PCA-tree: uma proposta para indexação multidimensional

Philipe Dalla Bernardina

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de Concentração : **Ciência da Computação**
Orientadora : **Prof. Dr. Nina S. T. Hirata**

- São Paulo, Maio de 2007 -

PCA-tree: uma proposta para indexação multidimensional

Este exemplar corresponde a versão final da dissertação devidamente corrigida e defendida por Philipe Dalla Bernardina e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Profa. Dra. Nina S. T. Hirata - IME-USP.
- Prof. Dr. João Eduardo Ferreira - IME-USP.
- Prof. Dr. Caetano Traina Junior - ICMC-USP.

Agradecimentos

A minha orientadora, Professora Dra. Nina S. T. Hirata, por sua grande ajuda na orientação, suporte e fortalecimento. Ela me ofereceu não só suporte técnico, tanto nos créditos quanto na pesquisa, como uma atitude rigorosa na pesquisa além do exemplo de vida acadêmica. Sinto-me afortunado por ter sido um orientando dela.

A todo o pessoal do laboratório de visão computacional (BIOINFO-IME), pela acolhida, abertura e disponibilidade em me ajudar. Principalmente ao colega David da Silva Pires que, como um dos administradores do Lab., sempre se mostrou prestativo no apoio técnico e acadêmico.

À Empresa Brasileira de Correios e Telégrafos que me privilegiou com uma suspensão temporária do contrato de trabalho demostrando interesse no incentivo ao aperfeiçoamento curricular de seus funcionários.

Ao professor Dr. João Eduardo Ferreira por me indicar para estágio em um projeto interno à Universidade que foi fundamental no suporte financeiro e para me manter próximo ao Instituto.

A minha mãe Maria Luiza e meu pai Ronaldo por terem atendido a todos os telefonemas e dado o suporte financeiro e emocional quando necessário.

Ao meu tio Ricardo, irmão Ronaldo e cunhada Hélena por serem meus únicos contatos familiares neste período temporário em São Paulo.

Resumo

Com o vislumbramento de aplicações que exigiam representações em espaços multidimensionais, surgiu a necessidade de desenvolvimento de métodos de acessos eficientes a estes dados representados em $\mathbb{R}^d$. Dentre as aplicações precursoras dos métodos de acessos multidimensionais, podemos citar os sistemas de geoprocessamento, aplicativos 3D e simuladores. Posteriormente, os métodos de acessos multidimensionais também apresentaram-se como uma importante ferramenta no projeto de classificadores, principalmente classificadores pelos vizinhos mais próximos. Com isso, expandiu-se o espaço de representação, que antes se limitava no máximo a quatro dimensões, para dimensionalidades superiores a mil. Dentre os vários métodos de acesso multidimensional existentes, destaca-se uma classe de métodos baseados em árvores balanceadas com representação em Rd. Estes métodos constituem evoluções da árvore de acesso unidimenisonal B-tree e herdam várias características deste último. Neste trabalho, apresentamos alguns métodos de acessos dessa classe de forma a ilustrar a idéia central destes algoritmos e propomos e implementamos um novo método de acesso, a PCA-tree. A PCA-tree utiliza uma heurística de quebra de nós baseada na extração da componente principal das amostras a serem divididas. Um hiperplano que possui essa componente principal como seu vetor normal é definido como o elemento que divide o espaço associado ao nó. A partir dessa idéia básica geramos uma estrutura de dados e algoritmos que utilizam gerenciamento de memória secundária como a B-tree. Finalmente, comparamos o desempenho da PCA-tree com o desempenho de alguns outros métodos de acesso da classe citada, e apresentamos os prós e contras deste novo método de acesso através de análise de resultados práticos.

Abstract

The advent of applications demanding the representation of objects in multi-dimensional spaces fostered the development of efficient multi-dimensional access methods. Among some early applications that required multi-dimensional access methods, we can cite geo-processing systems, 3D applications and simulators. Later on, multi-dimensional access methods also became important tools in the design of classifiers, mainly of those based on nearest neighbors technique. Consequently, the dimensionality of the spaces has increased, from earlier at most four to dimensionality larger than a thousand. Among several multi-dimensional access methods, the class of approaches based on balanced tree structures with data represented in $\mathbb{R}^d$ has received a lot of attention. These methods constitute evolues from the B-tree for unidimensional accesses, and inherit several of its characteristics. In this work, we present some of the access methods based on balanced trees in order to illustrate the central idea of these algorithms, and we propose and implement a new multi-dimensional access method, which we call PCA-tree. It uses an heuristic to break nodes based on the principal component of the sample to be divided. A hyperplane, whose normal is the principal component, is defined as the one that will split the space represented by the node. From this basic idea we define the data structure and the algorithms for the PCA-tree employing secondary memory management, as in B-trees. Finally, we compare the performance of the PCA-tree with the performance of other methods in the cited class, and present advantages and disadvantages of the proposed access method through analysis of experimental results.

Sumário

1 Introdução **1**

1.1 Contextualização 1

1.2 Proposta 2

1.3 Contribuições 3

1.4 Organização do Texto 3

2 Conceitos Básicos **5**

2.1 Indexação 5

2.1.1 Chave de acesso e Dados Satélite 7

2.1.2 Memória Secundária 7

2.1.3 Cache e Política de Gerenciamento 8

2.1.4 Tipos de Consultas 9

2.1.5 Tipos de Operações 10

2.1.6 Medidas de Desempenho 11

2.1.7 Avaliação de Desempenho 11

2.2 Análise de Componente Principal 12

2.3 Geometria Analítica 13

2.3.1 Centróide 14

2.3.2 Reta em $\mathbb{R}^d$ 14

2.3.3 Hiperplano em $\mathbb{R}^d$ 15

2.3.4 Hipercubo em $\mathbb{R}^d$ 15

2.3.5 Superelipse em $\mathbb{R}^d$ 15

2.3.6 Intersecção entre dois Hipercubos 17

2.3.7 Intersecção entre Hipercubo e Hiperplano 18

2.3.8 Intersecção entre Superelipse e Hiperplano 18
2.3.9 Intersecção entre Superelipse e Reta 19

3 Árvores B-tree e B+-tree 22
3.1 B-tree 22
3.1.1 Estrutura de Dados da B-tree 23
3.1.2 Algoritmo de busca da B-Tree 23
3.1.3 Algoritmo de Inserção da B-Tree 24
3.1.4 Algoritmo de exclusão da B-Tree 27
3.2 Árvore Balanceada B+-tree 32
3.2.1 Estrutura de Dados da B+-tree 32
3.2.2 Algoritmos de Busca, Inserção e Exclusão da B+-tree 33

4 KDB-tree e outros índices multidimensionais 37
4.1 KDB-Tree 37
4.1.1 Consulta em uma KDB-Tree 39
4.1.2 Algoritmo de Inserção 40
4.1.3 Algoritmo de Quebra 41
4.1.4 Algoritmo de Quebra de uma Folha 42
4.1.5 Algoritmo de Quebra de um Nó Interno 43
4.1.6 Algoritmo de quebra condicionada a um hiperplano 44
4.1.7 Algoritmo de Exclusão em uma KDB-Tree 46
4.2 R-Tree 47
4.2.1 Algoritmo de Busca da R-Tree 50
4.2.2 Algoritmo de Inserção da R-Tree 50
4.2.3 Algoritmo de Exclusão da R-Tree 50
4.2.4 Algoritmo de Quebra da R-tree 51
4.3 R+-Tree 53
4.4 X-Tree 53
4.4.1 Estrutura de Dados da X-Tree 54

5 PCA-Tree 55

5.1 Estrutura de dados 56
5.2 Propriedades 58
5.3 Busca Exata 58
5.4 Range-Query 59
5.5 KNN-Query 60
5.6 Algoritmo de Inserção 61
5.7 Algoritmo de Quebra de página folha 62
5.8 Algoritmo de Quebra de página interna 62
5.8.1 Algoritmo de quebra condicionada de nó 64
5.9 Algoritmos de Exclusão 64
5.10 Algoritmos de mesclagem de páginas folha 65
5.11 Algoritmos de mesclagem de páginas internas 65
5.12 PCA-RTree: uma extensão da PCA-Tree 66

6 Implementações 67
6.1 Índices e Estruturas de Dados 67
6.2 Interfaces Gráficas 68
6.2.1 Interface KDB-tree e R-tree 68
6.2.2 Interface PCA-tree 69
6.3 Interface exemplo para PCA-tree de Alta Dimensionalidade 70

7 Resultados 74
7.1 Dados Uniformemente Distribuídos 74
7.2 Dados com distribuição não uniforme 76
7.3 Análise dos resultados 78

8 Conclusão 79

Lista de Figuras

2.1 Árvore Balanceada(1) versus Árvore Degenerada(2). 6

2.2 Exemplo de aplicação do método PCA 13

2.3 Centróide de um triângulo. 14

2.4 Centróide de um conjunto de amostras. 14

2.5 Elipse(1), hiperelipse(2) e hipoelipse(3) rotacionadas. 17

3.1 Fluxograma do algoritmo de busca em uma B-tree. 25

3.2 Fluxograma do algoritmo de inserção na B-tree. 26

3.3 Exemplo de **quebra** na B-tree. 26

3.4 Fluxograma do algoritmo de exclusão. 28

3.5 Comportamento estrutural da B-tree à exclusão(caso 1). 30

3.6 Comportamento estrutural da B-tree à exclusão(caso 2). 30

3.7 Comportamento estrutural da B-tree à exclusão(caso 3). 31

3.8 Comportamento estrutural da B-tree à exclusão(caso 4). 31

3.9 Estrutura da B+-tree. 33

4.1 Exemplo de segmentação feita pela KDB-tree. 40

4.2 Ilustração de uma quebra de uma folha que resulta em nós com tamanhos significativamente diferentes. 42

4.3 Representação da estrutura antes da quebra condicionada. 45

4.4 Representação da estrutura depois do quebra condicionada(b). 45

4.5 Representação estrutural da quebra condicionada, antes e depois. 46

4.6 KDB-tree antes da exclusão com mesclagem. 47

4.7 KDB-tree depois da exclusão com mesclagem. 48

4.8 Exemplo 2 de segmentação da KDB-tree em um espaço em $\mathbb{R}^2$ 48

4.9 Exemplo de segmentação da R-tree 49
4.10 Exemplo de quebra da R+-tree 54

5.1 Exemplo da distribuição espacial de uma PCA-Tree. 56
5.2 Exemplo da estrutura de uma PCA-Tree. 57
5.3 Resultado Estrutural da quebra na PCA-tree 64

6.1 Interface gráfica para a KDB-Tree. 69
6.2 Interface gráfica para a R-Tree. 70
6.3 Interface gráfica para a PCA-Tree. 70
6.4 *Tabsheet* Imagem. 71
6.5 *Tabsheet* Máscara. 72
6.6 *Tabsheet* Máscara Projetada. 72
6.7 *Tabsheet* Range Query View. 73
6.8 *Tabsheet* Knn Query View. 73

7.1 Tempo (ms) para 10000 inserções de pontos uniformemente distribuídos em $\mathbb{R}^d$. 75
7.2 Comparação para *range queries* sobre pontos uniformemente distribuídos. 75
7.3 Comparação para *knn queries* sobre pontos uniformemente distribuídos. 76
7.4 Comparação para *knn queries*. 76
7.5 Imagem utilizada para extração de amostras. 77
7.6 Comparação para *inserções* sobre dados não uniformemente distribuídos. 77
7.7 Comparação para *range queries* sobre dados não uniformemente distribuídos. 77
7.8 Comparação para *knn queries* sobre dados não uniformemente distribuídos. 78

Capítulo 1

Introdução

1.1 Contextualização

Grandes volumes de dados são armazenados em bases de dados (BDs) e gerenciados por Sistemas de Gerenciamento de Banco de Dados(SGBDs). Esses sistemas permitem operações eficientes de busca, inserção e remoção de dados. A eficiência dessas operações é resultado do uso de mecanismos baseados em estruturas de dados e algoritmos de manipulação adequados, denominados índices. Similar aos índices de um livro, a idéia desses índices é que os dados de interesse possam ser operados rapidamente, sem necessidade de busca extensiva sobre todo o conjunto de dados.

Os objetos armazenados em BDs são em geral registros compostos por diferentes campos. Nos SGBDs tradicionais, as chaves (campos) usados para indexação são em geral numéricos e textuais. Os índices B-tree e seus derivados são, provavelmente, os mais utilizados em SGBDs tradicionais [3].

A realidade recente, em termos de avanços tecnológicos, vem impondo cada vez mais a necessidade de indexação de objetos mais complexos como som, imagem, video, séries temporais, etc. Os campos desses dados não são simplesmente numéricos ou textuais. Dentre estas aplicações podemos destacar: Recuperação de Imagens [35, 22, 25, 13, 30, 31, 7], Sistemas de Geoprocessamento [34], Bancos de Dados e Mineração de Dados [17], Classificadores pelos k vizinhos mais próximos [17, 23, 1, 14], etc. Essa realidade motivou o desenvolvimentos dos chamados índices multidimensionais. Enquanto a B-tree é baseada em chaves numéricas (valor em $\mathbb{R}$), no caso de indexação multidimensional, a chave é um ponto em $\mathbb{R}^d$.

Existem diferentes abordagens para indexação multidimensional. Dentre elas destacam-se as baseadas em

- Estruturas do tipo árvore,
- Grid,

- mapeamentos de $\mathbb{R}^d$ em $\mathbb{R}$.

As baseadas em estruturas do tipo árvore derivam-se da B-tree, que por sua vez é derivada das árvores de busca binária. As baseadas em grid subdividem o espaço $\mathbb{R}^d$ em z^d hipercubos adjacentes, de lados de tamanho igual, de mesmo volume. Finalmente, os baseados em mapeamento de $\mathbb{R}^d$ em $\mathbb{R}$ consistem em técnicas que usam uma função para mapear pontos de $\mathbb{R}^d$ em $\mathbb{R}$ e em seguida as B-trees ou outra forma de acesso unidimensional para indexá-los.

Dentre as técnicas baseadas em estruturas de árvore, a KDB-tree é a pioneira, tendo sido proposta em 1981 por Robinson [27]. A KDB-tree exige um grande volume de dados para realizar uma indexação eficiente, mais precisamente um valor maior que m^d pontos, onde m é o valor máximo de entradas em um nó da KDB-tree. O valor m geralmente é maior que 50. Para baixa dimensionalidade, a KDB-tree mostrou boa performance, entretanto, em alta dimensionalidade o valor de m^d pontos exigidos não é atendido por bases de dados convencionais. Tais dificuldades motivaram o surgimento de inúmeras outras técnicas de indexação. Dentre elas podemos citar: R-tree [16], R+-tree [29], R*-tree [4], X-tree [5], Hilbert R-tree [20], SS-tree[36], SR-tree [21], TV-tree [24], SF-tree [11], QSF-tree [37], M-tree [10], entre outros. Alguns artigos apresentam uma revisão de várias dessas variantes [15, 8].

Uma idéia central, tanto da KDB-tree como de outros índices baseados em árvore, consistem em promover subdivisões hierárquicas no espaço de busca. No caso da KDB-tree, essas subdivisões são necessariamente especificadas por um hiperplano ortogonal a um dos eixos em $\mathbb{R}^d$. Além disso, a árvore é construída de tal forma que, em todos os nós de um mesmo nível, o eixo em questão é o mesmo. Dessa forma, em muitos casos, nem todas as dimensões do espaço são levadas em consideração na construção da árvore. Sabe-se também que o desempenho da KDB-tree degrada para dimensões superiores a 8.

1.2 Proposta

Para tratar essa questão sobre o uso de todas as dimensões na construção do índice, propomos uma técnica que denominaremos PCA-tree. A idéia central é o uso de hiperplanos arbitrários, não necessariamente ortogonais a um dos eixos, para fazer a subdivisão hierárquica do espaço. A orientação do vetor normal a este hiperplano é determinada a partir do cálculo da componente principal sobre os pontos que pertencem ao subespaço a ser dividido. O prefixo PCA do nome PCA-tree, utilizado para denominar a proposta, refere-se ao método de extração de componente principal utilizado na quebra de páginas internas.

Nos índices convencionais, as entradas dos nós internos definem o subespaço que engloba as entradas dos nós filhos e não tem nenhuma relação com as entradas dos nós irmãos. Já na nova proposta, as entradas internas formam uma árvore binária de navegação. Esta nova estruturação exigiu algoritmos que não possuem semelhança com os

algoritmos de índices convencionais, entretanto, a complexidade computacional dos mesmos manteve-se. Esta estrutura interna em árvore binária é fundamental para que a utilização de hiperplanos arbitrários não ortogonais seja possível.

1.3 Contribuições

Como contribuições deste trabalho destacamos:

- Elaboração da PCA-tree (estrutura de dados, algoritmos de busca, inserção e remoção);
- Implementação dos seguintes algoritmos para a realização de estudos comparativos de desempenho:
  - KDB-tree
  - R-tree
  - PCA-tree
- Software Didático:

  Criou-se também uma interface gráfica que ilustra as diferentes operações para os três algoritmos implementados, para operações em $\mathbb{R}^2$. Este software pode ser usado para propósitos didáticos. Uma vez que os algoritmos são complexos de serem descritos e entendidos, a visualização de certos efeitos na subdivisão dos espaços de busca ajuda o processo de entendimento desses índices.

1.4 Organização do Texto

O texto foi organizado de forma a apresentar ao leitor alguns algoritmos de indexação unidimensional e multidimensional além da nova abordagem PCA-tree proposta. Apresentamos no segundo capítulo, de forma resumida, alguns conceitos básicos que são exigidos do leitor para entender os demais capítulos. Dentre estes conceitos básicos temos os conceitos de indexação, geometria espacial para manipulação de objetos em $\mathbb{R}^d$ e a apresentação do método PCA que é utilizado como parte do algoritmo de quebra da PCA-tree. No terceiro capítulo, apresentamos os índices unidimensionais B-tree e B+-tree que foram os precursores dos índices multidimensionais. Eles introduziram uma abordagem de gerenciamento de memória secundária e balanceamento da árvore que foi herdada pela maioria dos demais métodos de acesso multidimensionais. No quarto capítulo, apresentamos detalhes de alguns índices multidimensionais. A importância desse capítulo encontra-se no fato de permitir a compreensão da evolução dos índices unidimensionais para os índices multidimensionais. Estes índices também serão utilizados para comparação de resultados com a PCA-tree. No quinto capítulo, apresentamos a abordagem PCA-tree, sua estrutura

de dados e todos os algoritmos envolvidos. No sexto capítulo, apresentamos algumas implementações realizadas durante a pesquisa, incluindo interfaces gráficas didáticas para a KDB-tree, R-tree e PCA-tree em $\mathbb{R}^2$. No sétimo capítulo, apresentamos os resultados obtidos, utilizando algumas medidas de desempenho apresentadas na seção 2.1.6. Finalmente, na conclusão, resumimos os resultados e as melhorias que podem vir a ser realizadas.

Capítulo 2

Conceitos Básicos

Apresentamos neste capítulo alguns conceitos básicos pelos quais o leitor deverá estar ciente para uma boa leitura do trabalho. Não será dada grande ênfase na explicação destes conceitos, entretanto, será feita uma referência caso o leitor queira maiores detalhes sobre o assunto. O assunto do trabalho, Algoritmos de Indexação Multi-Dimensional, exige do leitor um certo conhecimento sobre as estruturas de dados utilizadas em algoritmos de indexação em geral. Também exige-se conhecimentos em Geometria Analítica e Álgebra Linear para lidar com representações de objetos no espaço em $\mathbb{R}^d$. Dentre estes objetos podemos destacar hipercubos, hiperesferas, poliedros, etc.

2.1 Indexação

A indexação é um termo utilizado em SGBD's como forma de representar os algoritmos e estruturas de dados para a otimização de consultas realizadas aos mesmos. Conceitualmente, os índices em SGBD's têm a mesma funcionalidade dos índices presentes em livros e outros tipos de documentos, como o próprio sumário desta monografia. Estruturalmente, os índices podem ser tabelas HASH, árvores binárias, índices binários e outros tipos de estruturas de dados que agilizem uma consulta. A estrutura de dados de indexação mais utilizada, e também a que é considerada nesta monografia, é a árvore balanceada com suporte a algoritmos de armazenamento e gerenciamento de memória secundária. Não entraremos em detalhes sobre outras estruturas de indexação que não sejam derivadas destas árvores balanceadas.

As árvores são estruturas de dados que organizam os principais tipos de índices. Em teoria dos grafos, uma árvore é um grafo conexo, ou seja, existe caminho entre quaisquer dois de seus vértices, e acíclico (não possui ciclos). Toda árvore é formada por nós que, em teoria dos grafos, correspondem aos vértices. Cada nó na árvore, exceto um nó especial chamado de **raiz** da árvore, tem um nó **pai**. Todos os nós da árvore podem possuir nenhum ou vários nós **filho**. Dois nós distintos que possuam o mesmo nó pai são chamados de **irmãos**. O nó raiz não possui pai. Um nó que não possui filhos é chamado de **folha** da

árvore. Um nó que não é folha da árvore é chamado de **nó interno**. O **nível** de um nó é sempre um nível superior ao nível do pai, ou seja, caso um nó possua pai que está localizado no nível 5, este mesmo nó está localizado no nível 6. A raiz da árvore possui nível 0. A **altura** de uma árvore é um valor igual do nível máximo da árvore somado de 1. O conjunto dos **descendentes** de um nó T é composto por todo nó da árvore cujo conjunto dos nós pertencentes ao caminho entre este nó e a raiz contenha T. Os **ancestrais** de um nó T são os nós pertencentes ao caminho entre este nó e a raiz, exceto o próprio nó T. Uma **sub-árvore** de um nó é composta do nó em si e de todos os seus descendentes.

As árvores binárias são estruturas de dados que dão suporte para que algoritmos de busca sejam executados em ordem de crescimento logarítmico, no caso médio. Entretato, as árvores binárias não previam um controle de balanceamento da árvore, o que, eventualmente, resultava em árvores ou ramos degenerados. Uma árvore degenerada, do tipo exemplificado no item 2 da figura 2.1, resulta em buscas de ordem de crescimento linear, ou seja, a complexidade dos algoritmos de busca em árvores binárias é de ordem linear no pior caso. No exemplo da figura, a árvore degenerada possui 7 nós e sua altura é igual a 7. Uma busca em uma árvore perfeitamente balanceada, como a exemplificada no item 1 da figura 2.1, é de ordem $1 + \lfloor lg(n) \rfloor$. No exemplo da figura, a árvore possui 7 nós e sua altura é igual a 3, ou seja, $3 = 1 + \lfloor lg(7) \rfloor$. Com isto, fica claro a necessidade de garantir um balanceamento na árvore. Algumas implementações de controle de balanceamento foram propostos para árvores binárias - como as árvores AVL [12], árvore vermelho e preto [12] - solucionando o problemas de árvores degeneradas. Entretato, estas árvores eram mantidas na memória principal sem o gerenciamento de acesso e armazenamento na memória secundária. O gerenciamento de memória secundária proposta pela B-tree foi fundamental para o uso de árvores balanceadas como métodos de acesso em bancos de dados.

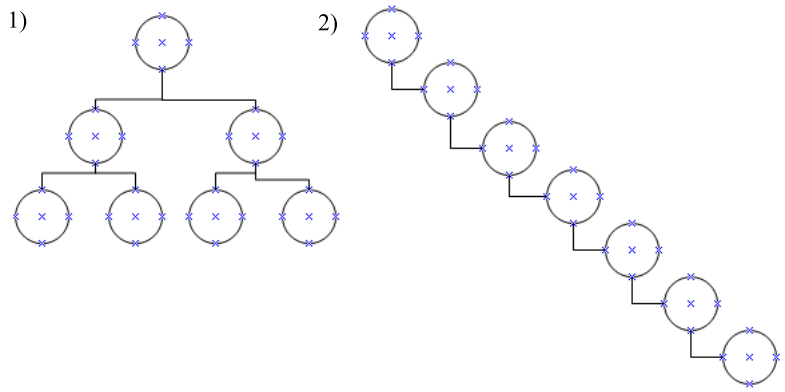

Figura 2.1: Árvore Balanceada(1) versus Árvore Degenerada(2).

Nesta seção apresenta-se alguns conceitos dos algoritmos de árvores binárias e B-trees, além dos tipos de consulta realizadas nestes índices. A seção 2.1.1 introdu o conceito de chave de acesso e dados satélite que são termos frequentemente utilizados no texto. Na seção 2.1.2 introduz-se o funcionamento da memória secundária e a importância da pagi-

nação desta. Na seção 2.1.3 introduz-se conceitos de políticas de gerenciamento aplicadas sobre os índices pelos SGBD's para melhoria de desempenho. Na seção 2.1.4 introduz-se os tipos de consultas que são disponibilizados aos usuários pelos SGBD's. Na seção 2.1.5 apresenta-se os tipos de operações realizadas sobre as tabelas e, consequentemente, sobre os respectivos índices. E, finalmente, nas seções 2.1.6 e 2.1.7 introduz-se alguns conceitos para análise de desempenho de índices e SGBD's. Não será detalhada a modelagem de dados e sim os conceitos voltados às estruturas de dados e que influam no desempenho dos algoritmos sobre o índice.

2.1.1 Chave de acesso e Dados Satélite

Em índices, as informações são hierarquicamente agrupadas pela informação contida na chave de acesso aos dados. Um registro em uma tabela pode conter várias informações como, por exemplo, em uma tabela de pessoal temos: matrícula, nome, sobrenome, idade, endereço, etc. Entretanto, quando vamos indexar os dados desta tabela, temos que escolher um subconjunto de campos que formarão a chave de acesso aos dados. Por exemplo, o número de matrícula poderia ser a chave de acesso de um índice I1 de forma que toda consulta por número de matrícula aos dados utilizariam este índice como meio de acesso. O nome poderia ser a chave de acesso de um segundo índice I2 e, este, seria utilizado como meio de acesso aos dados por buscas por nome. Quando a chave de acesso é composta por apenas um campo chamamos o índice de unidimensional e quando esta é composta por dois ou mais campos denominamos o índice de multi-dimensional. No caso de árvores de indexação, uma **entrada** de um nó é a estrutura de dados básica que forma o nó. O nó é composto por várias entradas. Para cada entrada do nó, existe apenas uma **chave de acesso** espacial à entrada. O valor desta chave de acesso determina a posição espacial da entrada. Estas definições ajudarão no entendimento das estruturas de dados e algoritmos a serem apresentados.

Além da chave de acesso aos dados, o banco de dados também armazena as informações satélites dos dados. Em relação à um índice, chamamos de informação satélite o subconjunto dos campos da tabela que não fazem parte da chave de acesso. Quando a informação satélite é armazenada pelo índice chamamos este índice de *clustered*. Quando a informação satélite é armazenada em estrutura de dados externas ao índice denominamos este índice de *unclustered*. Sendo assim, uma tabela, mesmo podendo ter vários índices de acesso aos dados, só poderá ter um índice *clustered*. Caso contrário, haveria informação satélite duplicada em diferente índices, o que exigiria a necessidade de políticas de replicação de dados satélites entre índices resultando em um mau aproveitamento de espaço.

2.1.2 Memória Secundária

Existem várias tecnologias que provêem capacidade de armazenamento de dados em sistemas de computação (memória) [12]. A memória primária (ou memória principal) de

um computador normalmente consiste de *chips* de silício de memória. Esta tecnologia tem, normalmente, um custo financeiro por bit armazenado com ordem de magnitude duas vezes maior que tecnologia de discos ou fitas magnéticas. A maioria dos sistemas de computadores também possuem uma memória secundária baseada em discos magnéticos; a quantidade de espaço de armazenamento na memória secundária, normalmente, excede em pelo menos duas ordens de magnitude o tamanho da memória principal.

Os discos magnéticos e o *hardware* envolvido utilizam-se de partes mecânicas para a leitura da mídia. Estes discos são organizados em cilindros constituídos de pratos magnetizáveis. Os pratos são divididos em trilhas magnéticas. A leitura da informação magnética contida nos pratos é feita através de braços e estes possuem cada um sua respectiva cabeça de leitura e escrita. Para melhor aproveitar esta estrutura física, e evitar movimentos desnecessários das partes físicas da memória secundária, os discos não são lidos bit a bit, ou byte a byte, e sim lidos e escritos página a página. Para discos magnéticos encontramos páginas de tamanho na ordem de 2^{11} a 2^{14} *bytes*. A partir do posicionamento do braço e da cabeça do disco magnético, as operações de leitura e escrita de página são feitas eletrônicamente sem mais nenhum movimento físico (a não ser a rotação constante dos pratos do cilindro). Nesta descrição da estrutura física dos discos magnéticos fica claro o motivo do baixo desempenho das operações de leitura e escrita em comparação a leitura feita em *chips* de silício.

As páginas descritas no parágrafo anterior são diretamente relacionadas ao sistema de paginação utilizados pelos índices B-tree (que serão detalhados mais a frente). Assim, ao fazer-se uma leitura de uma página do índice também otimiza-se a leitura da memória secundária para esta página. Portanto, as páginas em índices com gerenciamento de memória secundária devem ser projetadas com valor de tamanho múltiplo ao valor de tamanho das páginas do disco magnético.

2.1.3 Cache e Política de Gerenciamento

O termo página é utilizado no contexto do algoritmo B-tree como sendo um espaço físico contínuo na memória secundária. Quando uma página é carregada para a memória principal, o leitor do disco da memória secundária faz uma leitura contínua de toda a página. A relação entre uma página P de uma B-tree e uma página P' de disco físico apresentado na seção 2.1.2 é tal que P seja um valor múltiplo de P', geralmente com uma relação $\frac{P}{P'} = 1$, mas outros valores podem mostrar melhor desempenho do sistema. A página da B-tree não pode ser grande demais a ponto de sua leitura ser demorada e nem pequena demais a ponto de o número de leituras descontínuas à memória secundária ser grande. Estes valores variam muito e dependem do tipo de memória secundária utilizada e da quantidade e tipos de consultas realizadas sobre o índice. Usualmente, as páginas tem tamanho com valor girando em torno de 8k bytes, o que, em B-trees, representam cerca de 100 entradas.

Os índices são armazenados em páginas presentes na memória secundária. A página raiz do índice é a primeira página carregada para a memória quando uma consulta é

feita ao mesmo. As demais páginas são carregadas na medida em que a consulta exige a leitura das mesmas. Após o término da consulta, a memória primária utilizada para o armazenamento das páginas é liberada para que outros processos possam utilizá-la. Desta forma, quando é realizada uma seqüência de consultas ao índice, ou quando o número de consultas simultâneas ao índice é em demasiado grande, é interessante manter algumas ou todas as páginas na memória principal para evitar o carregamento e descarregamento desnecessário das mesmas. Este espaço onde mantêm-se as páginas na memória principal é chamado de *cache* e a decisão de quais páginas devem permanecer na memória ou não é chamado de política de gerenciamento.

Não entraremos em maiores detalhes sobre a política de gerenciamento e da forma de utilização do *cache* de memória, já que estes são independentes dos algoritmos de busca, inserção e exclusão dos índices que são o foco deste documento.

2.1.4 Tipos de Consultas

Para atender as necessidades do usuário de consulta a uma base de dados são definidos alguns tipos de consulta. Estes devem aproveitar a estrutura de dados do índice de forma a agilizar a obtenção do resultado. Dentre estas podemos citar:

- Busca Pontual ou de Casamento Exato

  Em um índice unidimensional, por exemplo, aplicados a uma tabela de pessoal, um usuário pode obter informações de um funcionário da empresa através de sua matrícula. Este tipo de consulta retorna apenas um registro e é denominado consulta pontual (do inglês, *point query*) ou consulta de casamento exato (do inglês, *exact match query*). Para exemplificar este tipo de consulta temos a seguinte situação. Foi detectada uma venda de valor muito acima do normal em um caixa de uma determinada loja de departamentos. Como medida de segurança deseja-se saber detalhes pessoais do cliente e da forma como foi atendido. Para isso, deseja-se saber qual funcionário estava atendendo no caixa em questão na data/horário em que a venda foi realizada. Assim, informa-se na consulta, o número do caixa e a data/horário da venda. O número do caixa e a data/horário da venda formam um valor pontual em um espaço bi-dimensional.

- Busca em Extensão

  Em situações reais, na maioria das vezes, exige-se mais que uma consulta pontual. Análises de dados geralmente lidam com faixa de valores ao invés de valores pontuais. Estas faixas implicam em uma margem de erro aceitável ao resultado o que, em dados de natureza inexata, dão maior credibilidade ao resultado que uma busca pontual. Por exemplo, podemos citar uma busca em uma base de imagens por uma imagem que apresente uma determinada cor próxima à cor da pele humana. A variedade nas cores de pele humana é relativamente grande. Para tanto é mais adequado trabalhar com faixas de valores de erro aceitável em torno de uma cor

específica. Outro exemplo seria em aplicações de visualização de mapas. Nestes o usuário visualiza uma região do mapa ao invés de um ponto do mapa. Desta forma, a busca no índice é feita utilizando-se uma região como parâmetro.

Este tipo de consulta é denominado **busca por extensão** [9] (do inglês, *Range Query*) e **janela de busca**(do inglês,*window query*) para o caso particular onde o subespaço de busca é representado por um hipercubo. Podemos defini-lo mais formalmente como: para cada dimensão i da chave de acesso, onde $1 \leq i \leq d$, existe um raio de busca, ou seja, um valor aceitável de erro para os objetos a serem retornados; a partir daí, todos os objetos que fazem intersecção com o subespaço definido por este intervalo em $\mathbb{R}^d$ serão retornados pela busca. Também podemos implementar outros tipos de *range-query* onde a região de consulta ao hiperespaço pode ser uma superelipse, hiperelipse, hipercubo, ou alguma outra representação para regiões. Entretanto, dependendo da estrutura do índice e da complexidade de representação desta região em uma estrutura de dados, algumas destas representações podem resultar em algoritmos de baixo desempenho.

- K vizinhos mais próximos

  Outro tipo de busca é denominado de **K vizinhos mais próximos** (KNN,do inglês *K Nearest Neighbors*) [9] [1] [23] em $\mathbb{R}^d$. Esta busca é interessante quando um usuário procura por um padrão em um banco de dados. A consulta retorna os K vizinhos mais próximos, segundo algum critério de similaridade, e o usuário seleciona aquele que se adequar melhor à sua necessidade. Usando o exemplo do item anterior, ao invés do usuário informar uma faixa de valores de cor de pele, este informa uma cor de pele pontual e solicita os K objetos cuja distância da cor presente nestes objetos e a cor pontual informada são as K menores em toda a base de dados.

2.1.5 Tipos de Operações

Uma característica importante em banco de dados é manter a dinâmica de seus dados, ou seja, permitir que novos dados sejam inseridos e dados existentes sejam alterados ou excluídos. Estas operações, além de alterarem os dados, também implicam em uma reestruturação dos índices envolvidos. Uma inserção, por exemplo, pode ocasionar um estouro de tamanho de uma página do índice causando a criação de uma nova página. Uma exclusão pode causar a mesclagem de duas páginas causando o desalocamento de uma página da memória secundária. E uma alteração pode causar a mudança de posição de uma entrada no índice.

Muitas vezes é necessária uma alteração grande em uma base de dados como a exclusão de todos os registros do ano anterior ocasionado pela abertura de um novo ano e pela necessidade de se manter uma base de dados mais "leve". Estes tipos de operação podem envolver algoritmos específicos e, muitas vezes, até envolvem recriação total dos índices.

2.1.6 Medidas de Desempenho

Utilizaremos neste trabalho algumas medidas de desempenho de índices para avaliar a nova estrutura proposta. Dentre estas medidas de desempenho podemos citar:

- Page Reads

  Número de páginas lidas pelo algoritmo. Quando há necessidade de acesso à informação contida em uma página, esta é lida da memória secundária para a memória principal.

- Page Writes

  Número de páginas salvas na memória secundária. Ou seja, quando há a criação ou a alteração da informação contida em uma página há a necessidade de salvar estas informações na memória secundária.

- Número de Splits

  Um *split* é um algoritmo que é executado quando há o estouro de uma página. Seja $P1$ a página sobrecarregada, este algoritmo envolve a criação de uma nova página $P2$ além da divisão das entradas de $P1$ entre $P1$ e $P2$. Desta forma, a diminuição do número de *splits* realizados resulta na diminuição de alocação de páginas na memória secundária além da diminuição no número de escritas de páginas.

As medidas de desempenho utilizadas são restritas às que envolvem a escrita ou leitura à memória secundária. Isso porque o acesso à memória principal é muito mais rápido que o acesso à memória secundária.

2.1.7 Avaliação de Desempenho

Muito embora as medidas de desempenho mostrem valores quantitativos a respeito da melhoria de desempenho do índice, a avaliação de desempenho também envolve uma avaliação qualitativa do desempenho do sistema como um todo, e não mais focando no desempenho do índice em si. Para isso, alguns fatores que também devem ser levados em consideração são apresentados abaixo.

- relacionados aos dados

  tipo de dado, distribuição espacial dos dados, volume (quantidade) e escalabilidade, tamanho e formato de objetos espaciais de dimensão não-zero, grau de sobreposição entre objetos espaciais de dimensão não-zero e ordem de inserção dos dados.

- relacionados à carga de trabalho

  tipo de consulta (ex.: nearest neighbour query), características associadas a parâmetros de consulta (tais como tamanho, formato e distribuição espacial dos

retângulos relativos às janelas de consulta de *range queries* ou distribuição espacial dos pontos base de consultas do tipo *point query*), dinâmica dos dados (operações de inserção, remoção e modificação) e seletividade dos dados.

- relacionados ao gerenciamento de cache

  política de gerenciamento, tamanho total do *cache* e tamanho da página de disco. A variação deste último fator conduz, para um mesmo SAM, a diferentes agrupamentos de objetos. Já os demais fatores determinam quais e quantos objetos estarão presentes em memória principal, requisitos fundamentais para otimizar os acessos a disco.

- relacionados a parâmetros da estrutura

  a variação de alguns parâmetros, tal como o número mínimo de entradas por nó, permite um ajuste diferenciado na estrutura de dados e por conseguinte um melhor ou pior desempenho.

2.2 Análise de Componente Principal

A Análise das Componentes Principais (PCA, do inglês *Principal Component Analysis*) é um método utilizado no reconhecimento de padrões em dados, e expressa os dados de maneira a enfatizar as suas similaridades e diferenças. Dado que o reconhecimento de padrões em bases de dados é um problema difícil em espaços de alta dimensionalidade, onde a visualização gráfica e intuitiva dos dados não está disponível, o PCA é uma ferramenta poderosa para a análise destes dados. No PCA, inicialmente é feito o cálculo da matriz de covariância das amostras. A partir do cálculo da matriz de covariância, é extraído a matriz de autovalores e de autovetores ordenados de forma crescente por autovalor correspondente. A componente principal é o autovetor com maior autovalor correspondente e esta tem por significado ser o vetor de direção cujo valor da correlação das amostras projetadas sobre este ser maximizado.

O método PCA pode ser entendido como uma rotação dos eixos de coordenadas de modo a minimizar o erro quadrático médio das amostras em relação às componentes principais. Em termos intuitivos, imagine-se um objeto de forma cúbica disposta em um espaço tridimensional. A posição/rotação do cubo de modo a aumentar a área de visualização é aquela onde um dos vértices fica no centro da imagem, o vértice oposto a este vértice fica escondido, e os outros 6 vértices do cubo ficam dispostos de maneira a formar um hexágono regular em volta do primeiro vértice. Nesta posição temos uma maximização da área de visualização do cubo, permitindo-se ver 3 faces do mesmo. Entretanto, quando lidamos com amostras, e não objetos, ou seja, pontos descontínuos, e estes estão dispostos em um espaço multi-dimensional, temos o PCA como metodologia utilizada para o propósito de encontrar esta melhor visualização. Na figura 2.2 temos um exemplo da aplicação do método PCA em um conjunto de amostras dispostas em um espaço bidimensional. No primeiro gráfico temos um conjunto de amostras em $\mathbb{R}^2$. No

segundo gráfico temos a representação da transformação pelo método PCA onde o centro do gráfico foi transladado para o centro das amostras e as linhas tracejadas diagonais representam os autovetores resultantes.

Apesar do cálculo da componente principal ser de vital importância na PCA-tree proposta, não entraremos em detalhes sobre o assunto. Para a implementação utilizamos a biblioteca JAMA (do inglês *Java Matrix Package*) que já disponibiliza as rotinas necessárias para a aplicação do método. Uma boa referência sobre a biblioteca JAMA pode ser encontrada em

$$http://math.nist.gov/javanumerics/jama/,$$

enquanto uma boa referência ao método PCA pode ser encontrado em [33].

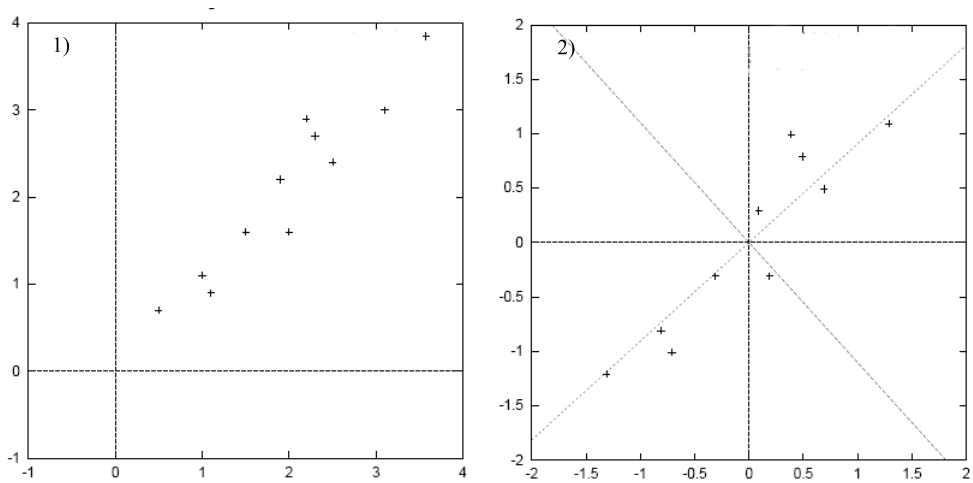

Figura 2.2: Exemplo de aplicação do método PCA

Neste trabalho o PCA é utilizado na escolha do vetor normal ao hiperplano que melhor divide as amostras em dois subespaços. Ou seja, esse método é utilizado pelo algoritmo de quebra de nó que será detalhado mais a frente.

2.3 Geometria Analítica

Apresentamos nesta seção a representação de objetos em $\mathbb{R}^d$ na forma de equações matemáticas. Também apresentamos algumas equações de relação entre estes objetos como, por exemplo, cálculo da intersecção entre objetos. A noção destas representações será importante para entender a estrutura de dados proposta no trabalho além dos algoritmos envolvidos.

2.3.1 Centróide

Em geometria, o centróide, ou centro de massa, de um objeto X em um espaço d-dimensional é a intersecção de todos os hiperplanos que dividem X em duas partes de momento equivalentes em relação ao hiperplano. Informalmente, a centróide equivale à média de todos os pontos pertencentes a X. No decorrer do texto o termo centróide será usado para indicar o ponto de média de um conjunto de amostras.

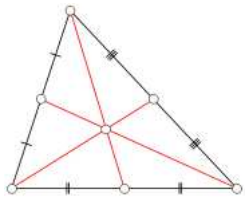

Figura 2.3: Centróide de um triângulo.

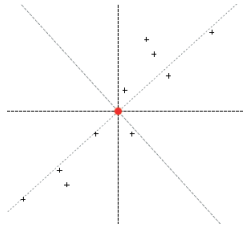

Figura 2.4: Centróide de um conjunto de amostras.

2.3.2 Reta em $\mathbb{R}^d$

Seja $\mathbf{c}$ um ponto em $\mathbb{R}^d$ pertencente a uma reta r e $\vec{\mathbf{v}}$ um vetor de direção de r, representados, respectivamente, pelas tuplas $\mathbf{c} = (c_1, c_2, ..., c_d)$ e $\vec{\mathbf{v}} = (v_1, v_2, ..., v_d)$. Então, a reta pode ser expressa por:

$$\mathbf{x} = \mathbf{c} + \alpha\vec{\mathbf{v}}.$$

De onde segue que, para qualquer $i \in \{1, 2, ..., d\}$,

$$\alpha = \frac{(x_i - c_i)}{v_i}$$

e

$$x_i = c_i + \frac{v_k}{v_i}(x_k - c_k), k \in \{1, 2, ..., d\}.$$

2.3.3 Hiperplano em $\mathbb{R}^d$

Um hiperplano em $\mathbb{R}^d$ pode ser representado por um ponto $\mathbf{c}$ pertencente a este hiperplano e um vetor normal $\vec{\mathbf{v}}$ a este hiperplano. A representação do ponto é dada através de uma tupla $\mathbf{c} = (c_1, c_2, ..., c_d)$ e a representação das componentes do vetor normal é dada pela tupla $\vec{\mathbf{v}} = (v_1, v_2, ..., v_d)$. Seja $\mathbf{x}$ um ponto qualquer pertencente ao hiperplano. Então, o vetor dado por $\overrightarrow{(\mathbf{x} - \mathbf{c})}$ é ortogonal a $\vec{\mathbf{v}}$, ou seja, $\overrightarrow{(\mathbf{x} - \mathbf{c})}^t * \vec{\mathbf{v}} = 0$. De posse dos valores de $\mathbf{c}$ e $\vec{\mathbf{v}}$, extrai-se a equação característica do hiperplano dada por:

$$\sum_{i=1}^{d} v_i(x_i - c_i) = 0.$$

Seja $\mathbb{S} \subseteq \mathbb{R}^d$, o hiperplano que divide o espaço em dois subespaços representados por $\mathbb{S}_{left}$ e $\mathbb{S}_{right}$, também chamados de *halfspaces*, estes subespaços são dados por:

$$\mathbb{S}_{left} = \{\mathbf{x} \in \mathbb{S} | \sum_{i=1}^{d} (v_i x_i - v_i c_i) < 0\}$$

e

$$\mathbb{S}_{right} = \{\mathbf{x} \in \mathbb{S} | \sum_{i=1}^{d} (v_i x_i - v_i c_i) \geq 0\}.$$

2.3.4 Hipercubo em $\mathbb{R}^d$

Um hipercubo em $\mathbb{R}^d$ é representado por uma tupla de intervalos $H = (I_1, I_2, ..., I_d)$. Cada intervalo I_i é dado por $I_i = [min_i, max_i|$. Um hipercubo divide um subespaço $\mathbb{S}$ em $\mathbb{S}_{in}$ e $\mathbb{S}_{out}$, onde:

$$\mathbb{S}_{in} = \{\mathbf{x} \in \mathbb{S} | min_i \leq x_i < max_i, \forall i\}$$

e

$$\mathbb{S}_{out} = \{\mathbf{x} \in \mathbb{S} | x_i < min_i \text{ ou } x_i \geq max_i, \forall i\}.$$

2.3.5 Superelipse em $\mathbb{R}^d$

Em $\mathbb{R}^2$, dados um ponto $\mathbf{c} = (c_1, c_2)$ e dois reais positivos r_1 e r_2, a equação

$$\frac{(x_1 - c_1)^2}{r_1^2} + \frac{(x_2 - c_2)^2}{r_2^2} = 1$$

define uma elipse não rotacionada com centro $\mathbf{c}$ e raios $r1$ e $r2$, respectivamente.

Seja R uma matriz de rotação, diferente da matriz identidade, dada por:

$$R = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix}$$

e defina

$$f_1(\mathbf{x}) = v_{11}x_1 + v_{12}x_2$$

e

$$f_2(\mathbf{x}) = v_{21}x_1 + v_{22}x_2.$$

O par $(f_1(\mathbf{x}), f_2(\mathbf{x}))$ pode ser entendido como a coordenada do ponto $\mathbf{x} = (x_1, x_2)$ após a rotação por R. Ao aplicarmos a rotação sobre todos os pontos da elipse temos a elipse rotacionada por R, dada por,

$$\frac{(f_1(\mathbf{x}) - f_1(\mathbf{c}))^2}{r_1^2} + \frac{(f_2(\mathbf{x}) - f_2(\mathbf{c}))^2}{r_2^2} = 1.$$

De forma similar, expandindo a equação característica de $\mathbb{R}^2$ para $\mathbb{R}^d$ temos que uma elipse rotacionada em $\mathbb{R}^d$ é dada pela equação

$$\sum_{1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{c})^2)}{r_i^2} = 1$$

com centro em $\mathbf{c} = (c_1, c_2, ..., c_d)$ e raios r_1,r_2,...,r_d , respectivamente.

Uma superelipse divide um subespaço $\mathbb{S}$ em $\mathbb{S}_{in}$ e $\mathbb{S}_{out}$, onde:

$$\mathbb{S}_{in} = \{\mathbf{x} \in \mathbb{S} | \sum_{i=1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{c}))^2}{r_i^2} \leq 1\}$$

e

$$\mathbb{S}_{out} = \{\mathbf{x} \in \mathbb{S} | \sum_{i=1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{c}))^2}{r_i^2} > 1\}.$$

O termo elipse é utilizado quando o espaço de representação é em $\mathbb{R}^2$. O termo superelipse é um termo genérico para um espaço $\mathbb{R}^d$. A título de curiosidade, já que não utilizaremos estas formas neste texto, também são encontrados os termos: hiperelipse e hipoelipse. Estes estão ligados ao grau do expoente dos termos da equação. Uma elipse e superelipse tem o valor do expoente de seus termos igual a 2. Uma hiperelipse é definida

quando este valor de expoente é maior que dois. Uma hipoelipse é definida quando este valor de expoente é menor que dois.

Na figura 2.5 temos um esboço de uma superelipse, um hiperelipse e uma hipoelipse, todas em $\mathbb{R}^2$ e rotacionadas pelo vetor de rotação:

$$R = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}$$

Os raios relativos possuem valores 13 e 35, respectivamente. Note que o vetor de rotação define o novo eixo de coordenadas esboçadas pelas linhas diagonais. O valor do expoente que define a hiperelipse exemplificada é 4 e o valor do expoente para a hipoelipse exemplificada é 0.6.

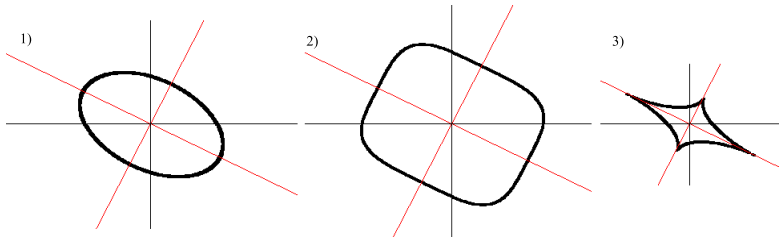

Figura 2.5: Elipse(1), hiperelipse(2) e hipoelipse(3) rotacionadas.

2.3.6 Intersecção entre dois Hipercubos

A intersecção entre dois hipercubos (não rotacionados) $H1$ e $H2$ definidos respectivamente pelas tuplas de intervalos $(I_1, I_2, ..., I_d)$ e $(J_1, J_2, ..., J_d)$ pode ser definida pelo algoritmo:

- Seja k a variável que indica se a intersecção existe. Faça k receber valor inicial igual a $true$.
- Para i de 1 até d, verifique se existe intersecção entre os intervalos definidos por I_i e J_i e caso não exista faça $k = false$

A intersecção entre dois hipercubos existe se e somente se para toda dimensão i a intersecção entre os intervalos correspondentes também existe. O algoritmo de verificação de intersecção entre hipercubos tem ordem de crescimento linear no número de dimensões d. Por esta linearidade os hipercubos são extremamente utilizados em algoritmos que manipulam objeto em $\mathbb{R}^d$. Muitos destes algoritmos representam um objeto em $\mathbb{R}^d$ por seu hipercubo mínimo de envolvimento (MBR, do inglês *Minimum Bounding Rectangle*), ou seja, um hipercubo de tamanho mínimo que comporta completamente o objeto. Em contrapartida, dois objetos próximos podem ter MBR's que possuem intersecção entre si gerando representação espacial duplicada. Em alta dimensionalidade, a probabilidade de intersecção entre MBR's aumenta significativamente, o que torna necessária a pesquisa por outras formas de representação, além dos MBR's.

2.3.7 Intersecção entre Hipercubo e Hiperplano

Dados um hipercubo e um hiperplano, seja $\mathbb{S}_{in}$ o subespaço interno ao hipercubo, $\mathbb{S}_{out}$ o subespaço externo ao hipercubo, $\mathbb{S}_{left}$ o subespaço à esquerda do hiperplano e $\mathbb{S}_{right}$ o subespaço à direita do hiperplano; então, temos que a intersecção entre o hipercubo e o hiperplano existe caso:

$$\mathbb{S}_{in} \bigcap \mathbb{S}_{left} \neq \emptyset$$

e

$$\mathbb{S}_{in} \bigcap \mathbb{S}_{right} \neq \emptyset$$

ou seja, se

$$\{\mathbf{x} \in \mathbb{S} | min_i \leq x_i < max_i, \sum_1^d (v_i x_i - v_i c_i) < 0\} \neq \emptyset$$

e

$$\{\mathbf{x} \in \mathbb{S} | min_i \leq x_i < max_i, \sum_1^d (v_i x_i - v_i c_i) \geq 0\} \neq \emptyset.$$

Dado que $\mathbb{S}_{in}$ é um subespaço convexo e ambos $\mathbb{S}_{left}$ e $\mathbb{S}_{right}$ também são subespaços convexos, podemos melhorar o algoritmo de verificação de intersecção validando a equação $\mathbb{S}_{in} \bigcap \mathbb{S}_{left} \neq \emptyset$ e $\mathbb{S}_{in} \bigcap \mathbb{S}_{right} \neq \emptyset$ para todos os pontos vértices do hipercubo. Assim, verifica-se a existência de pelo menos um vértice do hipercubo que satisfaz $\mathbb{S}_{left}$ e pelo menos um vértico do hipercubo que satisfaz $\mathbb{S}_{right}$.

O número de vértices de um hipercubo em $\mathbb{R}^d$ é igual a 2^d vértices. Um intervalo em $\mathbb{R}$ possui 2 vértices, um quadrado em $\mathbb{R}^2$ possui 4 vértices, um cubo em $\mathbb{R}^3$ possui 8 vértices e assim por diante. Ou seja, a ordem de grandeza do algoritmo de intersecção entre um hipercubo e um hiperplano é exponencial em d.

Métodos numéricos podem ser usados de forma a verificar a interseção entre em um hipercubo e um hiperplano em tempo polinomial. Algoritmos baseados em distâncias *minimax* [28] são exemplos de heurísticas utilizadas neste sentido.

2.3.8 Intersecção entre Superelipse e Hiperplano

Analogamente ao caso do hipercubo, deve-se fazer a verificação:

$$\mathbb{S}_{in} \bigcap \mathbb{S}_{left} \neq \emptyset \tag{2.1}$$

e

$$\mathbb{S}_{in} \bigcap \mathbb{S}_{right} \neq \emptyset, \tag{2.2}$$

ou seja,

$$\{\mathbf{x} \in \mathbb{S} | \sum_1^d (v_i x_i - v_i c_i) < 0, \sum_1^d \frac{(f_i(\mathbf{x}) - f_i(\mathbf{o}))^2}{r_i^2} \leq 1\} \neq \emptyset$$

e

$$\{\mathbf{x} \in \mathbb{S} | \sum_{1}^{d} (v_i x_i - v_i c_i) > 0, \sum_{1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{o}))^2}{r_i^2} \leq 1\} \neq \emptyset.$$

Note que, na equação da superelipse, trocamos o símbolo **c** por **o** já que o símbolo **c** já é usado para representar o ponto pertencente ao hiperplano. Assim, **o** representa o centro da superelipse.

Para simplificarmos a notação, definimos as funções

$$f_e(\mathbf{x}) = \sum_{1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{o}))^2}{r_i^2}$$

e

$$f_p(\mathbf{x}) = \sum_{1}^{d} (v_i x_i - v_i c_i).$$

Ambas $f_e(\mathbf{x})$ e $f_p(\mathbf{x})$ projetam um ponto $\mathbf{x}$ em $\mathbb{R}^d$ em um espaço $\mathbb{R}$.

Então, podemos expressar as condições das eq. 2.1 e 2.2 como:

$$\{\mathbf{x} \in \mathbb{S} | f_p(\mathbf{x}) < 0, f_e(\mathbf{x}) \leq 1\} \neq \emptyset$$

e

$$\{\mathbf{x} \in \mathbb{S} | f_p(\mathbf{x}) > 0, f_e(\mathbf{x}) \leq 1\} \neq \emptyset.$$

Pode-se deduzir que a intersecção existe se e somente se:

$$\{\mathbf{x} \in \mathbb{S} | f_p(\mathbf{x}) + f_e(\mathbf{x}) - 1 \leq 0\} \neq \emptyset.$$

2.3.9 Intersecção entre Superelipse e Reta

A intersecção entre uma reta e uma superelipse é feita isolando-se apenas uma variável na equação da superelipse e substituindo as demais variáveis pelo valor dado na equação da reta. Sendo assim:

Conforme visto, uma superelipse é dada pela equação

$$\sum_{i=1}^{d} \frac{(f_i(\mathbf{x}) - f_i(\mathbf{o}))^2}{r_i^2} = 1.$$

Expandindo as funções f_i dadas pela matriz de rotação, temos:

$$\sum_{i=1}^{d} \frac{\left(\sum_{j=1}^{d} v_{ij}x_i - \sum_{j=1}^{d} v_{ij}o_i\right)^2}{r_i^2} = 1. \tag{2.3}$$

Por outro lado, uma reta é dada pelas equações

$$x_j = c_j - \frac{v_k}{v_j}(x_k - c_k), i, k \in \{1, 2, ..., d\}. \tag{2.4}$$

Substituindo 2.4 em 2.3, temos

$$\sum_{i=1}^{d} \frac{\left(\sum_{j=1}^{d} v_{ij}[c_j - \frac{v_k}{v_j}(x_k - c_k)] - \sum_{j=1}^{d} v_{ij}o_i\right)^2}{r_i^2} = 1$$

na qual o valor do índice k é escolhido e fixado tal que $v_k \neq 0$.

A expanção desta equação, dado um k fixo, resulta em uma equação de segundo grau em x_k onde os coeficientes da equação são:

$$a = \sum_{i=1}^{d} \frac{\left(\sum_{j=1}^{d} \frac{v_{ij}}{v_j}\right)^2}{r_i^2},$$

$$b = -\sum_{i=1}^{d} \frac{2 * \left(\sum_{j=1}^{d} \frac{v_{ij}}{v_j}\right) \left[\sum_{j=1}^{d} v_{ij} * (c_j + \frac{v_k}{v_j} * c_k - o_i)\right]}{r_i^2}$$

e

$$c = -1 + \sum_{i=1}^{d} \frac{\left[\sum_{j=1}^{d} v_{ij} * (c_j + \frac{v_k}{v_j} * c_k - o_i)\right]^2}{r_j^2}$$

as raizes desta equação de 2º grau são

$$x_{k_{1,2}} = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}.$$

Posteriormente substitui-se o valor de x_{k_1} e x_{k_2} nas equações da reta para obter-se os valores de x_{i_1} e $x_{i_2} \forall i \in 1, 2, ..., d$. Com isso obtêm-se os valores pontuais $\mathbf{x_1} =$

$(x_{1_1}, x_{2_1}, ..., x_{d_1})$ e $\mathbf{x_2} = (x_{1_2}, x_{2_2}, ..., x_{d_2})$ que representam os pontos de intersecção entre a reta e a superelipse.

Note que os valores a, b e c possuem duas somatórias em d intercaladas, ou seja, o algoritmo para este cálculo é de ordem d^2. Entretanto, quando a superelipse não é rotacionada, ou seja, o vetor de rotação da superelipse é igual ao vetor identidade, as somatórias internas são desnecessárias, resultando em um algoritmo de ordem linear.

Capítulo 3

Árvores B-tree e B+-tree

Neste capítulo apresentaremos as árvores B-tree e B+-tree que são estruturas de dados de indexação unidimensional.

3.1 B-tree

A árvore B-tree é um índice unidimensional que reduz o número de acessos a memória secundária. Para isso, as entradas são agrupadas em nós da árvore, de forma que cada nó fique armazenado em apenas uma página da memória secundária e uma página da memória secundária armazene apenas um nó da árvore. Esta definição é a base do gerenciamento de memória secundária da B-tree. Para maximizar a capacidade de armazenamento de cada página da memória secundária, existe um número mínimo e máximo de entradas definidos pela B-tree para cada nó. No decorrer do texto referenciaremos a propriedade de número mínimo de entradas por nó de n_{tree} e a propriedade de número máximo de entradas por nó de m_{tree}. Um nó **cheio** é um nó que possui número de entradas maior que m_{tree}, também usaremos o termo **estouro de nó** para indicar a ocorrência de um nó cheio. Um nó **vazio** é um nó que possui número de entradas menor que n_{tree}. Um nó **preenchido** é um nó que possui número de entradas entre os valores n_{tree} e m_{tree}. Cada um dos nós é representado por um subespaço convexo do espaço total de busca. Este subespaço é dado por um intervalo [a,b|. Os valores a e b representam, respectivamente, o menor e maior valor de chave contido no subespaço representado pelo nó. Ou seja, todas as entradas contidas neste nó ou em algum de seus descendentes possuem valor de chave no intervalo [a,b|. Assim, em uma busca, serão lidas da memória secundária apenas as páginas cujo valor buscado está contido no subespaço representados por estas. Uma outra característica que reduz o número de acessos a memória secundária é a propriedade na qual a B-tree deve ser uma árvore balanceada, ou seja, todas as folhas estão em um mesmo nível na árvore. Chamaremos de **características de balanceamento** de uma B-tree a satisfação de duas propriedades: todos os seus nós estão preenchidos e todas as suas folhas estão em um mesmo nível da árvore.

O algoritmo de inserção em uma B-tree deve ficar atento para que o tamanho da página onde a entrada será inserida não ultrapasse m_{tree}. Quando isto for inevitável, a página deverá ser quebrada em duas novas páginas. Cada uma das novas páginas deverá conter pelo menos n_{tree} entradas. Com o mesmo intuito, o algoritmo de exclusão de uma entrada em uma B-tree não poderá permitir páginas com um número de entradas menor que n_{tree}. Caso isso ocorra, o nó com número inferior a n_{tree} deverá ser mesclado com um dos, ou ambos os, nós adjacentes. Chamamos de nós adjacentes os irmãos imediatamente a esquerda e imediatamente a direita de acordo com a ordenação crescente das chaves de acesso. Estes processos garantem que todas as páginas da B-tree fiquem com número de entradas sempre no intervalo [n_{tree},m_{tree}].

3.1.1 Estrutura de Dados da B-tree

A estrutura dos nós da B-tree está na forma definida por:

$$T = \langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, P_3, \langle K_3, Pr_3 \rangle, ..., \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$$

na qual $n_{tree} \leq q-1 \leq m_{tree}$. Cada P_i representa um ponteiro para um nó filho. Cada K_i representa uma chave de acesso ao objeto apontado por Pr_i. Para todo $2 \leq i \leq q-1$, $K_{i-1} < K_i$. Cada nó filho apontado por P_i contém valores pertencentes ao subespaço definido pelo intervalo $[K_{i-1}, K_i|$, exceto para $i = 1$ onde o subespaço é definido por $[-\infty, K_1|$ e $i = q$ onde o subespaço é definido por $[K_{q-1}, \infty]$. Uma entrada de um nó é uma de suas tuplas $\langle K_i, Pr_i \rangle$, para $1 \leq i \leq q-1$. O número de entradas no nó T, denotado por $|T|$, é igual a $q-1$.

As folhas da árvore, por não terem filhos, possuem os ponteiros para os filhos com valor nulo (*NULL*). Todas as folhas da árvore estão no mesmo nível, uma característica que é mantida tanto pelos algoritmos de inserção como exclusão. A raiz tem uma propriedade diferente dos outros nós internos: nela é permitida um número de entradas menor que n_{tree}. O nó raiz tem pelo menos dois filhos, caso contrário, ele é o único nó presente na árvore e, por isso, o nó raiz também é um nó folha.

3.1.2 Algoritmo de busca da B-Tree

Uma busca em uma B-tree ocorre inicialmente carregando-se a página que representa o nó raiz para a memória primária (políticas de gerenciamento podem manter a raiz de todos os índices de um SGBD na memória principal - seção 2.1.3 - otimizando este passo; entretanto, desconsideraremos qualquer otimização realizada por estas políticas). As demais páginas do índice ficam intactas na memória secundária. Com a raiz na memória principal, percorre-se as entradas do nó buscando a entrada que representa o subespaço onde o valor da chave de busca está contido. Ao se encontrar esta entrada, se o valor da chave de busca for igual ao valor da chave da entrada, o objeto buscado é exatamente o que é apontado por Pr_i desta entrada. Caso contrário, teremos duas situações:

- O nó raiz da árvore não possui filhos, portanto o nó raiz também é uma folha da árvore. Neste caso a chave buscada não está contida na árvore.
- O nó raiz da árvore não é uma folha da árvore. Neste caso a chave pode estar contida na subárvore cuja raiz é a entrada encontrada. Conseqüentemente, a página da memória secundária que representa este nó filho deverá ser carregada para a memória principal recorrendo-se ao mesmo processo de busca ocorrido no nó raiz da árvore.

Como todas as folhas da árvore estão em um mesmo nível e como todos os nós têm tamanho entre n_{tree} e m_{tree}, temos que o número de páginas visitadas em uma busca é, no máximo, igual à altura da árvore. Dado que n é o número de objetos indexados pela árvore, a altura da árvore tem ordem $O(log_{m_{tree}}(n))$. Portanto, o algoritmo de busca é da ordem de $O(log_{m_{tree}}(n))$. Abaixo seguem os passos realizados pelo algoritmo de busca, vide fluxograma na figura 3.1:

1. Seja T a raiz da árvore e seja c o valor de chave do objeto buscado.
2. Enumere as entradas, já ordenadas por valor de chave, em K_i. O valor de i é um índice entre 1 e o tamanho de T.
3. Enquanto $K_i < c$ e i for menor que $|T|$, incremente o índice i ($i = i + 1$).
4. Caso $K_i = c$, retorne o objeto apontado por Pr_i e termine.
5. Caso T seja uma folha, retorne um valor indicando que o objeto não foi encontrado e termine.
6. Faça T apontar para P_i e retorne ao item 2.

Algumas B-trees podem ser implementadas para aceitar valores duplicados de chave de acesso. Quando há valores de chave duplicados, a busca deve retornar uma estrutura de dados (uma lista encadeada por exemplo) de todos os casamentos encontrados. Os valores de chave duplicados também são armazenados em listas encadeadas na B-tree. Ou seja, os objetos apontados por Pr_i não são mais acessos diretos aos objetos e sim acesso às listas encadeadas de objetos com o mesmo valor de chave. Dado esta simplicidade no tratamento de chaves duplicadas, iremos desconsiderar o uso de chaves duplicadas nos demais algoritmos, apresentando apenas alguns comentários quando necessário.

3.1.3 Algoritmo de Inserção da B-Tree

As características de balanceamento da B-tree devem ser garantidas pelo algoritmo de inserção. Uma maneira de garantir que todas as folhas da B-tree estejam em um mesmo nível é inserindo-se cada novo objeto na folha da B-tree correspondente ao seu valor de

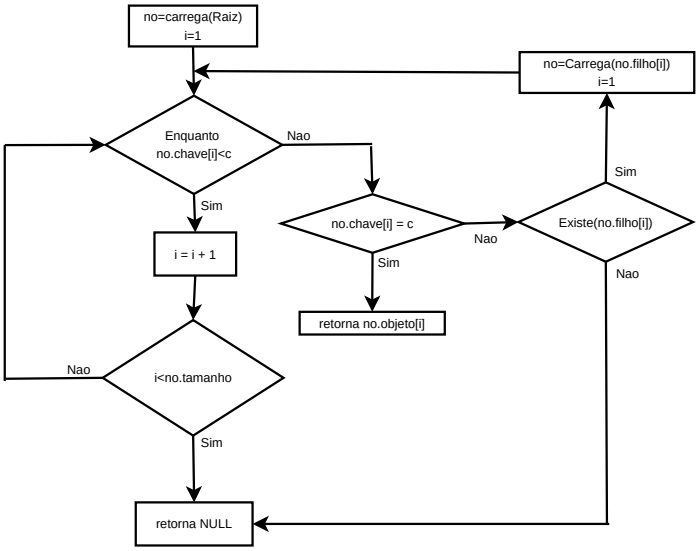

Figura 3.1: Fluxograma do algoritmo de busca em uma B-tree.

chave. Desta maneira, a árvore cresce na direção partindo das folhas para a raiz. Uma nova entrada só será inserida em um nó interno da B-tree caso algum de seus filhos tenha ultrapassado o valor de m_{tree}. Quando isto ocorre, o nó filho é dividido em um novo nó (vide figura 3.3). A entrada com valor de chave igual a mediana do conjunto formado pelos valores de chaves do nó é **promovida** para ser inserida no nó pai. As entradas com valor de chave menor ao valor da mediana são mantidas no nó atual. O novo nó criado receberá as entradas com valor de chave maior que a mediana. A este processo de divisão de um nó cheio damos o nome de **quebra** ou, do termo em inglês, ***split***. Caso seu ancestral direto também esteja cheio, a mediana promovida ao nó pai quebrará a propriedade m_{tree} deste. Assim, este também deverá ser dividido em um novo nó promovendo a mediana correspondente para o próximo ancestral. Desta maneira, a altura da árvore só será incrementada quando a chave for inserida em um nó folha cheio cujo todos os ancestrais também estejam cheios. Isto garante que todas as folhas da árvore permaneçam sempre no mesmo nível.

Na figura 3.2 temos o fluxograma do algoritmo de inserção de um objeto com valor de chave igual a **c**. Inicialmente é feita a busca pela folha onde o novo objeto será inserido. Em seguida, insere-se o objeto na folha mantendo sua ordenação. Caso a folha estoure, a folha deverá ser quebrada. Neste caso, a mediana será promovida para ser inserida no nó pai desta folha. A verificação de estouro também é feita no pai e a propagação da quebra e estouro deve ser verificada até o nó raiz.

Na figura 3.3 temos uma visão do comportamento estrutural da B-tree ao se aplicar o algoritmo de inserção com quebra. No caso deste exemplo, o nó folha mais a direita está com tamanho igual a 7. Seja $m_{tree} = 6$, então temos que $7 > m_{tree}$, o que implica na necessidade de quebra. Os elementos em cinza são os elementos que são remanejados na árvore. Note que a mediana de valor 8 foi promovida ao nó pai. Note também que não houve necessidade de modificação no nó folha mais a esquerda.

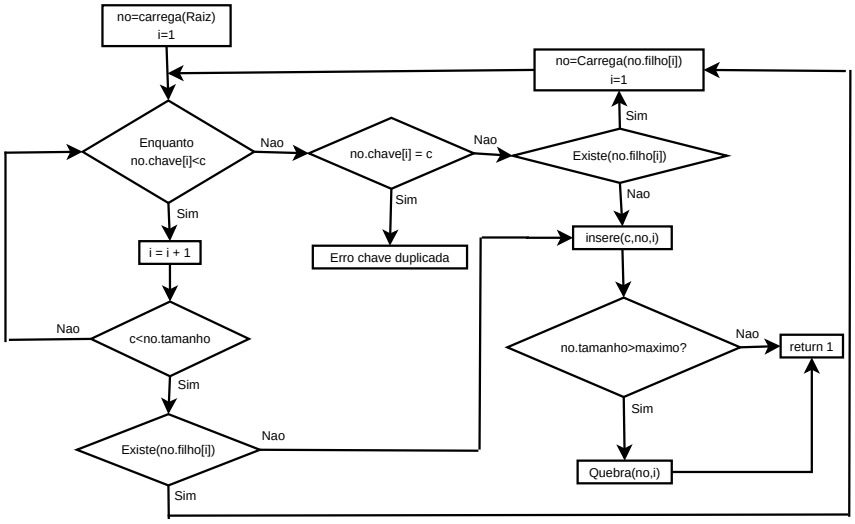

Figura 3.2: Fluxograma do algoritmo de inserção na B-tree.

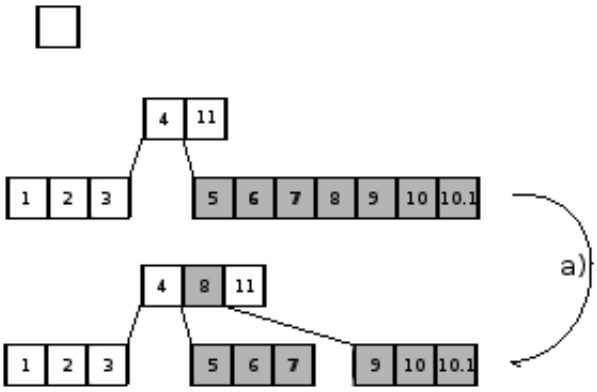

Figura 3.3: Exemplo de **quebra** na B-tree.

Os passos do algoritmo de inserção podem ser resumidos em:

1. Seja T a raiz da árvore e seja c o valor de chave do objeto inserido.
2. Caso T seja uma folha vá para o item 5.
3. Enquanto $K_i < c$ incremente o índice i ($i = i + 1$).
4. Faça T apontar para P_i e retorne ao item 2.

5. Crie uma nova entrada $\mathcal{E}$ em T e atribua o valor de c como o valor de chave de $\mathcal{E}$. A entrada $\mathcal{E}$ será inserida em uma posição i de T de forma a mantê-lo ordenado. Ao ponteiro de objeto Pr_i da entrada $\mathcal{E}$ atribua o endereço onde está localizado o objeto inserido.

6. Caso o tamanho de T seja menor ou igual a m_{tree}, termine.

7. Faça $\mathcal{E}$ a entrada de T cujo valor de chave é igual a mediana do conjunto formado por todos os valores de chave de T. Faça i o índice de posição da entrada $\mathcal{E}$ em T.

8. Crie um novo nó T'. Mova todas as entradas de índice maior que i e todos os ponteiros P_j de índice maior ou igual a i para T'.

9. Caso T seja a raiz da árvore vá para o item 13.

10. Insira a entrada $\mathcal{E}$ no nó pai de T removendo-a de T. Esta inserção deve manter o nó pai de T ordenado. Atribua a i a posição no pai de T onde $\mathcal{E}$ foi inserido.

11. Faça T igual ao pai de T.

12. Faça P_i de T apontar para T' e vá para o item 6.

13. Crie um novo nó R' e mova a entrada $\mathcal{E}$ para o mesmo. Aponte o ponteiro P_1 de R' para o endereço de T e P_2 para T'. Mude o endereço da raiz da árvore para R' e termine.

3.1.4 Algoritmo de exclusão da B-Tree

O algoritmo de exclusão, vide fluxograma na figura 3.4, também deverá garantir as características de balanceamento da B-tree. A exclusão é feita em duas etapas. A primeira etapa é a exclusão da entrada da árvore. A segunda etapa é a verificação e correção das características de balanceamento da árvore. Na primeira etapa, a entrada a ser excluída pode ser uma entrada de um nó interno na árvore. Neste caso, esta entrada deverá ser movida até uma das folhas da árvore antes de ser excluída. Abaixo seguem os passos realizados pela primeira etapa do algoritmo para a exclusão do elemento de valor c:

1. Seja T a raiz da árvore;

2. Faça $i = 0$;

3. Enquanto $c < K_i$ e $i < |T|$, $i = i + 1$;

4. Caso $c \neq K_i$ e T não seja uma folha, faça $T = P_i$ e retorne ao item 2;

5. Caso $c \neq K_i$ e T seja uma folha, termine informando que o objeto de valor c não se encontra na árvore.

6. Caso $c = K_i$ e T seja uma folha, exclua a entrada i de T, vá para a segunda etapa do algoritmo;

7. Caso contrário, faça uma troca de posição entre os valores de K_i de T e K_1 de P_i, também troque as posições dos valores de Pr_i de T e Pr_1 de P_i, faça $i = 1$ e $T = P_i$ e retorne ao item 6.

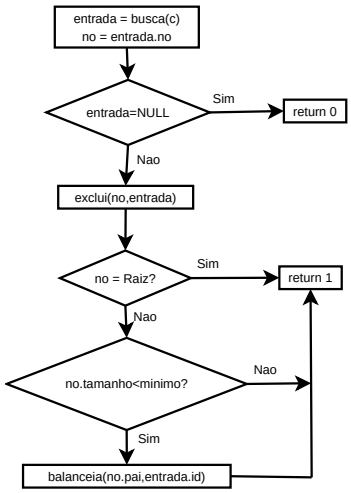

Figura 3.4: Fluxograma do algoritmo de exclusão.

O laço no item 7 do algoritmo anterior realiza a movimentação da entrada a ser excluída para uma folha, sempre trocando de posição, a cada iteração, com a primeira entrada imediatamente superior no nó filho. Quando esta entrada estiver em um nó folha ela poderá ser excluída pelo item 6. Em B-tree's que aceitam valores de chave duplicados a exclusão por um valor de chave c resulta na exclusão de todos os objetos presentes no índice que possuam este valor de chave.

A segunda etapa do algoritmo é a verificação e correção das características de balanceamento da árvore. Esta etapa começa a partir da folha T de onde a entrada foi excluída pela primeira etapa. Esta etapa só ocorre caso a chave a ser excluída tenha sido encontrada na árvore e removida, e caso a propriedade de tamanho mínimo de T tenha sido afetada. Para realizar o balanceamento da árvore, o algoritmo necessita fazer uma mesclagem de T com os nós pai de T, irmão da direita de T e irmão da esquerda de T. As entradas destes 4 nós serão recombinadas para formar uma subestrutura balanceada, vide exemplos nas figuras 3.5, 3.6, 3.7 e 3.8. Para isso alguns cálculos serão executados. Faça as seguintes considerações para a continuação do entendimento:

- Seja E uma referência ao nó irmão da esquerda, caso ele exista, caso contrário, $E = NULL$ e $|E| = 0$;

- Seja D uma referência ao nó irmão da direita, caso ele exista, caso contrário, $D = NULL$ e $|D| = 0$;

- Seja $Pai(N)$ uma referência ao nó pai do nó N, caso ele exista, caso contrário, $Pai(N) = NULL$;
- Seja $Indice_{pai}(T)$ o valor da posição i do ponteiro P_i em $Pai(T)$, tal que $P_i = T$;
- Seja S o conjunto de entradas formado pela união das entradas do nó T, E e D acrescentado com as entradas $Indice_{pai}(T)$ em $Pai(T)$ e $Indice_{pai}(E)$ em $Pai(T)$;
- Seja $|S|$ igual ao número de entradas do conjunto S.

No nó pai $Pai(T)$, apenas as entradas de índice $Indice_{pai}(T)$ e $Indice_{pai}(E)$ serão usadas e, se for necessário, excluídas. As entradas de S são as únicas entradas que serão remanejadas pelo algoritmo. Para um melhor entendimento visualize este passo como se fosse uma mesclagem dos três nós (E, T e D) em apenas um grande nó S seguido de uma quebra de S em nós que atendam a propriedade de n_{tree} e m_{tree}. Deve-se também minimizar a quantidade de nós gerados por esta quebra. Desta forma, a quantidade de nós gerada nesta quebra será igual a:

$$\alpha = \lceil \frac{|S|}{m_{tree}} \rceil.$$

Este valor minimiza o número de nós gerados dividindo a quantidade de nós total pelo valor máximo permitido por nó. Note que o valor de α é sempre menor ou igual a quantidade de nós inicial que formou S. Ou seja:

- Caso existam T,E e D, então $\alpha \leq 3$;
- Caso existam T e E, e $D = NULL$ então $\alpha \leq 2$;
- Caso existam T e D, e $E = NULL$ então $\alpha \leq 2$;

Dado o valor de α, o próximo passo é dividir igualmente as entradas entre estes nós. Este valor é dado por:

$$\beta = \lfloor \frac{|S| - (\alpha - 1)}{\alpha} \rfloor$$

O valor dado por $(\alpha - 1)$, no numerador da fração anterior, representa a quantidade de entradas pertencentes a S que estarão presentes em $Pai(T)$ ao final do processo. O valor β representa a quantidade de entradas que estarão em cada nó gerados pela quebra de S no final do processo, podendo variar de uma unidade por se tratar de números inteiros.

Para um melhor entendimento, dividiremos a explicação do algoritmo de exclusão em 4 casos. Apresentamos uma figura ilustrativa para cada um dos casos. Cada uma destas figuras é dividida em 3 partes. A primeira parte de cada figura ilustra a subestrutura inicial antes da exclusão, destacando o valor de chave que será excluído. A segunda parte

de cada figura ilustra a subestrutura local depois da exclusão e antes do balanceamento, destacando as entradas que formam o conjunto S. A terceira parte de cada figura ilustra a subestrutura local após o balanceamento, também destacando as entradas de S. Em todos os casos exemplificados a seguir é excluído o objeto com valor de chave igual a 3 e após esta exclusão o nó T fica vazio. Assumiremos também $m_{tree} = 6$ e $n_{tree} = 3$.

No primeiro caso, exemplificado na figura 3.5, temos $E = NULL$, $\alpha = 1$ e $\beta = 6$. Neste caso, a entrada $Indice_{pai}(T)$ e todas as entradas de D foram movidas para T. Houve a diminuição do número de entradas presentes em $pai(T)$. Houve a desalocação de D.

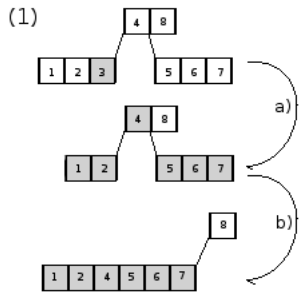

Figura 3.5: Comportamento estrutural da B-tree à exclusão(caso 1).

No segundo caso, exemplificado na figura 3.6, temos $E = NULL$, $\alpha = 2$ e $\beta = 4$. Neste caso, a entrada $Indice_{pai}(D)$ e a segunda entrada de D foram movidas para T. A primeira entrada de D substituiu a entrada $Indice_{pai}(D)$. Não houve a diminuição do número de entradas em $pai(T)$.

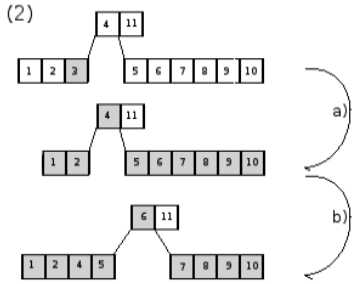

Figura 3.6: Comportamento estrutural da B-tree à exclusão(caso 2).

No terceiro caso, exemplificado na figura 3.7, $\alpha = 3$ e $\beta = 4$. Neste caso, a entrada $Indice_{pai}(D)$ e a primeira entrada de D foram movidas para T. A segunda entrada de D substituiu a entrada $Indice_{pai}(D)$. Não houve a diminuição do número de entradas em $pai(T)$.

No quarto caso, exemplificado na figura 3.8, $\alpha = 2$ e $\beta = 4$. Neste caso, a entrada $Indice_{pai}(T)$ e a primeira entrada de T foram movidas para E. A entrada $Indice_{pai}(D)$ foi movida para D. A segunda entrada de T substituiu a entrada $Indice_{pai}(D)$. Houve a diminuição do número de entradas em $pai(T)$. Houve a desalocação do nó T.

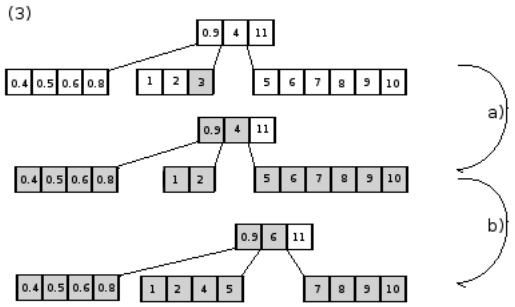

Figura 3.7: Comportamento estrutural da B-tree à exclusão(caso 3).

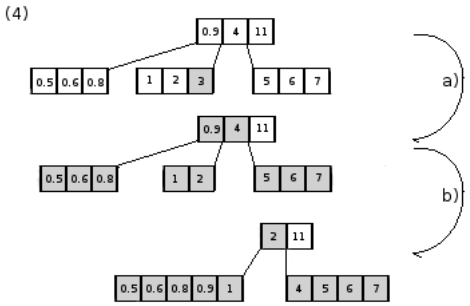

Figura 3.8: Comportamento estrutural da B-tree à exclusão(caso 4).

O algoritmo também deve prever outros casos particulares que não serão detalhados. Entretanto, estes 4 casos já dão uma visão geral do funcionamento do algoritmo. Quando há a diminuição do número de entradas do nó pai ocasionada pelo balanceamento dos filhos, este pai também deverá verificar seu balanceamento. Ou seja, o processo descrito é recursivo. Este processo será executado em todos os níveis, a partir da folha, cujo tamanho do nó é menor que n_{tree}. Assim, caso uma entrada seja excluída de uma folha cujo tamanho é n_{tree} e cujo tamanho de todos os seus ancestrais também sejam n_{tree}, então o rebalanceamento ocorrerá $O(log_{m_{tree}}(n))$ vezes até a raiz. Em particular, se, ao final do processo, a raiz for de tamanho igual a 0, T passará a ser a nova raiz da árvore, acarretando na diminuição da altura da árvore.

A figura 3.4 representa o fluxograma do algoritmo de exclusão do elemento de valor **c**. Como pode ser visualizado, inicialmente é feita a busca pelo elemento a ser excluído. Depois de excluir o elemento da folha, será feita a verificação de tamanho mínimo permitido. Caso o tamanho da folha seja menor que este tamanho mínimo, a folha será mesclada com as folhas da esquerda e da direita formando uma estrutura local balanceada. A mesclagem com nós vizinhos pode gerar a necessidade de exclusão de elementos no nó pai da folha, ou seja, o processo de verificação de tamanho mínimo e mesclagem com nós vizinhos será feita até a raiz da árvore.

3.2 Árvore Balanceada B+-tree

A diferença estrutural entre a B+-tree e a B-tree está nas entradas de seus nós internos. As entradas nos nós internos na B+-tree não possuem ponteiros para os objetos indexados. Os nós internos servem apenas de guia para se chegar até o nó folha onde está o ponteiro ao objeto indexado. Esta diferença estrutural permite um melhor aproveitamento do espaço nos nós internos, o que diminui significativamente a altura da árvore. Dado que existe uma "sobra" de espaço nos nós internos, estes nós permitem uma quantidade maior de entradas do que os nós folhas. Com isso, os valores de n_{tree} e m_{tree} para nós internos são maiores que os valores dos nós folhas. Denote os valores de m_{tree} e n_{tree} para os nós internos com os respectivos símbolos m_i e n_i. Para denotar os valores de m_{tree} e n_{tree} para as folhas utilizaremos, respectivamente, os símbolos m_f e n_f. Uma vez que os nós internos não possuem acesso direto aos objetos indexados, valores de chave poderão aparecer repetidos na árvore. No entanto, caso um valor de chave já esteja presente em qualquer nó interno na árvore ele não se repetirá em nenhum outro nó interno. Caso duas entradas $\mathcal{E}_1$ e $\mathcal{E}_2$ possuam um mesmo valor de chave na árvore, obrigatoriamente, uma das entradas pertence a um nó folha e a outra a um nó interno. Ou seja, não existem mais de 2 entradas com valor de chave repetido na árvore. Os nós folhas da B+-tree também possuem dois ponteiros extras, um para o próximo nó folha, e um para o nó folha anterior. Com estes ponteiros, ao se chegar à folha que contém o elemento buscado, a árvore pode ser navegada como em uma lista encadeada.

A B+-tree também aceita objetos com valor de chave repetidos, para isso ela utiliza o mesmo mecanismo descrito na B-tree. Ela utiliza-se de listas encadeadas de objetos com o mesmo valor de chave.

3.2.1 Estrutura de Dados da B+-tree

Como os nós folhas diferem dos nós internos na B+-tree temos:

1. Nós internos

   Cada nó interno possui a forma $\langle\langle P_1, K_1\rangle, \langle P_2, K_2\langle, ..., \langle K_{q-1}, P_q\rangle\rangle$. Temos $K_i < K_{i+1}$, para $1 \leq i < q-1$. Cada nó filho apontado por P_i contém valores pertencentes ao subespaço definido pelo intervalo $[K_{i-1}, K_i|$, exceto para $i = 1$ onde o subespaço é definido por $[\infty, K_1|$ e $i = q$ onde o subespaço é definido por $[K_i, \infty]$. O valor dado por $q - 1$ obrigatoriamente está no intervalo $[n_i, m_i]$.

2. Nós Folha

   Cada nó folha possui a forma $\langle P_{ant}, \langle K_1, Pr_1\rangle, ..., \langle K_q, Pr_q\rangle, P_{pos}\rangle$.Temos $K_i \leq K_{i+1}$ para $1 \leq i < q$. A chave K_i refere-se ao objeto apontado por Pr_i. O ponteiro P_{ant} aponta para o nó anterior e P_{pos} para o nós posterior. O valor de q está obrigatoriamente no intervalo $[n_f, m_f]$

Na figura 3.9 esboçamos a estrutura de dados de uma B+-tree simples com apenas 24 objetos indexados. Assuma para esta figura $m_f = 6$, $n_f = 3$, $m_i = 12$ e $n_i = 6$. Os nós folha são representados em cinza e somente nelas temos acesso aos objetos indexados. Os valores de chave 7, 13 e 19 aparecem repetidos na árvore mas, estes, quando presentes em nós internos, servem apenas de guias para nós inferiores. As setas de duplo sentido representam os ponteiros P_{ant} e P_{pos} presentes nos nós folhas e exibem a navegabilidade que estes proporcionam. Com estes podemos navegar da entrada de valor 6 para a entrada de valor 7 sem termos que acessar novamente os nós ancestrais resultando em algoritmos de busca de melhor desempenho.

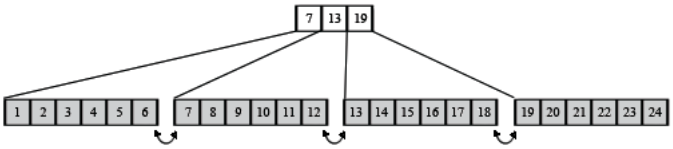

Figura 3.9: Estrutura da B+-tree.

3.2.2 Algoritmos de Busca, Inserção e Exclusão da B+-tree

Os algoritmos de busca, inserção e exclusão na abordagem B+-tree diferem dos correspondentes algoritmos da B-tree, obviamente, para manter-se a diferença estrutural entre as duas abordagens. No algoritmo de busca, a árvore será sempre percorrida até uma folha. Como os nós internos são apenas guias da busca, quando encontramos uma entrada com chave de valor igual ao valor buscado verificamos se esta entrada encontra-se em um nó folha. Caso seja um nó folha, haverá o ponteiro para o objeto buscado. Caso não seja um nó folha, o nó filho correspondente será carregado e a busca será efetuada novamente no nó filho. Este processo se repetirá até que encontremos um valor de chave igual ao buscado que, obrigatoriamente, esteja em um nó folha.

O algoritmo de inserção irá posicionar a chave e a referência ao objeto correspondente em um nó folha, assim como na B-tree. Contudo, existem algoritmos distintos de quebra para os nós folha e nós internos. Na quebra de uma folha a entrada de valor de chave igual a mediana escolhida para ser promovida ao nó pai será mantida em uma das subdivisões da folha. Será criada uma nova entrada no nó pai com valor de chave igual ao nó promovido. Com isto, duas entradas de mesmo valor de chave estarão na árvore. Na quebra da folha também são alterados os ponteiros de navegação de P_{ant} e P_{pos} dos nós resultantes para se manter a navegabilidade. Para os nós internos o algoritmo de quebra é idêntico ao algoritmo de quebra da B-tree.

O algoritmo de exclusão [18] também tem dois casos distintos. O primeiro é quando o valor de chave a ser excluído não está em nenhum nó interno e a exclusão não resulta em um nó folha vazio. A simples exclusão da entrada na folha correspondente é executada. Quando o valor de chave também estiver presente em algum nó interno, a entrada onde

este ocorre deverá ter seu valor de chave substituída pelo valor de chave imediatamente superior. Quando o algoritmo resulta em uma folha vazia, se faz necessária a mesclagem com as folhas vizinhas para manter o balanceamento da árvore. Na B+-tree, ao contrário da B-tree, a mesclagem entre folhas ignora os valores de chave, e suas respectivas entradas, do nó pai pelo fato de ser um valor de chave repetido. Caso a mesclagem não implique na exclusão de uma entrada do nó pai, o valor de chave correspondente à mediana irá substituir o valor de chave da entrada no nó pai. A partir da exclusão da entrada do nó pai, resultante da mesclagem das folhas, o algoritmo de mesclagem para nós internos é idêntico ao algoritmo de mesclagem da B-tree.

Para apresentarmos o algoritmo de exclusão assuma:

- K_i ser o valor de chave da i-ésima entrada;
- P_i ser o i-ésimo filho;
- Pr_i ser o objeto apontado pela i-ésima entrada, este ponteiro só existe nas folhas;
- $|T|$ ser o número de entradas contidas em T;
- $Pai(T)$ ser o nó pai do nó T;
- $Indice(T)$ ser o valor i de P_i em $Pai(T)$, onde P_i é igual a T;
- e c ser o valor de chave a ser excluído.

Abaixo segue os passos executados em uma exclusão:

1. Faça T receber a raiz da árvore;
2. Faça $i = 1$;
3. Enquanto $i < |T|$ e $c < K_i$, faça $i = i + 1$;
4. Caso $i = |T|$ e T seja uma folha, termine;
5. Caso $c = K_i$ e T seja um nó folha, vá para o item 8;
6. Caso $c = K_i$ faça $B = T$ e $j = i$;
7. Faça $T = P_i$ e retorne ao item 2;
8. Caso B esteja definido, faça K_j do nó B receber o valor de K_{i+1} da folha T. Isso corrige o valor de chave encontrado em um nó interno substituindo-o por um valor imediatamente superior;
9. Exclua a entrada K_i de T;
10. Caso $|T| \geq n_f$ ou T seja a raiz, termine;

11. Continue no próximo algoritmo.

Após a busca pelo nó folha T onde a entrada com valor de chave a ser excluído se encontra e após a realização desta exclusão pelo algoritmo anterior, é necessário fazer, caso $|T| < n_f$, a mesclagem de T com nós vizinhos de forma a preenchê-lo.

1. Faça D e E serem, respectivamente, os irmãos da direita e esquerda de T. Caso D não exista, assuma $|D| = 0$ e caso E não exista, assuma $|E| = 0$;
2. Caso D exista, mova todas as entradas de T para D e em seguida, mova todas as entradas de E para D de forma que D seja um único nó contendo todas as entradas ordenadas;
3. Caso D não exista, mova todas as entradas de E para T de forma que T seja um único nó contendo todas as entradas ordenadas. Faça $D = T$ e $T = E$;
4. Seja $\alpha = \lceil \frac{|D|}{m_f} \rceil$;
5. Seja $\beta = \lfloor \frac{|D|}{\alpha} \rfloor$;
6. Caso α seja igual a 1, faça i igual a $Indice(T)$ e exclua a chave K_i e o ponteiro P_i do $Pai(T)$. Destrua T e vá para o item 9;
7. Caso α seja igual a 2:
   - altere o valor de chave do nó pai de T $K_{pai(D,T)}$ para receber o valor de K_β.
   - Mova as β últimas entradas de D para T;
   - Caso ambos D, T e E estiverem definidos, faça i igual a $Indice(T)$ e exclua a chave K_i e o ponteiro P_i do $Pai(T)$. Destrua E e vá para o item 9;
   - Caso contrário termine;
8. Caso α seja igual a 3:
   - altere os valores de chave do nó pai de T $K_{Indice(T)}$ e $K_{Indice(E)}$ para receberem, respectivamente o valor de K_β e $K_{2*\beta}$;
   - Mova as β últimas entradas de D para E e em seguida mova as β últimas entradas de D para T;
   - Termine;
9. Faça $T = Pai(T)$;
10. Caso $|T| < n_i$, aplique o algoritmo de mesclagem de nós internos;

Este algoritmo de mesclagem só é aplicável nas folhas da B+-tree, sendo o algoritmo de mesclagem dos nós internos da B+-tree um algoritmo a parte. Lembrando que a propriedade de valor mínimo de entradas dos nós internos n_i é diferente dos nós folhas n_f. O algoritmo de mesclagem dos nós internos da B+-tree é similar ao algoritmo de mesclagem na B-tree, já que, a partir do primeiro nó interno, não é mais permitido chaves repetidas na árvore.

Capítulo 4

KDB-tree e outros índices multidimensionais

Apresentaremos neste capítulo alguns índices multidimensionais e a forma como eles representam objetos dispersos em um espaço multidimensional. Para ajudar a descrição dos algoritmos que manipulam estes índices, exemplos em $\mathbb{R}^2$ são apresentados nas figuras. São abordados de maneira mais detalhada os índices KDB-tree e R-tree. Nos demais, enfatizamos apenas as diferenças estruturais e algorítmicas e as vantagens e desvantagens em relação a esses.

4.1 KDB-Tree

A representação do subespaço de busca de um nó na B+-tree é feita através de um intervalo $[a, b|$ em $\mathbb{R}$. Na B+-tree, o filho apontado por P_i representa o subespaço dado pelo intervalo $[a, b|$, onde a é o valor de chave da entrada i e b o valor de chave da entrada $i+1$. Analogamente, para o espaço multi-dimensional em $\mathbb{R}^d$ indexado pela KDB-tree [26, 27], a representação do subespaço se dará por d intervalos, um para cada dimensão. Sendo assim, cada nó da árvore representa um hipercubo H em $\mathbb{R}^d$. Cada nó interno da KDB-tree é composto por um conjunto de pares na forma $\langle H, Ponteiro\ para\ Filho\rangle$, sendo que H representa o descritor de um hipercubo em $\mathbb{R}^d$. O ponteiro para o filho refere-se a página na memória secundária onde o nó filho está armazenado. Assim como na B+-tree, os nós internos não possuem referências diretas aos objetos indexados, servindo apenas como guia para a busca. As entradas nos nós folha da KDB-tree não representam hipercubos e sim pontos em $\mathbb{R}^d$. Para um melhor aproveitamento de espaço, devido a essa diferença estrutural, os nós internos também possuem valores de tamanho máximo m_i e mínimo n_i diferente dos valores de tamanho máximo m_f e mínimo n_f dos nós folha. Abaixo segue a representação das estruturas dos nós internos e nós folha:

1. Nós internos

Cada nó interno possui a forma

$$T = \langle parent, pointCount, splitDimension, \langle H_1, P_1\rangle, \langle H_2, P_2\rangle, ..., \langle H_q, P_q\rangle\rangle.$$

Caso o nó filho apontado por P_i seja folha, este contém chaves (pontos) contidas no subespaço definido pelo hipercubo representado em H_i. Caso P_i seja um nó interno, este contém hipercubos contidos no subespaço definido pelo hipercubo H_i. O valor de q, em um índice ótimo, obrigatoriamente está no intervalo $[n_i, m_i]$. Entretanto, devido a multidimensionalidade, q pode aparecer com um valor menor que n_i. O hipercubo H_i tem a forma:

$$H_i = \langle (H_{i_{1_l}}, H_{i_{1_u}}), (H_{i_{2_l}}, H_{i_{2_u}}), ..., (H_{i_{d_l}}, H_{i_{d_u}})\rangle$$

na qual utilizamos a representação $H_{i_{j_{(u|l)}}}$ em que o índice i representa a posição da entrada no nó, o índice j representa a dimensão, e os valores u ou l representam, respectivamente, o menor ou maior valor na dimensão j que delimitam o hipercubo H_i. Um hipercubo H_i dado por $\langle (1, 2), (2, 5)\rangle$ em $\mathbb{R}^2$ é um retângulo cujo lado na dimensão de índice 1 tem tamanho $(2 - 1) = 1$, o lado na dimensão de índice 2 tem tamanho $(5 - 2) = 3$, e o volume deste retângulo é dado por $(3 \cdot 1) = 3$.

O atributo *parent* contém um ponteiro para o nó pai de T. Caso T seja a raiz, então $parent = NULL$. O atributo *splitDimension* representa a dimensão de quebra do nó. Esta é determinada pelo nível da árvore onde o nó se encontra. A *splitDimension* de T é a dimensão que separa os hipercubos armazenados em T. O *pointCount* armazena a quantidade de pontos indexados pela subárvore enraizada em T. Este atributo é utilizado na heurística de escolha do hiperplano de quebra no algoritmo de quebra de nó.

2. Nós Folha

   Cada nó folha possui a forma $T = \langle parent, \langle K_1, Pr_1\rangle, ..., \langle K_q, Pr_q\rangle\rangle$. A chave K_i é um ponto em $\mathbb{R}^d$, ou seja, caracterizado por d coordenadas. A representação de K_i é na forma $(K_{i_1}, K_{i_2}, K_{i_3}, ..., K_{i_d})$, na qual o valor K_{i_j} representa o valor da j-ésima dimensão. Cada chave K_i refere-se ao objeto apontado por Pr_i. O valor de q, em um índice ótimo, obrigatoriamente está no intervalo $[n_f, m_f]$, entretanto, devido a multidimensionalidade, q pode aparecer com um valor menor que n_f. Caso T não seja raiz, o atributo *parent* é um ponteiro para o nó pai de T e, caso T seja raiz, $parent = NULL$. Os nós folha não possuem o atributo *pointCount*, entretanto, assumamos que, quando for citado o valor do atributo *pointCount* de um nó folha, estaremos referenciado o valor da quantidade de pontos armazenadas nesta folha.

Abaixo apresentamos algumas propriedades da estrutura de dados da KDB-tree.

1. O MBR de um nó T é o retângulo mínimo de encapsulamento (do inglês, *minimum bounding rectangle*) de todos hipercubos (ou pontos, caso T seja folha) representados em T.

2. Para todo nó T, que possui um MBR H_T, e todo nó N filho de T, representado pelo MBR H_N, $H_N \subseteq H_T$.

3. Para todo hipercubo H_A e H_B pertencentes a um mesmo nível da árvore, temos $H_A \cap H_B = \phi$, ou seja, os hipercubos de um mesmo nível da árvore não se interceptam;

4. O hipercubo que representa o MBR dos hipercubos do nó raiz abrange todo o espaço de busca do índice;

5. Seja $\langle H_i, P_i \rangle$ uma entrada de um nó interno, cujo nó filho P_i também é interno, o hipercubo H_i representa o MBR de todos os hipercubos representados nas entradas de P_i;

6. Seja $\langle H_i, P_i \rangle$ uma entrada de um nó interno, cujo nó filho P_i é folha, então, todas as entradas de P_i representam um ponto contido em H_i e H_i é o MBR de todos os pontos contidos em P_i;

7. Todos os nós em um mesmo nível da árvore possuem uma mesma dimensão de quebra, denominada de *splitDimension*, utilizada no algoritmo de quebra;

8. Todos os hipercubos representados pelos filhos de um nó não possuem interseção entre si.

Na figura 4.1 apresentamos um exemplo em $\mathbb{R}^2$ de segmentação de espaço realizado pela KDB-tree. Neste exemplo, a raiz possui duas entradas e sua dimensão de quebra é y. O primeiro filho possue 7 entradas e o segundo filho possui 4 entradas. Ambos possuem dimensão de quebra igual x, dado que ambos estão no segundo nível da árvore.

4.1.1 Consulta em uma KDB-Tree

Como definido em [27], uma busca em extensão é uma busca por pontos que pertencem a um subespaço. Este subespaço de busca é definido por um hipercubo B representado por intervalos em cada dimensão. Seja

$$B = \langle (B_{1_l}, B_{1_u}), ..., (B_{d_l}, B_{d_u}) \rangle.$$

Se algum intervalo (B_{j_l}, B_{j_u}) representar todo o domínio coberto pela árvore na dimensão j a busca é chamada de busca em extensão parcial, do inglês *partial range query*. Se existe um ou mais intervalos em que B_{j_l} é igual a B_{j_u} a busca é chamada busca de casamento parcial, do inglês *partial match query*. Se para todo $1 \leq i \leq d$ temos $B_{j_l} = B_{j_u}$, a busca é chamada de busca por casamento exato, do inglês *exact match query*.

O algoritmo que retorna todos os elementos que satisfazem uma busca em extensão na KDB-tree [27] é dado por:

1. Carrega a raiz em T. Faça o conjunto de pontos retornados R inicialmente vazio.

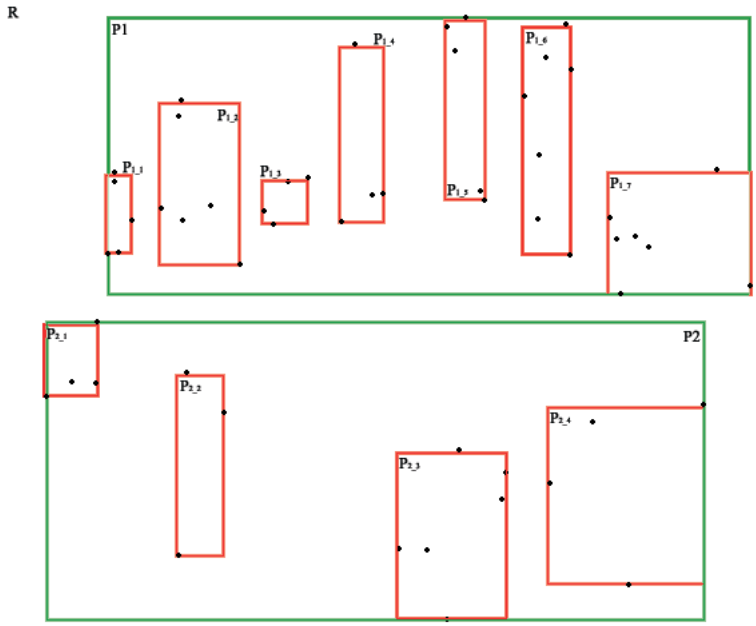

Figura 4.1: Exemplo de segmentação feita pela KDB-tree.

2. Caso T seja um nó folha, todos os pontos representados em T que intercedem o espaço de busca B são agregados à R.

3. Caso T seja um nó interno, a busca é realizada para todos os nós filhos de T cujo hipercubo H_i intercede com o espaço de busca B. O conjunto de pontos retornados por cada uma destas recursões é agregado a R.

Portanto, a busca em extensão poderá ter múltiplos caminhos entre a raiz e os nós folha. Mas, de acordo com a propriedade 2 da KDB-tree apresentada anteriormente, os caminhos não são redundantes. Já em uma busca de casamento exato, de acordo com esta mesma propriedade, haverá um caminho único de busca entre a raiz e as folhas. O algoritmo de casamento exato é o mesmo que a busca em extensão, basta informar $B_{j_l} = B_{j_u}$ para todo j no subespaço de busca.

4.1.2 Algoritmo de Inserção

Seja $\mathbf{p} = (p_1, p_2, ..., p_d)$ a chave a ser inserida no índice. A busca pelo nó folha no qual $\mathbf{p}$ será inserido percorre a árvore a partir da raiz até uma folha, escolhendo-se sempre o nó filho que "melhor comporta" $\mathbf{p}$.

Para entender o que significa "melhor comporta $\mathbf{p}$", seja

$$T = \langle parent, pointCount, splitDimension, \langle H_1, P_1 \rangle, \langle H_2, P_2 \rangle, ..., \langle H_q, P_q \rangle \rangle$$

um dos nós no caminho em percorrimento (inicialmente, o nó raiz). O próximo nó a ser percorrido é, entre os q filhos P_1, P_2,... e P_q, aquele cujo intervalo do hipercubo na dimensão $splitDimension$ está mais próximo de $p_{splitDimension}$. Mais formalmente, dado um hipercubo H_i e uma dimensão j, seja

$$growth(H, \mathbf{p}, i) = min(abs(p_i - H_{i_l}), abs(p_i - H_{i_u})).$$

Então, o índice do hipercubo cujo intervalo na dimensão $splitDimension$ está mais próximo de $p_{splitDimension}$ é dado por

$$k = argmin_{j=1}^{|T|}(growth(H_j, \mathbf{p}, splitDimension)). \tag{4.1}$$

O nó filho P_k correspondente ao hipercubo H_k é dito ser o filho que "melhor comporta $\mathbf{p}$" e será o próximo nó a ser considerado no percorrimento.

Este processo é repetido até se chegar a um nó folha. Após feita a inserção é realizada a quebra como descrito na próxima seção.

4.1.3 Algoritmo de Quebra

Seja $C = \langle T_1, T_2, ..., T_k \rangle$ o conjunto de nós que formam o caminho entre a folha onde o ponto será inserido e a raiz, ou seja, T_1 é a folha, T_k é a raiz e T_{i+1} é pai de T_i para todo $1 \leq i \leq k-1$. Seja $C' = \langle T_1, T_2, ..., T_j \rangle$ uma subsequencia de C, onde $j \leq k$, tal que todos os nós pertencentes a esta seqüencia estejam cheios. O algoritmo de quebra é realizado, inicialmente em T_j, seguindo a seqüencia inversa de C' até T_1. Durante este processo, a seqüência C' pode ser alterada devido ao rearranjo ocasionado por chamadas recursivas de quebras e mesclagens. Ou seja, se T_i for quebrado em T_i e T_i', o caminho até o ponto inserido deverá passar ou por T_i ou por T_i'. Se este caminho passar por T_i', T_i' substituirá T_i na seqüência C', caso contrário, não haverá alterações em C'. Mesmo com estas mudanças, C' nunca aumenta de tamanho, ou seja, este algoritmo é executado, no máximo j vezes.

Fazendo-se uma comparação com a quebra realizada nas árvores B-tree e B+-tree, a KDB-tree tem comportamento semelhante. Na B-tree e B+-tree, a seqüência C' também existe, entretanto, a quebra é realizada de baixo para cima, começando inicialmente em T_1 e terminando em T_j. Se este algoritmo, na B-tree e B+-tree, fosse alterado para ser executado de cima para baixo, começando em T_j e terminando em T_1, o resultado final seria o mesmo, tanto estruturalmente quanto em desempenho. Já na KDB-tree, uma quebra de baixo para cima, começando em T_1 e terminando em T_j, pode apresentar pior desempenho que a quebra de cima para baixo. Isso dá-se devido ao fato de que, na KDB-tree, uma quebra de um nó interno pode vir a resultar em quebras de nós descendentes. Assim, começando em nós superiores, quebras em níveis inferiores são evitadas.

Existe uma diferenciação entre os algoritmos de quebra de nós folha e nós internos. Esta diferença é evidente considerando-se as diferenças estruturais entre os dois tipos de

nós. A KDB-tree também exige um novo tipo de quebra denominado quebra condicionada. A seguir apresentamos separadamente cada um destes algoritmos.

4.1.4 Algoritmo de Quebra de uma Folha

Quando ocorre o estouro de um nó folha T, o conjunto de pontos deve ser dividido em 2 subconjuntos A e B de mesmo tamanho (diferença máxima 1). Entretanto, devido a multidimensionalidade, esta divisão pode vir a não gerar conjuntos de mesmo tamanho. Veja o exemplo na figura 4.2. Considere a direção horizontal na figura como a dimensão *splitDimension*. Nesta figura, a divisão dos 9 pontos em 2 subconjuntos ideais de 4 e 5 pontos, respectivamente, não é possível. Pelo fato de haver 8 pontos com mesmo valor de componente em *splitDimension*, estes 8 pontos, obrigatoriamente, ficarão em um mesmo conjunto. (Entretanto, se considerarmos uma distribuição uniforme dos pontos presentes no índice em $\mathbb{R}^d$, a probabilidade deste evento ocorrer é muito pequena.) O hiperplano J na figura representa esta divisão. Finalmente, o algoritmo de quebra de um nó folha consiste em:

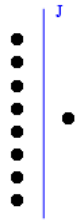

Figura 4.2: Ilustração de uma quebra de uma folha que resulta em nós com tamanhos significativamente diferentes.

1. Seja p_{i_j} o valor da j-ésima componente do ponto do i-ésimo ponto armazenado em T;

2. Caso $pai(T) = NULL$, faça $splitDimension = 1$, caso contrário, faça $splitDimension$ receber o valor do atributo $splitDimension$ de $pai(T)$;

3. O hiperplano de divisão é dado por $J = (j_1, j_2, ..., j_d)$ onde $j_i = 0$ para todo $i \neq splitDimension$ e
$$j_{splitDimension} = \frac{\sum_{i=1}^{pointCount} p_{i_{splitDimension}}}{pointCount};$$

4. Crie novo nó T' e mova todos os pontos de T que satisfaçam $p_{i_{splitDimension}} < j_{splitDimension}$ para T';

5. Faça $T'.parent = T.parent$, $T'.splitDimension = T.splitDimension$ e $P = pai(T)$;

6. Caso $T'.parent = NULL$, crie um novo nó interno, defina-o como raiz da árvore, e adicione duas entradas $\langle H_1, P_1 \rangle$ e $\langle H_2, P_2 \rangle$ onde $H_1 = MBR(T)$, $P_1 = T$, $H_2 = MBR(T')$ e $P_2 = T'$; termine.

7. Seja i a entrada de P tal que $P_i = T$, faça $H_i = MBR(T)$;

8. Crie uma nova entrada em P acessada em P_{q+1} e faça $P_{q+1} = T'$ e $H_{q+1} = MBR(T')$.

4.1.5 Algoritmo de Quebra de um Nó Interno

Na quebra de nós internos, o hiperplano J tem o i-ésimo eixo de coordenadas no espaço $\mathbb{R}^d$ como o seu vetor normal, onde $i = splitDimension$ em T. Ou seja, $J_i = 0$ para todo $i \neq splitDimension$ e $J_i = x$ para $i = splitDimension$, onde x representa o único valor a ser calculado para representar J. Uma heurística simples para a escolha de x é defini-lo tal que:

- seja H_i o hipercubo que representa o MBR do i-ésimo filho de T;
- seja q a quantidade de filhos de T;
- seja $pointCount_i$ a quantidade de pontos no i-ésimo filho de T,

então,

$$x = \frac{\sum_{k=1}^{q} pointCount_k (H_{k_l} + \frac{H_{k_u} - H_{k_l}}{2})}{\sum_{k=1}^{q} pointCount_k}.$$

A quebra de um nó interno T consiste em:

1. Calcular o hiperplano J de divisão;

2. Quebrar todos os nós filhos de T que fazem interseção com J condicionando a quebra ao hiperplano J.

3. Criar novo nó T' , mover todos os nós filhos de T e a direita de J para T';

4. Criar nova entrada em pai de T apontando para T';

5. Recalcular os MBR's e o valor dos atributos *pointCount* de T e T' presentes em pai de T;

6. Para todo filho de T, verificar se este se encontra vazio e, caso verdade, aplicar o algoritmo de mesclagem neste filho vazio;

7. Para todo filho de T', verificar se este se encontra vazio e, caso verdade, aplicar o algoritmo de mesclagem neste filho vazio;

O item 2 introduz um algoritmo, extensão do algoritmo de quebra, que é apresentado na próxima seção.

4.1.6 Algoritmo de quebra condicionada a um hiperplano

Uma nova heurística de quebra condicionada, não presentes nos índices unidimensionais, é introduzida na seção anterior. Para exemplificarmos este efeito, mostramos a quebra de uma raiz R. Quando uma raiz R é quebrada em uma nova página, após uma seqüência de inserções, será criada uma nova raiz N. O valor *splitDimension* de N será o valor *splitDimension* de R incrementado de 1.(Quando o valor *splitDimension* de N supera o valor d de quantidade de dimensões do índice, ou seja, $splitDimension = d + 1$, o *splitDimension* de N passa a ter valor igual a 1). Uma nova página R' também será criada para armazenar as entradas movidas de R que estão a direita de J, mantendo as entradas a esquerda de J em R. R' terá o valor de seu *splitDimension* igual a R já que R e R' estão em um mesmo nível na árvore. Desta maneira temos que toda página presente em um mesmo nível da árvore possui um mesmo valor de *splitDimension*.

Neste processo, caso R possua nós filhos representados por hipercubos que façam interseção com J, estes nós filhos também deverão ser quebrados por J. Na prática, quando for a primeira vez em que a i-ésima dimensão de quebra for utilizada, ou seja, ainda não houve nenhuma quebra realizada sobre a dimensão i, a maioria dos descendentes de R também serão quebrados por J. Este evento de propagação da quebra aos nós descendentes é denominado de *quebra condicionada*. As figuras 4.3 e 4.4 ilustram este evento. Assuma $m_{tree} = 8$ e $n_{tree} = 4$ para estas figuras. Na figura 4.3, a raiz R representa todo o espaço de busca, os retângulos representam os hipercubos das entradas de R que apontam para as folhas, filhos de R. As folhas armazenam os pontos contidos em seus respectivos retângulos. R possui 8 entradas. Quando alguma das folhas estourar será criado um novo retângulo e exigirá uma nova entrada em R e, conseqüentemente, R estoura. Uma nova raiz N é criada e estes 8 retângulos (P_1,P_2, ..., P_8) são quebrados por J em mais 8 novos retângulos (P'_1,P'_2, ..., P'_8) que farão parte de uma nova página R' irmã de R. O resultado final é ilustrado na figura 4.4.

Na figura 4.4, uma nova raiz N foi criada com duas entradas, uma apontando para a raiz antiga R e outra apontando para a nova página R' criada. Foram criadas mais 8 novas páginas folha. O quantidade total de escritas de página foi de 19 páginas. Ou seja, esta quebra reescreveu todo o índice. Entretanto, com o crescimento do número de objetos indexados e com o aumento do número de páginas na árvore, a probabilidade deste evento ocorrer diminui drasticamente. Assim, em uma análise amortizada, a ordem de crescimento do algoritmo de inserção continua sendo $log(n)$. A figura 4.5 ilustra a alteração da estrutura de dados causada pela quebra para o mesmo exemplo.

A quebra condicionada tem algoritmo equivalente tanto para nós internos quanto para

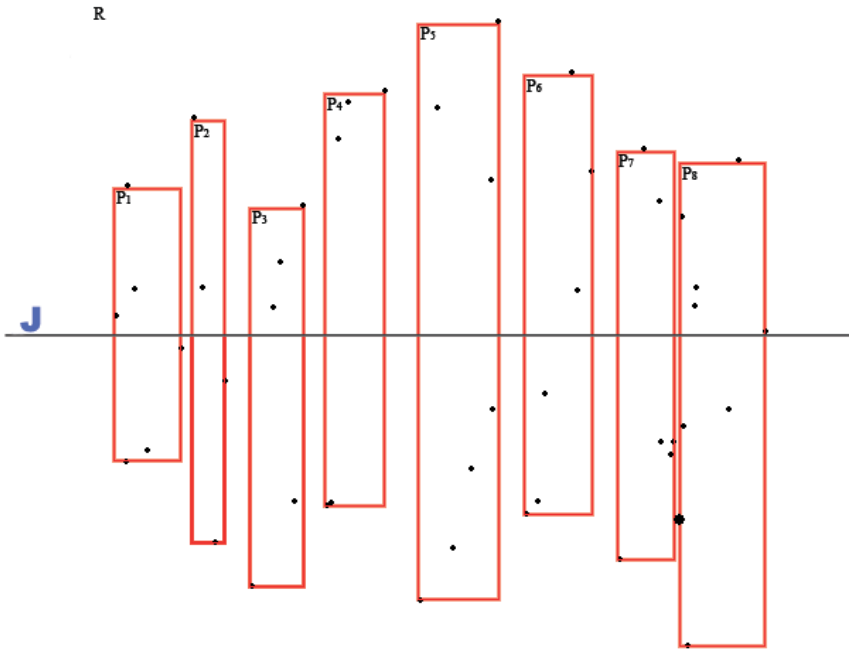

Figura 4.3: Representação da estrutura antes da quebra condicionada.

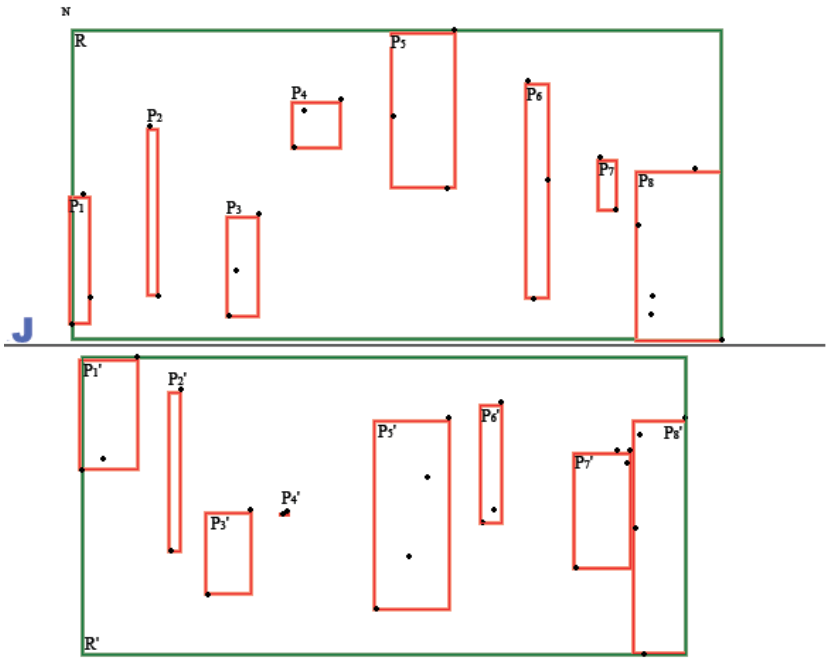

Figura 4.4: Representação da estrutura depois do quebra condicionada(b).

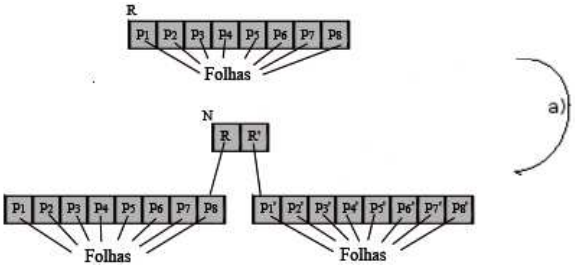

Figura 4.5: Representação estrutural da quebra condicionada, antes e depois.

nós folha. A diferença em relação a quebra não condicionada consiste apenas no fato de que o hiperplano de quebra J não é calculado e sim informado como parâmetro de quebra.

4.1.7 Algoritmo de Exclusão em uma KDB-Tree

O algoritmo de exclusão [27] se resume em: encontrar, através de uma busca de casamento exato, a folha T onde o ponto $\mathbf{p}$ está armazenado e excluir a entrada correspondente a $\mathbf{p}$; atualizar o valor dos hipercubos deste nó e de todos os seus ancestrais; caso o nó fique vazio, mesclá-lo com algum ou alguns dos nós irmãos. Esta mesclagem é análoga a da B+-tree cujos nós irmãos escolhidos são os nós da esquerda e da direita. Na KDB-tree, entretanto, dentre todos os irmãos, os dois nós irmãos escolhidos são aqueles que estão à direita e à esquerda de acordo com o valor de *splitDimension*, dimensão de quebra, do nó. Caso o número de nós deste novo nó ultrapasse o tamanho máximo permitido, este será quebrado em dois nós da mesma forma que o algoritmo de quebra definido anteriormente.

Seja $C = \langle T_1, T_2, ..., T_k \rangle$ o conjunto de nós que formam o caminho entre a raiz e a folha onde o ponto será excluído tal que: T_1 seja a folha, T_k seja a raiz e T_{i+1} seja pai de T_i para todo $1 \leq i \leq k-1$. Seja $C' = \langle T_1, T_2, ..., T_j \rangle$ uma subseqüência de C, onde $j \leq k$, tal que todos os nós pertencentes a esta seqüência estejam vazios. Assim como a quebra, a mesclagem é aplicada de cima para baixo, ou seja, de T_j em direção a T_1.

A mesclagem de um nó interno difere da mesclagem de nós folhas. Quando dois nós internos, de um mesmo nível da árvore, são mesclados através de sua *splitDimension*, os nós filhos destes nós mesclados podem apresentar interseção entre si em uma outra dimensão $i \neq splitDimension$. Veja o exemplo ilustrado pelas figuras 4.6 e 4.7. Na figura 4.6 temos uma KDB-tree imeditamente antes da exclusão de um ponto que gerou uma mesclagem. Na figura 4.7 temos a mesma KDB-tree depois das mesclagens. Assumamos para este exemplo $n_{tree} = 4$ e $m_{tree} = 8$. Denotemos a direção vertical das figuras representando a dimensão Y e denotemos a dimensão horizontal das figuras representando a dimensão X. A dimensão de quebra do nível 0 da árvore na figura 4.6 é Y (separa P_1 e P_2). A dimensão de quebra do nível 1 da árvore na figura 4.6 é X (separa P_{1_1} de P_{1_2}, P_{2_1} de P_{2_2}, etc.). O nó P_2 possui 4 filhos. Quando alguma exclusão resultar em uma diminuição do tamanho deste nó para 3 filhos, o nó P_2 deverá ser mesclado com o nó P_1 na dimensão Y (esta dimensão é a *splitDimension* que separa P_1 e P_2 na figura 4.6). Os nós filhos de P_1

e P_2 não possuem interseção entre si na dimensão Y, entretanto, alguns destes possuem interseção entre si na dimensão X. Para se manter a propriedade de apenas uma dimensão de quebra por nível na árvore, este fato exige uma mesclagem dos nós filhos de P_1 e P_2 que possuem interseção em X. Na figura 4.6, podemos notar que o nó P_{1_4} faz interseção, na dimensão X, com P_{2_2} e P_{2_3}. Sendo assim, estes também serão mesclados, em um processo recursivo, quando P_1 e P_2 forem mesclados. Este tipo de mesclagem também pode disparar quebras. No caso, ao se mesclar P_{1_4}, P_{2_2} e P_{2_3} temos como resultado um nó P_{1_4} cheio. Este nó resultante será quebrado em P_{1_4} e P'_{1_4} mostrados na figura 4.7. O mesmo ocorre com os nós P_{1_2}, P_{1_3} e P_{2_1}. Como resultado gerou-se os nós P_{1_2} e P'_{1_2} mostrados na figura 4.7. Já os nós P_{1_5} e P'_{1_5} foram gerados das mesclagens de P_{1_5} e P_{2_4}.

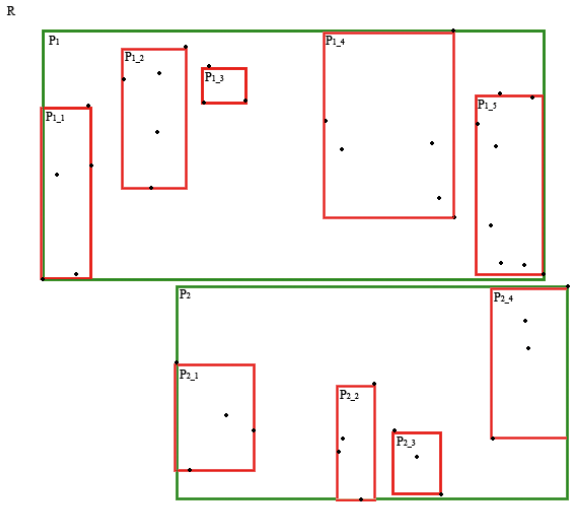

Figura 4.6: KDB-tree antes da exclusão com mesclagem.

A figura 4.8 ilustra uma KDB-tree após a inserção de 500 pontos em $\mathbb{R}^2$. Neste exemplo o número mínimo de entradas é igual a 5 e o número máximo de entradas é igual a 10. Os pontos em preto representam a posição espacial dos objetos indexados e os hipercubos representam as páginas de acesso. Note também que a árvore possui altura igual a 4.

4.2 R-Tree

A principal diferença entre as árvores de Regiões [16] (R-trees, do inglês Region Trees) e a KDB-tree é que os objetos por ela indexados são regiões no espaço $\mathbb{R}^d$ ao invés de pontos singulares. Isto é uma vantagem pelo fato de reduzirmos significativamente a quantidade de informação indexada através de um pré-processamento que agrupa a informação pontual em regiões. Entretanto, há a desvantagem de que os objetos nos nós folha da árvore são indivisíveis, o que requer alterações nos algoritmos de inserção, exclusão e busca. Estas regiões indivisíveis levam a em hipercubos com interseção entre si para um mesmo

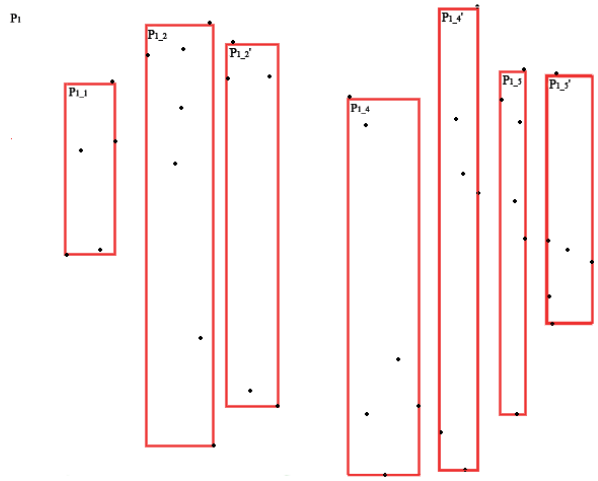

Figura 4.7: KDB-tree depois da exclusão com mesclagem.

Figura 4.8: Exemplo 2 de segmentação da KDB-tree em um espaço em $\mathbb{R}^2$

nível da árvore. Com isso, um mesmo subespaço de busca pode estar representado em mais de um nó da árvore, deteriorando o desempenho do algoritmo de busca. Na figura 4.9

exemplificamos como esta interseção pode ser gerada. Nela, um nó contendo 9 entradas foi dividido em dois nós, um com 4 entradas e outro com 5 entradas. Os objetos do primeiro nó resultante estão rotulados pelo valor 1 e os objetos do segundo nó resultante estão rotulados pelo valor 2. Os MBR's destes nós possuem interseção entre si como ilustrado.

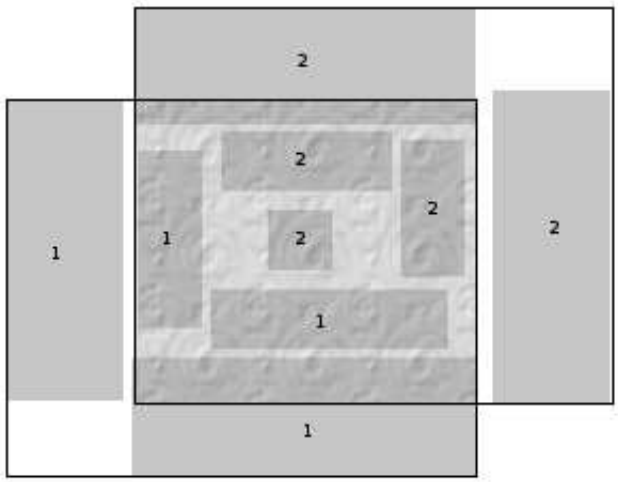

Figura 4.9: Exemplo de segmentação da R-tree

A estrutura de dados dos nós internos da R-tree são idênticas à estrutura dos nós internos da KDB-tree. Entretanto, as R-trees não possuem o atributo *splitDimension*, pois o algoritmo de quebra da R-tree não se baseia neste conceito. Já nas folhas, as chaves de acesso não são mais pontos em $\mathbb{R}^d$ e sim regiões no espaço $\mathbb{R}^d$. A estrutura de dados de uma região é idêntica a estrutura de dados de um hipercubo ($R_i = \langle (R_{i_{1_l}}, R_{i_{1_u}}), (R_{i_{2_l}}, R_{i_{2_u}}), ..., (R_{i_{q_l}}, R_{i_{q_u}}) \rangle$). Mesmo com essa igualdade de representação, distinguiremos regiões de hipercubos pelas propriedades:

- Região

  As regiões são chaves de acesso aos objetos indexados e estão presentes apenas nas folhas da árvore. As regiões são as estruturas atômicas do espaço de busca do índice, portanto, indivisíveis.

- MBR

  Os MBR's ou retângulos mínimos de encapsulamentos(do inglês, *minimum bounding boxes*) são hipercubos utilizados como chaves de acesso aos nós de níveis superiores da árvore e estão presentes apenas nos nós internos da árvore. Os MBR's podem ser divididos ou mesclados com outros pelos algoritmos de inserção e exclusão.

4.2.1 Algoritmo de Busca da R-Tree

O algoritmo de busca [16] percorre a árvore a partir da raiz de forma semelhante a KDB-tree. Entretanto, mais de um filho pode ser consultado para cada nó da árvore, mesmo sendo uma busca de casamento exato. Isto ocorre pelo fato da R-tree possuir hipercubos que se interceptam em um mesmo nível da árvore. A busca por extensão de um hipercubo B em uma R-tree é dada pelos passos:

1. A busca é iniciada atribuindo a T o nó raiz da árvore;
2. Caso T não seja uma folha da árvore, para cada filho Pr_i de T cujo hipercubo H_i intercepta B é feita uma chamada recursiva da busca;
3. Caso T seja uma folha da árvore, toda região R_i que intercepta B é adicionada ao resultado da busca.

4.2.2 Algoritmo de Inserção da R-Tree

Abaixo seguem os passos do algoritmo de inserção:

1. A busca pelo nó folha onde o novo objeto será inserido é iniciada atribuindo a T o nó raiz da árvore;
2. Caso T não seja um nó folha da árvore, escolhe-se o filho de T cujo MBR de MBR(T) mais a região inserida é menor. Repete-se este item recursivamente atribuindo a T o valor do ponteiro para o filho escolhido.
3. Caso T seja um nó folha da árvore, o novo elemento é inserido em T.
4. Caso T tenha ultrapassado o tamanho máximo de elementos permitidos por nó, é chamado o algoritmo de quebra do nó em um novo nó, detalhado adiante. A quebra também é propagada na direção dos ancestrais de T, caso eles também ultrapassem o tamanho máximo permitido. São redefinidos todos os hipercubos que representam os ancestrais de T.
5. Caso o balanceamento tenha acarretado em um estouro de tamanho máximo do nó raiz, um novo nó raiz é criado e a árvore tem sua altura aumentada.

O algoritmo de tratamento de estouro de nó é detalhado na subseção 4.2.4.

4.2.3 Algoritmo de Exclusão da R-Tree

No algoritmo de exclusão [16], a busca será dada por uma busca de casamento exato. Este algoritmo difere do algoritmo da KDB-tree pelo fato de que na R-tree existem hipercubos

que se interceptam em um mesmo nível da árvore. O algoritmo de exclusão de uma região R é dado pelos passos:

1. Achar R na árvore através de uma busca. Faça T apontar para o nó onde R está armazenado e E apontar para a entrada referente a R;
2. Remova a entrada E do nó T;
3. Crie uma lista encadeada para armazenamento temporário de nós excluídos denominada Q.
4. Caso T seja o nó raiz vá para o item 8. Caso contrário, faça P ser o nó pai de T e E_T ser a entrada do nó pai que aponta para T.
5. Caso T possua menos entradas que o valor mínimo permitido, adicione T em Q e em seguida exclua-o de P.
6. Caso T não tenha sido eliminado, ajuste o descritor MBR da entrada E_T para ser o MBR das entradas de T.
7. Faça $T = P$ e repita a partir do item 4.
8. Reinsira todas as entradas dos nós excluídos que foram armazenados em Q na árvore. Para isso utilize o algoritmo de inserção descrito anteriormente.

Este algoritmo utiliza-se de uma lista auxiliar Q de nós vazios. Todos os nós vazios encontrados no caminho entre a raiz e o nó folha onde R estava presente são adicionados a esta lista excluindo-os da R-tree. Posteriormente é feita a reinserção destes nós na R-tree de maneira a manter as características de balanceamento. Seja Q uma lista tal que $Q = \langle Q_1, Q_2, ..., Q_n \rangle$. Para cada Q_i a inserção é feita na árvore no mesmo nível onde Q_i se encontrava anteriormente.

4.2.4 Algoritmo de Quebra da R-tree

Ao se inserir uma nova entrada em um nó folha T cheio, será necessário quebrar o nó. As regiões armazenadas no nó são indivisíveis, o que pode vir a resultar, durante a quebra do nó, em duas regiões que possuem interseção. A quebra deve ser realizada de modo a minimizar o número de vezes que ambos os nós serão lidos em buscas subseqüentes, ou seja, minimizar a interseção entre as regiões. A leitura de um nó pelo algoritmo de busca é feita quando há interseção entre o subespaço de busca e o MBR do nó, dessa forma a soma dos volumes dos dois novos MBRs criados para os dois novos nós também deve ser minimizada. Para conseguir este resultado, um algoritmo direto seria a escolha da melhor dentre todas as possibilidades de divisão. Entretanto, o custo deste algoritmo seria da ordem de $2^{m_{tree}+1}$. Dois algoritmos propostos por Guttman [16] foram:

- Algoritmo de custo quadrático

  1. Para todo par de duas entradas E_1 e E_2 de T, calcule $d = area(MBR(T)) - area(E_1) - area(E_2)$. Escolha o par que maximize o valor de d.
  2. Cada elemento do par será adicionado a grupos distintos.
  3. Caso algum dos grupos tenha tamanho suficientemente pequeno tal que a adição de todos os itens restantes faça com que este grupo tenha tamanho igual ao mínimo aceitável, faça isso e termine o processo.
  4. Para cada um dos itens ainda não inseridos, faça d_1 igual ao valor de crescimento de volume do grupo 1 pela posterior inserção do item e d_2 igual ao valor de crescimento de área do grupo 2 pela posterior inserção do item.
  5. Escolha o item que maximize o valor $|d1 - d2|$.
  6. Insira o item escolhido no grupo que exija menor crescimento do MBR total.
  7. Repita a partir do item 3 até que não haja elementos desagrupados.

- Algoritmo de custo linear

  1. Para cada dimensão i, $1 \leq i \leq d$, ache as entradas que possuem o maior valor mínimo em i e o menor valor máximo em i. Faça s o valor do módulo da diferença entre estes dois valores.
  2. Normalize s dividindo-o pela largura do MBR de T em i, ou seja, a largura que contém todos os hipercubos de T na dimensão i.
  3. Escolha o par de hipercubos que maximize o valor de s normalizado para todas as dimensões.
  4. Cada elemento do par será adicionado a grupos distintos.
  5. Caso algum dos grupos tenha tamanho suficientemente pequeno tal que a adição de todos os itens restantes faça com que este grupo tenha tamanho igual ao mínimo aceitável, faça isso e termine o processo.
  6. Escolha qualquer item ainda não inserido.
  7. Insira o item escolhido no grupo que resulta em menor crescimento do MBR total.
  8. Repita a partir do item 4 até que não haja elementos desagrupados.

A figura 4.9 ilustra uma situação em que 9 regiões no espaço $\mathbb{R}^2$ ($d = 2$), representadas por retângulos em cinza, estão em um mesmo nó de uma R-tree. Para o nó representado temos $m_{tree} = 8$. A quebra pelo algoritmo quadrático é exemplificado através dos rótulos 1 e 2 nas regiões. As regiões de rótulo 1 pertencem ao primeiro nó enquanto as regiões de rótulo 2 pertencem ao segundo nó. Os hipercubos que comportam estes novos nós estão representados como bordas em preto (retângulos). A sobreposição dos hipercubos está representada na área texturizada. É exatamente esta sobreposição que deteriora o algoritmo de busca.

Não há distinção entre algoritmo de quebra de nós folha e nós internos na R-tree, em nós internos, os hipercubos que representam seus descendentes são tratados como regiões.

4.3 R+-Tree

A árvore R+-tree é uma proposta de melhora de desempenho do algoritmo de busca em relação a R-tree. A R+-tree propõe uma solução para o problema de caminhos múltiplos de busca na R-tree causados pela interseção de hipercubos. A R+-tree difere da R-tree nos seguintes aspectos:

1. Objetos indexados podem estar referenciados em mais de uma folha.
2. Os nós internos possuem hipercubos que fazem interseção entre si em um mesmo nível. Mas, por haver duplicação de referências a objetos, o algoritmo de busca tem caminho único até a folha.
3. A propriedade n_{tree} é garantida repetindo referências a um mesmo objeto em folhas diferentes.

A duplicação de referências a objetos em diferentes folhas é necessária para que não haja a necessidade de fazer múltiplos caminhos na árvore para a busca de um objeto. Quando o algoritmo de quebra de um nó T é executado, pode haver uma interseção I entre os hipercubos dos nós resultantes. Neste caso, todos os objetos referenciados por T que fazem interseção com I devem ser referenciados em ambos os nós resultantes. Isto garante que, em uma busca, qualquer nó que seja escolhido contenha a referência ao objeto indexado. Na R-tree, ambos os nós seriam lidos em busca de objetos em I. Desta forma, o algoritmo de busca tem seu desempenho melhorado.

Apesar da aparente melhora do algoritmo de busca, a R+-tree exige maior número de entradas para a mesma quantidade de objetos indexados. Isto implica em um aumento da altura da árvore em relação a R-tree. Ou seja, dependendo do grau de interseção dos hipercubos representados nos nós da árvore, o aumento de desempenho pode vir a não ser tão significativo.

A figura 4.10 ilustra uma quebra de um nó. Nesta figura, o número máximo m_{tree} é igual a 8. A quebra das 9 regiões presentes na figura resultou em regiões repetidas nos novos nós. As regiões rotuladas pelo número 1 estão presentes no primeiro nó. As regiões rotuladas pelo número 2 estão presentes no segundo nó. As regiões rotuladas tanto pelo número 1 quanto pelo número 2 estão presente nos dois nós.

4.4 X-Tree

Um dos trabalhos realizados para a diminuição de sobreposição de hipercubos nos espaços multi-dimensionais foi a especificação da árvore de nós estendida(X-tree, do inglês

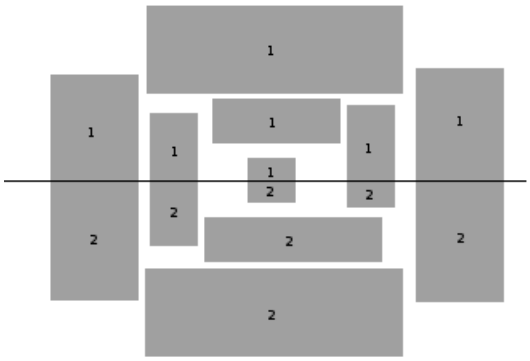

Figura 4.10: Exemplo de quebra da R+-tree

eXtended node tree) [5]. A X-tree pode ser vista como uma abordagem híbrida entre estruturas hierárquicas baseadas em R-tree e vetores lineares. Quando a indexação é baseada em baixa dimensionalidade, as estruturas hierárquicas se mostram mais eficientes devido a baixa incidência de hipercubos sobrepostos. Já em alta dimensionalidade, as estruturas hierárquicas tendem a deteriorar-se devido ao aumento da probabilidade de hipercubos sobrepostos. Quando a dimensionalidade é alta, em apenas um nó de índice R-tree podemos ter todas as suas entradas com hipercubos sobrepostos. Já que para este caso todas as entradas devem ser analisadas em uma busca, uma estrutura linear resultaria em melhor desempenho que uma estrutura hierárquica.

4.4.1 Estrutura de Dados da X-Tree

A X-tree possui 3 tipos de nós. Os nós de dados da X-tree contém tuplas de descritores MBR e apontadores para os objetos, ou seja, eles tem forma semelhante aos nós folha da R-tree. Os nós chamados de diretórios possuem tuplas de descritores MBR e apontadores para nós filho, assim como os nós internos de uma R-tree. Os nós denominados super-nós são nós baseados em vetores lineares que possuem tamanho maior, ou muito maior, que o máximo permitido nos nós de diretórios e nos nós de dados. A decisão de se criar um super-nó ou não é baseada no grau de interseção entre as regiões de um nó. Por exemplo, dado a melhor divisão de um nó T em dois nós N_1 e N_2, o grau de interseção entre estes nós é dado por $\frac{Volume(N_1 \cap N_2)}{Volume(N_1 \cup N_2)}$. Caso este valor seja maior que um valor de corte, é determinado que um super-nó deverá ser criado ao invés de ser realizada a divisão de T em $N1$ e $N2$. Geralmente, o tamanho dos super-nós é um múltiplo do tamanho máximo permitido nos nós internos. O objetivo dos super-nós é evitar a quebra de nós que possuam uma grande porcentagem de interseção, o que resultaria em uma estrutura hierárquica ineficiente. Uma estrutura hierárquica, para o caso de alta sobreposição de regiões, resultaria em uma estrutura que faria uma leitura de todos os nós folha em um algoritmo de busca.

Capítulo 5

PCA-Tree

Na KDB-Tree, todos os nós pertencentes a um mesmo nível da árvore possuem uma mesma dimensão de quebra no espaço de indexação $\mathbb{R}^d$. Isto significa que, para que todas as d dimensões sejam indexadas, a altura da árvore deve ser maior ou igual a d. Em uma KDB-Tree cujo valor do número mínimo de entradas por nó é igual a n, o número mínimo de entradas necessárias para que todas as dimensões sejam indexadas é igual a n^d entradas. Em aplicações reais de indexação de espaços de alta dimensionalidade, o valor de n gira em torno de 25 a 200 entradas. Assumamos $d > 50$ como alta dimensionalidade, assim, temos um valor mínimo de entradas igual a 25^{50}. Este valor é significativamente alto até para aplicações mais complexas onde o número de entradas gira em torno de 10^{12}. Uma boa modelagem para o uso de uma KDB-Tree para aplicações desta ordem de grandeza seria utilizar um valor de $m = 10$ e utilizar, no máximo 12 dimensões para cada valor de chave de entrada. Qualquer valor acima disso, resultaria em uma ou mais dimensões não indexadas.

A PCA-Tree é uma proposta de indexação, também baseada na KD-tree e B-tree, que ao invés de escolher uma dimensão de quebra para cada nível da árvore, realiza a quebra através de um hiperplano cujo vetor normal é a componente principal dos pontos a serem particionados. Esta componente principal é extraída através da aplicação da análise de componente principal (PCA, do inglês *Principal Component Analysis*) citada no Capítulo 2. Assim, já na primeira quebra realizada na árvore, todas as dimensões relevantes poderão ser utilizadas. Além disso, a PCA-Tree tem a propriedade de ser acessada como uma árvore binária, ou seja, as regiões são agrupadas hierarquicamente de duas em duas, o que proporciona uma segmentação hierarquica mais detalhada do espaço.

A figura 5.1 exemplifica uma PCA-tree com número de objetos indexados igual a 11. Nesta, o número máximo de entradas por página é igual a 10 e o número mínimo de entradas por página igual a 5. Ela possue duas folhas, sendo a primeira T com 5 entradas e a segunda T' com 6 entradas. Também possui uma página raiz R com apenas um nó $\mathcal{N}$. Na figura, a reta representa o hiperplano h que divide as páginas folhas, sendo h definido pelo centróide $\mathbf{c}_{\mathcal{N}}$ e vetor normal $\overrightarrow{v_{\mathcal{N}}}$.

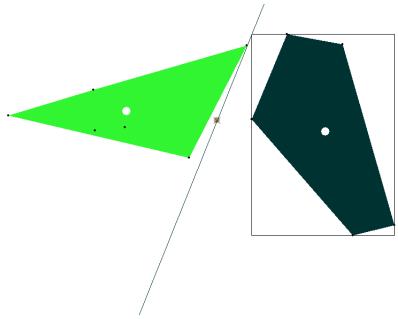

Figura 5.1: Exemplo da distribuição espacial de uma PCA-Tree.

5.1 Estrutura de dados

A PCA-Tree proposta possui uma estrutura de dados diferenciada da KDB-Tree. A KDB-Tree é uma árvore n-ária onde cada nó pode ser representado como um conjunto de entradas filho, sendo que cada nó é armazenado em uma página da memória secundária. A PCA-Tree possui duas perspectivas do ponto de vista de estrutura de dados, uma como sendo uma árvore binária para navegação e acesso a memória principal; e outra como sendo uma árvore n-ária do ponto de vista de gerenciamento de memória secundária e balanceamento. Por ter estas duas perspectivas, a partir deste ponto do texto, teremos que diferenciar o termo nó da árvore binária e nó da árvore n-ária. Assim, quando nos referenciarmos ao termo **nó** estaremos nos referindo a um nó da árvore do ponto de vista binário e utilizaremos o termo **página** para nos referenciarmos a um nó da árvore do ponto de vista n-ário. Estes termos também serão expandidos para os termos conseqüentes, ou seja, uma página interna é um nó interno da árvore n-ária, um nó interno é um nó interno da árvore binária e assim por diante (vide Figura 5.2). Cada página da PCA-Tree contém uma sub-árvore binária local (Seção 2.1). Como na KDB-tree, cada página também possui um número máximo e mínimo de entradas, onde estas entradas representam nós da árvore binária ao invés de um acesso direto a páginas inferiores. Estes nós binários internos podem ter tanto filhos presentes na mesma página quanto presentes em páginas diferentes. Denominaremos **sub-árvore local** de uma página I a sub-árvore binária cujos nós estejam todos armazenados em I e incluindo os nós raizes das páginas imediatamente inferiores. Denominaremos **raiz local** de I, o nó da sub-árvore binária local armazenado em uma página I cujo pai está armazenado em uma página $I' \neq I$. Denominaremos de **folhas locais** de I os nós da sub-árvore local de I que não são armazenados em I, ou seja, o conjunto das folhas locais de I é formado pelas raizes das sub-árvores locais das páginas imediatamente inferiores a I. Uma página que contenha x entradas armazenadas, independente da configuração da sub-árvore binária local e de acordo com as propriedades de toda árvore binária, terá sempre $x + 1$ páginas filhos. Uma página não pode ter tamanho maior que m e menor que n entradas. A figura 5.2 ilustra esta estrutura de dados indicando a diferença entre um nó e uma página. Observe que, nesta figura, todas as páginas internas e a página raiz possuem uma mesma estrutura de sub-árvore binária local, entretanto, em uma PCA-tree, estas sub-árvores locais são independentes podendo

apresentar configurações diferentes.

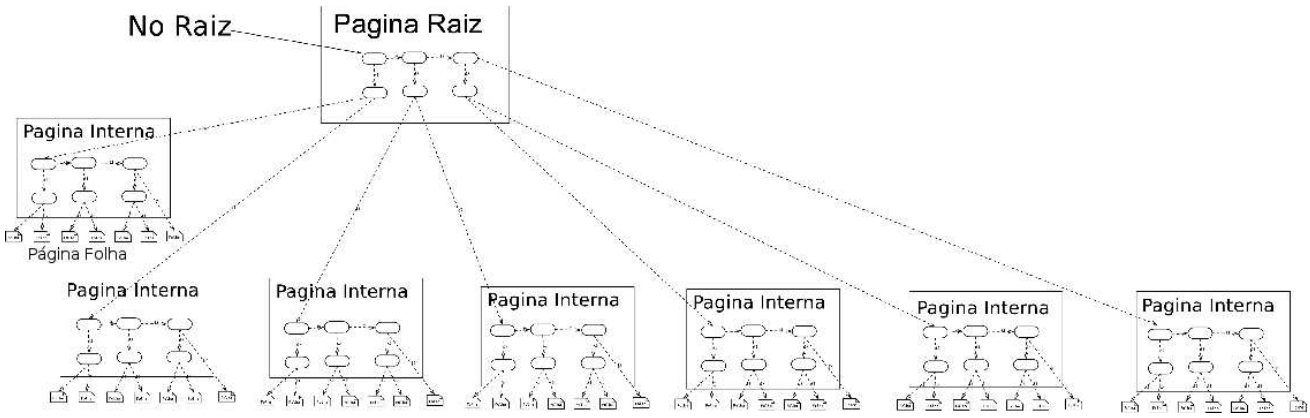

Figura 5.2: Exemplo da estrutura de uma PCA-Tree.

As páginas folha não representam uma subárvore da árvore binária. Suas entradas representam, diretamente, o ponto do espaço multidimensional que é utilizado como chave de acesso à informação. As páginas folha também possuem valores de m e n diferentes das páginas internas.

A estrutura de dados de uma página folha P da PCA-tree é resumida em uma lista de chaves de acesso(pontos em $\mathbb{R}^d$) e seus respectivos endereços de acesso à informação. O valor da centróide $\mathbf{c_P}$ e da quantidade de pontos $pointCount_P$ armazenados em P também são mantidos.

A estrutura de dados de uma página interna I da PCA-tree mantém a sub-árvore local, exceto os nós raizes das páginas descendentes. Os nós internos são armazenados em uma lista onde o primeiro nó da lista é a raiz da sub-árvore local. O valor da centróide $\mathbf{c_I}$ e da quantidade de pontos $pointCount_I$ armazenados nas páginas folha descendentes de I também são mantidos e persistidos em memória secundária. Diferente das páginas folhas, onde $\mathbf{c_P}$ e $pointCount_P$ são calculados, os valores de $\mathbf{c_I}$ e $pointCount_I$ são persistidos, evitando-se assim o acesso à informações em páginas descendentes.

Um nó interno $\mathcal{T}$ de uma página interna I possue as seguintes informações:

- Centróide $\mathbf{c}_\mathcal{T}$ - centróide aproximada de todos os pontos pertencentes ao subespaço representado pelo nó;
- Vetor Normal $\overrightarrow{v_\mathcal{T}}$ - vetor normal à componente principal que divide o subespaço representado pelo nó em dois subespaços, o subespaço do nó filho esquerdo e o subespaço do nó filho direito;
- Quantidade de pontos $pointCount_\mathcal{T}$ acessados através do nó;
- Filho da esquerda $Left_\mathcal{T}$ - ponteiro para o nó filho da esquerda do nó;
- Filho da direita $Right_\mathcal{T}$ - ponteiro para o nó filho da direita do nó.

Um nó folha $\mathcal{N}$ de uma página folha P possue as seguintes informações:

- Ponto do espaço multidimensional $\mathbf{k}_{\mathcal{N}}$ que representa a chave de acesso à informação satélite;
- Informação satélite ou endereço de acesso $Pr_{\mathcal{N}}$ à informação satelite.

5.2 Propriedades

A PCA-Tree deverá atender as propriedades de gerenciamento de memória secundária originários da B-Tree. Ou seja, do ponto de vista de árvore n-ária de páginas, a PCA-Tree deverá ser uma árvore na qual todos os caminhos entre uma página folha e a página raiz possuam um mesmo tamanho, sendo este tamanho medido em número de páginas acessadas pelo caminho. Todas as páginas internas devem possuir no máximo $m+1$ e no mínimo $n+1$ páginas filho, o que implica que todas as páginas internas armazenam no máximo m e no mínimo n entradas da árvore binária. A página raiz da árvore é a única que pode vir a ter menos que n entradas. As páginas folha também possuem uma quantidade mínima n e máxima m de nós folhas que deve ser atendida. Estes valores, entretanto, não precisam ser iguais aos valores das páginas internas. Todas as propriedades de uma árvore binária também estão embutidas na PCA-Tree por ela também definir esta forma de navegação. Assumiremos, a priori, a familiaridade com estas.

Nas páginas internas, os valores $\mathbf{c_I}$ e $pointCount_I$ devem ser atualizados sempre que houver uma inserção, exclusão ou alteração em I. Isto não ocorre no valor de $\mathbf{c}_{\mathcal{T}}$ e $\overrightarrow{v_{\mathcal{T}}}$, dado que estes definem o hiperplano que divide o subespaço em direita e esqueda. Entretanto, inserções, alterações ou exclusões que acessem um nó interno $\mathcal{T}$ deverão atualizar o valor de $pointCount_{\mathcal{T}}$. Nas páginas folhas, os valores $\mathbf{c_P}$ e $pointCount_P$ também devem ser atualizados sempre que houver uma inserção, exclusão ou alteração em P.

Os algoritmos apresentados a seguir se propõem a atender estas propriedades após a execução dos mesmos. Entretanto, o rebalanceamento da árvore de forma a atender a todas as propriedades descritas acima pode vir a ter um desempenho não aceitável. Desta forma, algumas páginas com números de entradas menor que n e alguns caminhos de altura diferente dos demais podem vir a ocorrer de forma a diminuir este grande custo de rebalanceamento.

5.3 Busca Exata

Uma busca exata por um ponto $\mathbf{p}$ é definida pela busca do caminho que leva ao nó folha que define o subespaço onde o ponto está contido. Dado um nó interno $\mathcal{T}$, um ponto $\mathbf{p}$ está contido no subespaço definido pelo filho esqueda de $\mathcal{T}$ caso a distância entre $\mathbf{p}$ e o hiperplano h, definido pelo vetor normal $\overrightarrow{v_{\mathcal{T}}}$ e centroide $\mathbf{c}_{\mathcal{T}}$, seja menor que zero. Caso contrário, o ponto $\mathbf{p}$ está contido no subespaço definido pelo filho direita de $\mathcal{T}$. Este cálculo

é feito recursivamente a partir do nó raiz até que se chegue a um nó folha $\mathcal{N}$ da árvore. Finalmente, basta verificar se o ponto $\mathbf{p}$ está presente na página folha P onde o nó folha $\mathcal{N}$ está contido. Pelo fato da PCA-tree ser uma árvore com gerenciamento de memória secundária, durante o processo de descida na árvore pode ser encontrado um ou mais nós descarregados, ou seja, sua informação ainda não foi carregada da memória secundária para a memória principal. Neste caso, o algoritmo deve executar uma rotina de leitura da página carregando as informações deste nó para a memória principal. Ao final da busca, todas as páginas lidas da memória secundárias são desalocadas da memória principal. Entretanto, podem ser aplicadas outras políticas de gerenciamento como definidas no capítulo 2.

5.4 Range-Query

Uma *Range Query*, ou busca em extensão, é definida como uma busca por todos os pontos presentes no índice que fazem intersecção com um subespaço $\mathbb{S} \in \mathbb{R}^d$. Este subespaço geralmente é definido como um hipercubo (neste caso, denomina-se a busca pelo termo *window query*). Como veremos a seguir, a representação de um subespaço por um hipercubo resulta em um algoritmo de ordem exponencial em d e, portanto, também apresentaremos algoritmos de busca por hiperesferas, superelipses e hiperelipses. Assim temos algoritmos de busca de ordem linear ou polinomial em d.

Um hipercubo H em $\mathbb{R}^d$ é definido por d intervalos $[min_i, max_i]$ onde $0 \leq i < d$. Denote de vértice do hipercubo H, qualquer ponto dado por $< P_1, P_2, ..., P_d >$ onde $P_i = min_i$ ou $P_i = max_i$ para todo $0 \geq i < d$. Desta forma, um hipercubo definido em $\mathbb{R}^d$ possui 2^d vértices. Dado um nó interno $\mathcal{T}$ e seu hiperplano h definido por $\mathbf{c}_\mathcal{T}$ e $\overrightarrow{v_\mathcal{T}}$, a intersecção entre o hipercubo e o subespaço definido pelo nó filho esquerda de $\mathcal{T}$ existe caso exista algum vértice $\mathbf{v}$ cuja distancia entre $\mathbf{v}$ e o hiperplano h seja menor que zero. Analogamente, a intersecção entre o hipercubo e o subespaço definido pelo nó filho direita de $\mathcal{T}$ existe caso exista algum vértice $\mathbf{v}$ cuja distancia entre $\mathbf{v}$ e h seja maior ou igual a zero. Assim, o algoritmo de cálculo de intersecção tem ordem $\theta(2^d)$.

Uma hiperesfera (capítulo 2) é definida pela equação característica:

$$\sum_{i=0}^{d-1} (x_i - c_i)^2 = R^2$$

Onde R é o raio da hiperesfera e c_i são os valores das componentes do centro $\mathbf{c}$ da hiperesfera.

Dados R, $\mathbf{c}$, $\mathcal{T}$ e o hiperplano h definido por $\mathbf{c}_\mathcal{T}$ e $\overrightarrow{v_\mathcal{T}}$, a intersecção entre a hiperesfera e o filho esquerda de $\mathcal{T}$ existe caso a distância *dist* entre $\mathbf{c}$ e o hiperplano h seja menor que zero ou o módulo de *dist* seja menor que o raio. Análogamente, a intersecção entre a hiperesfera e o filho da direita existe caso a distância *dist* entre $\mathbf{c}$ e o hiperplano h seja maior ou igual a zero ou o módulo de *dist* seja menor que o raio. Este algoritmo proposto

para o cálculo de intersecção tem ordem de crescimento $\theta(d)$.

Os algoritmos de decisão de decida na árvore para uma superelipse ou para uma superelipse rotacionada funcionam de forma análoga ao algoritmo de intersecção da hiperesfera diferenciando apenas na equação característica entre a superelipse, superelipse rotacionada e a hiperesfera. A ordem de crescimento do algoritmo de intersecção para a superelipse é igual a $\theta(d)$. A ordem de crescimento do algoritmo de intersecção para uma superelipse rotacionada é de $\theta(d^2)$ devido a múltiplicação pela matriz dXd de rotação.

5.5 KNN-Query

A busca pelos k vizinhos mais próximos (capítulo 2) consiste em, a partir de um ponto $\mathbf{p}$, encontrar o menor valor de raio para uma hiperesfera centrada em $\mathbf{p}$ que faça intersecção com no mínimo k pontos presentes no índice. Um algoritmo intuitivo é, partindo do valor de raio igual a zero, crescer este valor de raio até que a hiperesfera englobe k vizinhos. Para cada valor de raio definido será realizada uma Range-Query no índice. A questão é, qual a taxa de crescimento do valor do raio em cada passo do algoritmo?

Uma outra abordagem seria estimar um raio inicial. Após esta estimativa, faz-se uma contagem de pontos internos à hiperesfera definida por este raio e caso este valor seja maior que k o raio deverá diminuir de tamanho, caso contrário deverá aumentar de tamanho. Uma estimativa inicial simples seria, dada a página raiz da árvore, escolher a menor dentre as distâncias formadas pelo ponto de busca e os centróides dos filhos diretos da página raiz da árvore. Esta abordagem seria top-down (de cima para baixo), ou seja, partindo da raiz em direção as folhas. Uma abordagem bottom-up (de baixo para cima) seria uma busca inicial pelo nó folha $\mathcal{N}$ onde o ponto se encontra e o uso da distância entre o ponto de busca e o centróide do pai de $\mathcal{N}$ como raio inicial. A medida que a hiperesfera definida por este raio conter menos que k pontos, crescer o raio pela distância entre o ponto de busca e o centróide do próximo pai, até que finalmente chegue na raiz da árvore.

Seja $\mathcal{N}$ um nó interno inicialmente configurado como a raiz da árvore; seja $\mathbf{c}_{\mathcal{N}}$ o centróide deste nó; seja $Right(\mathcal{N})$ o nó filho direito de $\mathcal{N}$; seja $Left(\mathcal{N})$ o nó filho esquerda de $\mathcal{N}$; seja $Pagina(\mathcal{N})$ a página onde $\mathcal{N}$ está contido; seja $h_{\mathcal{N}}$ o hiperplano definido pelo ponto $\mathbf{c}_{\mathcal{N}}$ e o vetor normal $\overrightarrow{v_{\mathcal{N}}}$ de $\mathcal{N}$; seja L uma lista de páginas folha, inicialmente vazia, acessadas pela busca; seja L_h uma lista de hiperplanos, inicialmente vazia; seja L_r uma lista de pontos que armazena os k vizinhos mais próximos retornados pelo algoritmo; seja S uma hiperesfera de raio inicialmente igual a 0 e centro em $\mathbf{p}$; seja k o número de vizinhos mais próximos que deseja-se encontrar; seja i um contador de vizinhos mais próximos encontrados. A abordagem implementada para busca KNN neste trabalho consiste em:

1. Caso $\mathcal{N}$ seja folha, adicione $Pagina(\mathcal{N})$ em L;

2. Caso $\mathcal{N}$ seja nó interno:

(a) Caso S interceda $h_{\mathcal{N}}$, adicione $h_{\mathcal{N}}$ em L_h, recorra a partir do item 1 para $\mathcal{N} = Right(\mathcal{N})$ e $\mathcal{N} = Left(\mathcal{N})$;

(b) Caso $\mathbf{p}$ esteja à direita de $h_{\mathcal{N}}$ recorra a partir do item 1 para $\mathcal{N} = Right(\mathcal{N})$;

(c) Caso $\mathbf{p}$ esteja à esquerda de $h_{\mathcal{N}}$ recorra a partir do item 1 para $\mathcal{N} = Left(\mathcal{N})$;

3. Ache o ponto $\mathbf{x}$ mais próximo de S dentre os pontos em L e ache o hiperplano h' mais próximo de S dentre os hiperplanos em L_h;

4. Caso a distância entre S e $\mathbf{x}$ seja maior que a distância entre h' e S, faça o raio de S igual ao raio de S acrescentado da distância entre S e h', faça $\mathcal{N}$ igual ao nó interno que define h' e recorra ao item 1;

5. adicione $\mathbf{x}$ em L_r e faça o raio de S igual ao raio de S acrescido do valor da distância entre S e $\mathbf{x}$;

6. Caso o tamanho de L_r seja menor que k recorra ao item 3;

Este algoritmo inicia com o raio de S igual a 0 e possui um caminho único entre a raiz e um nó folha até chegar no item 3. Neste item, pode haver um ou mais hiperplanos dentro deste caminho que possuam uma distância a S menor que qualquer distância a pontos na folha. Isto indica que há a probabilidade de haver algum outro ponto mais próximo de S "do outro lado"destes hiperplanos. Assim, haverá a necessidade de buscar outros caminhos além do caminho inicial, o que é indicado pelo item 4. O raio de S é acrescido a medida que vão se encontrando pontos ou hiperplanos mais próximos até que S englobe k pontos.

5.6 Algoritmo de Inserção

O algoritmo de inserção de um objeto com valor de chave de acesso representado por um ponto $\mathbf{p}$ em $\mathbb{R}^d$ consiste em:

1. Fazer uma busca exata pela página folha P que representa o subespaço onde o ponto $\mathbf{p}$ está contido;

2. Inserir um novo nó folha em P com chave de acesso $\mathbf{p}$ e referência ao objeto;

3. Seja C o conjunto de páginas que definem o caminho entre a página raiz e P, então, ajuste o valor do centróide e número de pontos de todas as páginas pertencentes a C;

4. Verificar se houve estouro de página em P;

5. Caso tenha ocorrido estouro de página, aplicar o algoritmo de quebra em P para rebalanceamento da árvore.

5.7 Algoritmo de Quebra de página folha

A quebra de uma página folha P dá-se através dos passos:

1. Através do método PCA aplicado sobre os pontos presentes em P, encontrar a componente principal $\vec{\mathbf{v}}$ que define o vetor normal ao hiperplano h que dividirá estes pontos em dois conjuntos;
2. Criar uma nova página E;
3. Seja $\mathbf{c_P}$ o valor da centróide dos pontos presentes em P, então, para cada ponto $\mathbf{p}'$ presente em P cuja distancia até h for menor que zero, mover seu nó correspondente para E;
4. Recalcular os valores de centróide e contagem de pontos em P e E;
5. Crie um novo nó interno $\mathcal{N}$, faca $c_{\mathcal{N}} = c_P$ e $pointCount_{\mathcal{N}} = pointCount_P + pointCount_E$;
6. Definir o primeiro nó folha em P como filho direita de $\mathcal{N}$;
7. Definir o primeiro nó folha em E como filho esquerda de $\mathcal{N}$;
8. Caso P seja raiz, crie nova página raiz R inserindo $\mathcal{N}$ em R. Ajuste os valores de página pai de P e E para R, faça $c_R = c_{\mathcal{N}}$, faça $pointCount_R = pointCount_{\mathcal{N}}$ e termine.
9. Caso contrário, seja T a página pai de P, então, insira $\mathcal{N}$ em T;
10. Ajuste o valor de página pai de E para T;
11. Seja $\mathcal{E}$ o primeiro nó de P, seja $PAI(\mathcal{E})$ o nó pai de $\mathcal{E}$, então, caso $\mathcal{E}$ seja filho da esquerda de $PAI(\mathcal{E})$, faça o pai de $\mathcal{N}$ ser $PAI(\mathcal{E})$ e o filho da esquerda de $PAI(\mathcal{E})$ ser $\mathcal{N}$, caso contrário, faça o pai de $\mathcal{N}$ ser $PAI(\mathcal{E})$ e o filho da direita de $PAI(\mathcal{E})$ ser $\mathcal{N}$;
12. Caso haja estouro de página em T, aplicar o algoritmo de quebra de página interna em T;
13. Termine.

5.8 Algoritmo de Quebra de página interna

O algoritmo de quebra de uma página interna $\mathcal{I}$ pode ser resumido pelos passos:

1. Faça $\mathcal{I}'$ uma cópia completa de $\mathcal{I}$;

2. Seja C o conjunto de pontos formado pelas centróides dos nós folhas locais de $\mathcal{I}$.
3. Aplique o método PCA tendo C como conjunto de amostras e obtenha a componente principal $\vec{\mathbf{v}}$;
4. Faça $\mathbf{c}$ a centróide das amostras presentes em C;
5. Faça h o hiperplano definido pelo vetor normal $\vec{\mathbf{v}}$ e centróide $\mathbf{c}$.
6. Exclua de I' os pontos cuja distância até h seja menor que zero.
7. Exclua de I os pontos cuja distância até h seja maior ou igual a zero.
8. Corte os ramos vazios de I' e I;
9. Criar um novo nó $\mathcal{N}$;
10. Definir o nó raiz local de I' como filho direita de $\mathcal{N}$;
11. Definir o nó raiz local de I como filho esquerda de $\mathcal{N}$;
12. Caso I seja raiz, criar nova página raiz R e inserir $\mathcal{N}$ em R. Ajuste os valores de página pai de I e I' para R e termine.
13. Caso contrário, seja P a página pai de I, então, insira $\mathcal{N}$ em P;
14. Ajuste o valor de página pai de I para R;
15. Seja $\mathcal{E}$ o primeiro nó de I, seja $PAI(\mathcal{E})$ o nó pai de $\mathcal{E}$, então, caso $\mathcal{E}$ seja filho da esquerda de $PAI(\mathcal{E})$, faça o pai de $\mathcal{N}$ ser $PAI(\mathcal{E})$ e o filho da esquerda de $PAI(\mathcal{E})$ ser $\mathcal{N}$, caso contrário, faça o pai de $\mathcal{N}$ ser $PAI(\mathcal{E})$ e o filho da direita de $PAI(\mathcal{E})$ ser $\mathcal{N}$;
16. Caso haja estouro de página em P, aplicar o algoritmo de quebra de página interna em P;
17. Termine.

O algoritmo acima descrito exige uma grande quantidade de memória, já que este faz uma cópia completa de toda a árvore contida em I. Esta grande quantidade de alocação de memória também implica em um fraco desempenho do algoritmo.

Para solucionarmos este fraco desempenho, propõe-se um algoritmo de quebra recursivo de baixo para cima. Assim, após calcular-se $\vec{\mathbf{v}}$ e $\mathbf{c}$ que determina o hiperplano que dividirá a sub-árvore local, realiza-se a quebra inicialmente nas folhas da árvore, subindo em direção a raiz da sub-árvore local. Desta forma, evita-se a criação de páginas intermediárias desnecessárias.

A Figura 5.3 exemplifica o resultado estrutural da aplicação do algoritmo de quebra. Nesta, temos em 1, o esboço estrutural antes da quebra e, em 2), o esboço estrutural

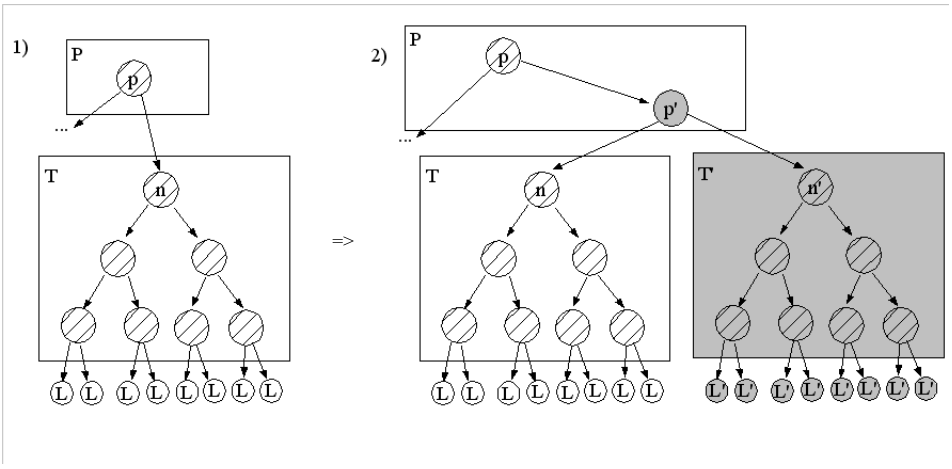

Figura 5.3: Resultado Estrutural da quebra na PCA-tree

depois da quebra. Seja n o nó a ser quebrado. Seja T a página a qual n pertence. Seja p o nó pai de n. Seja P a página a qual o p pertence. Seja L as páginas descendentes de T. Seja h o hiperplano que irá condicionar a quebra. Então, após a quebra teremos a criação de T', $L's$, n' e p'. Estes elementos são representados com preenchimento, em 2, para indicar que são as novas estruturas criadas e alocadas na memória secundária. Após a quebra, os pontos à "direita"do hiperplano h serão acessados por T, enquanto os pontos à "esquerda"do hiperplano h serão acessados por T'.

5.8.1 Algoritmo de quebra condicionada de nó

Dado a página inicial P, o nó interno $\mathcal{N}$ raiz da subárvore de P, e os valores de c e v como centróide e vetor normal que definem o hiperplano h, a quebra condicionada realiza quebra da sub-árvore de $\mathcal{N}$ em duas novas árvores separando os pontos de acordo com sua posição relativa a h. Esta é feita partindo das folhas da subárvore de $\mathcal{N}$ até $\mathcal{N}$. Ou seja, o algoritmo de quebra é inicialmente aplicado aos filhos de $\mathcal{N}$ e, em seguida, são feitas as modificações em relação a $\mathcal{N}$ necessárias na árvore. O resultado do algoritmo é equivalente ao apresentado na seção 5.7. A diferença entre uma quebra condicionada e uma quebra (não condicionada) é que na quebra condicionada o hiperplano h é passado como parâmetro, enquanto que na quebra o hiperplano h é calculado.

5.9 Algoritmos de Exclusão

O algoritmo de exclusão de um objeto com valor de chave de acesso representado por um ponto $\mathbf{p}$ em $\mathbb{R}^d$ consiste em:

1. Fazer uma busca exata por $\mathbf{p}$.

2. Caso a busca não encontre um ponto correspondente presente no índice, terminar.

3. Caso contrário, seja T a página folha que contém $\mathbf{p}$ e seja C o conjunto de páginas no caminho de busca entre a raiz e T.

4. Excluir de T a entrada correspondente a $\mathbf{p}$.

5. Ajustar o valor do centróide e número de pontos de todas as páginas e nós internos pertencentes a C;

6. Se o número de entradas de T for menor que n, aplicar algoritmo de mesclagem para rebalanceamento da árvore.

5.10 Algoritmos de mesclagem de páginas folha

Um página folha $L1$ possui um nó pai $\mathcal{P}$ presente em uma página P, pai de $L1$. O nó $\mathcal{P}$ possui dois filhos, dentre eles $L1$. O outro filho corresponde à página $L2$ a qual $L1$ será mesclada. Portanto, a mesclagem de dois nós folhas resume-se na união dos pontos de $L1$ e $L2$ em um novo conjunto L. Após esta união, caso L tenha tamanho maior que m, haverá a necessidade de redivisão de L e dois novos nós $L1'$ e $L2'$. A redivisão é equivalente ao algoritmo de quebra de um nó folha.

5.11 Algoritmos de mesclagem de páginas internas

Um página interna $I1$ possui um nó pai $\mathcal{P}$ presente em uma página P, pai de $I1$. O nó $\mathcal{P}$ possui dois filhos, dentre eles $I1$. O outro filho corresponde á página $I2$ a qual $I1$ será mesclada. Portanto, a mesclagem de dois nós internos resume-se a união dos pontos de $I1$ e $I2$ e a reindexação destes pontos. Dado que uma página interna I possui a quantidade de pontos na ordem de n^h, onde n é o número mínimo de entradas e h é a altura de I, a simples união e reindexação resulta em um algoritmo de baixo desempenho.

Um algoritmo de melhor desempenho é dado pela escolha de $I1$, dentre $I1$ e $I2$, e condicionamento da subárvore de $I2$ à subárvore presente em $I1$. Para uma melhor explicação assuma $\mathcal{N}\infty$ o nó raiz da subárvore de $I1$ e assuma $\mathcal{N}\in$ o nó raiz de $I2$. A partir daí execute os passos:

1. Quebre $I2$ condicionado ao hiperplano definido por $\mathcal{N}\infty$;

2. faça uma chamada recursiva ao item 1 passando $\mathcal{N}\infty$ igual ao filho esquerda de $\mathcal{N}\infty$ e $\mathcal{N}\in$ igual à subárvore esquerda gerada no item 1;

3. faça uma chamada recursiva ao item 1 passando $\mathcal{N}\infty$ igual ao filho direita de $\mathcal{N}\infty$ e $\mathcal{N}\in$ igual à subárvore direita gerada no item 1;

4. Quando ambos $\mathcal{N}\infty$ e $\mathcal{N}\in$ forem folhas faça uma mesclagem de nós folhas
5. Quando $\mathcal{N}\infty$ for uma folha, insira os pontos de $\mathcal{N}\infty$ em $\mathcal{N}\in$
6. Quando $\mathcal{N}\in$ for uma folha, insira os pontos de $\mathcal{N}\in$ em $\mathcal{N}\infty$

O algoritmo acima tem uma abordagem de baixo para cima, ou seja, a informação local nos níveis mais baixos são reajustados antes dos níveis mais altos da árvore. Esta abordagem diminui a quantidade de reajustes. Dado que em níveis mais altos o número de pontos indexados envolvidos é maior, esta abordagem melhora o desempenho do algoritmo.

5.12 PCA-RTree: uma extensão da PCA-Tree

A PCA-Tree divide um espaço de busca em dois semi-espaços através de um hiperplano. Numa busca por extensão, se o espaço de busca interceptar algum semi-espaço, é necessário prosseguir a busca nesse semi-espaço. Muitas vezes, porém, os elementos indexados que estão nesse semi-espaço não fazem interseção com o espaço de busca. Porém isso só pode ser constatado ao se atingir uma página folha ou ao se atingir um semi-espaço com o qual o espaço de busca não faz intersecção. Para evitar percorrimento nesses subespaços, pode-se, em cada semi-espaço gerado pela PCA-Tree, armazenar também o MBR dos pontos pertencentes a esse semi-epaço. Desta forma, ao se verificar se o espaço de busca intercepta o MBR, pode-se reduzir significativamente o número de caminhos percorridos em uma busca em extensão. Uma segunda consideração é que apesar de MBRs assim construídos poderem se interseptar entre si em um mesmo nível da árvore, o hiperplano que divide as regiões elimina esta interseção. Esta extensão da PCA-Tree denominou-se PCA-RTree.

Capítulo 6

Implementações

No decorrer deste trabalho foram implementados os índices KDB-tree, R-tree e PCA-tree e suas operações de inserção, exclusão, busca exata, busca em extensão e busca pelos vizinhos mais próximos. Além destes índices, também foram implementadas interfaces gráficas que oferecem uma forma didática de exibição do funcionamento destes em um espaço $\mathbb{R}^2$.

Estas implementações foram feitas utilizando a linguagem java e estão disponíveis em `http://www.vision.ime.usp.br/~philipe`.

6.1 Índices e Estruturas de Dados

Os três tipos de índice são instanciados através do método Btree. Exemplo:

```
Btree tree = new Btree(min, max,dimensionality),
```

onde, `min` representa o número mínimo de entradas em uma página, `max` representa o número máximo de entradas em uma página e `dimensionality` representa o número de dimensões de seu índice.

O tipo de índice a ser instanciado por este método (KDB-tree, R-tree ou PCA-tree) só depende do pacote ".jar" importado na cláusula "import". Para isso são oferecidos três pacotes: "pcatree.jar", "kdbtree.jar" e "rtree.jar". O objeto instanciado `tree` oferece os métodos:

- `void insert(Point p, Object o)`

  O método `insert` insere um novo ponto `p` no índice tendo o objeto `o` como informação satélite.

- `void delete(Point p)`

  O método `delete` exclui do índice as entradas correspondentes ao ponto `p`.

- `void deleteNearestNeighbor(Point p)`

  O método `delete` exclui do índice as entradas correspondentes ao ponto mais próximo de `p`.

- `LinkedList rangeQuery(Hipercube h)`

  O método `rangeQuery` executa a busca em extensão por todos os pontos presentes no índice que são internos ao hipercubo `h`, retornando uma lista encadeada das entradas correspondentes a estes pontos.

- `LinkedList knnQuery(Point p, int k)`

  O método `knnQuery` executa a busca pelos `k` vizinhos mais próximos de `p` presentes no índice, retornando uma lista encadeada das entradas correspondentes a estes pontos.

As estruturas de dados utilizadas estão todas presentes em código fonte no arquivo ".jar" correspondente ao tipo de árvore a ser escolhido. Dentre estas estruturas de dados podemos citar: hipercubos, hiperplanos, pontos, vetores, entradas internas, entradas folha, páginas internas, páginas folha, etc. O código foi organizado de forma que cada uma destas estruturas estejam codificadas em arquivos ".java" separados.

6.2 Interfaces Gráficas

As interfaces gráficas implementadas exemplificam o funcionamento dos índices implementados em um espaço $\mathbb{R}^2$ normalizado no intervalo $[0, 1]$. Nestas são oferecidas funcionalidades de: criação de novo índice, inserção de um conjunto de x pontos gerados aleatoriamente, inserção de pontos de acordo com a posição do *mouse*, exclusão de x pontos aleatoriamente selecionados, exclusão de pontos de acordo com a posição do *mouse*, busca em extensão de acordo com uma janela de tamanho e posição controlados pelo *mouse*, busca pelos k vizinhos mais próximos a um ponto p definido pelo *mouse*, salvar índice em memória secundária e carregar índice a partir da memória secundária. Além destas funcionalidades é disponibilizado uma visualização espacial em $\mathbb{R}^2$ contendo os pontos e hipercubos internos aos índices e uma visualização da estrutura hierárquica do índice.

6.2.1 Interface KDB-tree e R-tree

Na interface tanto da KDB-tree quanto da R-tree, ao se selecionar nós na estrutura de dado hierárquica, os correspondentes hipercubos são realçados na visualização espacial. Para isto, basta selecionar quais os nós desejados e, estes, serão realçados na visualização espacial do índice. Além disso, pode-se obter informações sobre a estrutura de dados do nó selecionado como: a quatidade de nós filhos, o fator de preenchimento, o nome do

arquivo que persiste o nó, o hipercubo que engloba o nó, etc. As consultas em extensão e de vizinhos mais próximos estão separadas em dois *tabsheets* correspondentes e com títulos intuitivos.

Na figura 6.1 temos uma captura de tela da interface gráfica implementada para a KDB-tree. Na figura 6.2 temos uma captura de tela da interface gráfica implementada para a R-tree. Nestas, podemos ver a estrutura hierárquica do índice à esquerda e a visualização espacial dos pontos e hipercubos à direita. Esta disponibiliza também 3 *tabsheets*, um para consultar detalhes sobre o nó selecionado, um para realizar consultas em extensão e outro para realizar a consulta de vizinhos mais próximos.

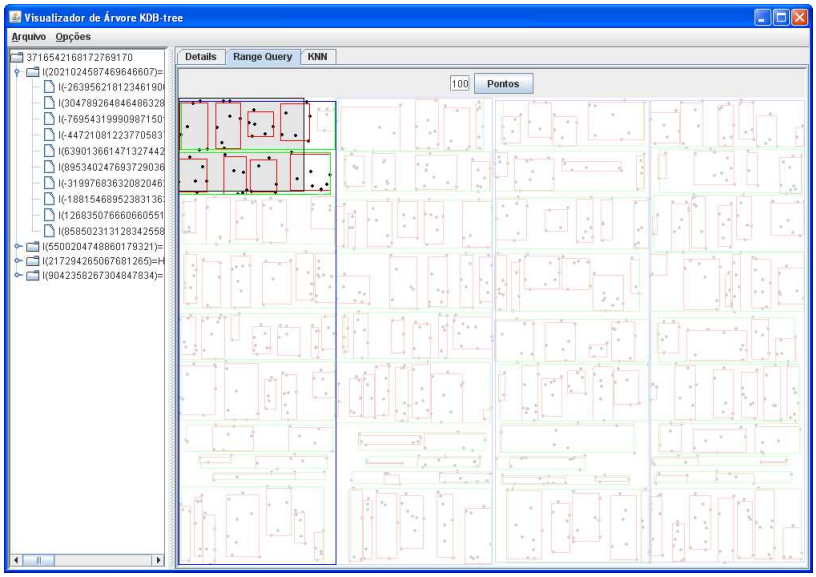

Figura 6.1: Interface gráfica para a KDB-Tree.

6.2.2 Interface PCA-tree

A interface gráfica para a PCA-tree oferece, além das funcionalidades presentes na interface gráfica da KDB-tree e R-tree, uma forma de navegação binária nos nós. Ou seja, a visualização hierárquica pode ser feita através da árvores de páginas da PCA-tree ou através da árvore binária de nós da PCA-tree. Outra diferença é que na PCA-tree não existe a definição de hipercubos. Para substituir a visualização de hipercubos utilizamos a visualização do diagrama de Voronoi [2] dos pontos presentes em uma página folha. Observamos, no entanto, que estes diagramas de Voronoi não estão representados estruturalmente na PCA-tree e são exibidos apenas para facilitar a visualização do efeito da segmentação do espaço realizada pelo índice. Na visualização da consulta dos vizinhos mais próximos não é disponibilizada a visualização dos pontos escondidos, ou seja, aqueles que pertencem a páginas folhas que não fazem interseção com o subespaço de busca. A figura 6.3 ilustra esta interface. Nesta podemos notar a presença do *tabsheet* nomeado *binary tree* que dá acesso à navegação binária na PCA-tree.

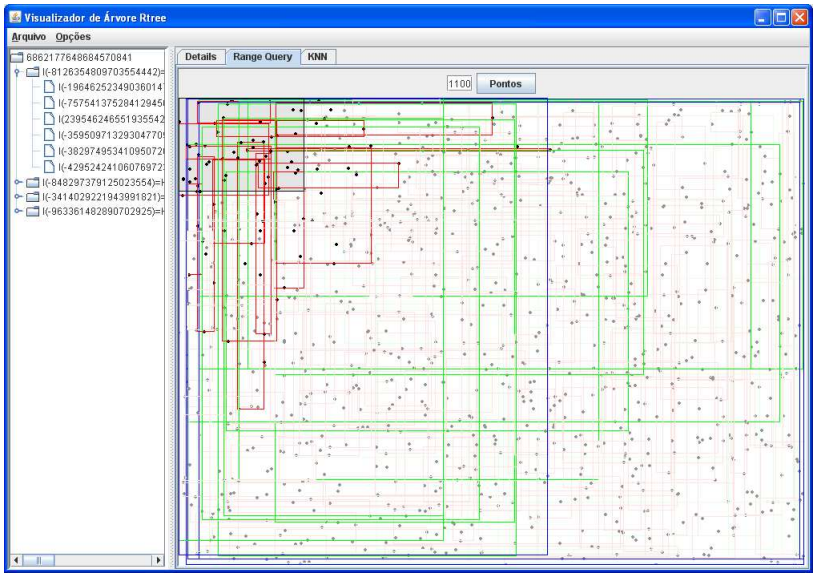

Figura 6.2: Interface gráfica para a R-Tree.

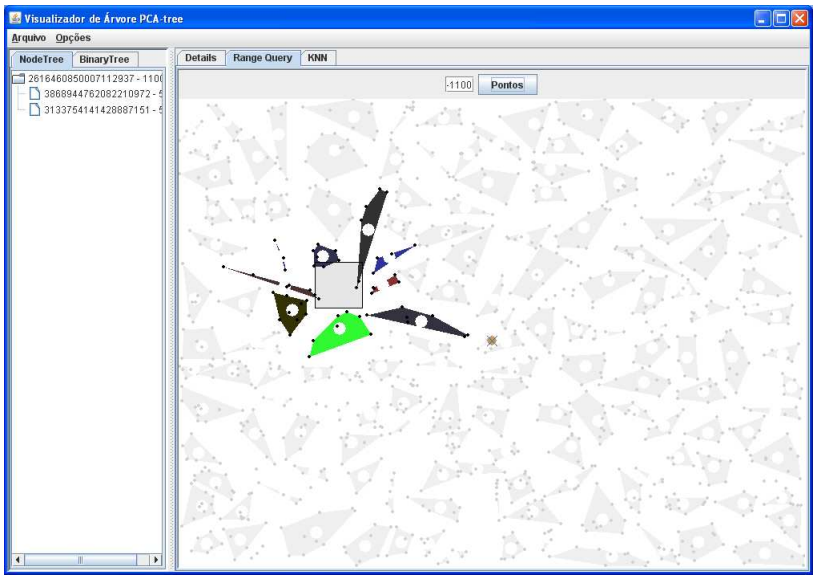

Figura 6.3: Interface gráfica para a PCA-Tree.

6.3 Interface exemplo para PCA-tree de Alta Dimensionalidade

Para exemplificar a utilização da PCA-tree em um espaço de busca $\mathbb{R}^d$ de alta-dimensionalidade, $d > 25$, implementamos uma aplicação para indexação de janelas convoluidas em uma imagem. Seja W_p uma janela de tamanho de 5 pixeis por 5 pixeis centrada no pixel p em uma imagem. Para todo pixel p pertencente a imagem, criamos um vetor de características de dimensionalidade igual a 75 dimensões, cujos valores de

cada dimensão representam os valores HSV [32] dos 25 pixeis da janela W_p centrada em p. A aplicação insere estes vetores em uma PCA-tree e permite uma visualização da partição realizada pelo índice, além da realização de buscas de vizinhos mais próximos e buscas em extensão. Esta aplicação, a princípio, não tem nenhuma utilização prática. Entretanto, ela permite uma avaliação da aplicabilidade da PCA-tree para espaços de alta-dimensionalidade.

Dentre as funcionalidades disponibilizadas estão: carregar(indexar) uma imagem selecionada, navegar pelas páginas e nós visualizando o conjunto de pontos que estes representam na imagem, realizar buscas de vizinhos mais próximos e buscas em extensão na PCA-tree criada. A seguir, apresentaremos uma breve explicação de cada um dos *tabsheets* disponibilizados na interface.

- Imagem

  Possibilita a visualização da imagem juntamente com a máscara gerada a partir dos nós/páginas selecionadas. Na figura 6.4 temos a exibição deste *tabsheet* pela interface. A região preta corresponde aos pixels na página selecionada à esquerda.

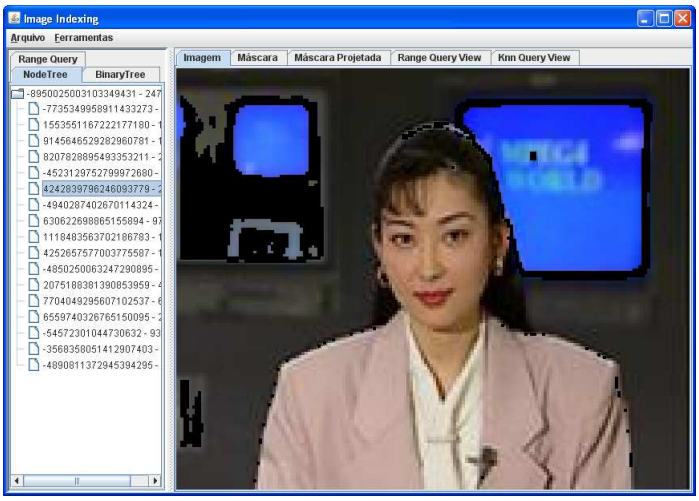

Figura 6.4: *Tabsheet* Imagem.

- Máscara

  A máscara é formada pelo conjunto de pixeis que representam os pontos pertencentes aos nós/páginas selecionados na visualização hierárquica da árvore. A máscara é desenhada em cor preta, enquanto os pontos que não pertencem a máscara permanecem em cor branca. Na figura 6.5 temos a exibição deste *tabsheet* pela interface.

- Máscara Projetada

  A máscara projetada é formada pelo conjunto de nós/páginas selecionados. Na visualização da máscara projetada, os pixeis que não pertencem a nenhum dos nós

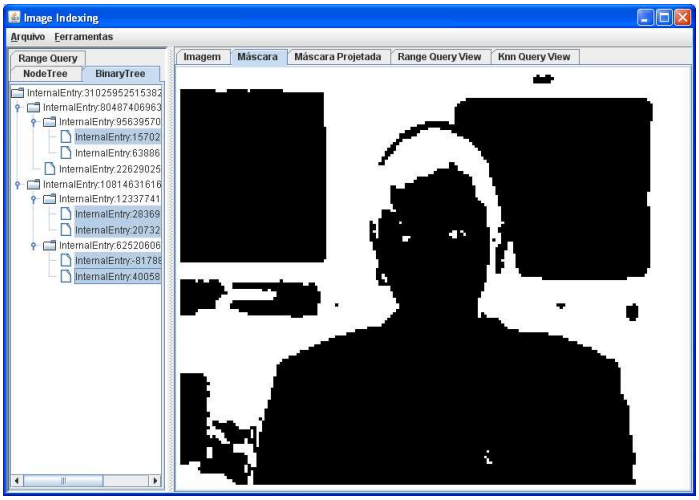

Figura 6.5: *Tabsheet* Máscara.

selecionados permanecem em cor branca. Os pixeis que pertencem a máscara projetada são desenhados com a cor definida pelo centróide do ancestral selecionado que possui menor profundidade. Seja um pixel p pertencente a um nó N selecionado que não possua ancestrais selecionados. O centróide de N define a cor na qual p será desenhado na tela. Na figura 6.6 temos a exibição deste *tabsheet* pela interface.

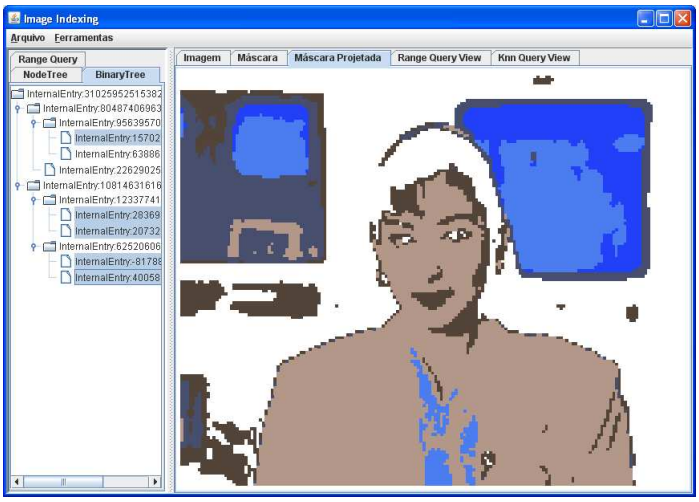

Figura 6.6: *Tabsheet* Máscara Projetada.

- Range Query View

  Neste *tabsheet* são desenhados apenas os pixeis que representam pontos que atendem a uma determinada busca em extensão realizada sobre o índice. A busca em extensão é definida no *tabsheet* "Range Query". A definição da hiperelipse para a busca em extensão se dá através da seleção de um ponto de referência como centróide e da seleção dos raios da hiperelipse. figura 6.7 temos a exibição deste *tabsheet* pela

interface.

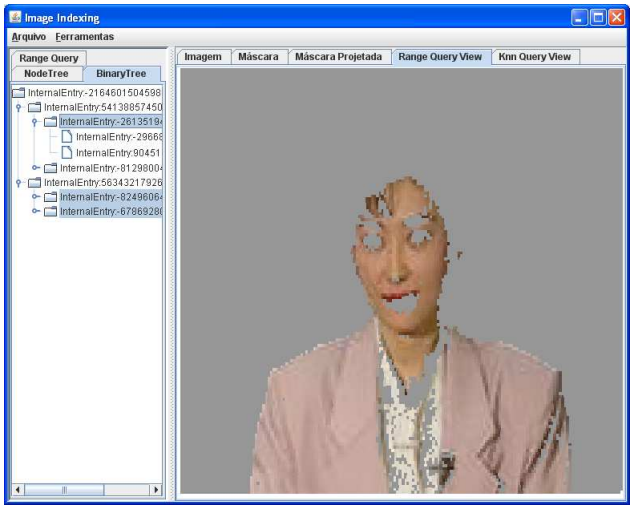

Figura 6.7: *Tabsheet* Range Query View.

- Knn Query View

  Neste *tabsheet* são exibidos apenas os pixeis que representam pontos que atendem a uma determinada busca por vizinhos mais próximos. Para definir esta busca, também neste *tabsheet*, utiliza-se o botão "Reset" para reiniciar a busca, o *mouse* para definir o ponto **p** de referência para a busca, os botões "Previous"e "Next"para procurar pelos prévios ou próximos k vizinhos mais próximos. O valor de k é definido em uma caixa de texto também presente no *tabsheet*. Na figura 6.8 temos a exibição deste *tabsheet* pela interface.

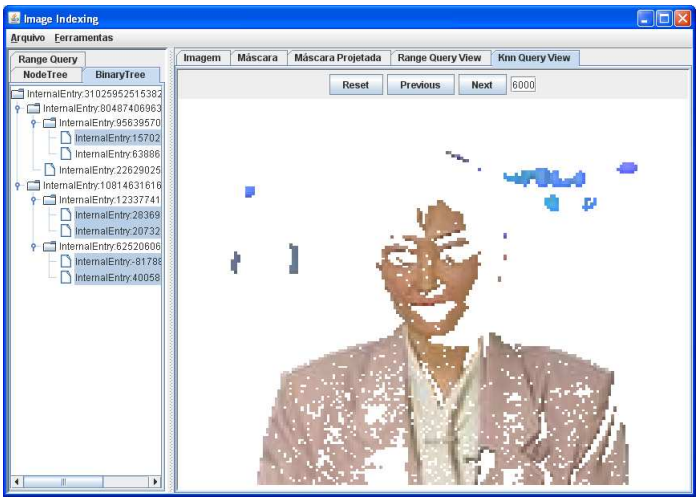

Figura 6.8: *Tabsheet* Knn Query View.

Capítulo 7

Resultados

Apresentaremos neste capítulo alguns resultados experimentais comparativos entre a R-tree, SR-tree, R+tree e PCA-tree (ou PCA-RTree). Não consideramos a KDB-tree pois sabe-se que a R-tree é superior a ela. No caso da PCA-tree e R-tree foram utilizadas a nossa implementação. Já para a SR-tree e R+-tree, foram utilizadas as implementações disponíveis em `http://www.comp.nus.edu.sg/~cuibin/research.htm`. Para todos os resultados mostrados abaixo utilizamos um valor de 50 entradas como número máximo de entradas por página e 25 entradas como número mínimo de entradas por página. Dividimos a análise em três subseções. A primeira mostra o resultado do desempenho dos algoritmos sobre dados uniformemente distribuídos. Na segunda, apresentamos o resultado do desempenho dos mesmos algoritmos aplicados sobre dados reais, ou seja, com distribuição não uniforme. Na terceira seção, discutimos os resultados obtidos.

7.1 Dados Uniformemente Distribuídos

Na figura 7.1 temos o resultado, em milisegundos, da criação da árvore seguida da inserção de 10000 pontos em $\mathbb{R}^d$ gerados aleatoriamente e uniformemente distribuídos. No eixo Y temos o tempo em milisegundos, no eixo X temos o número de dimensões d, variando de 2 a 25. O tempo envolve a operação de salvamento das páginas em memória secundária.

Na figura 7.2 temos o resultado, em número de páginas acessadas (*pageCount*), da execução de buscas em extensão de 100 hipercubos em $\mathbb{R}^d$ de tamanho de lado fixo e igual a 0.2 e centróides gerados aleatoriamente e uniformemente distribuidos em $\mathbb{R}^d$. As buscas são realizadas sobre um conjunto de pontos, também uniformemente distribuidos, de tamanho igual a 100000 pontos. No eixo Y temos o número de páginas acessadas, no eixo X temos o número de dimensões, variando de 2 a 25 dimensões.

Na figura 7.3 temos o resultado, em número de páginas acessadas (*pageCount*), da execução de 100 buscas pelos 5 vizinhos mais próximos a pontos gerados aleatoriamente e uniformemente distribuídos em $\mathbb{R}^d$. As buscas são realizadas sobre um conjunto de pontos, também uniformemente distribuídos, de tamanho igual a 100000 pontos. No eixo Y temos

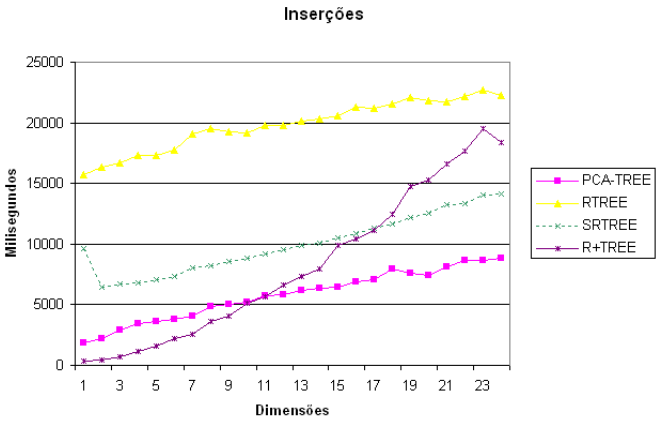

Figura 7.1: Tempo (ms) para 10000 inserções de pontos uniformemente distribuídos em $\mathbb{R}^d$.

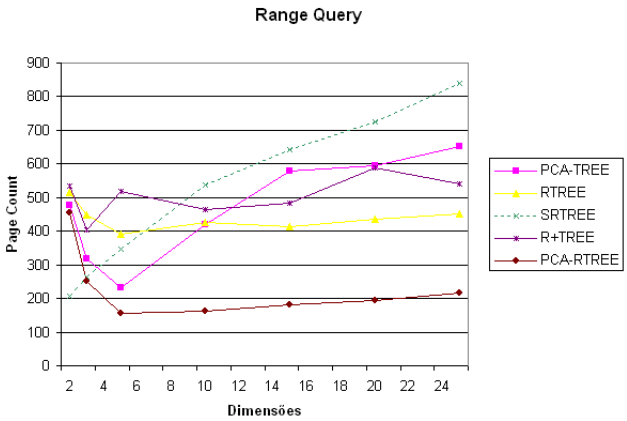

Figura 7.2: Comparação para *range queries* sobre pontos uniformemente distribuídos.

o número de páginas acessadas, no eixo X temos o número de dimensões, variando de 2 a 25 dimensões.

Na figura 7.4 temos o resultado, em número de páginas acessadas (*pageCount*), da execução de 100 buscas pelos k vizinhos mais próximos a pontos gerados aleatoriamente e uniformemente distribuídos em $\mathbb{R}^2$. As buscas são realizadas sobre um conjunto de pontos, também uniformemente distribuidos, de tamanho igual a 100000 pontos. No eixo Y temos o número de páginas acessadas, no eixo X temos o valor de k, variando de 5 a 100 vizinhos. Neste gráfico comparamos apenas o desempenho da PCA-tree,R-tree e R+-tree.

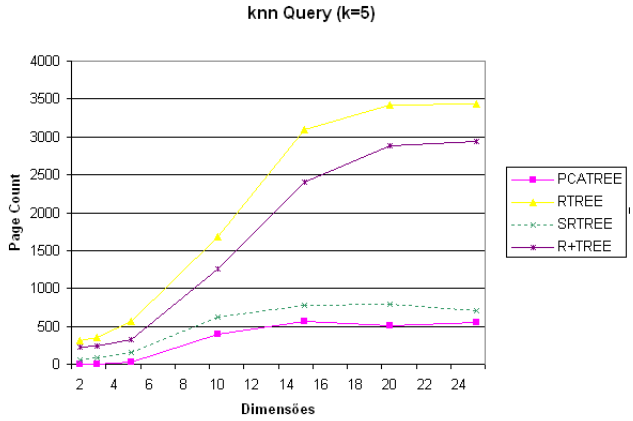

Figura 7.3: Comparação para *knn queries* sobre pontos uniformemente distribuídos.

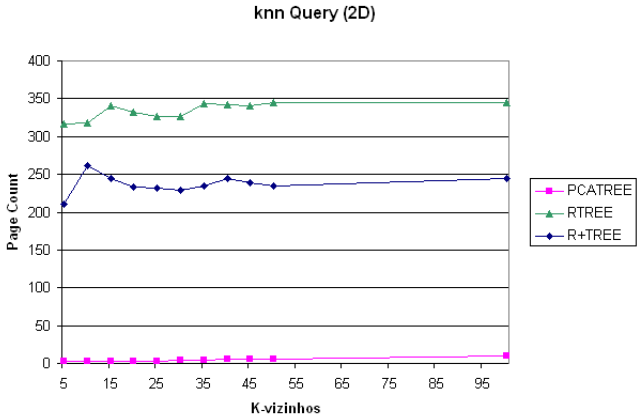

Figura 7.4: Comparação para *knn queries*.

7.2 Dados com distribuição não uniforme

Como fonte de dados não uniformemente distribuídos em $\mathbb{R}^d$, apresentando padrões em sua estrutura, utilizamos uma imagem para a amostragem. Para cada pixel presente na imagem, extraímos um vetor de características de d-dimensões que é utilizado como chave de indexação nos índices. As consultas em extensão e pelos vizinhos mais próximos não são geradas aleatóriamente em $\mathbb{R}^d$ e sim, utilizam-se de um dos valores de chave presente no índice, sendo estes escolhidos aleatoriamente. Foram extraídos vetores de características de 3, 6, 12, 18, 27, 36 e, no caso do gráfico de inserções, 48 dimensões. Para 3 dimensões, o vetor de caraterística de cada pixel é dado pelo valor das componentes RGB do mesmo. Para o caso de 6 dimensões, o vetor de característica é dado pelo valor RGB do pixel em questão e o valor RGB do pixel vizinho na horizontal, formando assim uma janela de 2×1 pixeis. Para 12, 18, 27, 36 e 48 dimensões; as respectivas janelas que caracterizam o vetor de características é de tamanho 2×2, 2×3, 3×3, 3×4 e 4×4. A imagem escolhida para extração de amostras está exposta na figura 7.5.

Figura 7.5: Imagem utilizada para extração de amostras.

Na figura 7.6 temos o resultado, em média de milisegundos, da execução da inserção de 25000 objetos no índice. Os objetos inseridos são pontos aleatoriamente escolhidos a partir do espaço de amostragem.

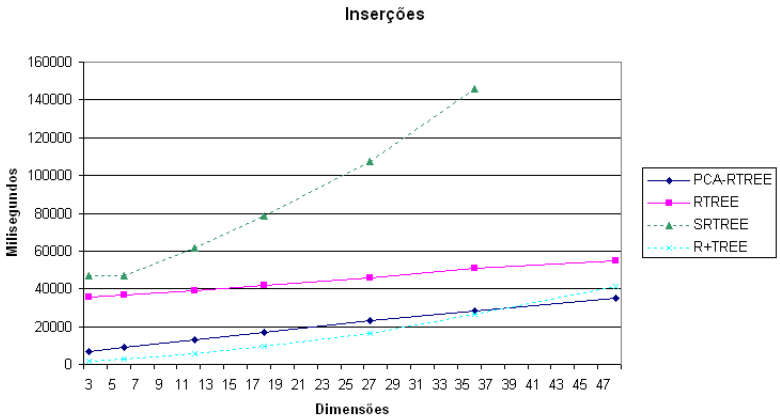

Figura 7.6: Comparação para *inserções* sobre dados não uniformemente distribuídos.

Na figura 7.7 temos o resultado, em média de páginas acessadas (*pageCount*), da execução de buscas em extensão de 100 hiperesferas em $\mathbb{R}^d$ de raio igual a 0.2. Os centróides destas hiperesferas são pontos aleatoriamente escolhidos a partir do espaço de amostragem.

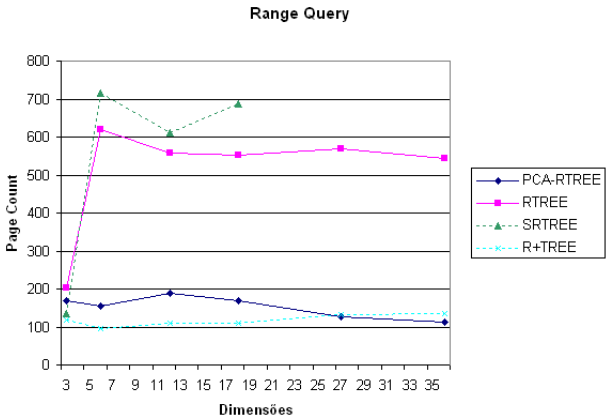

Figura 7.7: Comparação para *range queries* sobre dados não uniformemente distribuídos.

Na figura 7.8 temos o resultado, em média de páginas acessadas (*pageCount*), da execução de 100 buscas pelos 5 vizinhos mais próximos. Os pontos de refêrencia para a busca são aleatoriamente escolhidos a partir do espaço de amostragem.

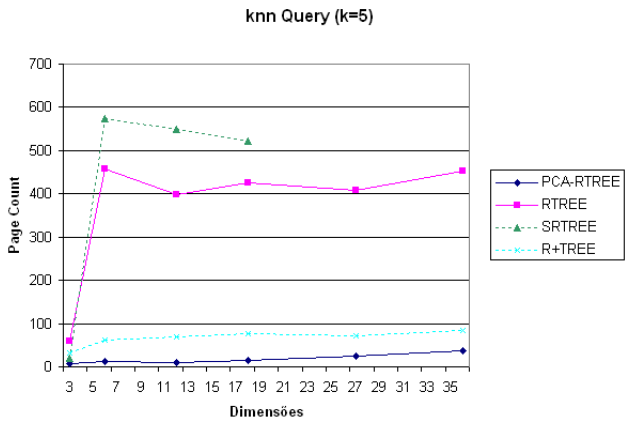

Figura 7.8: Comparação para *knn queries* sobre dados não uniformemente distribuídos.

7.3 Análise dos resultados

A PCA-tree apresentou excelente desempenho nas consultas cujo volume de busca é muito menor que o espaço indexado, mesmo comparado com a R+-tree. Quando o volume de busca é igual a zero, ou seja, busca de casamento exato, a PCA-tree chega a ter desempenho superior a 10 vezes a R+-tree. Entretanto, para volumes grandes, a PCA-tree apresentou desempenho pior que a R+-tree, embora ainda melhor que a R-tree e SR-tree.

A PCA-tree apresentou um ganho de desempenho considerável na inserção. Este ganho pode ser interpretado como sendo resultado do melhor desempenho da PCA-tree para consultas de casamento exato, já que, uma inserção envolve a busca da página folha onde o ponto será inserido. O ganho de desempenho da PCA-tree em buscas exatas, independente do número de dimensões, é devido a propriedade de caminho único entre a raiz e a página folha onde o ponto buscado está contido. Já na R-tree e SR-tree, devido a interseção de hipercubos, podem existir vários caminhos para uma busca exata.

A PCA-RTree apresenta melhor desempenho que a PCA-Tree para buscas em extensão de grandes volumes. A PCA-RTree, ao armazenar também o MBR dos pontos indexados em cada subespaço de busca, reduz significativamente o número de caminhos percorridos por uma busca em extensão. Isto possibilitou buscas em extensão e de vizinhos mais próximos de melhor desempenho mesmo comparados com a R+-Tree. Apesar desta melhoria, na PCA-RTree a estrutura de dados necessita do dobro de espaço de armazenamento que a PCA-tree, o que representa uma perda de desempenho considerável nas rotinas de leitura e escrita.

Capítulo 8

Conclusão

Neste trabalho descrevemos alguns algoritmos conhecidos de indexação unidimensional e multi-dimensional que são baseadas em estruturas do tipo árvore balanceada e com capacidade de armazenamento de nós em páginas de memória secundária. Mais precisamente, descrevemos a B-Tree, B+-Tree, KDB-Tree, R-Tree, R+-Tree. Introduzimos um novo método de indexação multi-dimensional que incorpora duas inovações: o uso de hiperplanos de direção arbitrária, não necessariamente ortogonais a um dos eixos do espaço multi-dimensional, para dividir um espaço, e a representação das divisões hierárquicas através de árvores binárias. Denominamos este novo método de PCA-Tree devido ao fato da escolha da direção do hiperplano ser baseado na componente principal dos elementos contidos no espaço a ser dividido. Foram elaborados as estruturas de dados da PCA-Tree e algoritmos para as operações de inserção, exclusão, buscas exatas, por extensão e pelos k vizinhos mais próximos. Além desses algoritmos, foram elaborados os algoritmos de mesclagem e quebra de nós, necessários para restaurar o balanço das árvores.

Os algoritmos elaborados para a PCA-Tree foram implementados assim como os algoritmos relacionados à KDB-Tree e R-Tree. Uma extensão da PCA-Tree, que denominamos PCA-RTree, que armazena também o MBR de cada subespaço foi implementada. Além desses algoritmos, foi implementada também uma interface gráfica que permite visualizar esses índices para dados em $\mathbb{R}^2$. A interface ilustra a estrutura do índice bem como o efeito das operações sobre os índices e constitui uma ferramenta didática que pode auxiliar no entendimento desses índices. Todas as implementações realizadas encontram-se disponíveis em `http://www.vision.ime.usp.br/~philipe`. Essas são as principais contribuições deste trabalho.

Para avaliar o método de indexação multi-dimensional proposto, comparamos seu desempenho com o desempenho da R-Tree, R+-Tree e SR-Tree. No caso da R+-Tree e SR-Tree, foram utilizadas as implementações disponíveis em `http://www.comp.nus.edu.sg/~cuibin/research.htm`. Foram comparadas os desempenhos referentes às operações de inserção, busca exata, busca em extensão e busca pelos vizinhos mais próximos. Os resultados experimentais mostram que a PCA-Tree é superior na operação de inserção. Para a operação de busca por extensão, para espaços de busca com pequeno volume a PCA-Tree

é superior aos demais, porém para volumes grandes a PCA-Tree tem desempenho inferior à R+-Tree. Ainda com relação a busca em extensão, a PCA-RTree mostrou desempenho equivalente ao da R+-Tree, superando as demais. Em relação à busca pelos vizinhos mais próximos, a PCA-Tree mostrou-se superior aos demais.

Futuramente, o desenvolvimento de um algoritmo que calcula a mínima distância entre um ponto e um politope pode ser desenvolvido para o cálculo de interseção entre o volume de busca e um nó. Nos algoritmos apresentados, a interseção é verificada apenas para o hiperplano local no nó. Desta forma, ignora-se o fato de que o nó local representa um politope formado pelos hiperplanos do caminho entre a raiz e o nó em questão.

Pesquisas futuras também podem ser feitas de forma a comparar a PCA-tree com outros tipos de índices. Dentre estes podemos citar os que usam métricas para indexação (M-tree [10], Slim-tree [19]) e os que usam um mapeamento $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ para indexação (*Pyramid-Technique* [6]).

Referências Bibliográficas

[1] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu, *An optimal algorithm for approximate nearest neighbor searching fixed dimensions*, J. ACM **45** (1998), no. 6, 891–923.

[2] Franz Aurenhammer, *Voronoi diagrams: a survey of a fundamental geometric data structure*, ACM Comput. Surv. **23** (1991), no. 3, 345–405.

[3] R. Bayer and E. M. McCreight, *Organization and maintenance of large ordered indices*, Acta Inf. **1** (1972), 173–189.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The R*-tree: an efficient and robust access method for points and rectangles*, SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 1990, pp. 322–331.

[5] S. Berchtold, D. A. Keim, and H. P. Kriegel, *The X-tree: An index structure for highdimensional data*, International Conference on Very Large Databases (1996), 28–39.

[6] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel, *The pyramid-tree: Breaking the curse of dimensionality*, SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA (Laura M. Haas and Ashutosh Tiwary, eds.), ACM Press, 1998, pp. 142–153.

[7] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario, *Efficient matching and indexing of graph models in content-based retrieval*, IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE **23** (2001), no. 10, 1089–1105.

[8] Christian Bohm, Stefan Berchtold, and Daniel A. Keim, *Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases*, ACM Comput. Surv. **33** (2001), no. 3, 322–373.

[9] _____, *Searching in high-dimensional spaces - index for improving the performance of multimedia databases*, ACM Comput. Surv. (2002).

[10] P. Ciaccia, M. Patella, and P. Zezula, *M-tree: An efficient access method for similarity search in metric spaces*, Presented at the international conference on Very large databases (1997).

[11] R. R. Ciferri, A. C. Salgado, V.C . Times, M. A. Nascimento, and G. C. Magalhaes, *A Performance Comparison among the Traditional R-trees, the Hilbert R-tree and the SR-tree*, Chilean Computer Science Society (2003), 3–12.

[12] T. H. Cormem, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 2nd ed., The MIT Press, 2003.

[13] Yining Deng and B. S. Manjunath, *An efficient low dimensional color indexing scheme for region-based image retrieval*, Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP) **6** (1999), 3017–3020.

[14] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*, 2nd ed., John Wiley & Sons, 2000.

[15] Volker Gaede and Oliver Günther, *Multidimensional access methods*, ACM Computing Surveys **30** (1998), no. 2, 170 – 231.

[16] Antonio Guttman, *R-trees: a dynamic index structure for spatial searching*, Proceedings of the 1984 ACM SIGMOD international conference on Management of data (1984), 47–57.

[17] J. Han and M. Kamber, *Data mining*, 2nd ed ed., Morgan Kaufmann Publishers, 2006.

[18] J. Jannink, *Implementing Deletion in B+-trees*, ACM SIGMOD Record **24** (1995), no. 1, 33–38.

[19] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos, *Slim-trees: High performance metric trees minimizing overlap between nodes.*, EDBT, 2000, pp. 51–65.

[20] Ibrahim Kamel and Christos Faloutsos, *Hilbert R-tree: An Improved R-tree using Fractals*, VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1994, pp. 500–509.

[21] Norio Katayama and Shin'ichi Satoh, *The SR-tree: an index structure for high-dimensional nearest neighbor queries*, SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 1997, pp. 369–380.

[22] Byoung Chul Ko, Hae-Sung Lee, and Hyeran Byun, *Region-based image retrieval system using efficient feature description*, Proceedings of the 15th International Conference on Pattern Recognition **4** (2000), 283–286.

[23] Ju-Hong Lee, Guang-Ho Cha, and Chin-Wan Chung, *A model for k-nearest neighbor query cost in multidimensional index structures.*

[24] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos, *The TV-tree: An index structure for high-dimensional data*, The VLDB Journal The International Journal on Very Large Data Bases **03** (1994), no. 04, 517–542.

[25] Guojun Lu and Atul Sajjanhar, *Region-based shape representation and similarity measure suitable for content-based image retrieval*, Multimedia Systems **7** (1999), 165–174.

[26] Ratiko Orlandic and Byunggu Yo, *Implementing KDB-trees to support high-dimensional data*, Proceedings International Database Engineering and Applications Symposium **01** (2001), 58–67.

[27] John T. Robinson, *The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*, Proceedings of ACM SIGMOD Conference (1981), 10–18.

[28] Nick Roussopoulos, Stephen Kelley, and Frederic Vincent, *Nearest neighbor queries*, SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 1995, pp. 71–79.

[29] T. Settis, N. Roussopoulos, and C. Faloutsos, *The R+-Tree: A dynamic index for multi-dimensional objects*, VLDB Conference **13** (1987).

[30] T. K. Shih, C. S. Wang, A. Y. Chang, and C. H. Kao, *Indexing and retrieval scheme of the image database based on color and spatial relation*, International Conference on Multimedia and Expo (2000).

[31] A. W. M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain, *Content-based image retrieval at the end of the early years*, IEEE Transactions on Pattern Analysis and Machine Inteligence **22** (2005), no. 12.

[32] Alvy Ray Smith, *Color gamut transform pairs*, SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM Press, 1978, pp. 12–19.

[33] L. I. Smith, *A tutorial in principal component analysis*, 2002.

[34] Henry Tom, *The geographic information systems (GIS) standards infrastructure*, StandardView **2** (1994), no. 3, 133–142.

[35] Roger Weber and Michael Mlivoncic, *Efficient region-based image retrieval*, CIKM 03 (2003).

[36] David A. White and Ramesh Jain, *Similarity Indexing with the SS-tree*, ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering (Washington, DC, USA), IEEE Computer Society, 1996, pp. 516–523.

[37] B. Yu, R. Orlandic, and M. Evens, *Simple QSF-Trees: An Efficient and Scalable Spatial Access Method*, Proceedings of the 8th International Conference on Information and Knowledge Management (1999).