quantidade de objetos do bloco folha da estrutura da esquerda. Cada lista contém k quantidade de vizinhos mais próximos e estas listas são mantidas ordenadas aumentando assim o custo da execução.

As listas de resultados locais são necessárias para minimizar a quantidade de acessos a disco e de cálculos de distância. Isto acontece porque a inclusão de um par de objetos não é garantia de que tal par faça realmente parte da resposta.

5.2.3 Junção por Proximidade

Na implementação da junção por proximidade mostrada no Algoritmo 5.6, deve-se navegar simultaneamente em cada nível hierárquico de duas regiões de cobertura de ambas as estruturas envolvidas na junção.

A junção por proximidade é dividida em dois algoritmos: o primeiro apresentado no Algoritmo 5.6 e o segundo é mostrado no Algoritmo 5.4 que deve ser modificado para remover os objetos excedentes ao k , minimizando o raio máximo com o valor da maior distância de pares de objetos.

Para minimizar o número de acessos a disco, pode-se habilitar um *buffer*. O consumo de memória sem o uso do *buffer* desse algoritmo é dado pela alocação de i blocos da estrutura da esquerda e j blocos da estrutura da direita, onde i e j são as alturas de cada estrutura respectivamente.

5.3 Junção por Similaridade por Particionamento

A implementação das consultas de junção nos métodos de acessos por Partição Local e Partição Local+ também está sendo proposto neste trabalho. Essas estruturas usam a distância entre os objetos mapeadores para organizar os seus níveis hierárquicos. Nessas implementações, cada *bucket* é visitado simultaneamente em ambas as estruturas que compõem a junção. Essa visita irá restringir, pela sobreposição das estruturas, quais *sub-buckets* deverão ser navegados.

5.3.1 Junção por Abrangência

Na implementação da junção por abrangência apresentado, no Algoritmo 5.7, deve-se navegar simultaneamente em cada nível hierárquico de ambas as estruturas envolvidas na junção.

O Algoritmo 5.7 mostra, como é feita a implementação da junção por abrangência nas estruturas de região de cobertura. A técnica que mantém um *buffer* para a estrutura interna não pode ser aplicada nessa estrutura, pois mais de uma chave pode redirecionar para um mesmo *bucket*.

Algoritmo 5.6 JuncaoProxCob(noEsq,cobEsq,repEsq,noDir,cobDir,repDir,k,raioMax,distRep,ativaBuf)

```

se noEsq e noDir sao INDICE então
  para cada objeto objEsq[i] do noEsq faça
    se (ativaBuf) então
      vetSubNoDir ← todos os sub-nos do noDir
    fim-se
    se distRep ≤ dist. repres. objEsq[i] + coberturaobjEsq[i] + cobEsq + raioMax então
      para cada objeto objDir[j] do noDir faça
        se distRep ≤ dist. repres. objDir[i] + coberturaobjDir[i] + cobDir + raioMax então
          dist ← distancia(objEsq[i], objDir[j])
          se dist ≤ coberturaobjEsq[i] + coberturaobjDir[j] + raioMax então
            noEsq' ← sub-no do objEsq[i]
            cobEsq ← cobertura objEsq[i]
            se ativaBuf então
              cobDir ← cobertura objDir[i]
              JuncaoProxCob(noEsq', cobEsq, objEsq[i], vetSubNoDir[j], cobDir, objDir[j], k, raioMax, dist, ativaBuf)
              // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
              JuncaoAbranCob'(noEsq', cobEsq, objEsq[i], vetSubNoDir[j], cobDir, objDir[j], k, raioMax, dist, ativaBuf)
            senão
              noDir' ← sub-no do objDir[j]
              cobDir ← cobertura objDir[i]
              JuncaoProxCob(noEsq', cobEsq, objEsq[i], noDir', cobDir, objDir[j], k, raioMax, dist, ativaBuf)
              // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
              JuncaoAbranCob'(noEsq', cobEsq, objEsq[i], noDir', cobDir, objDir[j], k, raioMax, dist, ativaBuf)
          fim-se
        fim-se
      fim-se
    fim-para
  fim-se
senão
  se noEsq é INDICE então
    para cada objeto objEsq[i] do noEsq faça
      se distRep ≤ dist. repres. objEsq[i] + coberturaobjEsq[i] + cobEsq + raioMax então
        dist ← distancia(objEsq[i], repDir)
        se dist ≤ cobertura objEsq[i] + raioMax então
          noEsq' ← sub-no do objEsq[i]
          cobEsq ← cobertura objEsq[i]
          JuncaoProxCob(noEsq', cobEsq, objEsq[i], noDir, cobDir, repDir, k, raioMax, dist, ativaBuf)
          // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
          JuncaoAbranCob'(noEsq', cobEsq, objEsq[i], noDir, cobDir, repDir, k, raioMax, dist, ativaBuf)
        fim-se
      fim-se
    fim-para
  senão
    se noDir é INDICE então
      se (ativaBuf) então
        vetSubNoDir ← todos os sub-nos do noDir
      fim-se
      para cada objeto objDir[j] do noDir faça
        se distRep ≤ dist. repres. objDir[i] + coberturaobjDir[i] + cobDir + raioMax então
          dist ← distancia(repEsq, objDir[j])
          se dist ≤ cobertura objDir[j] + raioMax então
            se ativaBuf então
              cobDir ← cobertura objDir[i]
              JuncaoProxCob(noEsq, cobEsq, repEsq, vetSubNoDir[j], cobDir, objDir[j], k, raioMax, dist, ativaBuf)
              // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
              JuncaoAbranCob'(noEsq, cobEsq, repEsq, vetSubNoDir[j], cobDir, objDir[j], k, raioMax, dist, ativaBuf)
            senão
              noDir' ← sub-no do objDir[j]
              cobDir ← cobertura objDir[i]
              JuncaoProxCob(noEsq, cobEsq, repEsq, noDir', cobDir, objDir[j], k, raioMax, dist, ativaBuf)
              // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
              JuncaoAbranCob'(noEsq, cobEsq, repEsq, noDir', cobDir, objDir[j], k, raioMax, dist, ativaBuf)
          fim-se
        fim-se
      fim-para
    senão
      para cada objeto objEsq[i] do noEsq faça
        se distRep ≤ dist. repres. objEsq[i] + cobEsq + raioMax então
          para cada objeto objDir[j] do noDir faça
            se distRep ≤ dist. repres. objDir[i] + cobDir + raioMax então
              dist ← distancia(objEsq[i], objDir[j])
              se dist ≤ raioMax então
                adiciona par objEsq[i] e objDir[j] na lista de resultado
                se numero de elementos em resultado ≤ k então
                  remove em resultado os elementos excedentes ao k
                  raioMax ← maior distância dos elementos de resultado
              fim-se
            fim-se
          fim-para
        fim-se
      fim-para
    fim-se
  fim-se

```

A Figura 5.3(a) mostra como duas funções mapeadoras dividem o espaço métrico de dois *buckets* índice, cada uma pertencente a uma Partição Local.

Todos os objetos representantes da estrutura que está sendo usada do lado esquerdo da junção, com suas respectivas regiões de cobertura, são representados pela letra *a* e os da estrutura que está

Algoritmo 5.7 *JuncaoAbrangPart*(*buckEsq*, *inicEsq*, *termEsq*, *buckMapEsq*, *buckDir*, *inicDir*, *termDir*, *buckMapDir*, *raio*, *ativaBuf*)

```

1: se buckEsq e buckDir são INDICE então
2:   inicEsq ← função mapeadora de início de faixa do buckEsq para o buckDir e o raio
3:   termEsq ← função mapeadora de término de faixa do buckEsq para o buckDir e o raio
4:   inicDir ← função mapeadora de início de faixa do buckDir para o buckEsq e o raio
5:   termDir ← função mapeadora de término de faixa do buckDir para o buckEsq e o raio
6:   buckMapEsq ← buckEsq
7:   buckMapDir ← buckDir
8:   para cada coordenada chaveEsq entre (inicEsq e termEsq) faça
9:     para cada coordenada chaveDir entre (inicDir e termDir) faça
10:      buckEsq ← bloco que subBucket[chaveEsq] referencia sem ler mais de uma vez
11:      buckDir ← bloco que subBucket[chaveDir] referencia sem ler mais de uma vez
12:      JuncaoAbrangPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir,
13:        buckMapDir, raio, ativaBuf)
14:    fim-para
15:  fim-para
16: senão
17:   se buckEsq é INDICE então
18:     inicEsq ← função mapeadora de início de faixa do buckEsq para o buckMapDir e o raio
19:     termEsq ← função mapeadora de término de faixa do buckEsq para o buckMapDir e o raio
20:     buckMapEsq ← buckEsq
21:     para cada coordenada chaveEsq entre (inicEsq e termEsq) faça
22:       buckEsq ← bloco que subBucket[chaveEsq] referencia sem ler mais de uma vez
23:       JuncaoAbrangPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir,
24:         buckMapDir, raio, ativaBuf)
25:     fim-para
26:   senão
27:     se buckDir é INDICE então
28:       inicDir ← função mapeadora de início de faixa do buckDir para o buckMapEsq e o raio
29:       termDir ← função mapeadora de término de faixa do buckDir para o buckMapEsq e o raio
30:       buckMapDir ← buckDir
31:       para cada coordenada chaveDir entre (inicDir e termDir) faça
32:         buckDir ← bloco que subBucket[chaveDir] referencia sem ler mais de uma vez
33:         JuncaoAbrangPart(buckEsq, inicEsq, termEsq, buckMapDir, buckDir, inicDir, termDir,
34:           buckMapDir, raio, ativaBuf)
35:       fim-para
36:     senão
37:       para cada bucket colisaoEsq ← buckEsq da lista de buckets FOLHA faça
38:         para cada objeto objEsq[i] do buckEsq faça
39:           para cada bucket colisaoDir ← buckDir da lista de buckets FOLHA faça
40:             para cada objeto objDir[j] do buckDir faça
41:               se coordenada métrica de objEsq[i] está entre a faixa inicEsq e termEsq então
42:                 se coordenada métrica de objDir[j] está entre a faixa inicDir e termDir então
43:                   se  $\text{distancia}(\text{objEsq}[i], \text{objDir}[j]) \leq \text{raio}$  então
44:                     adiciona par objEsq[i] e objDir[j] na lista de resultado
45:                   fim-se
46:                 fim-se
47:             fim-para
48:           fim-para
49:         fim-para
50:       fim-para
51:     fim-se
52:   fim-se

```

sendo usada do lado direito da junção são representados pela letra *b*.

A Figura 5.3(b) mostra duas funções de mapeamento de duas estruturas de Partição Local sendo que, a do lado esquerdo tem 3 objetos mapeadores e a do lado direito tem 4 objetos mapeadores. Pode-se observar que existe sobreposição entre os eixos dos objetos $A_0 - B_0$, $A_1 - B_1$ e $A_2 - B_2$.

Figura 5.3: Regiões que se sobrepõem entre duas estruturas Partição Local: (a) Regiões sem raio de consulta e (b) Regiões com o raio de consulta aumentando o número de cobertura de cada nó

No entanto, quando é realizada uma operação de junção por abrangência, existe um raio que é passado para a consulta que incrementa essa sobreposição, como pode ser visto na Figura 5.3(b).

Pode-se observar que as divisões que ficaram sobrepostas foram: 1 e 2 do mapeador A_0 entre 1 do mapeador B_0 ; 1 e 2 do mapeador A_1 entre 1, 2 e 3 do mapeador B_1 ; 2 do mapeador A_2 entre 3 do mapeador B_2 .

Portanto, serão visitados os sub-nós da estrutura do lado esquerdo da junção que têm as coordenadas métricas $\{2,1,1\}$, $\{2,1,2\}$, $\{2,2,1\}$ e $\{2,2,2\}$ e da estrutura do lado direito da junção, os sub-nós que têm as coordenadas $\{3,1,3\}$, $\{3,2,3\}$ e $\{3,3,3\}$.

5.3.2 Junção por Vizinhos mais Próximos

Na implementação da junção por vizinhos mais próximos navega-se em uma estrutura de cada vez; primeiro na estrutura do laço externo até chegar em um nó folha, e depois na estrutura do laço interno.

A junção por vizinhos mais próximos, mostrado no Algoritmo 5.8, deve descer somente do laço externo até chegar ao nó folha. A partir desse momento, inicia-se a descida do lado direito da estrutura realizando uma consulta por vizinhos mais próximos para cada nó da estrutura do laço externo. Quando o algoritmo completar a lista de resposta com k elementos, deve atualizar o raio máximo com o valor da maior distância de pares de objetos.

O consumo de memória desse algoritmo é dado pela alocação de i blocos da estrutura da esquerda e j blocos da estrutura da direita, onde i e j são as alturas de cada estrutura respectivamente. Além disso, deve-se manter n listas com resultados locais, onde n é a quantidade de objetos do bloco folha da estrutura da esquerda. Cada lista contém k quantidade de vizinhos mais próximos e são mantidas ordenadas aumentando assim o custo da execução.

As listas de resultados locais são necessárias para minimizar a quantidade de acessos a disco e de

Algoritmo 5.8 *JuncaoVizPart(buckEsq, buckMapEsq, buckDir, inicDir, termDir, k, raioMax, ativaBuf, resLocal)*

```

1: se buckEsq é INDICE então
2:   buckMapEsq  $\leftarrow$  buckEsq
3:   para cada coordenada chaveEsq do buckEsq faça
4:     buckEsq  $\leftarrow$  bloco que subBucket[chaveEsq] referencia sem ler mais de uma vez
5:     JuncaoVizPart(buckEsq, buckMapEsq, buckDir, inicDir, termDir, k, raioMax, ativaBuf, NULO)
6:   fim-para
7: senão
8:   se buckDir é INDICE então
9:     cria um vetor de listas de resultados resLocal[] com o numero de elementos do buckEsq
10:    para cada bucket colisaoEsq  $\leftarrow$  buckEsq da lista de buckets FOLHA faça
11:      para cada objeto objEsq[i] no colisaoEsq faça
12:        inicDir  $\leftarrow$  função mapeadora de início de faixa do buckDir para o buckMapEsq
13:        termDir  $\leftarrow$  função mapeadora de término de faixa do buckDir para o buckMapEsq
14:        para cada coordenada chaveDir entre (inicDir e termDir) faça
15:          buckDir  $\leftarrow$  bloco que subBucket[chaveDir] referencia sem ler mais de uma vez
16:          JuncaoVizPart(buckEsq, buckMapEsq, buckDir, inicDir, termDir, k, raioMax, ativaBuf, resLocal[i])
17:          // consulta por abrangência que elimina os excedentes ao k e minimiza raioMax
18:          ConsultaAbrangPart'(buckDir, inicDir, termDir, objEsq[i], k, raioMax, resLocal[i])
19:        fim-para
20:      fim-para
21:    fim-para
22:    copia o vetor resLocal para a lista resultados
23:  senão
24:    para cada bucket colisaoDir  $\leftarrow$  buckDir da lista de buckets FOLHA faça
25:      para cada objeto objDir[j] no colisaoDir faça
26:        se coordenada métrica de objDir[j] está entre a faixa inicDir e termDir então
27:          se  $\text{distancia}(\text{objEsq}[i], \text{objDir}[j]) \leq \text{raioMax}$  então
28:            adiciona par objEsq[i] e objDir[j] na lista de resLocal
29:          se numero de elementos em resLocal  $\leq k$  então
30:            remove em resLocal os elementos excedentes ao k
31:            raioMax  $\leftarrow$  maior distância dos elementos de resLocal
32:          fim-se
33:        fim-se
34:      fim-para
35:    fim-para
36:  fim-para
37: fim-se
38: fim-se

```

cálculos de distância. Isto acontece porque a inclusão de um par de objetos não é garantia de que tal par faça realmente parte da resposta.

5.3.3 Junção por Proximidade

Na implementação da junção por proximidade deve-se navegar simultaneamente em cada nível hierárquico de ambas as estruturas envolvidas na junção.

A junção por proximidade é muito parecida com o Algoritmo 5.5 que é dividido em dois algoritmos: o primeiro apresentado no Algoritmo 5.9 e o segundo é mostrado no Algoritmo 5.7 que deve ser modificado para remover os objetos excedentes ao *k*, minimizando o raio máximo com o valor da maior distância de pares de objetos.

O consumo de memória desse algoritmo, é dado pela alocação de *i* blocos da estrutura da esquerda

e j blocos da estrutura da direita, onde i e j são as alturas de cada estrutura respectivamente.

5.4 Técnicas usando Árvore AVL para Lista Global de Resposta

A literatura cita que, em todo algoritmo de consulta por vizinhos mais próximos, seja para qualquer estrutura, deve ser mantida em memória primária uma lista global. Essa lista deve ser ordenada pelas distâncias do objeto de consulta para os objetos de resposta.

Durante a execução do algoritmo de consulta por vizinhos mais próximos, uma inserção ordenada na lista pode gerar a retirada de um objeto excedente à quantidade k , ou mesmo de vários objetos se a consulta deve recuperar também a lista de empate.

Existem duas formas tradicionais de implementação dessa lista: usando o vetor estático ou lista duplamente encadeada. A implementação usando vetor estático apresenta um problema quando a consulta exige a lista, pois não é possível prever o tamanho do vetor. A vantagem é que a operação de retirada não é custosa, pois pode-se navegar a partir do final do vetor para o início e remover os objetos excedentes. O problema aparece nas operações de inserção, pois embora sejam beneficiadas pela possibilidade de se usar busca binária, existe a necessidade de deslocamento dos elementos até o final do vetor.

A implementação usando lista duplamente encadeada usa um pouco mais de memória, pois em cada nó devem ser armazenados também os endereços para os nós antecessor e sucessor, mas não precisa reservar memória para uma possível lista de empate. A vantagem também está na operação de retirada que não é custosa, pois pode navegar do final da lista para o início removendo os excedentes. O problema aparece novamente na inserção, pois embora não se desloquem elementos por causa da alocação dinâmica, não se pode usar agora a busca binária, obrigando que se realize uma busca sequencial até encontrar a posição de inserção do novo elemento.

Assim, qualquer forma de implementação apresenta alto custo quando a consulta de seleção dos k vizinhos mais próximos retorna uma grande quantidade de resposta, ou quando a estrutura indexa vários níveis e assim, a todo momento acontece inserção e retirada de objetos excedentes na lista.

O algoritmo básico da consulta por abrangência não necessita dessa lista global. Entretanto, se além da operação de consulta for requisitada, também a ordenação das distâncias de resposta, faz-se necessário a implementação dessa lista global. Essa situação se agrava quando é realizada uma operação de junção, pois o conjunto de resposta normalmente é bem maior.

Para solucionar esse problema, optou-se por implementar uma lista global usando-se uma árvore binária balanceada AVL criada em 1962 pelos matemáticos russos Adelson-Velskii e Landis [65]. A inserção é feita através de navegação em altura na árvore e acessa $\log_2(n)$ elementos, onde n é a

Algoritmo 5.9 JuncaoProxPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir, buckMapDir, k, raioMax, ativaBuf)

```

1: se buckEsq e buckDir sao INDICE então
2:   inicEsq ← função mapeadora de início de faixa do buckEsq para o buckDir
3:   termEsq ← função mapeadora de término de faixa do buckEsq para o buckDir
4:   inicDir ← função mapeadora de início de faixa do buckDir para o buckEsq
5:   termDir ← função mapeadora de término de faixa do buckDir para o buckEsq
6:   buckMapEsq ← buckEsq
7:   buckMapDir ← buckDir
8:   para cada coordenada chaveEsq entre (inicEsq e termEsq) faça
9:     para cada coordenada chaveDir entre (inicDir e termDir) faça
10:      buckEsq ← bloco que subBucket[chaveEsq] referencia sem ler mais de uma vez
11:      buckDir ← bloco que subBucket[chaveDir] referencia sem ler mais de uma vez
12:      JuncaoProxPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir, buckMapDir,
13:        k, raioMax, ativaBuf)
14:      // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
15:      JuncaoAbrangPart'(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir,
16:        buckMapDir, k, raioMax, ativaBuf)
17:   fim-para
18: fim-para
19: senão
20:   se buckEsq é INDICE então
21:     inicEsq ← função mapeadora de início de faixa do buckEsq para o buckMapDir
22:     termEsq ← função mapeadora de término de faixa do buckEsq para o buckMapDir
23:     buckMapEsq ← buckEsq
24:     para cada coordenada chaveEsq entre (inicEsq e termEsq) faça
25:       buckEsq ← bloco que subBucket[chaveEsq] referencia sem ler mais de uma vez
26:       JuncaoProxPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir, buckMapDir,
27:         k, raioMax, ativaBuf)
28:       // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
29:       JuncaoAbrangPart'(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir,
30:         buckMapDir, k, raioMax, ativaBuf)
31:     fim-para
32:   senão
33:     se buckDir é INDICE então
34:       inicDir ← função mapeadora de início de faixa do buckDir para o buckMapEsq
35:       termDir ← função mapeadora de término de faixa do buckDir para o buckMapEsq
36:       buckMapDir ← buckDir
37:       para cada coordenada chaveDir entre (inicDir e termDir) faça
38:         buckDir ← bloco que subBucket[chaveDir] referencia sem ler mais de uma vez
39:         JuncaoProxPart(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir, buckMapDir,
40:           k, raioMax, ativaBuf)
41:         // junção por abrangência que elimina os excedentes ao k e minimiza raioMax
42:         JuncaoAbrangPart'(buckEsq, inicEsq, termEsq, buckMapEsq, buckDir, inicDir, termDir,
43:           buckMapDir, k, raioMax, ativaBuf)
44:       fim-para
45:     fim-para
46:   para cada bucket colisaoEsq ← buckEsq da lista de buckets FOLHA faça
47:     para cada objeto objEsq[i] no colisaoEsq faça
48:       para cada bucket colisaoDir ← buckDir da lista de buckets FOLHA faça
49:         para cada objeto objDir[j] no colisaoDir faça
50:           se coordenada métrica de objEsq[i] está entre a faixa inicEsq e termEsq então
51:             se coordenada métrica de objDir[j] está entre a faixa inicDir e termDir então
52:               se distancia(objEsq[i], objDir[j]) ≤ raioMax então
53:                 adiciona par objEsq[i] e objDir[j] na lista de resultado
54:               se numero de elementos em resultado ≤ k então
55:                 remove em resultado os elementos excedentes ao k
56:                 raioMax ← maior distância dos elementos de resultado
57:             fim-se
58:           fim-se
59:         fim-se
60:       fim-para
61:     fim-para
62:   fim-se
63: fim-se

```

quantidade de elementos. A remoção dos excedentes navega a partir do último elemento à direita para que ele seja removido. Como a árvore está balanceada, essa operação também acessa $\log_2(n)$ elementos. Se a consulta exigir a lista de empate, antes de remover todos os objetos repetidos, deve-se garantir que a lista não vai ficar menor que k objetos.

5.5 Experimentos

O conjunto de dados usado neste experimento é o conjunto de cidades do projeto *GEOnet Names Server* [60] que contém 3.976.221 pontos geográficos (latitude e longitude). Os dados foram filtrados para compor quatro subconjuntos, que contêm as cidades de quatro países: Brasil, com 47.518 pontos; Peru, com 36.831 pontos; Argentina, com 28.027 pontos e o Equador, com 9.230 pontos.

A máquina utilizada para realizar esses experimentos possui a seguinte configuração: processador Intel Xeon de 2.08GHz com *cache* L2 de 512K, 3Gb de memória RAM, HD 200Gb 7200rpm. Embora a máquina tenha dois processadores, os algoritmos são sequenciais ou seja, processados em um único processador. Um dado importante da configuração é o *cache* de 512 para o processador que permite realizar os cálculos de distância das proteínas e das palavras em *cache* do que em memória RAM. A função de distância para o conjunto de cidades é a mesma utilizada no capítulo 4.

Para cada conjunto de dados foram construídas estruturas sequencial e quatro métodos de acesso: Sequencial; *Slim-Tree* com quebra pelo algoritmo MINMAX e otimizado pelo algoritmo *SlimDown*; Partição Local com escolha de mapeador pelo randômico, e Partição Local+ com escolha de mapeador pelo randômico. Além disso, para cada método de acesso e conjunto de dados, foram construídas as estruturas usando diferentes tamanhos de bloco em disco: 512 *bytes*, 1024 *bytes* e 2048 *bytes*, como pode ser visto na Tabela 5.1.

As características mostradas na Tabela 5.1 são: a base que é o subconjunto de dados do GEONet; o tamanho do bloco; a altura para as estruturas *Slim-Tree* e a altura máxima das estruturas Partição Local e Partição Local+; a quantidade de nó índice; a quantidade de nó folha; o tamanho do arquivo gerado; a porcentagem de objetos em relação ao total de registros que estão em regiões de colisão, sendo que essa característica só se aplica para as estruturas baseadas em coordenadas métricas; o número de acessos a disco realizados na construção das estruturas; o número de cálculos de distância realizados na construção das estruturas; e o tempo de execução para a construção das estruturas.

O tempo, número de acessos a disco e números de cálculos de distância mostrado para a estrutura *Slim-Tree* refere-se ao total para a construção e execução do algoritmo de *Slim-Down*.

A partir dos valores da Tabela 5.1, podemos concluir que:

- os métodos de acesso baseado em coordenadas métricas geram estruturas com até o dobro

Tabela 5.1: Características das Estruturas usadas na Junção da Base GEONet

Estrutura	Base	Tam. Bloco	Alt.	Qt.Nó Índice	Qt.Nó Folha	Tam.Arq (bytes)	% Obj. Colisão	Acesso Disco	Cálc. Dist.	Tempo (ms)
Sequencial	Argentina	512	0	0	905	463.872	0,0000%	28.026	0	297
Slim Down	Argentina	512	5	343	2.197	1.300.992	0,0000%	136.794	1.480.473	1.625
Part.Local	Argentina	512	8	491	1.761	1.153.536	0,3390%	136.132	520.148	1.078
Part.Local+	Argentina	512	9	707	2.476	1.630.208	1,5985%	137.423	464.406	1.094
Sequencial	Brasil	512	0	0	1.533	785.408	0,0000%	47.518	0	485
Slim Down	Brasil	512	5	581	3.695	2.189.824	0,0000%	238.409	2.587.942	2.828
Part.Local	Brasil	512	9	839	2.981	1.956.352	2,1886%	245.443	1.092.167	1.968
Part.Local+	Brasil	512	9	1.119	4.076	2.660.352	5,9956%	273.901	981.930	2.015
Sequencial	Equador	512	0	0	298	153.088	0,0000%	9.229	0	140
Slim Down	Equador	512	5	120	712	426.496	0,0000%	38.507	506.320	547
Part.Local	Equador	512	7	168	593	390.144	1,4410%	38.102	197.043	343
Part.Local+	Equador	512	7	228	827	540.672	1,8527%	39.056	226.126	375
Sequencial	Peru	512	0	0	1.189	609.280	0,0000%	36.831	0	359
Slim Down	Peru	512	5	446	2.908	1.717.760	0,0000%	185.503	2.008.688	2.203
Part.Local	Peru	512	10	662	2.309	1.521.664	0,9693%	198.615	881.978	1.594
Part.Local+	Peru	512	9	785	3.056	1.967.104	3,5592%	738.621	689.932	2.140
Sequencial	Argentina	1.024	0	0	445	456.704	0,0000%	28.026	0	297
Slim Down	Argentina	1.024	4	85	1.120	1.234.944	0,0000%	105.646	2.360.670	2.078
Part.Local	Argentina	1.024	7	232	860	1.119.232	0,2640%	119.859	711.042	1.157
Part.Local+	Argentina	1.024	9	359	1.241	1.639.424	3,8962%	142.614	621.809	1.219
Sequencial	Brasil	1.024	0	0	755	774.144	0,0000%	47.518	0	468
Slim Down	Brasil	1.024	4	146	1.855	2.050.048	0,0000%	186.326	5.235.217	4.156
Part.Local	Brasil	1.024	7	408	1.457	1.910.784	0,5893%	208.584	961.335	1.797
Part.Local+	Brasil	1.024	9	627	2.145	2.839.552	0,1473%	224.903	757.123	1.844
Sequencial	Equador	1.024	0	0	147	151.552	0,0000%	9.229	0	110
Slim Down	Equador	1.024	3	24	359	393.216	0,0000%	28.245	1.337.910	969
Part.Local	Equador	1.024	7	74	278	361.472	1,2459%	39.256	485.003	515
Part.Local+	Equador	1.024	7	124	415	552.960	0,4225%	35.286	220.052	375
Sequencial	Peru	1.024	0	0	585	600.064	0,0000%	36.831	0	375
Slim Down	Peru	1.024	4	108	1.447	1.593.344	0,0000%	141.198	3.816.530	3.094
Part.Local	Peru	1.024	8	329	1.148	1.513.472	0,7684%	171.074	1.613.853	1.906
Part.Local+	Peru	1.024	8	475	1.701	2.229.248	0,0000%	172.317	486.012	1.343
Sequencial	Argentina	2.048	0	0	221	454.656	0,0000%	28.026	0	297
Slim Down	Argentina	2.048	3	17	517	1.095.680	0,0000%	83.181	3.988.615	2.953
Part.Local	Argentina	2.048	5	86	439	1.077.248	0,0000%	98.069	347.396	1.047
Part.Local+	Argentina	2.048	6	109	617	1.488.896	0,0000%	99.308	354.329	1.109
Sequencial	Brasil	2.048	0	0	375	770.048	0,0000%	47.518	0	578
Slim Down	Brasil	2.048	3	33	911	1.935.360	0,0000%	142.893	12.626.802	8.141
Part.Local	Brasil	2.048	5	130	734	1.771.520	0,0000%	180.983	642.862	1.907
Part.Local+	Brasil	2.048	6	198	1.068	2.594.816	0,0000%	173.905	627.264	1.953
Sequencial	Equador	2.048	0	0	73	151.552	0,0000%	9.229	0	109
Slim Down	Equador	2.048	3	8	178	382.976	0,0000%	25.632	5.830.755	3.344
Part.Local	Equador	2.048	4	22	137	327.680	0,0000%	27.153	92.317	312
Part.Local+	Equador	2.048	5	41	207	509.952	0,0000%	27.612	93.028	312
Sequencial	Peru	2.048	0	0	291	598.016	0,0000%	36.831	0	407
Slim Down	Peru	2.048	3	27	715	1.521.664	0,0000%	110.120	12.027.615	7.422
Part.Local	Peru	2.048	5	97	558	1.343.488	0,0000%	135.410	481.857	1.422
Part.Local+	Peru	2.048	5	133	824	1.961.984	0,0000%	138.282	498.576	1.546

de níveis do que as estruturas baseadas em região de cobertura. Essa diferença mostra que a escolha dos objetos mapeadores que compõem a função de mapeamento local não foi boa para qualquer variação de tamanho de bloco;

- as estruturas Partição Local possuem em média 25% menos blocos do tipo folha do que a estrutura *Slim-Tree*;
- as estruturas Partição Local+ possuem em média 13% mais blocos do tipo folha do que a estrutura *Slim-Tree*;

- a colisão em algumas base chegou até 7.7% da base, influenciadas pela escolha ruim dos objetos mapeadores; e
- as estruturas Partição Local são construídas em 1,6 vezes menos tempo do que a estrutura *Slim-Tree*, pois realiza menos cálculos de distância e menos acessos a disco;

Junção por Abrangência

Os gráficos das Figuras 5.4 e 5.5 mostram o número de acessos a disco, número de cálculos de distância e o tempo de execução para a junção por abrangência entre os pontos geográficos da Argentina e do Brasil, onde o raio foi 10, 20, 30, 40, 50, 60, 70, 80, 90 e 100 km.

Pelos valores da Tabela 5.1, podemos calcular o número de acessos a disco para qualquer tamanho de raio na junção por abrangência em bloco aninhado, em arquivos sequenciais, multiplicando a quantidade de nó folhas dos arquivos envolvidos. Portanto, o número de acessos a disco entre a base da Argentina e do Brasil é de: 1.387.365 para blocos de 512 *bytes*; 335.975 para blocos de 1024 *bytes* e 82.875 para blocos de 2048 *bytes*.

Da mesma maneira, o número de cálculos de distância para qualquer tamanho de raio na junção por abrangência em bloco aninhado, em arquivos sequenciais, é dado pela multiplicação da quantidade de objetos dos arquivos envolvidos. Portanto, o número de cálculos de distância entre a base da Argentina e do Brasil é de 1.331.786.986 independente do tamanho dos blocos.

Comparando os gráficos das Figuras 5.4 e 5.5 com os resultados da Tabela 5.1 pode-se concluir que:

- o número de acessos a disco é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo sequencial. A estrutura que obteve os melhores resultados com o raio valendo 100 km, foi a Partição Local, com os seguintes números de acessos a disco: 3.704 ou 0,26% do acesso da junção por abrangência em bloco aninhado com bloco de tamanho de 512 *bytes*; 2.192 ou 0,65% do acesso da junção por abrangência em bloco aninhado com bloco de tamanho de 1024 *bytes* e 1.790 ou 2,16% do acesso da junção por abrangência em bloco aninhado com bloco de tamanho de 2048 *bytes*;
- o número de cálculos de distância é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo sequencial. A estrutura que obteve os melhores resultados com o raio valendo 100 km, foi a Partição Local+, com os seguintes números de cálculos de distância: 185.117 ou 0,014% da junção por abrangência em bloco aninhado com blocos de tamanho de 512 *bytes*; 253.762 ou 0,019% da junção por

Figura 5.4: Gráficos com números de acessos a disco (1^a coluna) e de número de cálculos de distância (2^a coluna) para junção por abrangência entre as bases Argentina e Brasil

abrangência em bloco aninhado com bloco de tamanho de 1024 *bytes* e 370.570 ou 0,027% da junção por abrangência em bloco aninhado com bloco de tamanho de 2048 *bytes*; e

- as estruturas obtiveram tempo de execução semelhantes pois, o número de cálculos de distância é superior ao acesso a disco.

Os gráficos das Figuras 5.6 e 5.7 mostram o número de acessos a disco, número de cálculos de

(a)

(b)

(c)

Figura 5.5: Gráficos de tempo de execução para junção por abrangência entre as bases Argentina e Brasil

distância e o tempo de execução para a junção por abrangência entre os pontos geográficos do Equador e Peru, onde o raio foi 10, 20, 30, 40, 50, 60, 70, 80, 90 e 100 km.

Pelos valores da Tabela 5.1, podemos calcular o número de acessos a disco para qualquer tamanho de raio na junção por abrangência em bloco aninhado, em arquivos seqüenciais, multiplicando a quantidade de nó folhas dos arquivos envolvidos. Portanto, o número de acessos a disco entre a base do Equador e do Peru é de: 354.322 para blocos de 512 *bytes*; 85.995 para blocos de 1024 *bytes* e 21.243 para blocos de 2048 *bytes*.

Da mesma maneira, o número de cálculos de distância para qualquer tamanho de raio na junção por abrangência em bloco aninhado, em arquivos seqüenciais, é dado pela multiplicação da quantidade de objetos dos arquivos envolvidos. Portanto, o número de cálculos de distância entre a base do Equador e do Peru é de 339.950.130 independente do tamanho dos blocos.

Comparando os gráficos das Figuras 5.6 e 5.7 com os resultados da Tabela 5.1, pode-se concluir que:

- o número de acessos a disco é menor nas estruturas baseadas em região de cobertura e nas es-

Figura 5.6: Gráficos com números de acessos a disco (1ª coluna) e de número de cálculos de distância (2ª coluna) para junção por abrangência entre as bases Equador e Peru

truturas baseadas em coordenadas métricas comparado com o arquivo seqüencial. A estrutura que obteve os melhores resultados com o raio valendo 100 km, foi a Partição Local, com os seguintes números de acessos a disco: 14.263 ou 4,02% do acesso da junção por abrangência em bloco aninhado de tamanho de 512 *bytes*; 5.343 ou 6,21% do acesso da junção por abrangência em bloco aninhado de tamanho de 1024 *bytes* e 900 ou 4,23% do acesso da junção por

(a)

(b)

(c)

Figura 5.7: Gráficos de tempo de execução para junção por abrangência entre as bases Equador e Peru

abrangência em bloco aninhado de tamanho de 2048 *bytes*;

- o número de cálculos de distância é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo sequencial. A estrutura que obteve os melhores resultados com o raio valendo 100 km, foi a Partição Local+, com os seguintes números de cálculos de distância: 2.077.100 ou 0,61% da junção por abrangência em bloco aninhado de tamanho de 512 *bytes*; 1.960.784 ou 0,57% da junção por abrangência em bloco aninhado de tamanho de 1024 *bytes* e 2.982.244 ou 0,87% da junção por abrangência em bloco aninhado de tamanho de 2048 *bytes*; e
- as estruturas obtiveram tempo de execução semelhantes, pois o número de cálculos de distância é superior ao acesso a disco.

Junção aos Vizinhos mais Próximos

Os gráficos da Figura 5.8 mostram o número de acessos a disco, número de cálculos de distância e o tempo de execução para a junção aos vizinhos mais próximos entre os pontos geográficos do Equador e Brasil em que o k foi 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10 pontos geográficos.

(a)

(b)

(c)

Figura 5.8: Gráficos de Número de Acessos a Disco, Número de Cálculos de Distância e o Tempo de Execução para junção aos vizinhos mais próximos entre as bases Equador e Brasil

Pelos valores da Tabela 5.1, pode-se calcular o número de acessos a disco para qualquer tamanho de raio na junção aos vizinhos mais próximos em bloco aninhado, em arquivos seqüenciais, multiplicando a quantidade de nós folha dos arquivos envolvidos. Portanto, o número de acessos a disco entre a base do Equador e do Brasil é de 456.834 para blocos de 512 *bytes*.

Da mesma maneira, o número de cálculos de distância para qualquer tamanho de raio na junção aos vizinhos mais próximos em bloco aninhado, em arquivos seqüenciais, é dado pela multiplicação da quantidade de objetos dos arquivos envolvidos. Portanto, o número de cálculos de distância entre a base do Equador e do Peru é de 438.591.140 independente do tamanho dos blocos.

Comparando o gráfico da Figura 5.8 com os resultados da Tabela 5.1, pode-se concluir que:

- o número de acessos a disco é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo seqüencial. A estrutura que obteve os melhores resultados com o k valendo 10 unidades, foi a *Slim-Tree* com 193.522 ou 42,36% acessos da junção aos vizinhos mais próximos em bloco aninhado de tamanho de 512 *bytes*; e
- o número de cálculos de distância é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo seqüencial. A estrutura que obteve os melhores resultados com o k valendo 10 unidades foi a *Slim-Tree* com 3.710.163 ou 0,84% da junção aos vizinhos mais próximos em bloco aninhado de tamanho de 512 *bytes*.

Junção por Proximidade

Os gráficos da Figura 5.9 mostram o número de acessos a disco, número de cálculos de distância e o tempo de execução para a junção por proximidade entre os pontos geográficos do Peru e Brasil em que o k foi 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10 pontos geográficos.

(a)

(b)

(c)

Figura 5.9: Gráficos de Número de Acessos a Disco, Número de Cálculos de Distância e o Tempo de Execução para junção por proximidade entre as bases Peru e Brasil

Pelos valores da Tabela 5.1, pode-se calcular o número de acessos a disco para qualquer tamanho de raio na junção por proximidade em bloco aninhado, em arquivos seqüenciais, multiplicando a quantidade de nós folha dos arquivos envolvidos. Portanto, o número de acessos a disco entre a base do Peru e do Brasil é de 1.822.737 para blocos de 512 *bytes*.

Da mesma maneira, o número de cálculos de distância para qualquer tamanho de raio na junção por proximidade em bloco aninhado, em arquivos seqüenciais, é dado pela multiplicação da quantidade de objetos dos arquivos envolvidos que é de 1.750.135.458 cálculos.

Comparando o gráfico da Figura 5.9 com os resultados da Tabela 5.1, pode-se concluir que:

- o número de acessos a disco é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo seqüencial. A estrutura que obteve os melhores resultados com o *k* valendo 10 unidades foi a *Slim-Tree* com 697.833 ou 38, 28% acesso da junção por proximidade em bloco aninhado de tamanho de 512 *bytes*; e.
- o número de cálculos de distância é menor nas estruturas baseadas em região de cobertura e nas estruturas baseadas em coordenadas métricas comparado com o arquivo seqüencial. A estrutura que obteve os melhores resultados com o *k* valendo 10 unidades foi a *Slim-Tree* com 3.710.163 ou 0, 21% da junção por proximidade em bloco aninhado de tamanho de 512 *bytes*.

Discussão dos Resultados

Os resultados mostraram que a organização das estruturas de baseadas em região de cobertura realizam menor número de cálculos de distância, menor número de acessos a disco e utilizam pouco tempo para realizar os algoritmos de junção.

As estruturas baseadas em particionamento apresentam um problema que deverá ser tratado na função mapeadora que determina as coordenadas de início e fim de ambas estruturas. Como mencionado anteriormente, a função mapeadora usa as distâncias entre os mapeadores para determinar as coordenadas de início e fim. O problema está em, como não existe uma distância máxima para cada mapeador, a função mapeadora pode habilitar uma região desnecessária (a região externa) em cada *bucket* navegado. Isso faz com que as estruturas baseadas em particionamento façam mais acessos a disco, cálculos de distância e conseqüentemente gastam mais tempo.

Uma proposta para solucionar esse problema é alterar os *buckets* índices (responsáveis pelo armazenamento das funções mapeadores) nas estruturas de particionamento para armazenar essa distância. Essa distância máxima não é necessária para os algoritmos de seleção, mas deve minimizar bastante o número de cálculos de distância, número de acessos a disco e tempo

5.6 Conclusão

Os testes da junção por abrangência mostraram que as estruturas de particionamento e de região de cobertura apresentam tempos próximos.

As estruturas baseadas em região de cobertura sempre ganham para responder junção aos vizinhos mais próximos e junção por proximidade.

As estruturas baseadas em particionamento apresentam um problema na função de mapeamento para determinar as coordenadas de início e fim da junção.

No próximo capítulo serão apresentadas as conclusões sobre esse trabalho e as propostas de projetos futuros.

CAPÍTULO 6 Conclusões

As primeiras pesquisas com armazenamento de dados convencionais relatam a utilização de estruturas baseadas em árvore e, em uma segunda fase surgiram as estruturas baseadas em funções de mapeamento local.

Esse trabalho definiu duas funções de mapeamento local com a característica de extensibilidade que apresentou um menor custo em todos os aspectos: acessos a disco e cálculos de distância, refletindo um menor tempo de resposta.

Um outro aspecto é que as consultas por similaridade por abrangência e dos vizinhos mais próximos têm sido considerados como equivalentes ao operador de seleção. Porém, para que essas consultas possam ser incorporadas aos gerenciadores, exige-se que essas suportem também as operações de junção.

Neste trabalho foram propostas técnicas para realizar junção por abrangência, aos vizinhos mais próximos e por proximidade nas estruturas de região de cobertura e de particionamento disjunto.

6.1 Resultados

Os principais resultados deste trabalho foram:

- um novo conceito para particionar o espaço métrico extensivelmente, baseando em distância entre os objetos mapeadores que dividem o espaço métrico em regiões disjuntas;
- uma estrutura baseada em funções de mapeamento local onde cada *bucket* da estrutura particiona localmente o espaço métrico, chamada Partição Local. Essa estrutura é a que realiza menos acessos a disco e assim, é recomendável para conjuntos de dados em que a função de distância não é custosa;
- uma extensão na estrutura Partição Local que armazena nos seus nós folhas a distância para o primeiro mapeador do *bucket* do nível acima, chamada Partição Local+. Essa estrutura realiza

menos cálculos de distância, sendo recomendável para conjuntos de dados em que a função de distância é custosa;

- uma melhora no custo de execução da consulta por abrangência usando estrutura baseada em particionamento sob as estruturas em região de cobertura, o que resultou em um ganho no conjunto de pontos geográficos, para blocos de 512 *bytes*, de até 5,07 vezes menos cálculos de distância; 13,23 vezes menos acessos a disco; e 185% menos tempo de execução.
- uma melhora no custo de execução da consulta dos vizinhos mais próximos usando estrutura baseada em particionamento sob as estruturas em região de cobertura, o que resultou em um ganho no conjunto de pontos geográficos, para blocos de 512 *bytes*, de até 18,79 vezes menos cálculos de distância; 43,29 vezes menos acessos a disco; e 177% menos tempo de execução.
- as estruturas baseadas em particionamento, apresentaram melhor escalabilidade para a realização das consultas por abrangência e aos vizinhos mais próximos do que as estruturas baseadas em região de cobertura. Isso acontece pois, a medida em que se aumenta a quantidade de objetos nas estruturas baseadas em região de cobertura aumenta o custo de realizar consultas de abrangência e de vizinhos mais próximos, comparado com as estruturas de particionamento;
- os algoritmos de junção por abrangência, junção aos vizinhos mais próximos e junção por proximidade nas estruturas de região de cobertura e nas estruturas de particionamento;
- os algoritmos de junção por abrangência apresentaram pouca diferença em tempo entre as estruturas de particionamento e as estruturas de região de cobertura;
- os algoritmos de junção aos vizinhos mais próximos e por proximidade apresentaram melhor desempenho nas estruturas baseadas em região de cobertura do que as estruturas de particionamento;
- a técnica para manter em *buffer* todas as sub-páginas da estrutura do lado direito da junção de uma página que foi visitada que reduz o número de acessos a disco, mas consome mais memória;
- uma técnica que usa uma árvore AVL para a lista de resposta que minimiza o custo da lista global de resposta para as consultas por similaridade: vizinhos mais próximos e consulta por abrangência com ordenação das distâncias da resposta. A árvore AVL foi aplicada nos algoritmos de junção por similaridade;

- uma melhora no custo de execução da junção por abrangência usando a junção hierárquica nas estruturas *Slim-Tree*, Partição Local e Partição Local+ sob as estruturas sequenciais. Essa junção consumiu: 0,26% do número de acessos a disco na junção por abrangência da sequencial; e 0,014% do número de cálculos de distância na junção por abrangência da sequencial.
- duas formas de escolha do novo mapeador para a função de mapeamento local nas estruturas de particionamento do espaço métrico, além do randômico, baseadas na maior variância do valor da coordenada métrica e na maior contagem do valor da coordenada métrica diferentes. Essas técnicas não apresentaram ganhos constantes e por isso esse trabalho recomenda a técnica de escolha randômica, pois torna mais rápida a inserção dos objetos na estrutura.

6.2 Trabalhos Futuros

Os resultados desse trabalho abrem pesquisas para:

- Criação de uma estrutura de particionamento que usa uma única função de mapeamento global e extensível, ou seja, uma estrutura de *hashing*;
- Priorização de sub-páginas para os algoritmos de junção aos vizinhos mais próximos e por proximidade. Uma escolha melhor pode minimizar bastante o raio de busca nesses algoritmos e diminuir a quantidade de acessos a disco e cálculos de distância;
- Suporte às atuais estruturas métricas para o armazenamento de objetos sem que seja necessário aumentar o tamanho de bloco. Um exemplo são as proteínas que podem variar de poucos aminoácidos até mais que 20 mil aminoácidos. Para indexar proteínas desse tamanho as atuais estruturas devem usar blocos superiores a 64 *kbytes*, que são maiores do que os que têm suporte aos atuais sistemas de arquivos;
- Indexar maior quantidade de dados e conseguir menos tempo de resposta para as consultas, distribuindo as estruturas em várias máquinas para criar uma estrutura de *grid*;
- Suporte nas estruturas de particionamento local para resolver consultas usando os operadores lógicos: E, OU e NÃO;
- Explorar novas formas de escolha dos mapeadores que melhorem o espalhamento dos valores de coordenadas métricas e que não geram colisões;
- Explorar formas para restringir as sub-páginas que serão navegadas pelos algoritmos de junção por vizinhos mais próximos e por proximidade na técnica de junção hierárquica; e

- Novas formas de otimização da junção quando são utilizadas em expressões de consulta enca-deando tanto com operadores lógicos quanto operadores de seleção.

Referências Bibliográficas

- [1] Jeffrey K. Uhlmann. Satisfying General Proximity/Similarity Queries with Metric Trees. *Information Processing Letter*, 40(4):175–179, 1991.
- [2] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins using R-Trees. *ACM SIGMOD Record*, 22(2):237–246, June 1993.
- [3] Christian Böhm and Florian Krebs. High Performance Data Mining Using the Nearest Neighbor Join. In *Proceedings. 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 43–50, Maebashi City, Japan, December 2002. IEEE Computer Society.
- [4] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.
- [5] Ramez Elmasri, Gene T. J. Wu, and Yeong-Joon Kim. The Time Index: An Access Structure for Temporal Data. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 1–12, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
- [6] J. Lue, K. Brown, and W. B. Samson. Indexing Techniques for Temporal Data. In *Database and Expert Systems Application, 6th International Conference DEXA*, pages 134–141, London, UK, 1995.
- [7] Cheng Hian Goh, Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. Indexing Temporal Data Using Existing B+-trees. *Data Knowledge Engineering*, 18(2):147–165, 1996.
- [8] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. In *The VLDB Journal*, pages 345–356, 1998.
- [9] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

- [10] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM Press.
- [11] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings 21st International Conference Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 1995.
- [12] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+—Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB'87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.
- [13] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An Efficient and Robust Access Method for Points and Rectangles. In *In Proc. ACM SIGMOD International Conference on Management of Data*, pages 322–331, May 23–25 1990.
- [14] Dimitris Papadias, Timos Sellis, Yannis Theodoridis, and Max J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In *SIGMOD Conference*, pages 92–103, San Jose, 1995. ACM Press.
- [15] Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.
- [16] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, and Zenon Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. In *The VLDB Journal*, pages 215–226, Bombay, India, 1996. Morgan Kaufmann.
- [17] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "Nearest Neighbor" Meaningful? In *ICDT'99: Proceeding of the 7th International Conference on Database Theory*, pages 217–235, London, UK, 1990. Springer-Verlag.
- [18] Apostol Natsev, Rajeev Rastogi, and Kyuseok Shim. WALRUS: A Similarity Retrieval Algorithm for Image Databases. In *SIGMOD Conference*, pages 395–406, 1999.
- [19] Atsuo Yoshitaka and Tadao Ichikawa. A Survey on Content-Based Retrieval for Multimedia Databases. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):81–93, 1999.

- [20] Peter N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA '93: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.
- [21] Tolga Bozkaya and Meral Ozsoyoglu. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Transactions on Database Systems*, 24(3):361–404, 1999.
- [22] Tolga Bozkaya and Meral Ozsoyoglu. Distance-Based Indexing for High-Dimensional Metric Spaces. In *ACM Int'l Conference on Data Management - SIGMOD*, pages 357–368. AMC Press, 1997.
- [23] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. In Matthias Jarke, Michael Carey, Klaus R. Dittrich, Fred Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *The VLDB Journal*, pages 426–435, Athens, Greece, August 1997. Morgan Kaufmann Publishers, Inc.
- [24] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *EDBT'00: Proceedings of the 7th International Conference on Extending Database Technology*, volume 1777 of *Lecture Notes in Computer Science*, pages 51–65, London, UK, March 27-31 2000. Springer.
- [25] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity Search in Metric Databases through Hashing. In *Proceedings of the 2001 ACM Workshops on Multimedia: Multimedia Information Retrieval*, pages 1–5, New York, NY, USA, April 2001. ACM Press.
- [26] Christos Faloutsos. Indexing of Multimedia Data. In *Multimedia Databases in Perspective*, pages 219–245. 1997.
- [27] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
- [28] Paolo Ciaccia and Marco Patella. Searching in Metric Spaces with User-Defined and Approximate Distances. *ACM Transactions Database Systems*, 27(4):398–437, December 2002.
- [29] W. A. Burkhard and R. M. Keller. Some Approaches to Best-Match File Searching. *Communications of the ACM*, 16(4):230–236, 1973.

- [30] David A. White and Ramesh Jain. Similarity Indexing with the SS-tree. In *ICDE'96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.
- [31] D. Randall Wilson and Tony R. Martinez. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, 6:1–34, 1997.
- [32] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics - Doklady*, 10(8):707–710, February 1966.
- [33] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.
- [34] Flip Korn, Bernd-Uwe Pagel, and Christos Faloutsos. On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):96–111, January 2001.
- [35] Caetano Taina Jr., Agma J. M. Traina, C. Faloutsos, and B. Seeger. Fast Indexing and Visualization of Metric Datasets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(2):244–260, 2002.
- [36] Christian Digout. Metric Techniques for High-Dimensional Indexing. Technical Report TR 04–19, University of Alberta, Edmonton, Alberta, Canada, September 2004.
- [37] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for Processing K-Closest-Pair Queries in Spatial Databases. *Data Knowledge Engineering*, 49(1):67–104, April 2004.
- [38] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 189–200, Dallas, TX, USA, May 2000. ACM.
- [39] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-Trees Using the Concept of Fractal Dimension. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 4–13, 1994.
- [40] Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Stephane Bressan. Indexing the Edges – A Simple and yet Efficient Approach to High-Dimensional Indexing. In *PODS '00: Proceedings of the*

- Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 166–174, New York, NY, USA, May 2000. ACM Press.
- [41] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, USA, August 1996. Morgan Kaufmann Publishers.
- [42] Dennis Shasha and Tsong-Li Wang. New Techniques for Best-Match Retrieval. *ACM Transactions on Information Systems*, 8(2):140–158, 1990.
- [43] Roberto F. Santos Filho, Agma J. M. Traina, Jr. Caetano Traina, and Christos Faloutsos. Similarity Search without Tears: The OMNI Family of All-Purpose Access Methods. In *Proceedings of the 17th International Conference on Data Engineering*, pages 623–630, Washington, DC, USA, 2001. IEEE Computer Society.
- [44] Melliya Annamalai, Rajiv Chopra, and Samuel DeFazio. Indexing Images in Oracle8i. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 539–547. ACM, May 16-18 2000.
- [45] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB '98: Proceedings of the 24th International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA, August 1998. Morgan Kaufmann Publishers Inc.
- [46] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-Index: Distance Searching Index for Metric Data Sets. *Multimedia Tools and Applications*, 21(1):9–33, September 2003.
- [47] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.
- [48] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, second edition, 1994.
- [49] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. *AMC SIGMOD Record*, 14(2):1–8, June 1984.

- [50] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-Step Processing of Spatial Joins. *SIGMOD Conference*, 23(2):1997–208, May 1994.
- [51] M. G. Martynov. Spatial Joins and R-Trees. In *Proceedings of the Second International Workshop on Advances in Databases and Information Systems - ADBIS'95*, pages 160–164, Moscow, 1995. Phasis.
- [52] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
- [53] Christian Böhm and Florian Krebs. The k-Nearest Neighbor Join: Turbo Charging the KDD Process. *Knowledge Information Systems*, 6(6):728–749, January 2004.
- [54] Christian Böhm and Florian Krebs. Supporting KDD Applications by the K-Nearest Neighbor Join. In *Proceedings on the 14th International Conference Database and Expert Systems Applications*, Lecture Notes in Computer Science, pages 504–516, Prague, Czech Republic, September 2003. Springer.
- [55] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. *SIGMOD Record*, 30(2):379–388, June 2001.
- [56] Dmitri V. Kalashnikov and Sunil Prabhakar. Similarity Join for Low-and High-Dimensional Data. In *DASFAA'03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, pages 7–16, Washington, DC, USA, March 2003. IEEE Computer Society.
- [57] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An Efficient Method for KNN Join Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 756–767, Toronto, Canada, August 31/September 3 2004. Morgan Kaufmann.
- [58] Manli Zhu, Dimitris Papadias, Jun Zhang, and Dik Lun Lee. Top-k Spatial Joins. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):567–579, April 2005.
- [59] Christian Böhm and Hans-Peter Kriegel. A Cost Model and Index Architecture for the Similarity Join. In *Proceedings of the 17th International Conference on Data Engineering*, pages 411–420, Heidelberg, Germany, April 2001. IEEE Computer Society.
- [60] National Geospatial-Intelligence Agency. GEOnet Names Server. http://earth-info.nga.mil/gns/html/geonames_dd_dms_date_20050701.zip(Atualizada em 12/08/2005).

-
- [61] The Universal Protein Resource. TrEMBL Release 30.3, June. ftp://ftp.uniprot.org/pub/databases/uniprot/knowledgebase/uniprot_trembl.xml.gz.
- [62] Richard Holt. Open Office: Spell Checking Dictionaries. http://lingucomponent.openoffice.org/spell_dic.html(Atualizada em 13/06/2005).
- [63] Hal Berghel and David Roach. An Extension of Ukkonen's Enhanced Dynamic Programming ASM Algorithm. *ACM Transactions on Information Systems (TOIS)*, 14(1):94–106, January 1996.
- [64] Weijia Xu and Daniel P. Miranker. A Metric Model of Amino Acid Substitution. *Bioinformatics*, 20(8):1214–1221, May 2004.
- [65] Adel'son-Vel'skii G.M. and Landis E.M. An Algorithm for the Organization of Information. *Dokl. Akad. Nauk USSR*, 142(2):263–266, 1962.