Universidade Estadual de Londrina

RODRIGO MIMURA SHIMOMURA

ANÁLISE E COMPARAÇÃO DE ESTRUTURAS DE INDEXAÇÃO PARA CONSULTAS POR SIMILARIDADE COM CONDIÇÕES ADICIONAIS

LONDRINA

2023

RODRIGO MIMURA SHIMOMURA

ANÁLISE E COMPARAÇÃO DE ESTRUTURAS DE INDEXAÇÃO PARA CONSULTAS POR SIMILARIDADE COM CONDIÇÕES ADICIONAIS

Versão Preliminar de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Daniel dos Santos Kaster

LONDRINA

2023

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Sobrenome, Nome.
Título do Trabalho : Subtitulo do Trabalho / Nome Sobrenome. - Londrina, 2017.
100 f. : il.

Orientador: Nome do Orientador Sobrenome do Orientador.
Coorientador: Nome Coorientador Sobrenome Coorientador.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2017.
Inclui bibliografia.

1. Assunto 1 - Tese. 2. Assunto 2 - Tese. 3. Assunto 3 - Tese. 4. Assunto 4 - Tese. I. Sobrenome do Orientador, Nome do Orientador. II. Sobrenome Coorientador, Nome Coorientador. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Este trabalho é dedicado às crianças adultas
que, quando pequenas, sonharam em se
tornar cientistas.

AGRADECIMENTOS

"Não vos amoldeis às estruturas deste mundo, mas transformai-vos pela renovação da mente, a fim de distinguir qual é a vontade de Deus: o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2))

SHIMOMURA, R. M.. **Análise e comparação de estruturas de indexação para consultas por similaridade com condições adicionais**. 2023. 58f. Trabalho de Conclusão de Curso – Versão Preliminar (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

Com o desenvolvimento acelerado de tecnologias e meios de comunicação, surge a necessidade de armazenar, gerenciar e recuperar não apenas dados tradicionais como números e textos, mas também informações complexas como fotos, vídeos, sequências genéticas e coordenadas geográficas, por exemplo. Os sistemas de gerenciamento de banco de dados, originalmente projetados para dados simples, enfrentam dificuldades na padronização e recuperação desses tipos de informações complexas, armazenados em formato binário. Para superar esse obstáculo, a busca por similaridade emerge como uma abordagem eficaz, otimizando consultas e retornando resultados relevantes. Esta pesquisa objetiva a realização de uma análise comparativa de diferentes estruturas de indexação para dados complexos, avaliando métricas de desempenho como acessos a arquivos de índices e dados, tempo médio de consulta com foco especial na busca dos k-vizinhos mais próximos, visando a identificação das situações mais adequadas para a aplicação de cada solução proposta.

Palavras-chave: Buscas por similaridade. Dados complexos. Slim-tree. cx-Sim* tree. S2I. BSlim. Inverted Linear Quadtree

SHIMOMURA, R. M.. **Comparison and analysis of indexing structures for similarity queries with additional conditions**. 2023. 58p. Final Project – Draft Version (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

With the accelerated development of technologies and means of communication, there is a need to store, manage and recover not only traditional data such as numbers and texts, but also complex information such as photos, videos, genetic sequences and geographic coordinates, for example. Database management systems, originally designed for storing simple data, face difficulties in standardization and retrieval of these types of complex information, stored in binary format. To overcome this obstacle, similarity search emerges as an effective approach, optimizing queries and returning relevant results. This research aims to carry out a comparative analysis of different indexing structures for complex data, evaluating performance metrics such as access to indexes and data files, average query time with a special focus on searching for the k-nearest neighbors, aiming to identify the most suitable situations for the application of each proposed solution.

Keywords: Similarity search. Complex data. Slim-tree. cx-Sim* tree. S2I. BSlim. Inverted Linear Quadtree

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BLOBs	*Binary Large Objects*
CBVR	*Content based video retrieval*
${}_cKNNq$	*Conditional k-Nearest Neighbors query*
cx-Sim tree*	*Condition-eXtended Similarity tree*
DWT	*Discrete Wavelet Transform*
ECG	*Eletrocardiogramas*
GLC	*Gray co-occurrence matrix*
HDD	*High Dimensional Data*
IL-tree	*Inverted Linear Quadtree*
ISOMAP	*Isometric Mapping*
k-FNq	*k-Farthest Neighbors query*
k-NNq	*k-Nearest Neighbors query*
LDA	*Linear Discriminant Analysis*
LLE	*Locally Linear Embedding*
LSH	*Locality Sensitive Hashing*
LSI	*Latent Semantic Indexing*
MAM	*Metric access method*
MDS	*Multi-dimensional scaling*
MFCC	*Mel-Frequency Cepstral Coefficients*
MRMR	*Minimum Redundancy Maximum Relevance*
MST	*Minimum Spanning Tree*
PCA	*Principal Component Analysis*
RkNN	*Reverse k-Nearest Neighbors query*
Rq	*Range query*

RSFS	*Recursive Sequential Forward Selection*
S2I	*Spatial Inverted Index*
SFS	*Sequential Forward Selection*
SFFS	*Sequential Floating Forward Selection*
SGBD	Sistema de gerenciamento de banco de dados
STFT	*Short Time Fourier Transform*
TOPK-SK	*Top-k Spatial Keyword search*

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, o mundo vem testemunhando a produção e troca massiva de dados digitais devido ao acelerado desenvolvimento dos meios de comunicação e das tecnologias [5]. Nesse cenário, a produção de dados abrange não somente os dados tradicionais como números e textos, mas também dados complexos que são informações não representáveis por apenas um dado simples, como representações visuais, registro de vídeos, coordenadas geográficas, séries temporais e sequências de DNA, por exemplo. À medida que a sociedade está imersa com esse fluxo em profusão de informações, a necessidade de armazenar, gerenciar e recuperar tanto os dados tradicionais/simples quanto os dados complexos se torna cada vez mais indispensável [6, 7].

Os sistemas de gerenciamento de banco de dados (SGBDs) foram projetados com foco no armazenamento e recuperação de informações de natureza simples, como valores numéricos e blocos de texto. Diante do exposto, torna-se evidente a existência de desafios na padronização e recuperação de dados complexos, dado que esses são armazenados no formato *BLOBs* (*Binary Large Objects*) [8], o qual guarda as informações em strings binárias de baixo nível. O problema de realizar a recuperação de dados complexos está no fato de que raramente consultas envolvendo igualdade são realizadas com esse tipo de informação, o que pode levar as buscas a se tornarem ineficientes em bancos de dados com abundância de dados.

Diante do exposto, o modo mais utilizado para recuperar dados complexos é através de buscas por similaridade, por otimizar os processos que o SGBD terá que realizar visando retornar os resultados das consultas [9]. Para realizar as buscas, são extraídas características relevantes ao dado, guardando-as em *feature vectors* (vetores de características) e posteriormente definindo um critério de similaridade para cada tabela do banco de dados. Por exemplo, a criação dos vetores de características de eletrocardiogramas (ECG) utiliza os coeficientes do método DWT (*Discrete Wavelet Transform*) (após aplicados para remoção de ruídos e melhoramento da qualidade dos exames) [10]. Após ter criado os vetores de características, o último passo seria definir o critério de similaridade, e posteriormente a utilização de uma função de distância para os cálculos de (dis)similaridade.

1.1 Motivação

Um ponto a ser levado em consideração na realização das buscas é que geralmente os dados complexos são armazenados em conjunto com dados tradicionais, os quais devem ser utilizados na filtragem dos resultados e operações de similaridade visando otimizar

e responder consultas complexas [2]. Diante do exposto, os mecanismos de busca por similaridade mais frequentemente utilizados são a consulta por abrangência (*Range query - Rq*), consulta dos k vizinhos mais próximos (*k-Nearest Neighbor query - k-NNq*), consulta dos k vizinhos mais próximos reverso (*RkNN*) [11]. Entretanto, aplicações de SGBDs que possuem alguns desses algoritmos implementados, foram criadas e estruturadas de maneira muito específica para cada caso, fato que dificulta a reutilização e a manutenção dos códigos que muitas vezes são ineficazes e custosos.

Algumas estruturas de indexação que possuem dados complexos em seus elementos são a *Slim-tree* [12], *cx-Sim* tree* (*Condition-eXtended Similarity tree*) com suas 4 variações (*cx-sim* Simple, cx-sim* Covering, cx-sim* Chained, cx-sim* Composite*) [3], *S2I (Spatial Inverted Index)* [13], *Inverted Linear Quadtree* [4] e *BSlim* [3], cada uma com suas peculiaridades, vantagens e desvantagens para cada caso de uso. Cada um dos autores mencionados já procedeu à implementação e execução de experimentos relativos a cada uma das estruturas, entretanto, o fizeram de forma independente. Tal circunstância suscita o interesse na condução de uma análise comparativa entre todas as estruturas acima mencionadas, levando em consideração distintos cenários de busca e conjuntos de dados diversos.

1.2 Objetivos do trabalho

O objetivo deste trabalho de conclusão de curso é realizar uma análise comparativa das estruturas de indexação de dados complexos, com ênfase na avaliação das métricas de desempenho como acessos ao arquivo de índices e dados, além do tempo médio de consulta, principalmente para a busca dos k-vizinhos mais próximos. A proposta será elaborar uma maneira de comparar as estruturas, bem como evidenciar as melhores situações de uso para cada tipo de maneira de indexação.

1.3 Organização do Trabalho

- O capítulo 2 apresenta as definições dos conceitos relacionados às buscas por similaridade, além dos tipos de busca utilizados atualmente.
- O capítulo 3 enumera e descreve o funcionamento e peculiaridades de cada estrutura de indexação de dados complexos que será utilizada na comparação no capítulo 4
- O capítulo 4 irá apresentar a proposta de contribuição, detalhar o ambiente criado para a realização dos testes, mostrar as bases de dados utilizadas e os resultados obtidos.
- Por fim, o capítulo 5 irá trazer as conclusões obtidas neste trabalho de conclusão de curso.

2 BUSCAS POR SIMILARIDADE

Este capítulo irá abordar sobre os conceitos fundamentais para o desenvolvimento e entendimento deste trabalho. Será explicado o significado de similaridade para o ser humano, com a abstração necessária para transferência ao ambiente computacional. Ademais, uma análise detalhada da representação dos dados complexos, com a criação dos vetores de características, e suas técnicas de preparo será conduzida. Outros conceitos importantes como a definição do espaço métrico e funções de distâncias serão brevemente discutidos. Por fim, serão apresentados os principais tipos de busca por similaridade, bem como a justificativa para a escolha da *Range query - Rq* e *k-Nearest Neighbors query - k-NNq*.

2.1 O conceito de similaridade

A percepção do conceito de similaridade e dissimilaridade remete a um dos aspectos fundamentais da cognição humana [14]. O processo de definir critérios de similaridade adequados às situações presentes em nossas vidas é complexo e subjetivo, em decorrência da natureza dos objetos presentes na comparação. Isso pode ser observado, por exemplo, na maneira como é definido a similaridade entre duas pinturas e entre duas músicas. Enquanto a comparação no primeiro caso leva em consideração características como cores, texturas, formatos e tamanho, a segunda considera critérios como ritmo, timbre, harmonia e melodia, por exemplo. Os conceitos de similaridade e dissimilaridade são de grande importância nas ciências da computação, tendo grande expressividade em áreas como aprendizado de máquina, reconhecimento de padrões, mineração de dados e inteligência artificial.

Intuitivamente, pode-se definir o valor de similaridade entre dois objetos como as diferenças entre suas características, de tal forma que distância deve refletir a percepção humana. Assim, imagens similares devem ter valores de distância menores em comparação às imagens diferentes, por exemplo, [15]. A figura 1 demonstra uma situação onde três cachorros podem ser considerados similares através da definição critérios como tamanho do animal, formato do rosto, cor, tipo de pelagem, de tal forma que pode-se dizer que pertencem à mesma raça.

2.1.1 Representação dos dados complexos e similaridade

Sequências temporais, imagens, vídeos, coordenadas geográficas, sequências de DNA, resultados de experimentos científicos e arquivos de texto extensos e compactados (.pdf) são chamados dados complexos devido à incapacidade de representá-los com

Figura 1 – Cachorros da raça *Boxer*

apenas um tipo de dado simples como um número inteiro, cadeia de caracteres ou ponto flutuante [16]. A maioria dos domínios complexos não possuem relações de ordem total entre seus elementos, o que implica a incapacidade de utilizar operadores de comparação relacionais ($>, \geq, <, \leq$). Já operadores "="e "$\neq$"não possuem utilidade, uma vez que não faz sentido procurar um áudio exatamente igual ao fornecido, salvo exceções quando o objetivo da consulta é verificar a existência do objeto de referência no banco de dados [9, 2].

A principal maneira de representar esses dados complexos para realizar comparações por similaridade consiste em 2 etapas primordiais. A primeira consiste em escolher um método para realizar a extração e seleção de características do dado em questão visando armazená-las em um vetor de características, por exemplo, a extração de informações relevantes de imagens consiste em 2 tipos de informação: características de baixo nível como cor, textura e forma, além de características de alto nível como objetos na imagem (utilizando as características de baixo nível) [17]. A segunda etapa está na elaboração da função de distância entre elementos do conjunto, indicando a dissimilaridade entre dois vetores de características, portanto, a similaridade entre dois elementos é inversamente proporcional à sua distância [18].

2.1.2 Criação dos vetores de características

A transformação dos dados complexos visando torná-los comparáveis envolve a criação de vetores de características. Essa abstração permite a conversão da complexidade das informações em valores numéricos e significativos. Esse processo é executado em algumas etapas como a extração primária das características, seguido da redução de dimensionalidade dos dados através da transformação e seleção de características [1, 2]. Todas as etapas desse processo de criação dos vetores de características possibilitam uma análise mais eficaz, e a formulação de *insights* valiosos para o conjunto de dados em questão.

2.1.2.1 Extração de características

As técnicas de extração de características pertinentes aos dados variam conforme a natureza das informações. Vídeos são um exemplo de combinação de dados complexos, onde sequências de imagens variando de 24 a 60 quadros por segundo são exibidos, juntamente com faixas de áudio. As imagens são principalmente definidas por texturas, formas, cores e os áudios pela amplitude, frequência e formato da onda. Atualmente a busca e recuperação de vídeos se baseia nos sistemas de CBVR (*Content based video retrieval*) onde o conteúdo do vídeo é analisado [19]. É de suma importância destacar que por combinar várias características simultaneamente, o problema da maldição da dimensionalidade ocorre conforme comentado em [19, 20].

Os autores em [19] enumeram algumas *features* utilizadas no vetor de características de vídeos:

- Texturas
- Formas
- Cores (descritores, histogramas, correlogramas)
- Características de alto nível semântico
- Áudio (tom, frequência de pausa, cromagrama, perceptual latente)
- Representação de quadros-chave

Algumas técnicas mais utilizadas para extrair essas características são: *Gray co-occurrence matrix* (GLC) para texturas (Julesz (1975)), histograma normalizado para cores [21],*Mel-Frequency Cepstral Coefficients* (MFCC) e *Short Time Fourier Transform* (STFT) para faixas de áudio [22, 23].

2.1.2.2 Redução de dimensionalidade

Cada característica que pode ser extraída de um dado aumenta a dimensionalidade do vetor que irá representá-lo. Aplicações com *High Dimensional Data* (HDD) são encontradas em várias áreas do conhecimento como medicina, educação, negócios e redes sociais em diversos formatos, como textos, imagens e vídeos [24]. Como comentado por [20], a maldição da dimensionalidade é um dos fatores que aumentam a dificuldade da extração de características relevantes ao contexto da análise. Dessa maneira, torna-se indispensável a redução de dimensionalidade, a fim de otimizar a criação de vetores de características que consigam representar, de forma clara e objetiva, os dados complexos.

De acordo com [1], a redução de dimensionalidade é um processo realizado visando a diminuição da quantidade de informações presentes no vetor de características, sem a perda de dados importantes. Isso se torna possível, dado que apenas partes irrelevantes, redundantes e ruídos são eliminados do conjunto de dados, fato que colabora para a redução do tempo, memória e poder de processamento necessário para o processamento das informações. Uma das principais vantagens obtidas ao realizar esse processo, como mostra [25] é o aumento da qualidade dos dados, melhora da eficiência dos algoritmos, aumento da acurácia e simplificação da visualização e análise dos resultados para os pesquisadores. Assim sendo, as técnicas de redução de dimensionalidade são separadas em dois grupos principais: seleção e transformação de características.

2.1.2.3 Seleção de características

A seleção de características é um método empregado para diminuir a dimensionalidade, consistindo na identificação do subconjunto de características capaz de descrever os dados de forma eficaz. Isso é obtido através da seleção de características importantes e relevantes, removendo informações irrelevantes e redundantes [26]. Algumas vantagens obtidas no processo são redução do tamanho dos dados, diminuição do armazenamento necessário, aumento da acurácia, evita o *overfitting*, reduz o tempo de execução e treinamento dos algoritmos [1]. A figura 2 ilustra o processo geral dos métodos de seleção de características

A escolha do subconjunto das características pode ser dividida em quatro frentes gerais [1]:

1. *Filter*: Analisa as características presentes a partir de quatro critérios: informação, dependência, consistência e distância. O ranqueamento do subconjunto selecionado é feito a partir de dados estatísticos.

2. *Wrapper*: Envolve o processo de seleção de características de acordo com a acurácia ou a taxa de erro obtida no algoritmo de aprendizagem, para classificação

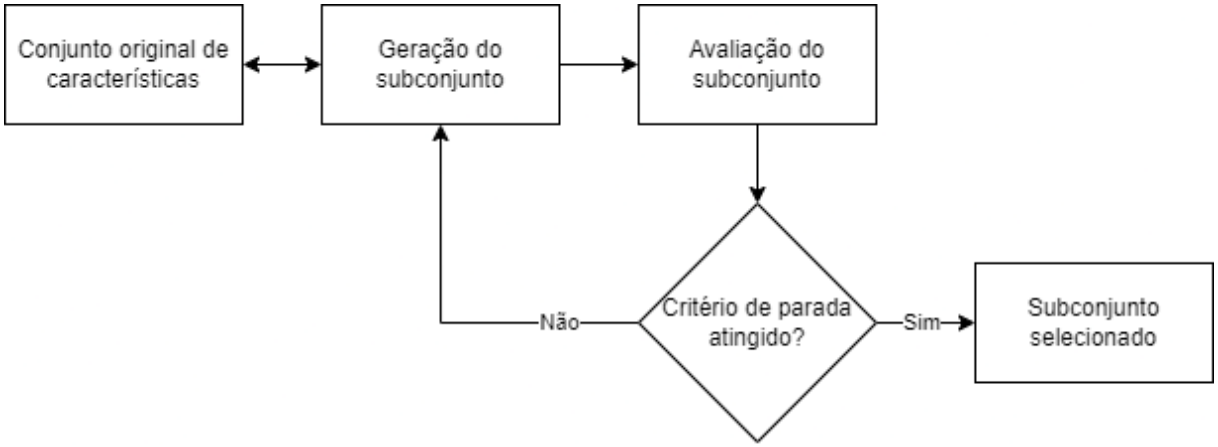

Figura 2 – Processo de seleção de características. Adaptado de [1]

da qualidade do subconjunto escolhido. Esse processo ocorre de forma separada do treinamento do modelo.

3. *Embedded*: A seleção de características ocorre no algoritmo de aprendizagem, e usa as propriedades para a avaliação do subconjunto.

4. *Hybrid*: O modo híbrido consiste na junção de dois ou mais métodos anteriores. A vantagem está na união dos pontos fortes de cada abordagem, e normalmente se usa o método *Filter* juntamente com o método *Wrapper*.

Existem outros métodos específicos para a seleção de características de acordo com [26]:

1. *Sequential Forward Selection* (SFS): O subconjunto é constantemente atualizado, adicionando-se novas características relevantes. É mais recomendado para conjunto de dados pequenos, geralmente com menos de 9 características. Eventualmente pode produzir soluções diferentes da ótima, além de possuir alta complexidade.

2. *Sequential Floating Forward Selection* (SFFS): Uma versão melhorada do SFS, eliminando os problemas da versão anterior. Porém, sofre de alto custo computacional quando aplicado em conjuntos de dados grandes. Na maior parte dos casos, não converge para um subconjunto de tamanho fixo.

3. *Minimal Redundancy Maximal Relevance* (MRMR): Seleciona as características que são mutuamente independentes, mantendo alta relevância para a variável de classificação.

4. *Random Subset Feature Selection* (RSFS): Usado para descobrir o subconjunto médio que é melhor que o atual. As características são escolhidas de maneira aleatória e repetidamente para todas as combinações de características.

2.1.2.4 Transformação de características

O processo de transformação de características envolve a criação de novas *features*, que dependem das já existentes. Isso é feito visando a redução da dimensionalidade do vetor, e não perder grandes quantidades de informações, por conservar as distâncias originais entre características [1]. A figura 3 ilustra, de forma simples, a ideia geral dos métodos de transformação de características.

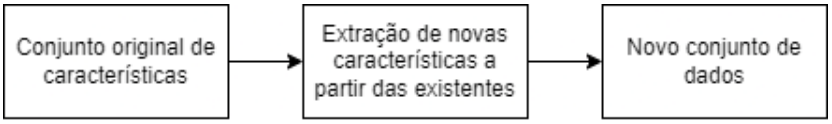

Figura 3 – Processo de extração de características. Adaptado de [1]

Os métodos mais comuns de realizar esse método de acordo com os autores em [2, 1, 25, 27] são:

- *Principal Component Analysis* (PCA)
- *Multi-Dimensional Scaling* (MDS)
- *Isometric Mapping* (ISOMAP)
- *Locally Linear Embedding* (LLE)
- *Linear Discriminant Analysis* (LDA)
- *Latent Semantic Indexing* (LSI)

2.1.3 Espaço métrico

As buscas por similaridade exigem ao menos um elemento de referência e um critério de seleção de objetos similares. Esse critério utiliza a função de distância e pode ser um número k de objetos, ou uma dissimilaridade máxima em relação ao ponto de comparação fornecido. Dessa maneira, o ranqueamento dos objetos mais similares é ditado pela função de distância.

Restringindo os tipos de distância a serem utilizados através dos postulados métricos, um espaço métrico pode ser definido como o par $(\mathbb{S}, d)$, sendo $\mathbb{S}$ representando o domínio dos objetos e d representando a função de distância [18]. As propriedades da função $d : \mathbb{S} \times \mathbb{S} \to \mathbb{R}$ para $\forall x, y, z \in \mathbb{R}$ são tipicamente caracterizadas como:

- Não negatividade: $d(x, y) \geq 0$
- Simetria: $d(x, y) = d(y, x)$

- Identidade: $x = y \Leftrightarrow d(x, y) = 0$
- Desigualdade triangular: $d(x, z) \leq d(x, y) + d(y, z)$

2.1.4 Funções de distância

Existem diferentes tipos de funções de distância presentes na literatura [28]. As mais utilizadas para realizar os cálculos das funções de dissimilaridade são as da família *Minkowski*. A equação 2.1 define essa família, onde n é a dimensão do vetor de características e p é um valor inteiro ($1 \leq p \leq \infty$). Os valores mais utilizados de p são: $p = 0$ ou ∞ (distância Chebyshev), $p = 1$ (distância de Manhattan) e $p = 2$ (distância Euclidiana).

$$L_p[(x_1, ..., x_n), (y_1, ..., y_n)] = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p} \tag{2.1}$$

Em especial, a distância Euclidiana é utilizada em profusão em problemas geométricos e de agrupamento. Além disso, ela satisfaz os quatro postulados métricos, sendo assim, considerada métrica [29].

2.2 Tipos de busca por similaridade

Os principais tipos de busca por similaridade podem ser divididos em alguns grupos como exposto por [18]:

- *Range query*
- *Nearest Neighbor Query*
- *Reverse Nearest Neighbor Query*
- *Similarity Join*
- *Combination of queries*
- *Complex Similarity Queries*

As alternativas mais utilizadas atualmente se concentram na *Range Query* e *Nearest Neighbor Query*. Um exemplo de aplicação dessas técnicas é mostrado em [30] com a utilização de *Range queries* em redes de sensores sem fio para consultas do tipo: *"Retorne todos os eventos onde a temperatura do local varie entre 50° e 60°"* e *"Retorne os locais onde os níveis de luz variam entre 10 e 15 lúmens"*. Outro exemplo é elucidado por [31], onde os autores utilizam os dois tipos de busca para realizar consultas por similaridade em banco de dados de objetos 3D, com a utilização de vetores de características.

Este trabalho dará uma ênfase maior para a consulta *k-NNq* na parte das comparações a serem feitas no capítulo 4.

2.2.1 *Range Query - Rq*

A consulta $Rq(q, r)$ retorna para um grau de tolerância $r \in \mathbb{R}^+$ todos os objetos s que satisfazem a condição $d(s, q) \leq r$. Situações comuns da utilização dessa consulta, majoritariamente focam em questões de distância como: *"Retorne os restaurantes em um raio de até 2 km da minha localização"*. Em notação de conjunto, temos a equação 2.2:

$$Resultado = \{s_i \in S | d(q, s_i) \leq r\} \tag{2.2}$$

O autor em [2] descreve em notação algébrica a consulta por abrangência por similaridade. Considerando um elemento de referência s_q, o operador de similaridade θ, uma função de distância entre elementos δ, um limiar de consulta, nesse caso o raio, ξ, um atributo da relação de entrada S_j pertencente ao domínio $\mathbb{S}$ e aplicados na relação R, temos a notação mostrada na notação 2.3:

$$\hat{\sigma}_{S_j\ \theta[\delta,\xi]\ s_q}(R) \tag{2.3}$$

É importante ressaltar que o uso da notação $\hat{\sigma}$ é utilizada para referir-se a consultas por abrangência, pontual e abrangência reversa. Para diferenciar cada uma das operações, altera-se o termo θ e no caso da consulta pontual, o parâmetro ξ. As notações 2.4, 2.5 e 2.6 descrevem, respectivamente, consulta por abrangência, consulta pontual e consulta por abrangência reversa:

$$\hat{\sigma}_{S_j\ Rq[\delta,\xi]\ s_q}(R) \tag{2.4}$$

$$\hat{\sigma}_{S_j\ Rq[\delta,0]\ s_q}(R) \tag{2.5}$$

$$\hat{\sigma}_{S_j\ Rq^{-1}[\delta,\xi]\ s_q}(R) \tag{2.6}$$

Consultas com abrangência reversa alteram apenas a comparação feita antes de considerar um elemento pertencente ao conjunto resposta. Enquanto a abrangência normal realiza comparações do tipo $d(s, q) \leq r$, a abrangência reversa é o seu complemento, em decorrência de realizar a comparação $d(s, q) > r$. A figura 4 ilustra um exemplo de uma *Range query* com $r = 1.4$, com a notação algébrica sendo $\hat{\sigma}_{Rq[L_2,1.4]s_q}(R)$. O atributo S_j foi omitido em consequência de ser uma consulta simples por abrangência sem considerar outros atributos para filtragem.

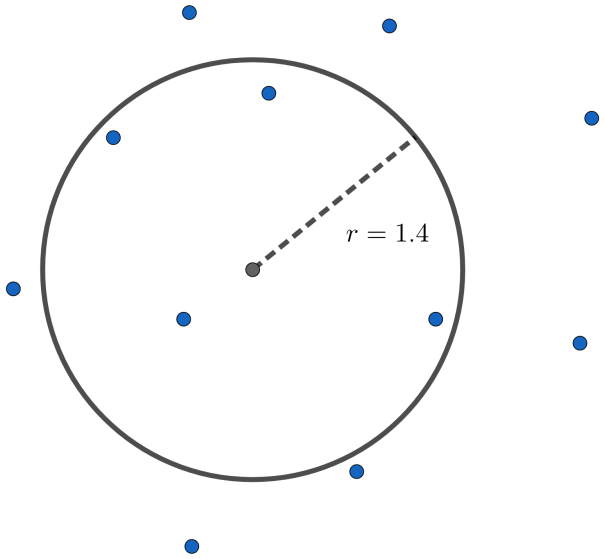

Figura 4 – Exemplo da Rq com $r = 1.4$

2.2.2 *k-Nearest Neighbors query - k-NNq*

A consulta *k-NNq(q, k)* retorna os k vizinhos mais próximos do elemento de referência q. Como exemplificado por [2], consultas do tipo: *"Retorne as 5 imagens de exames médicos mais semelhantes à referência fornecida"* são solucionadas utilizando essa técnica. Em notação de conjunto, temos a equação 2.7:

$$Resultado = \{s_i \in S | \forall s_j \in S \setminus K, |K| = k, d(s_q, s_i) \leq d(s_q, s_j)\} \tag{2.7}$$

Novamente o autor em [2] demonstra a notação algébrica capaz de descrever a consulta *k-NNq* de maneira similar à Rq na notação 2.8. Nessa notação, o símbolo $\ddot{\sigma}$ é utilizado para descrever consultas dos k vizinhos mais próximos ou mais distantes. As diferenças em relação à notação 2.3 estão no uso do parâmetro k que é a quantidade de vizinhos a ser considerado na busca:

$$\ddot{\sigma}_{S_j\ \theta[\delta,k]\ s_q}(R) \tag{2.8}$$

Como mencionado, existem 2 funções disponíveis: os k vizinhos mais próximos (notação 2.9) (*k-NN*), podendo também utilizar $k = 1$ para a consulta do vizinho mais próximo (notação 2.10) e os k vizinhos mais distantes (*k-FN - k-Farthest Neighbors*) (notação 2.11)

$$\ddot{\sigma}_{S_j\ kNN[\delta,k]\ s_q}(R) \tag{2.9}$$

$$\ddot{\sigma}_{S_j\ kNN[\delta,1]\ s_q}(R) \tag{2.10}$$

$$\ddot{\sigma}_{S_j\ kFN[\delta,k]\ s_q}(R) \tag{2.11}$$

De maneira análoga à consulta por abrangência reversa da Rq, o operador $k-FNq$ retorna o complemento da consulta $k-NNq$, os vizinhos mais distantes. A figura 5 mostra um exemplo de uma *k-Nearest Neighbors query* com $k = 4$, onde a notação algébrica que descreve essa consulta é $\ddot{\sigma}_{kNN[L_2,4]s_q}(R)$. Assim como mostrado no exemplo de notação da figura 4, o atributo S_j foi omitido por se tratar de uma consulta sem condição adicional.

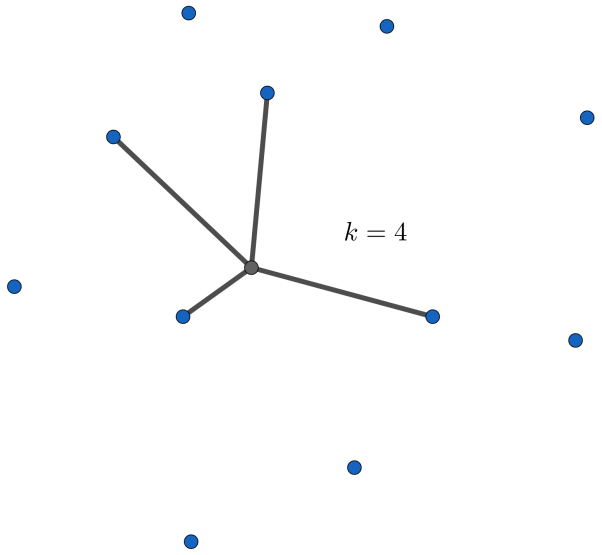

Figura 5 – Exemplo de $kNNq$ com $k = 4$

2.3 Consultas por similaridade com condições adicionais

Segundo os autores em [2, 3], a utilização de atributos simples atrelados aos dados complexos possibilita realizar processos de filtragem, conferindo otimizações durante o processamento das consultas. Isso ocorre em decorrência do uso de condições com alta seletividade, reduzindo o conjunto a ser analisado. Entretanto, o autor em [2] comenta sobre as dificuldades encontradas na integração e otimização do processamento de consultas *k-NNq* sobre dados complexos, devido ao número reduzido de propriedades algébricas. Dessa forma, essa operação é muito limitada na questão do posicionamento no plano de execução da consulta. Adicionalmente, o autor mostra outras variações do *k-NNq* que não conseguem ser representadas utilizando o operador simples. Dessa maneira, é importante definir as peculiaridades de consultas por similaridade com condições adicionais, objetivando torná-las possíveis de serem implementadas (principalmente a *k-NNq* com condições).

2.3.1 *Range Query* com condições adicionais

A busca por abrangência com condições adicionais não possui problemas de implementação, como mostra os autores de [32] conseguindo aplicar condições adicionais na consulta em um banco de dados de *Blockchain*, a partir do mapeamento de uma *Rq* para uma consulta booleana. Outro trabalho mostrando a implementação de consultas por abrangência em relações criptografadas pode ser encontrado em [33], onde os autores demonstram a possibilidade de construir condições simples e compostas ainda que os dados estejam encriptados, desde que uma função de comparação seja estabelecida.

2.3.2 *k-Nearest Neighbors Query* com condições adicionais

A busca pelos k vizinhos mais próximos com condições adicionais foi uma contribuição proposta por [2]. O autor descreve em detalhes as operações e propriedades algébricas desse tipo de consulta, além de propor um novo operador de similaridade: ${}_c k$-*NNq* (*condition-extended k-NN query*). A definição desse operador é descrito pela notação 2.12:

$$\ddot{\sigma}_{S_j \ {}_c kNN \ [\delta,\Delta,k,cond] \ Q}(R) \tag{2.12}$$

Os termos presentes na notação são: $\ddot{\sigma}$ caracterizando uma *k-NNq*, uma coluna da relação de entrada S_j, o operador ${}_c kNN$, com a função de similaridade δ, a função agregadora de distâncias δ representada por Δ, k vizinhos mais próximos, uma condição para filtrar a busca denotada por *cond*, o elemento de referência Q e a relação onde a consulta será aplicada R.

As formas canônicas de *Rqs* e *kNNqs* possuem a função agregadora de distâncias representada por Δ, entretanto a definição desta é necessária somente quando existe um conjunto de elementos de referência ao invés de um único elemento. Dessa maneira, é possível omitir a função Δ para consultas com apenas um elemento de referência. Assim como mostrado em [34], considerando um conjunto de elementos S do banco de dados, com um elemento $s \in S$, com o fator de agregação $g \in \mathbb{R}^+$, w_q sendo o peso correspondente a cada elemento de consulta s_q e Q o conjunto de elementos de referência, uma família de funções agregadoras de distâncias pode ser expresso como mostra a equação 2.13:

$$d_g(s, Q) = \sqrt[g]{\sum_{s_q \in Q} (\delta(s, s_q)^g * w_q)} \tag{2.13}$$

Acerca do atributo *cond*, existem 2 tipos de condições que podem ser inseridas na consulta: as baseadas em tuplas e as baseadas em agregação, sendo possível também a combinação de condições. Assim como definido por [2], condições baseadas em tuplas são definidas como condições que podem ser verificadas mediante uso de dados disponíveis

somente na tupla, de tal forma que a análise é feita individualmente tupla a tupla. Já para as condições baseadas em agregação, a análise deve ser feita levando em consideração um conjunto de tuplas. De maneira geral, o autor define o formato das condições baseadas em agregação da seguinte maneira:

$$f(agAtr, min-op, tcond)\ \theta\ c \tag{2.14}$$

Sobre cada termo da representação, temos que f é a função de agregação utilizada (por exemplo, MIN, MAX, COUNT, SUM e AVG) $agAtr$ é o atributo a ser levado em consideração na análise da função, $min-op$ é a opção de minimização a ser utilizada (utilizada principalmente em funções do tipo SUM e AVG) que será explorada na seção a seguir, *tcond* é a condição baseada em tupla para realizar a seleção de tuplas, θ é o comparador de comparação $\{>, \geq, <, \leq, =, \neq\}$ e por último, c é o valor utilizado na comparação com o resultado da consulta.

2.3.3 Peculiaridades das funções utilizadas nas consultas baseadas em agregação

Cada uma das funções mencionadas anteriormente possui alguns detalhes importantes para a execução das consultas. O objetivo dessa subseção é elucidar os argumentos utilizados em cada uma das funções e, principalmente, dar significado para o conjunto resposta da consulta. Para isso, é obrigatório definir os parâmetros θ e c. A seguir, serão mostradas as formas gerais de cada uma dessas condições por agregação.

1. Contagem: *COUNT(agAtr, tcond)* θ *c*
2. Soma ou média: *SUM/AVG(agAtr, min-op, tcond)* θ *c*
3. Mínimo ou máximo: *MIN/MAX(agAtr, tcond)* θ *c*

Para a utilização de uma função de contagem, é necessário um atributo não nulo $agAtr$ (podendo ser *) e uma condição envolvendo os atributos da tupla (representado por *tcond*). O autor em [2] fornece exemplos de consultas considerando uma base de dados de informações geográficas e populacionais de cidades dos Estados Unidos. Dois exemplos adaptados de consultas são:

1. Retorne as 7 cidades mais próximas à cidade A, de maneira que no resultado existam ao menos 4 cidades com a população maior ou igual a 12.500 habitantes.

$$\ddot{\sigma}_{coord\ ,kNN[L_2,7,COUNT(*,pop\geq 12.500)\geq 4]\ ACoord}\ (Cidades) \tag{2.15}$$

2. Retorne os 5 restaurantes mais próximos ao evento, sendo que no máximo 2 deles estejam a 15 km ou mais do aeroporto.

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,5,COUNT(*,L_2(coord,\ aerCoord)\geq 15km)\leq 2]\ eventCoord}\ (Restaurantes) \quad (2.16)$$

É de grande importância notar que a condição de agregação COUNT é aplicada após o conjunto resposta ter sido criado, além disso, o autor ressalta a possibilidade de se incluir a função DISTINCT durante operações de contagem para evitar tuplas repetidas.

O funcionamento de operações de soma ou média é muito similar ao de contagem por requerer a definição de um *agAtr* e uma *tcond*, o diferencial está na escolha da opção de minimização *min-op*, dado que este altera a maneira como o algoritmo de seleção irá operar, sendo que as principais escolhas básicas são:

1. *min-sum*: Minimização de soma das distâncias ao elemento de referência Q, ou seja, durante a seleção de elementos que pertencerão ao conjunto resposta, o algoritmo irá priorizar aqueles cuja soma/média do valor de dissimilaridade seja a menor possível.
2. *min-min*: Minimização da distância mínima a Q, visando priorizar a construção de um conjunto resposta que contém o elemento mais similar àquele usado de referência (podendo até mesmo ser o próprio elemento).
3. *min-max*: Minimização da distância máxima a Q, focando na criação de um conjunto resposta onde o maior valor de dissimilaridade seja o menor possível.

Para exemplificar a utilização dessas *min-op*, considere a relação *Passeios(Nome, preço, distância)* que representa os possíveis locais de visitação que um hotel na praia oferece para seus hóspedes. A tabela 1 enumera 10 passeios disponíveis, juntamente com os preços e a distância em relação ao hotel em Kms:

Considere as consultas, suas notações e os resultados:

1. Retorne os 6 passeios mais próximos do hotel, de maneira que a soma dos preços de cada passeio seja no máximo R$ 200,00 e **o passeio mais próximo ter a menor distância possível** (min_min)

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,6,SUM(preco,min\_min)\leq 200{,}00]\ hotel}\ (Passeios) \quad (2.17)$$

   - **Resultado**: {*Balsa, Caverna dos cristais, Flutuação, Mercado histórico, Mergulho e Surfe*}
   - **Preço total**: R$196,00

Nome	**Preço (R$)**	**Distância (km)**
Balsa	60	10
Buggy	25	15
Caverna dos cristais	10	80
Dunas	38	20
Flutuação	11	17
Mercado histórico	25	13
Mergulho	20	21
Praia das conchas	40	30
Recifes	37	19
Surfe	70	4

Tabela 1 – Exemplo de 10 tuplas da relação *Passeios*

- **Somatório das distâncias**: 145 km

2. Retorne os 6 passeios mais próximos do hotel, de maneira que a soma dos preços de cada passeio seja no máximo R$ 200,00, e **o passeio mais distante seja o mais próximo possível** (min_max)

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,6,SUM(preco,min\_max)\leq 200{,}00]\ hotel}\ (Passeios) \tag{2.18}$$

- **Resultado**: {*Balsa, Buggy, Dunas, Flutuação, Mercado histórico, Recifes*}
- **Preço total**: R$196,00
- **Somatório das distâncias**: 94 km

3. Retorne os 6 passeios mais próximos do hotel, de maneira que a soma dos preços de cada passeio seja no máximo R$ 200,00 e **a menor soma possível** (min_sum)

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,6,SUM(preco,min\_sum)\leq 200{,}00]\ hotel}\ (Passeios) \tag{2.19}$$

- **Resultado**: {*Buggy, Flutuação, Mercado histórico, Mergulho, Recifes, Surfe*}
- **Preço total**: R$188,00
- **Somatório das distâncias**: 89 km

É possível verificar as propriedades de cada tipo de opção de minimização em cada um dos conjuntos resposta das consultas. Uma situação prática de uso de cada *min-op* seria:

- *min-min*: O hóspede está ansioso pela visita ao local e deseja chegar o mais rápido o possível em algum passeio oferecido pelo hotel.

- *min-max*: Uma família está planejando os passeios que farão durante 6 dias de estadia no hotel e desejam estar próximos do hotel mesmo no passeio mais longe, em caso de emergência.
- *min-sum*: Um casal deseja passear todos os dias da viagem, tendo de se deslocar a menor distância total possível.

E para finalizar, as operações de mínimo e máximo possuem um formato similar a operação de contagem, dado que exigem a definição de um *agAtr* e um *c*, sem precisar definir uma *min-op* em virtude desta opção de minimização não afetar os operadores *MIN* e *MAX*. A semântica das consultas desses operadores é restringir os limites inferior e superior da busca dos k vizinhos mais próximos com condições considerando a *tcond* fornecida. O autor demonstra a não necessidade de se usar uma *min-op*, além de comentar sobre as diferenças entre o sentido conferido quando se utiliza $\theta \in \geq, >$ e $\leq, <$

Um exemplo do uso de operação de mínimo, juntamente com sua notação foi extraído de [2] é considerando um conjunto de cidades americanas, onde existem dados dos nomes das cidades, estado, população, coordenadas e porcentagem de pessoas que utilizam meios de transportes públicos é:

> "Retorne as 20 cidades mais próximas de *Bistrol city*, de forma que o resultado inclua cidades com 100 mil habitantes ou mais, ou cidades com menos de 100 mil habitantes cuja porcentagem de habitantes que usa transporte público seja, no mínimo, 5%."

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,20,MIN(usaTransportePublico,pop<100000)\geq 5\%]\ BistrolCoord}\ (CidadesAmericanas) \tag{2.20}$$

No conjunto resposta desta consulta, pode existir somente cidades com mais de 100.000 habitantes, porém, caso existam cidades com menos de 100.000, a menor porcentagem de pessoas que utilizam transporte público tem que ser, no mínimo, 5%. Caso a *tcond pop* < 100000 for retirada, todas as cidades pertencentes ao conjunto resposta deverão ter a porcentagem de pessoas que utilizam transporte público maior ou igual a 5%, resultando na notação:

$$\ddot{\sigma}_{coord\ {}_{c}kNN[L_2,20,MIN(usaTransportePublico)\geq 5\%]\ BistrolCoord}\ (CidadesAmericanas) \tag{2.21}$$

3 ESTRUTURAS DE INDEXAÇÃO PARA DADOS COMPLEXOS

Este capítulo será destinado para a apresentação e detalhamento das estruturas que realizam o processo de indexar os vetores de características criados a partir dos dados complexos, como explicado na seção 2.1.2.

Antes de realizar a apresentação das estruturas, é importante definir o conceito de métodos de acesso métrico (*metric access methods* - MAM). Um MAM deve ser capaz de organizar uma grande quantidade de objetos em um espaço métrico, partindo do pressuposto que apenas a função de distância, satisfazendo as regras de simetria, não negatividade e desigualdade triangular, está disponível para ser utilizada. Dessa maneira, as operações primitivas como adição e subtração não estão disponíveis [12]. Os MAMs a serem explorados a seguir, possuem suporte para principalmente buscas do tipo *range queries* e *k-nearest neighbors queries*. Assim como descrito por [12], a eficiência de um MAM é influenciada por alguns fatores como: número de acesso de disco, custo computacional para realizar os cálculos de distância entre os objetos no espaço e a quantidade de armazenamento utilizada para indexar todos os dados.

Foram selecionadas ao total 7 abordagens distintas para solucionar problemas encontrados durante o processo de tratar e armazenar os dados complexos nos SGBDs. As 6 primeiras abordagens que serão apresentadas conseguem resolver qualquer tipo de consulta por similaridade com condições adicionais, enquanto a última irá tratar de um caso especial de consulta por similaridade.

3.1 Estruturas para consultas por similaridade

Existem diversos tipos de estruturas para realizar buscas por similaridade, cada uma com suas peculiaridades e características. Podemos dividir essas estruturas em alguns grupos como: estruturas baseadas em árvores como a *Slim-tree* [12], a *DBM-tree* [35] e a *M-tree* [36], estruturas baseadas em técnicas de *hashing* como o LSH (*Locality Sensitive Hashing*) [37] e estruturas baseadas em grafos de proximidade [38, 39]. Na presente seção, será apresentada uma descrição da estrutura *Slim-tree*, a qual é utilizada como base das próximas estruturas a serem apresentadas neste trabalho. A escolha desta estrutura se deu em virtude de outros autores como [2, 3] terem desenvolvido trabalhos utilizando-a como base para a criação de novas estruturas e algoritmos de indexação para dados complexos.

3.1.1 *Slim-tree*

A *Slim-tree* proposta por [12] é um MAM dinâmico, e foi criada com o objetivo de aumentar o desempenho de consultas por similaridade, através de algoritmos de particionamento dos objetos no espaço, e fatores que medem a quantidade de *overlaps* ocorridos durante esse processo.

A *Slim-tree* é uma árvore balanceada e dinâmica que cresce no sentido *bottom-up*, a partir das folhas até a raiz. Todos os objetos do espaço métrico são armazenados nas folhas. Assim como em outras árvores métricas, eles são agrupados em páginas de disco com espaço fixo. Cada página representa um nó da árvore em questão. Dessa maneira, uma estrutura hierárquica é criada através da utilização de um elemento representante como centro da região de cobertura dos elementos presentes nas subárvores. Uma característica importante dessa estrutura, é que os cálculos de distâncias, entre o novo elemento e o elemento representante, são realizados durante a inserção e persistidos na árvore, fato que colabora para aumentar o desempenho desse MAM, como explicado na seção 3.

Existem 2 tipos de nós nessa estrutura:

- Nós folha: armazenam um vetor de tuplas de 3 elementos no formato:

$$< id_i, d(s_i, s_{rep}), s_i >$$

  em ordem, temos o identificador do elemento, a distância em relação ao elemento representante e o elemento propriamente dito.

- Nós índice: armazenam um vetor de tuplas de 5 elementos no formato:

$$< s_i, r_i, d(s_i, s_{rep}), Ptr(T_{s_i}), \#Ent(Ptr(T_{s_i})) >$$

  onde s_i é o elemento representante da subárvore apontada por $Ptr(T_{s_i})$, r_i é o raio da região circular coberta pelo nó (distância entre o representante e o elemento mais distante desse nó), $d(s_i, s_{rep})$ é a distância entre s_i e s_{rep}, por fim, $\#Ent(Ptr(T_{s_i}))$ é o número de entradas da subárvore apontada por $Ptr(T_{s_i})$.

As regiões circulares presentes nos nós da árvore podem se sobrepor, o que colabora para o aumento do número de caminhos e nós a serem visitados durante as buscas por similaridade. A figura 6 ilustra os elementos representantes como pontos pretos, enquanto os outros elementos são representados em cinza. Cada círculo branco representa um nó folha, e cada círculo cinza um nó índice. É possível observar também a sobreposição de

regiões como ocorre nos pontos S10, S15 e S16, fato que torna possível perceber que a inserção de novos pontos pode elevar consideravelmente a quantidade de interseções entre as regiões.

No contexto de buscas por similaridade com condições estendidas, é importante ressaltar que essa estrutura foi aprimorada por [2] com a criação de dois algoritmos: *Table-Slim* e *Covering-Slim*. Esses dois algoritmos possibilitam a verificação das condições adicionais presentes nas consultas realizadas nas estruturas:

- *Table-Slim*: Em uma *Slim-tree* sem modificações, os vetores de características e o *rowId* para cada elemento indexado são armazenados. Durante uma consulta, primeiro verifica-se a dissimilaridade entre os elementos e depois as condições adicionais. Como na estrutura original apenas o vetor de características e *rowId* são armazenados, para realizar a verificação das condições, utiliza-se o *rowId* recuperado do índice para acessar o arquivo de dados e recuperar o restante das informações do elemento. As principais vantagens deste algoritmo, como mostrado em [2, 3], estão na velocidade superior em relação à busca sequencial para condições com seletividade moderada, além disso, apenas a existência do índice no atributo complexo é necessária, permitindo a capacidade de responder à condições com quaisquer outros atributos da tabela. A principal desvantagem está no elevado número de acessos a disco para realizar a verificação das condições adicionais da consulta.

- *Covering-Slim*: Esse algoritmo implica na modificação das informações armazenadas nos nós índice, dado que agora eles irão armazenar também atributos simples. Assim como realizado no algoritmo *Table-Slim*, primeiro verifica-se a dissimilaridade entre os elementos e depois, para as condições adicionais, como o(s) atributo(s) já está(ão) no índice, existe uma redução drástica do número de acessos a disco. O desempenho desse algoritmo mostrado por [2, 3] é sempre maior que o *Table-Slim* e muito mais rápido que uma busca sequencial. Os pontos negativos são: a necessidade da existência de um índice de cobertura apropriado e pior desempenho quando comparado com a busca sequencial em situações onde as condições possuem alta seletividade.

3.1.1.1 Construção da *Slim-tree*

A construção da *Slim-tree* é composta de algumas etapas. O processo inicia com a inserção de um novo elemento na árvore, a partir da raiz, buscando um nó que consiga abranger o elemento. Caso não exista, o nó com o centro mais próximo é escolhido. Caso existam 2 ou mais nós possíveis, o algoritmo de seleção de subárvore é executado para decidir. Esses passos são executados recursivamente para todos os elementos inseridos na *Slim-tree*. Caso um nó da árvore n não comporte mais elementos, um novo nó n' é alocado

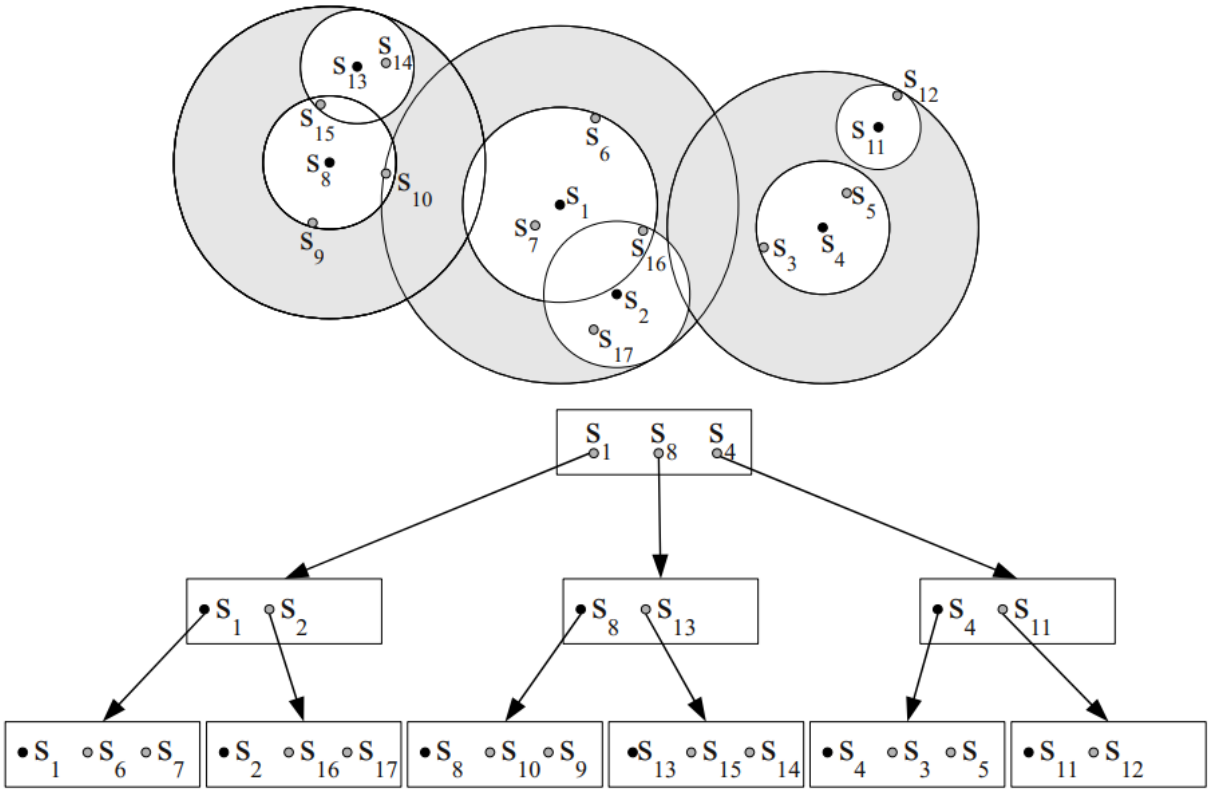

Figura 6 – Representação gráfica da *Slim-tree*, extraído de [2]

no mesmo nível e os elementos são redistribuídos. Quando o nó raiz sofre o processo de *split*, uma nova raiz é criada em um nível superior e a árvore cresce um nível.

Em relação aos algoritmos disponíveis para a escolha da subárvore, existem 3 alternativas:

- *random*: Escolha aleatória de um dos nós aptos.
- *mindist*: Escolha do nó que possui a menor distância do seu centro em relação ao novo elemento inserido.
- *minoccup*: Escolha do nó com a menor ocupação possível.

O método *minoccup* utiliza o atributo $\#Ent(Ptr(T_{s_i}))$ disponível nos nós índice para realizar a verificação da ocupação. Ainda que armazenar essa informação utilize mais espaço, os autores mostram que apenas um *byte* é suficiente, possibilitando uma queda no número de acessos a disco, em consequência do aumento da taxa de ocupação de nós. Outra categoria importante de algoritmos, que são utilizados para construir a árvore, são os utilizados durante o processo de *split* visando a criação de novos nós. Os autores enumeraram algumas opções de algoritmos:

- *Random*: O mais rápido entre os três. Dois novos centros são aleatoriamente selecionados, e todos os outros objetos presentes no nó são distribuídos em relação a eles. É importante ressaltar que o critério envolvendo a ocupação mínima dos novos nós deve ser respeitado.

- *minMax*: Utilizado também na *M-tree*, esse algoritmo é o mais promissor quando estamos falando sobre *performance* durante a consulta. Todas as combinações possíveis de pares de nós são criadas, e logo em seguida, submetidas a testes do raio de cobertura necessário para encobrir todos os objetos. O par que obtiver o menor raio de cobertura possível irá ser escolhido.

- *MST (minimum spanning tree)*: Possibilita a construção das *Slim-trees* com *performance* parecida com o *minMax*, porém mais rápido.

De maneira simples, os autores propuseram o processo de *split* de um nó utilizando uma *MST* seguindo os passos:

1. Construa a *MST*.
2. Remova a maior aresta.
3. Crie 2 grupos distintos resultantes do passo 2.
4. Escolha um representante de cada grupo seguindo um critério pré-estabelecido.

3.1.1.2 Desafios da *Slim-tree*

Um dos principais desafios encontrados pelos autores durante a elaboração da *Slim-tree*, foi criar maneiras avaliar a qualidade da organização dos objetos em uma árvore métrica através de um único número. Isso leva em consideração, principalmente, a quantidade de *overlaps* (número de elementos que estão presentes em mais de uma região simultaneamente) presentes na árvore, dado que isso aumenta a quantidade de nós a serem visitados durante uma busca. Para circunvir essa dúvida, foram criados duas métricas que são capazes de avaliar a qualidade da distribuição: o *fat-factor* e o *bloat-factor*.

3.1.1.3 *Fat-Factor*

O *Fat-Factor* é uma medida de desempenho criada para avaliar a qualidade da construção da árvore como mencionado anteriormente. Para definir o *Fat-Factor*, os autores partiram de dois pressupostos: primeiro, a avaliação da qualidade da árvore será feita apenas através de buscas por abrangência (Rq), e segundo, a distribuição dos centros de cada nó irá seguir a distribuição dos objetos, sendo assim, eles esperam que as consultas tendam a visitar regiões com alta densidade de objetos. Outra parte importante sobre o

primeiro pressuposto é que os autores comentam sobre a possibilidade de aplicação do *Fat-Factor* para consultas do tipo *k-NNq*, em consequência de serem um tipo especial de *Rq*. Os valores possíveis para esse fator estão no intervalo de $[0, 1]$, onde valores próximos de 0 representam árvores ideais e valores próximos de 1 representam os piores cenários de *overlaps*.

A equação 3.1 descreve os cálculos realizados para a obtenção do *Fat-Factor*, com as variáveis:

- T: Árvore métrica
- H: Altura da árvore
- M: Quantidade de nós ($M \geq 1$)
- N: Número de elementos
- I_c: Número total de acessos aos nós para responder uma busca Rq com $r = 0$ (*point query*) para cada um dos N elementos presentes na árvore

$$fat(T) = \frac{I_c - H * N}{N} * \frac{1}{(M - H)} \tag{3.1}$$

As figuras 7 e 8 foram adaptadas de [12] e mostram os fatores calculados utilizando a equação 3.1. A linha pontilhada em cada uma das figuras delimita o nó raiz. A tabela 2 exibe os valores das variáveis e o resultado para cada uma das figuras:

Figura	H	M	N	I_c	$fat()$
7	2	4	6	12	0
8	2	3	6	14	$\frac{1}{3}$

Tabela 2 – Valores para o cálculo do *Fat-factor*

3.1.1.4 *Bloat-Factor*

A utilização do *Fat-factor* pressupõe que as árvores tenham o mesmo número de nós, de maneira que a árvore com o menor *Fat-factor* possui um número menor de *overlaps* de regiões, ocasionando uma diminuição da quantidade de acessos a disco. Para um mesmo conjunto de elementos, considerando duas *Slim-trees* distintas T_1 e T_2 com $M_1 > M_2$, o fato do número de nós não ser igual implica na impossibilidade de comparação dos *Fat-factors* de cada uma das árvores, uma vez que T_2 possivelmente terá um *Fat-factor* maior pelos raios de cobertura dos nós serem maiores. Entretanto, a quantidade de acessos a disco pode ser menor para T_2, uma vez que menos nós terão de ser acessados para realizar uma busca em T_2.

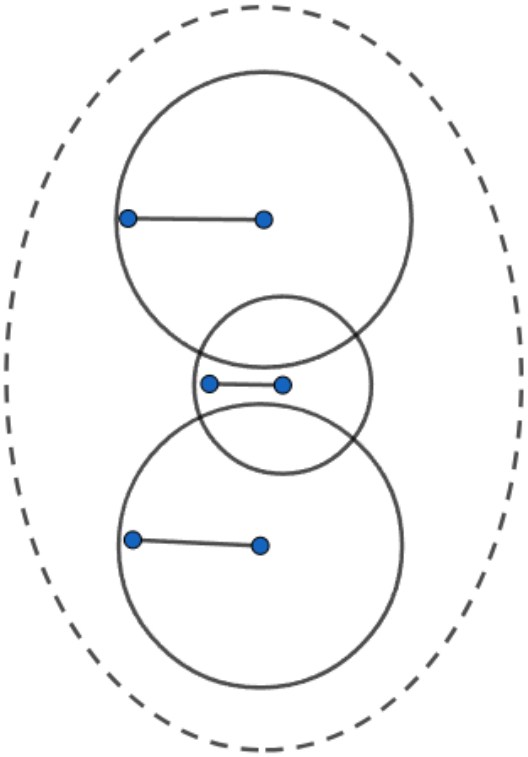

Figura 7 – *Slim-tree* sem *overlap*

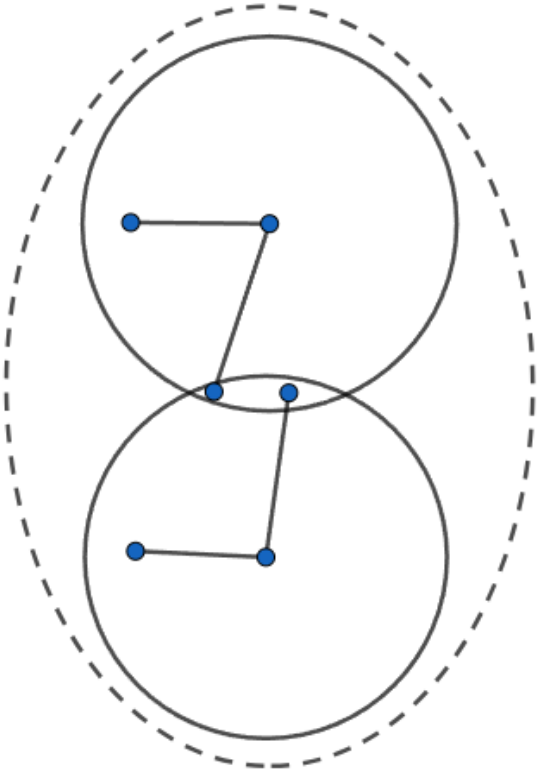

Figura 8 – *Slim-tree* com *overlap*

Os autores comentam sobre a necessidade de penalizar árvores que utilizam mais nós do que o necessário. Para isso, eles trocaram as variáveis utilizadas no *Fat-factor* para levar em consideração a altura mínima e número mínimo de nós necessários para construção da árvore. Dessa maneira, o *Bloat-factor* foi criado para realizar a comparação entre árvores com diferentes números de nós, sendo definido pela equação 3.2 com as variáveis:

- T: Árvore métrica
- H_{min}: Altura da árvore mínima
- M_{min}: Quantidade de nós mínima ($M \geq 1$)
- N: Número de elementos
- I_c: Número total de acessos aos nós para responder uma busca Rq com $r = 0$ (*point query*) para cada um dos N elementos presentes na árvore

Os valores possíveis do *Bloat-factor* estão no intervalo $[0, \infty)$, e de maneira semelhante ao *Fat-factor*, quanto menor o seu valor, melhor é a árvore em análise.

$$bl(T) = \frac{I_c - H_{min} * N}{N} * \frac{1}{(M_{min} - H_{min})} \tag{3.2}$$

3.1.1.5 Algoritmo *Slim-down*

Levando em consideração as métricas apresentadas anteriormente (*Fat-factor* e *Bloat-factor*), elas são utilizadas para mostrar se existe possibilidade de melhorias a serem feitas nas *Slim-trees*. A maneira que os autores apresentaram uma técnica de diminuição de regiões de interseção, consequentemente reduzindo os valores obtidos nas métricas, é através de um algoritmo de reorganização de elementos entre os nós de um determinado nível da árvore chamado *Slim-down*.

A execução do algoritmo segue algumas etapas:

1. Para cada nó i em um nível da árvore, encontre o elemento mais distante c em relação ao representante b.

2. Encontre o nó irmão j de i, de tal maneira que j também cobre c. Se j existir e não estiver cheio, mova o elemento c do nó i para o nó j. Em seguida, corrija o raio de cobertura de i.

3. Repetir os procedimentos 1 e 2 para todos os nós presentes no nível, até que nenhum elemento tenha movido de lugar.

3.1.1.6 Algoritmo de *Range Queries* na *Slim-tree*

A busca por abrangência na *Slim-tree* segue a abordagem *branch-and-bound* onde a ordem de busca inicia-se a partir da raiz da estrutura de indexação [2, 36]. E quando os dados estão em um espaço métrico, a utilização da propriedade de desigualdade triangular é indispensável para realizar podas durante o processo de busca [2, 40]. Então, considerando um espaço métrico $\mathbb{M} = < \mathbb{S}, \delta >$, com um conjunto de elementos $S \subseteq \mathbb{S}$, com um elemento de referência $s_q \in \mathbb{S}$, um elemento representante de um nó do MAM e um limite de dissimilaridade ξ e um elemento $s_i \in \mathbb{S}$, esse processo é mostrado no algoritmo 1:

3.1.1.7 Algoritmo de *k-Nearest Neighbors Queries* na *Slim-tree*

O algoritmo implementado na *Slim-tree* para realizar consultas pelos k vizinhos mais próximos é encontrado no algoritmo 2, em contraste com o algoritmo de busca por abrangência, não possui um raio de abrangência pré-definido. Dessa maneira o raio é atualizando de forma dinâmica durante a execução do algoritmo. O algoritmo implementado na *Slim-tree* é chamado *Best-First k-NN* que utiliza uma fila de prioridade com a prioridade sendo a distância mínima (*mindist*) entre o elemento de consulta e a região que uma subárvore cobre. Existe também a utilização da menor distância máxima (*minmaxdist*) entre o elemento de consulta e a região que uma subárvore cobre para reduzir o raio de abrangência em vigor. As definições dessas duas distâncias utilizadas no algoritmo são:

Algoritmo 1: Busca por abrangência na *Slim-tree*

Entrada: *node*, s_q, ξ
Saída: result

```
1  início
2      result ← ∅;
3      se node é um nó índice então
4          para cada s_i em node faça
5              se |δ(s_q, s_rep) − δ(s_rep, s_i)| ≤ ξ + r_i então
6                  distance ← δ(s_q, s_i);
7                  se distance ≤ ξ + r_i então
8                      retorna rangeQuery(Ptr(T_{s_i}), s_q, ξ)
9      senão
10         para cada s_i em node faça
11             se |δ(s_q, s_rep) − δ(s_rep, s_i)| ≤ ξ + r_i então
12                 distance ← δ(s_q, s_i);
13                 se distance ≤ ξ então
14                     result.add(s_i, distance);
15     retorna result;
16 fim
```

$$mindist(s_q, T_{s_i}) = max\{\delta(s_q, s_i), 0\} \tag{3.3}$$

$$minmaxdist(s_q, T_{s_i}) = \delta(s_q, s_{rep}) + r_i \tag{3.4}$$

3.2 Estruturas para consultas por similaridade com condições estendidas

A partir da seção 3.1.1, foi possível observar que a *Slim-tree* é uma estrutura de indexação que possui um bom desempenho para consultas por similaridade, porém, a verificação das condições adicionais nos atributos tradicionais é dependente do acesso ao arquivo de dados, podendo causar lentidão durante o processamento da consulta. Dessa maneira, outro autor propôs novas estruturas de indexação que utilizam a *Slim-tree* como base, e aprimoraram-na para que fosse possível realizar consultas por similaridade com condições estendidas, tentando evitar o acesso ao disco por meio índices criados em outras estruturas.

As próximas 5 soluções conseguem conferir um maior desempenho para a consulta por criar índices que armazenam alguns atributos tradicionais, visando evitar o acesso ao

Algoritmo 2: Busca por k-vizinhos mais próximos na *Slim-tree*

```
   Entrada: node, s_q, k
   Saída: result
 1 início
 2     result ← ∅;
 3     priorQueue ← ∅;
 4     result.setMaxDistance(∞);
 5     enquanto priorQueue ≠ ∅ faça
 6         node ← priorQueue.removeMin();
 7         se node é um nó índice então
 8             para cada s_i em node faça
 9                 se |δ(s_q, s_rep) − δ(s_rep, s_i)| ≤ result.getMaxDistance() + r_i
                    então
10                     minDistance ← mindist(s_q, T_{s_i});
11                     se minDistance ≤ result.getMaxDistance() então
12                         priorQueue.add(Ptr(T_{s_i}), minDistance);
13                         minMaxDistance ← minmaxdist(s_q, T_{s_i});
14                         se minMaxDistance ≤ result.getMaxDistance() então
15                             result.setMaxDistance(minMaxDistance);
16                             remova todas as entradas de priorQueue onde
                                mindist(s_q, T_{s_i}) > minMaxDistance;
17         senão
18             para cada s_i em node faça
19                 se |δ(s_q, s_rep) − δ(s_rep, s_i)| ≤ result.getMaxDistance() então
20                     distance ← δ(s_q, s_i);
21                     se distance ≤ result.getMaxDistance() então
22                         result.add(s_i, distance);
23                         se result.getNumElements() > k então
                            result.cutElements(k);
24                         se result.getMaxDistance() foi atualizado então
25                             remova todas as entradas de priorQueue onde
                                mindist(s_q, T_{s_i}) > result.getMaxDistance();
26             retorna result;
27 fim
```

disco para verificar as condições adicionais. Além disso, elas foram criadas especialmente para consultas do tipo ${}_cKNNq$.

3.2.1 *cx-Sim* Simple Tree*

Considerando que atributos adicionais, associados aos dados complexos, conseguem conferir maior precisão nas buscas de um SGBD, o autor em [3] propôs um novo MAM: a *cx-Sim tree (Condition-eXtended Similarity tree)*. Essa estrutura, como o autor

descreve, tem a capacidade de indexar múltiplos atributos, dinâmica, composta por duas camadas e foi criada para ser armazenada em disco.

3.2.1.1 Estrutura geral da *cx-Sim* Simple Tree*

A estrutura é composta pela união de duas ou mais estruturas, sendo que para dados complexos deve-se usar um método de acesso métrico respeitando as propriedades expostas na seção 2.1.3, e para os atributos tradicionais um método de acesso ordenado (geralmente uma B^+-*tree*). Isso configura a possibilidade de realizar comparações com os operadores de comparação tradicionais $(>, \geq, <, \leq)$, o que antes não era possível, além de possibilitar ao usuário a execução de buscas mais precisas em um SGBD com dados complexos. Outra vantagem proporcionada pela estrutura, é o fato da *cx-Sim tree* reduzir a quantidade de elementos no espaço métrico, em consequência do particionamento dos dados complexos. Tal fato ocorre, dado que em seu segundo nível apenas potenciais elementos que possuem o atributo condizente com a condição verificada no primeiro nível, estarão no espaço de busca. Dessa maneira, torna-se possível reduzir duas operações custosas para o desempenho das consultas: acessos a disco e cálculos de distâncias.

Acerca do formato da estrutura, como mencionado anteriormente, ela é separada em 2 camadas:

- Camada 1 - Realiza a indexação e armazenamento dos atributos tradicionais, utilizados na verificação das condições adicionais presentes na busca. A estrutura escolhida pelo autor é a B^+-*tree* considerando fatores como tempo de busca logarítmico e consultas envolvendo intervalo de valores. Um exemplo de consulta com intervalo seria: *"Retorne os 4 restaurantes mais próximos da minha localização, com capacidade de lotação entre 50 a 100 pessoas"*.
- Camada 2 - Responsável por indexar os dados complexos. Levando em consideração a possibilidade de existir valores como resposta da camada 1, nesta camada serão criadas florestas de árvores dinâmicas, como *R-trees*, *M-trees* ou a *Slim-tree* [12]. A escolha do tipo de árvore será decidida em função da natureza dos dados complexos.

O autor ressalta que a utilização de *Slim-trees* na camada 2 da *cx-Sim* Simple Tree* deve-se ao fato da possibilidade de usar algoritmos *Table-Slim*, para que assim, após verificar a primeira condição de similaridade, as condições envolvendo atributos simples são viabilizadas através do acesso ao arquivo de dados mediante os *rowId* recuperados. Dessa maneira, operações custosas como cálculo de distâncias e acessos a disco serão processados somente após filtragem dos dados em potencial, ademais, esse processo é executado somente se a condição de busca for composta. A figura 9 mostra a estrutura da *cx-Sim* Simple Tree*, com o fluxo de uma busca com o algoritmo *Table-Slim* destacado com as setas em cinza-claro.

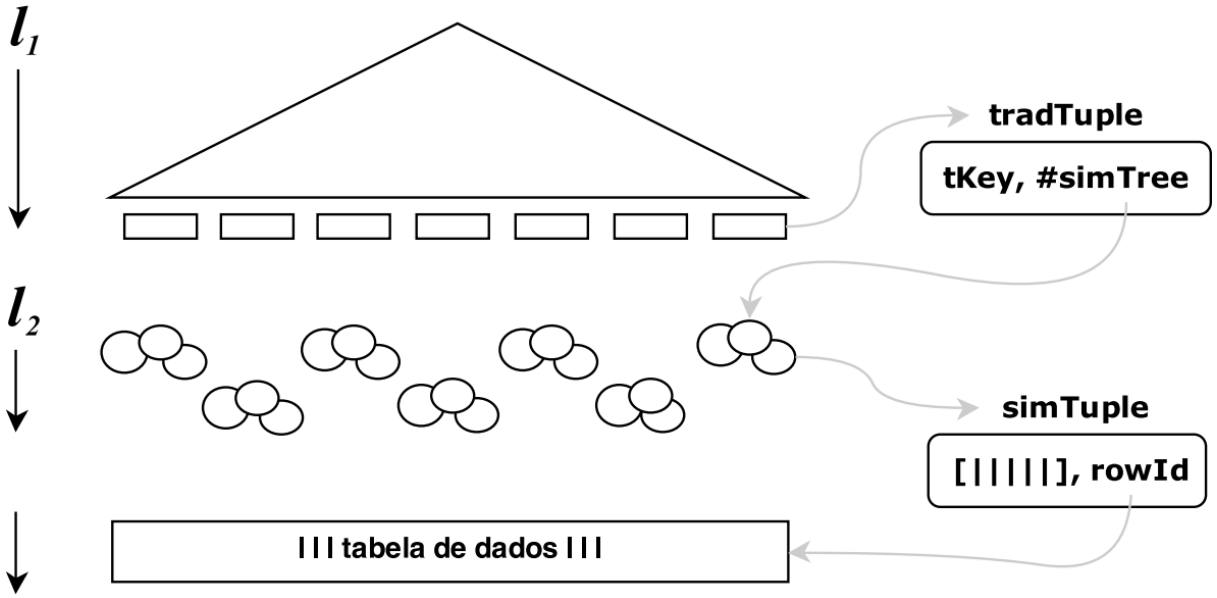

Figura 9 – Representação gráfica da *cx-Sim* Simple Tree*, extraído de [3]

3.2.1.2 Funcionamento da *cx-Sim* Simple Tree*

Quando a *cx-Sim* Simple Tree* é construída para indexar apenas um atributo tradicional e um complexo, o primeiro é indexado na B^+-*tree*, enquanto o segundo, por escolha do autor, é inserido em uma *Slim-tree*. A inserção consiste em 2 etapas:

1. Busca pelo atributo tradicional na B^+-*tree*. Caso o valor não seja encontrado, o elemento é inserido na B^+-*tree* com uma referência para uma nova *Slim-tree*, caso contrário, o algoritmo acessa a *Slim-tree* referenciada.

2. Inserção na *Slim-tree*. Utilizando informações como o vetor de características e o *rowId*, o algoritmo faz uma busca recursiva nos nós da árvore objetivando localizar o nó que irá sofrer o menor incremento da área de cobertura (consequentemente reduzindo a quantidade de sobreposições e mantendo os medidores *Fat-factor* e *Bloat-factor* menores o possível). Quando o nó é encontrado, o elemento é inserido e o raio de cobertura é atualizado.

A busca na *cx-Sim* Simple Tree* opera de maneira semelhante à inserção, consistindo em primeiro realizar a busca nas B^+-*trees* com a utilização dos comparadores tradicionais ($>, \geq, <, \leq, =, \neq$), já que estamos lidando com atributos simples/tradicionais e depois nas *Slim-trees* referenciadas. É importante ressaltar que em consultas com condições compostas envolvendo outros atributos tradicionais, é necessário recuperar o(s) *rowIds* das tuplas que satisfazem a condição do atributo tradicional indexado na B^+-*tree*, para então, acessar o arquivo de dados e recuperar o restante das informações do elemento.

O autor em [3] comenta sobre a capacidade que a *cx-Sim* Simple Tree* possui de responder consultas mesmo com condições compostas, envolvendo múltiplos atributos tradicionais, mediante acesso ao arquivo de dados. Entretanto, esse processo pode ser otimizado de maneira que a partir da construção de um índice de cobertura [2] contendo mais atributos tradicionais. Dessa maneira, o autor propôs 3 variações da *cx-Sim* Simple Tree* para múltiplos atributos tradicionais visando alterar algumas partes da estrutura básica para reduzir o número de acessos a disco efetuados durante a consulta.

3.2.2 *cx-Sim* Covering Tree*

Esta abordagem é fortemente inspirada no algoritmo de *Covering-slim* apresentado na seção 3.1.1 por corresponderem à definição de índice de cobertura. Os atributos adicionais, com exceção do primeiro, já estão presentes na *Slim-tree* com o vetor de características e o endereço físico da tupla (*rowId*) nas folhas da árvore. Isso implica na redução significativa de acessos ao arquivo de dados durante a verificação das condições adicionais. Em relação às mudanças feitas na estrutura, temos alterações na segunda camada da *cx-Sim* Simple Tree*, onde esta irá armazenar além do vetor de características, os outros atributos adicionais que serão utilizados nas comparações a serem feitas durante as buscas. Essa mudança é mostrada na figura 10 na *simTuple* com a presença da tKey2 (podendo haver mais chaves, como comentado anteriormente).

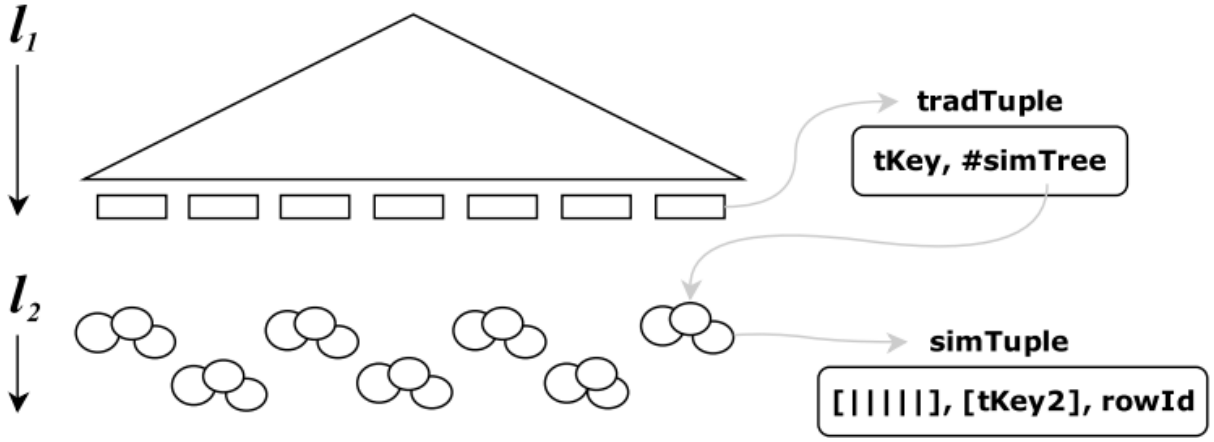

Figura 10 – Representação gráfica da *cx-Sim* Covering Tree*, extraído de [3]

A inserção nessa estrutura segue o padrão da inserção na *cx-Sim* Simple Tree*, em 2 etapas:

1. Busca pelo atributo tradicional na B^+-*tree*. Caso o valor não seja encontrado, o elemento é inserido na B^+-*tree* com uma referência para uma nova *Covering-Slim-tree*, caso contrário, o algoritmo acessa a *Covering-Slim-tree* referenciada.

2. Inserção na *Covering-Slim-tree*. Utilizando informações como o vetor de características e o *rowId*, o algoritmo faz uma busca recursiva nos nós da árvore objetivando

localizar o nó que irá sofrer o menor incremento da área de cobertura (consequentemente reduzindo a quantidade de sobreposições e mantendo os medidores *Fat-factor* e *Bloat-factor* menores o possível). Quando o nó é encontrado, o elemento é inserido e o raio de cobertura é atualizado.

A busca na *cx-Sim* Covering Tree* opera de maneira semelhante à inserção, consistindo em primeiro realizar a busca nas B^+-*trees* com os comparadores tradicionais $(>, \geq, <, \leq, =, \neq)$ e depois nas *Covering-Slim-trees* referenciadas. Caso o tipo de consulta for uma busca por abrangência, a estratégia apresentada no algoritmo 1 de *branch-and-bound* é utilizada para realizar as podas nas subárvores não promissoras. Em caso de uma busca dos k vizinhos mais próximos, o algoritmo 2 é utilizado para realizar a busca.

A principal vantagem levantada pelo autor ao utilizar a variação *cx-Sim* Covering Tree* é a redução da quantidade de acessos a disco em decorrência da pós-validação dos atributos adicionais, não necessitando do acesso da tabela de dados.

3.2.3 *cx-Sim* Chained Tree*

Nessa variação da *cx-Sim* Simple Tree*, o autor experimentou a possibilidade de alterar o nível l_1, local onde os atributos tradicionais são indexados. Dessa maneira, o objetivo dessa mudança é realizar mudanças na estrutura do primeiro nível que contém as B^+-*trees* para que assim, as condições envolvendo os atributos simples sejam verificados durante a travesia do nível 1. Isso se torna possível através da utilização de um encadeamento de B^+-*trees* (no caso, duas), onde o primeiro atributo é indexado na primeira B^+-*tree* e o segundo atributo é indexado na segunda B^+-*tree*, isto é nos níveis $l_1^{'}$ e $l_1^{''}$. É importante ressaltar que no nível $l_1^{'}$, os nós folha irão armazenar tuplas no formato: *<tKey, #bTree>*, onde *tKey* é o valor do atributo tradicional e *#bTree* é um ponteiro para a B^+-*tree* que armazena o outro atributo adicional.

O nível 2, onde as florestas de *Slim-trees* são armazenadas, permanece igual à estrutura da *cx-Sim* Simple Tree* (em contraste com a *cx-Sim* Covering Tree* que utiliza *Covering-Slim-trees*). A figura 11 mostra a estrutura da *cx-Sim* Chained Tree*, com o fluxo de uma busca com o algoritmo *Table-Slim* destacado com as setas em cinza-claro.

A inserção com essa mudança no nível l_1 segue o padrão da inserção na *cx-Sim* Simple Tree*, em 2 etapas:

1. Busca pelo atributo tradicional na primeira B^+-*tree* (como existem duas B^+-*trees*, a busca é realizada nas duas de maneira sequencial, primeiro no nível $l_1^{'}$ e depois no nível $l_1^{''}$). Caso o(s) valor(es) não forem encontrados, eles são inseridos na B^+-*tree* sendo que o primeiro atributo é inserido no nível $l_1^{'}$ com uma referência para uma nova B^+-*tree* no nível $l_1^{''}$, caso contrário, o algoritmo acessa a B^+-*tree* referenciada.

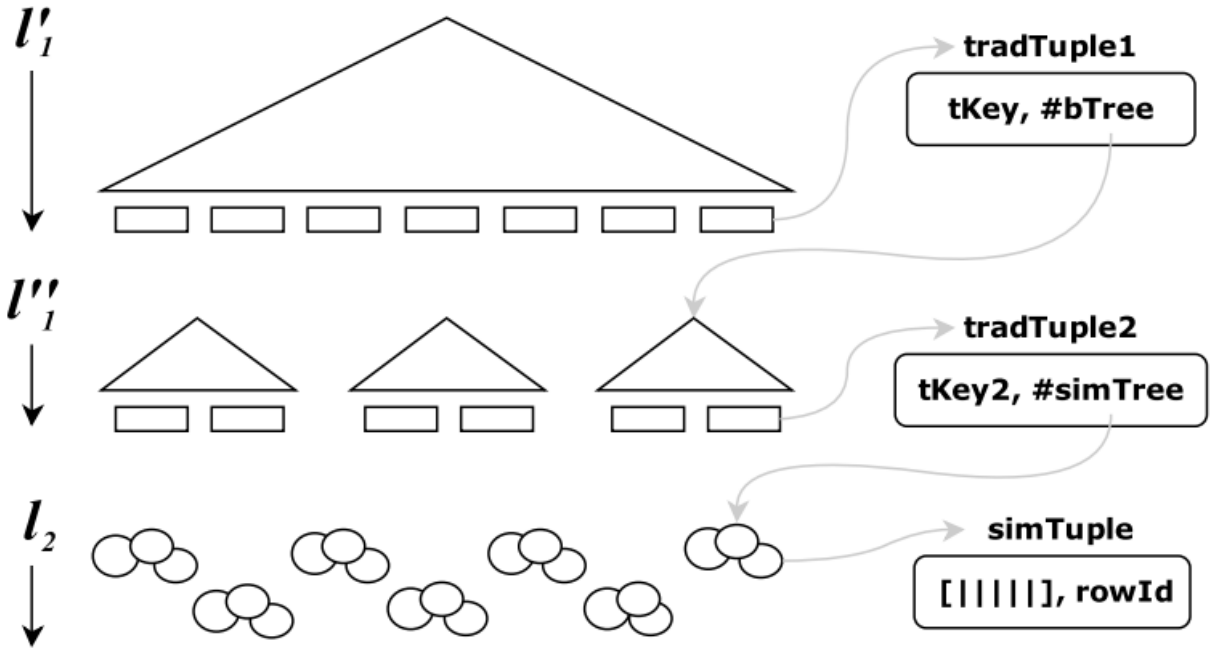

Figura 11 – Representação gráfica da *cx-Sim* Chained Tree*, extraído de [3]

2. Inserção na *Slim-tree*. Opera de forma igual à inserção na *cx-Sim* Simple Tree*.

A busca nesse estrutura segue o padrão encontrado na *cx-Sim* Simple Tree*, com a diferença que a busca no nível l_1 é realizada nas duas B^+-*trees* (no nível $l_1^{'}$ e depois no nível $l_1^{''}$), utilizando os operadores de comparação tradicionais ($>, \geq, <, \leq, =, \neq$) durante a travessia na B^+-*tree*. Caso o tipo de consulta for uma busca por abrangência, a estratégia apresentada no algoritmo 1 de *branch-and-bound* é utilizada para realizar as podas nas subárvores não promissoras. Em caso de uma busca dos k vizinhos mais próximos, o algoritmo 2 é utilizado para realizar a busca.

As vantagens apresentadas pelo autor são: a pré-validação das condições envolvendo atributos simples antes de acessar o segundo nível da estrutura onde a verificação complexa é executada, e a redução do tamanho do índice no disco, já que não há repetições das chaves de busca no primeiro nível.

3.2.4 *cx-Sim* Composite Tree*

Para finalizar as variações da *cx-Sim* Simple Tree*, o autor propôs uma estrutura com outra alteração no nível 1, onde a chave presente nas folhas da B^+-*tree* é uma composição dos atributos tradicionais. Dessa maneira, o primeiro nível terá uma B^+-*tree* com as chaves no formato *<tKey1, tKey2, ..., tKeyN, #simTree>*, onde *tKey1, tKey2, ..., tKeyN* são os atributos tradicionais e *#simTree* é um ponteiro para a *Slim-tree* que armazena os dados complexos. No segundo nível da estrutura, não existe alteração, permanecendo

igual à *cx-Sim* Simple Tree*, isto é, nesse nível temos apenas os atributos complexos, sem a presença de atributos tradicionais.

A figura 12 mostra a estrutura da *cx-Sim* Composite Tree*, com o fluxo de uma busca com o algoritmo *Table-Slim* destacado com as setas em cinza claro.

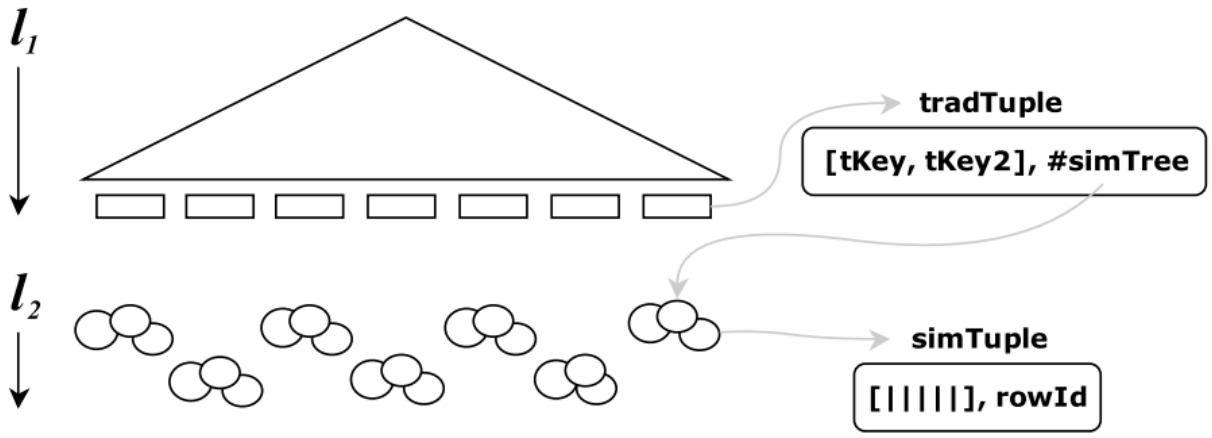

Figura 12 – Representação gráfica da *cx-Sim* Composite Tree*, extraído de [3]

A inserção considerando que no primeiro nível é uma composição de atributos tradicionais segue o padrão da inserção na *cx-Sim* Simple Tree*, em 2 etapas:

1. Busca pela chave composta pelos atributos adicionais. Caso o valor não seja encontrado, a chave composta é inserida na B^+-*tree* com uma referência para uma nova *Slim-tree*, caso contrário, o algoritmo acessa a *Slim-tree* referenciada. É importante notar que combinações dos atributos-chave serão armazenados no primeiro nível, reduzindo o tamanho do índice.

2. Inserção na *Slim-tree*. Opera de forma igual à inserção na *cx-Sim* Simple Tree*.

Por utilizar chaves compostas no primeiro nível, a busca usa a noção de *comparator* para realizar a busca na B^+-*tree*, com operadores tradicionais de comparação ($>, \geq, <, \leq, =, \neq$). Dessa maneira, a ordem dos atributos é importante durante o processo de busca, uma vez que a comparação das chaves é feita atributo por atributo. Considere, por exemplo, um banco de dados com uma relação LeiturasAmbientais contendo informações de leituras de sensores de temperatura e umidade em uma plantação, onde as tuplas dessa relação seguem o formato:

$$[id_{sensor}, data, temperatura, umidade, latitude, longitude]$$

Uma amostra dessa tabela pode ser vista na tabela 3.2.4.

Id	Data	Temperatura	Umidade	Latitude	Longitude
1	2023-01-15	25°C	60%	-25.4284	-49.2733
2	2023-01-15	22°C	75%	-23.5505	-46.6333
3	2023-01-15	28°C	50%	-24.9515	-53.4559
4	2023-01-15	30°C	55%	-23.5505	-46.6333
5	2023-01-15	26°C	68%	-25.4195	-49.2646

Tabela 3 – Exemplo de 5 tuplas da relação LeiturasAmbientais

Considere a consulta: *"Retorne as localizações de k sensores de temperatura e umidade realizadas no dia 15/01/2023, com temperatura entre 25° C e 30° C e umidade entre 50% e 60%"*.

Suponha que a chave composta utilizada na busca seja $[temperatura, umidade]$. O algoritmo começa primeiro verificando a temperatura, temos:

$$[25, 60], [28, 50], [30, 55], [26, 68]$$

Em seguida, o conjunto restante é filtrado pela umidade:

$$[25, 60], [28, 50]$$

Após a busca no primeiro nível finalizar, o algoritmo acessa o segundo nível, onde a verificação das condições complexas é realizada nas *Slim-trees* referenciadas pelos nós folha no primeiro nível.

A pré-validação das condições envolvendo atributos simples evitando completamente a comparação dos atributos complexos antes da avaliação da condição com os atributos tradicionais é uma das principais vantagens proporcionadas pela modificação. Além disso, a redução da quantidade de arquivos de índice necessários para armazenar os dados tradicionais implica na redução do espaço utilizado no disco.

3.2.5 Algoritmo *B-Slim*

O algoritmo B-Slim foi proposto também por [3] e surgiu a partir de um questionamento acerca do desempenho que a estrutura da *cx-Sim* Simple Tree* proporciona devido à arquitetura de duas camadas. A ideia então foi criar outra estrutura que consiste em uma *Slim-tree* e uma B^+*-tree* independentes, onde a condição adicional seria verificada na B^+*-tree* e a condição complexa na *Slim-tree*. O algoritmo basicamente consiste em algumas etapas:

1. Busca na B^+*-tree* pelos elementos que satisfazem a condição adicional com os atributos simples, retornando os *rowIds* das tuplas aptas.

2. Durante a busca na *Slim-tree*, o algoritmo verifica se o *rowId* do elemento em análise está no conjunto de *rowIds* que retornou da B^+-*tree*, sendo que esse conjunto é passado no argumento da busca por abrangência por pelos k vizinhos mais próximos.

O autor ressalta a importância de perceber que esse algoritmo difere de realizar buscas separadas nas duas estruturas e depois realizar a interseção dos conjuntos, como apresentado por [2], uma vez que caso a consulta fosse uma ${}_cKNNq$, o resultado estaria errado. Isso acontece porque a interseção dos conjuntos retornados pode ter menos de k elementos, ou até mesmo nenhum elemento. De maneira simples, para uma consulta ${}_cKNNq$, o algoritmo 2 seria modificado na linha 14, para incluir a verificacaoo se o rowId da tupla em análise está no conjunto de rowIds que retornou da B^+-*tree*, ou seja, a condição fica:

$$minMaxDistance \leq result.getMaxDistance() \wedge rowId \in rowIds$$

3.3 Estruturas para *Spatial keyword query*

Buscas por palavras-chave em dados geográficos utilizando índices, também conhecidas como *Spatial keyword query*, podem ser consideradas um caso especial de ${}_cKNNq$, onde a condição adicional é de igualdade em um atributo tradicional, enquanto o atributo complexo é a localização geográfica. Esse tipo de consulta é muito comum em aplicações de sistemas de navegação (GPS), onde o usuário deseja encontrar os k pontos de interesse mais próximos que incluem todas as palavras-chave da consulta, por exemplo, os k restaurantes mexicanos mais próximos. Algumas empresas que utilizam essa consulta são o Google Maps, Bing Maps, Foursquare, Flickr, Twitter [4, 41, 42].

Dado um conjunto de objetos espaço-textuais, uma posição de busca e um conjunto de palavras-chave, a consulta *TOPK-SK* recupera os k objetos mais próximos à posição de busca que incluem todas as palavras-chave. [13, 43].

Existem 2 principais grupos de índices para resolver esse tipo de consulta, assim como mostrado em [4]:

- *Keyword First Index*: Utilização do índice de palavras-chave para extração dos índices invertidos, para posteriormente análise do índice espacial para filtragem. Normalmente utilizado quando existe apenas uma palavra-chave na consulta. Exemplos de índices são a *Inverted R-tree* [44], *SFC-QUAD* [45] e o *S2I* [13]

- *Spatial First Index*: Utilização do índice espacial para extração dos índices invertidos, para posteriormente análise do índice de palavras-chave para filtragem. Recomendado quando existe mais de uma palavra-chave na consulta. Exemplos de

índices são a $\mathbb{R}^2$-*tree* [46], *KR*-tree* [47], $\mathbb{R}$-*tree* [48, 49], *WIBR-tree* [50], *SKI* [51] e a *IL-Quadtree* [4].

A seguir, será apresentado uma breve descrição sobre a *IL-Quadtree* proposta por [4]. A escolha dessa estrutura foi feita a partir de análises de resultados de buscas do tipo k vizinhos mais próximos, onde a *IL-Quadtree* consegue obter os melhores resultados na medida em que o valor k aumenta em comparação com as outras estruturas citadas acima como mostrado por [4].

3.3.1 *Inverted Linear Quadtree*

A *Inverted Linear Quadtree (IL-Quadtree)* proposta por [4] foi criada com o objetivo de aumentar a performance de consultas do tipo *Top k spatial keyword search (TOPK-SK).*

Os principais desafios encontrados em outras soluções para otimizar esses tipos de consulta como mostrado pelos autores, concentram-se em 2 categorias. A primeira é procurar maneira de reduzir o número de objetos presentes na região de busca, e a segunda, é como diminuir a probabilidade de sobrevivência dos objetos, isto é, a necessidade de carregar um objeto presente na região de busca. Para isso, os autores queriam construir uma estrutura de índice com algumas características:

1. O índice deve estar na categoria de índice invertido, onde os objetos são indexados por um índice espacial para cada palavras-chave, para que objetos que não possuam nenhuma palavra-chave serem eliminados imediatamente.
2. O índice deve ser adaptativo de acordo com a distribuição dos objetos para cada palavra-chave.
3. O índice deve ter a capacidade de podar grupos de objetos que não satisfaçam a condição de igualdade com a palavra-chave.

3.3.1.1 Estrutura geral da *IL-Quadtree*

Uma *Quadtree* é uma estrutura de dados que particiona o espaço em regiões de forma recursiva, onde cada nó representa uma região e cada filho representa um subespaço dessa região. Uma maneira eficiente de representar uma *Quadtree* é demonstrada por [52], onde a representação linear da estrutura capacita a utilização de uma estrutura auxiliar implementada em disco para seu armazenamento, como um B^+-*tree*. Os autores optaram por utilizar a codificação dos nós dessa estrutura seguindo a ordem Z (ou *Z-order*) [53], dado que essa codificação confere valores únicos para cada região criada. Dessa maneira, as regiões SW, SE, NW, NE são codificadas como 00, 01, 10, 11, respectivamente, assim como mostrado na figura 13.

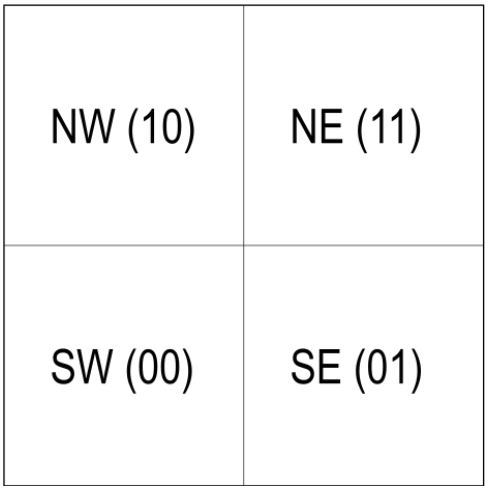

Figura 13 – Regiões e codificação usando Z-order, adaptado de [4]

Existem 2 tipos de nós na *IL-Quadtree*: nós internos e nós folha. Os nós internos incluem referências às sub-regiões que constituem a região do nó, e os nós folha contêm referências aos objetos presentes na região do nó. Cada um dos nós folha pode ser diferenciado pela presença ou não de objetos que satisfazem a condição de igualdade com a palavra-chave. Nós folha pretos indicam a presença de objetos que satisfazem a condição de igualdade com a palavra-chave, enquanto nós folha brancos indicam a ausência de objetos que satisfazem a condição de igualdade com a palavra-chave, assim como mostrado nas figuras 14 e 15.

Dessa maneira, apenas os nós folha pretos são armazenados na B^+-*tree* como mostrado na figura 16. Todo esse processo de construção da *IL-Quadtree* é realizado para cada palavra-chave presente na consulta. Para que um elemento esteja presente no resultado da consulta *TOPK-SK*, ele deve estar presente em todas as *IL-Quadtrees* de cada termo, isto é, o elemento deve estar presente em todos os nós folha pretos. O algoritmo 3 mostra o algoritmo da consulta *TOPK-SK* na *IL-Quadtree*. É importante dizer que o critério de ordenação do min-heap $\mathcal{R}$ é a distância entre o objeto e a posição de busca. A função $dist(o, q)$ retorna a distância entre o objeto o e a posição de busca q. Na linha 13, o contador o_{hit} conta quantas palavras-chave o objeto o contém, e l é a quantidade de palavras-chave da consulta.

Algoritmo 3: *TOPK-SK* query na *IL-Quadtree*

```
1  início
       Entrada: LQ: IL-Quadtree, q: consulta espaço-textual, k: quantidade de
                vizinhos
       Saída: R: resultado da consulta
2      R ← ∅;
3      H ← ∅;
4      λ ← ∞;
5      para cada LQ_i onde i ∈ q.keywords faça
6          H.add(LQ_i.root);
7      enquanto H ≠ ∅ faça
8          e ← H.pop();
9          se e é um nó folha preto então
10             se e está presente em todas as IL-Quadtrees então
11                 Carregue todos os objetos contidos em e;
12                 para cada objeto o contido em e faça
13                     o_hit ← o_hit + 1;
14                     se o_hit = l então
15                         Significa que o objeto contém todas as palavras-chaves
                            da consulta;
16                         se dist(o, q) ≤ λ então
17                             R.add(o);
18                             λ ← dist(o, q);
19         senão se e é um nó índice então
20             para cada nó filho e' de e faça
21                 se e não é um nó branco ∧ dist(e', q) ≤ λ então
22                     H.add(e');
23     retorna R;
24 fim
```

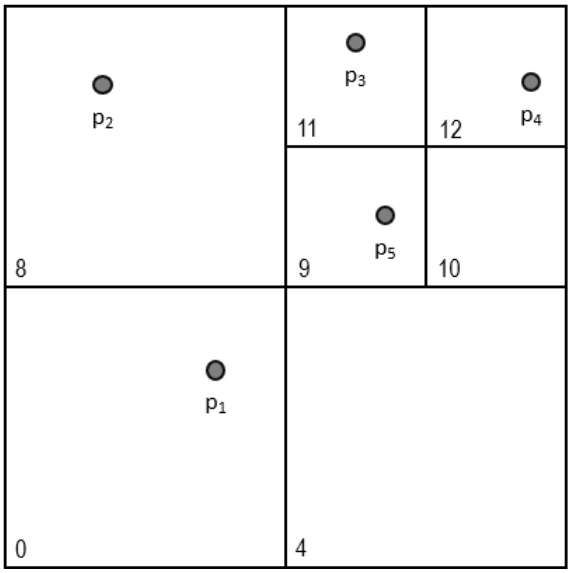

Figura 14 – Exemplo de dispersão dos objetos na *IL-Quadtree*, adaptado de [4]

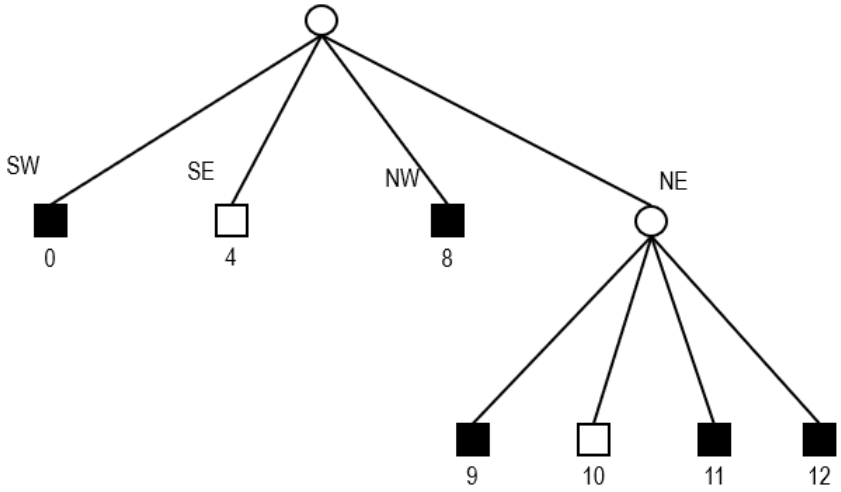

Figura 15 – *IL-Quadtree* referente à figura 14, adaptado de [4]

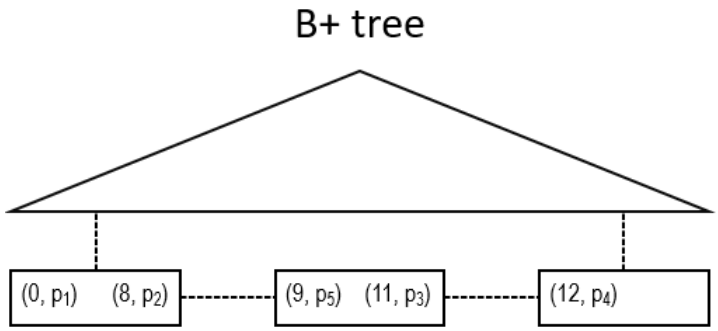

Figura 16 – Organização da B^+-*tree* referente à figura 14, adaptado de [4]

4 EXPERIMENTOS E RESULTADOS

4.1 Proposta de contribuição

Assim como comentado nas seções 1.1 e 1.2, os testes de performance das abordagens descritas na seção 3 foram realizados utilizando diferentes bases de dados com configurações de hardware diversas. Isso dificulta a comparação entre elas, uma vez que não existe um ambiente de testes padronizado. Dessa maneira, a proposta de contribuição deste trabalho de conclusão de curso será realizar uma análise comparativa do desempenho das estratégias supracitadas utilizando um ambiente controlado, com as mesmas bases de dados, e com as mesmas configurações de processador, memória e disco. Ademais, a ideia será criar rankings de performance, bem como identificar os melhores cenários para cada uma delas. Para isso, serão considerados métricas como tempo de busca, quantidade de acessos aos índices e arquivos de dados. O objetivo é fornecer insights valiosos para orientar a escolha da solução mais adequada para cada cenário.

5 CONCLUSÃO

REFERÊNCIAS

[1] ZEBARI, R. et al. A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. *Journal of Applied Science and Technology Trends*, v. 1, n. 2, p. 56–70, 2020.

[2] KASTER, D. d. S. *Tratamento de condições especiais para busca por similaridade em bancos de dados complexos*. Tese (Doutorado) — Universidade de São Paulo, 2012.

[3] SOARES, L. C.; KASTER, D. S. cx-sim: A metric access method for similarity queries with additional conditions. *Journal of Information and Data Management*, v. 4, n. 3, p. 437–437, 2013.

[4] ZHANG, C. et al. Inverted linear quadtree: Efficient top k spatial keyword search. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 28, n. 7, p. 1706–1721, 2016.

[5] NASCIMENTO, L. F. A sociologia digital: um desafio para o século xxi. *Sociologias*, SciELO Brasil, v. 18, p. 216–241, 2016.

[6] ANDRADE, F. G. d. et al. Sesdi: um arcabouço para a recuperação de dados geográficos em infraestruturas de dados espaciais. Universidade Federal de Campina Grande, 2012.

[7] MOTA, J. S. Estendendo quadtress para suporte ao armazenamento e recuperação de dados espaço-temporais. 2000.

[8] CHEN, T. et al. iblob: Complex object management in databases through intelligent binary large objects. In: SPRINGER. *Objects and Databases: Third International Conference, ICOODB 2010, Frankfurt/Main, Germany, September 28-30, 2010. Proceedings 3*. [S.l.], 2010. p. 85–99.

[9] CHÁVEZ, E. et al. Searching in metric spaces. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 33, n. 3, p. 273–321, 2001.

[10] RABEE, A.; BARHUMI, I. Ecg signal classification using support vector machine based on wavelet multiresolution analysis. In: IEEE. *2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA)*. [S.l.], 2012. p. 1319–1323.

[11] OLIVEIRA, W. D. d. *Operação de busca exata aos K-vizinhos mais próximos reversos em espaços métricos*. Tese (Doutorado) — Universidade de São Paulo, 2010.

[12] JR, C. T. et al. Slim-trees: High performance metric trees minimizing overlap between nodes. In: SPRINGER. *International Conference on Extending Database Technology*. [S.l.], 2000. p. 51–65.

[13] ROCHA-JUNIOR, J. B. et al. Efficient processing of top-k spatial keyword queries. In: SPRINGER. *Advances in Spatial and Temporal Databases: 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings 12*. [S.l.], 2011. p. 205–222.

[14] MARTINS, A. d. S. et al. Computação baseada em casos: contribuições metodológicas aos modelos de indexação, avaliação, ranking, e similaridade de casos. Universidade Federal de Campina Grande, 2000.

[15] ZHANG, D.; LU, G. Evaluation of similarity measurement for image retrieval. In: IEEE. *International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003*. [S.l.], 2003. v. 2, p. 928–931.

[16] BARIONI, M. C. N. Operações de consulta por similaridade em grandes bases de dados complexos. *São Carlos, SP: Universidade de São Paulo*, 2006.

[17] AGHAV-PALWE, S.; MISHRA, D. Feature vector creation using hierarchical data structure for spatial domain image retrieval. *Procedia Computer Science*, Elsevier, v. 167, p. 2458–2464, 2020.

[18] ZEZULA, P. et al. *Similarity search: the metric space approach*. [S.l.]: Springer Science & Business Media, 2006. v. 32.

[19] PATEL, B.; MESHRAM, B. Content based video retrieval systems. *arXiv preprint arXiv:1205.1641*, 2012.

[20] LI, M. et al. Fast hybrid dimensionality reduction method for classification based on feature selection and grouped feature extraction. *Expert Systems with Applications*, Elsevier, v. 150, p. 113277, 2020.

[21] WOODS, R. E.; GONZALEZ, R. C. *Digital image processing*. [S.l.]: Pearson Education Ltd., 2008.

[22] LOGAN, B. et al. Mel frequency cepstral coefficients for music modeling. In: PLYMOUTH, MA. *Ismir*. [S.l.], 2000. v. 270, n. 1, p. 11.

[23] RIZAL, A.; PRIHARTI, W.; HADIYOSO, S. Seizure detection in epileptic eeg using short-time fourier transform and support vector machine. *International Journal of Online & Biomedical Engineering*, v. 17, n. 14, 2021.

[24] AYESHA, S.; HANIF, M. K.; TALIB, R. Overview and comparative study of dimensionality reduction techniques for high dimensional data. *Information Fusion*, Elsevier, v. 59, p. 44–58, 2020.

[25] VELLIANGIRI, S.; ALAGUMUTHUKRISHNAN, S. et al. A review of dimensionality reduction techniques for efficient computation. *Procedia Computer Science*, Elsevier, v. 165, p. 104–111, 2019.

[26] PADMAJA, D. L.; VISHNUVARDHAN, B. Comparative study of feature subset selection methods for dimensionality reduction on scientific data. In: IEEE. *2016 IEEE 6th International Conference on Advanced Computing (IACC)*. [S.l.], 2016. p. 31–34.

[27] HUANG, X.; WU, L.; YE, Y. A review on dimensionality reduction techniques. *International Journal of Pattern Recognition and Artificial Intelligence*, World Scientific, v. 33, n. 10, p. 1950017, 2019.

[28] WILSON, D. R.; MARTINEZ, T. R. Improved heterogeneous distance functions. *Journal of artificial intelligence research*, v. 6, p. 1–34, 1997.

[29] MORAES, J. C. B. d. *Busca por similaridade utilizando grafo de interações NK*. Tese (Doutorado) — Universidade de São Paulo, 2020.

[30] LI, X. et al. Multi-dimensional range queries in sensor networks. In: *Proceedings of the 1st international conference on Embedded networked sensor systems*. [S.l.: s.n.], 2003. p. 63–75.

[31] BUSTOS, B. et al. Feature-based similarity search in 3d object databases. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 345–387, 2005.

[32] XU, C.; ZHANG, C.; XU, J. vchain: Enabling verifiable boolean range queries over blockchain databases. In: *Proceedings of the 2019 international conference on management of data*. [S.l.: s.n.], 2019. p. 141–158.

[33] LI, J.; OMIECINSKI, E. R. Efficiency and security trade-off in supporting range queries on encrypted databases. In: SPRINGER. *IFIP Annual Conference on Data and Applications Security and Privacy*. [S.l.], 2005. p. 69–83.

[34] RAZENTE, H. L. et al. Aggregate similarity queries in relevance feedback methods for content-based image retrieval. In: *Proceedings of the 2008 ACM symposium on Applied computing*. [S.l.: s.n.], 2008. p. 869–874.

[35] VIEIRA, M. R. et al. Dbm-tree: A dynamic metric access method sensitive to local density data. In: *SBBD*. [S.l.: s.n.], 2004. p. 163–177.

[36] CIACCIA, P. et al. M-tree: An efficient access method for similarity search in metric spaces. In: CITESEER. *Vldb*. [S.l.], 1997. v. 97, p. 426–435.

[37] INDYK, P.; MOTWANI, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1998. p. 604–613.

[38] PAREDES, R.; CHÁVEZ, E. Using the k-nearest neighbor graph for proximity searching in metric spaces. In: SPRINGER. *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings 12*. [S.l.], 2005. p. 127–138.

[39] OCSA, A.; BEDREGAL, C.; CUADROS-VARGAS, E. A new approach for similarity queries using neighborhood graphs. In: *SBBD*. [S.l.: s.n.], 2007. p. 131–142.

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

[41] CHEN, L. et al. Spatial keyword query processing: An experimental evaluation. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 6, n. 3, p. 217–228, 2013.

[42] CAO, X. et al. Spatial keyword querying. In: SPRINGER. *Conceptual Modeling: 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings 31*. [S.l.], 2012. p. 16–29.

[43] ZHANG, D.; TAN, K.-L.; TUNG, A. K. Scalable top-k spatial keyword search. In: *Proceedings of the 16th international conference on extending database technology*. [S.l.: s.n.], 2013. p. 359–370.

[44] ZHOU, Y. et al. Hybrid index structures for location-based web search. In: *Proceedings of the 14th ACM international conference on Information and knowledge management*. [S.l.: s.n.], 2005. p. 155–162.

[45] CHRISTOFORAKI, M. et al. Text vs. space: efficient geo-search query processing. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. [S.l.: s.n.], 2011. p. 423–432.

[46] FELIPE, I. D.; HRISTIDIS, V.; RISHE, N. Keyword search on spatial databases. In: IEEE. *2008 IEEE 24th International conference on data engineering*. [S.l.], 2008. p. 656–665.

[47] HARIHARAN, R. et al. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In: IEEE. *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*. [S.l.], 2007. p. 16–16.

[48] CONG, G.; JENSEN, C. S.; WU, D. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 2, n. 1, p. 337–348, 2009.

[49] WU, D.; CONG, G.; JENSEN, C. S. A framework for efficient spatial web object retrieval. *The VLDB Journal*, Springer, v. 21, p. 797–822, 2012.

[50] WU, D. et al. Joint top-k spatial keyword query processing. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 24, n. 10, p. 1889–1903, 2011.

[51] CARY, A.; WOLFSON, O.; RISHE, N. Efficient and scalable method for processing top-k spatial boolean queries. In: SPRINGER. *International Conference on Scientific and Statistical Database Management*. [S.l.], 2010. p. 87–95.

[52] GARGANTINI, I. An effective way to represent quadtrees. *Communications of the ACM*, ACM New York, NY, USA, v. 25, n. 12, p. 905–910, 1982.

[53] MORTON, G. M. A computer oriented geodetic data base and a new technique in file sequencing. International Business Machines Company New York, 1966.