Universidade Estadual de Londrina

RODRIGO MIMURA SHIMOMURA

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

LONDRINA

2024

RODRIGO MIMURA SHIMOMURA

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

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

2024

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.

RODRIGO MIMURA SHIMOMURA

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

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.

BANCA EXAMINADORA

Orientador: Prof. Dr. Daniel dos Santos Kaster
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca – Sigla instituição

---

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca – Sigla instituição

---

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto Membro da Banca – Sigla instituição

Londrina, A definir de 2024.

Este trabalho é dedicado à minha família e amigos, que sempre me apoiaram e incentivaram a seguir em frente.

AGRADECIMENTOS

Agradeço a Deus pela minha vida e oportunidades que me foram dadas.

Expresso minha profunda gratidão à minha familia, especialmente aos meus pais, Sidnei e Erika, e ao meu irmão Alexandre, por terem sempre me oferecido apoio, incentivo e estrutura ao longo da minha vida, especialmente nos momentos mais desafiadores.

Agradeço ao meu orientador, Prof. Dr. Daniel dos Santos Kaster, por toda a paciência, orientação e ensinamentos durante horas e horas de reuniões e discussões ao longo do desenvolvimento deste trabalho.

Ao corpo docente do Departamento de Computação da UEL, em especial àqueles que mostraram dedicação e comprometimento durante as aulas ministradas no curso de Ciência da Computação.

Aos meus amigos e colegas de curso, em especial, Enzo, Felipe, Mateus, Matheus e Rhuan por inúmeros dias de estudo e trabalho em grupo.

"A lesson without pain is meaningless. That's because no one can gain without sacrificing something. But by enduring that pain and overcoming it, he shall obtain a powerful, unmatched heart. A fullmetal heart..."

(Hiromu Arakawa)

SHIMOMURA, R. M.. **Análise e comparação de estruturas de indexação para consultas por similaridade com condições adicionais**. 2024. 89f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

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. As análises e experimentos realizados em uma plataforma controlada e automatizada mostram que existem diversos fatores a serem levados em consideração na escolha da estrutura de indexação, como o tamanho do conjunto de dados, seletividade da consulta, tipo de condição adicional e se são dados bidimensionais ou não.

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

SHIMOMURA, R. M.. **Comparison and analysis of indexing structures for similarity queries with additional conditions**. 2024. 89p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

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. The analyzes and experiments carried out in a controlled and automated platform show that there are several factors to be considered in the choice of the indexing structure, such as the size of the dataset, query selectivity, type of additional condition and whether the data is two-dimensional or not.

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

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*
GBDI	Grupo de Banco de Dados e Imagens
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*
XXL	*eXtensible and fleXible Library*

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 [4]. 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 [5, 6].

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*) [7], 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.

Assim sendo, 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 [8]. 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) [9]. 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.

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 Neighbors query - k-NNq*), consulta dos k vizinhos mais próximos reverso (*RkNN*) [10]. 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]. Algumas estruturas de indexação que possuem dados

complexos em seus elementos são a *Slim-tree* [11], *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)* [12] 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.

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 com condições adicionais.

A contribuição deste trabalho reside na análise comparativa entre as estruturas de indexação de dados complexos, realizada por meio de experimentos em uma plataforma controlada e automatizada. Além disso, destaca-se a identificação das situações mais adequadas para a aplicação de cada solução apresentada.

Os experimentos realizados neste trabalho mostram que existem diversos fatores a serem levados em consideração na escolha da estrutura de indexação, como o tamanho do conjunto de dados, seletividade da consulta, tipo de condição adicional e se são dados bidimensionais ou não.

O restante deste trabalho está organizado da seguinte forma.

- O capítulo 2 apresenta as definições dos conceitos relacionados às buscas por similaridade, além dos tipos de busca utilizados atualmente. Outro ponto de grande importância nesse capítulo é a apresentação da introdução das condições adicionais nas buscas por similaridade. Para isso, são apresentadas as peculiaridades de cada tipo de condição, juntamente com exemplos de casos de uso.
- 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 5.
- O capítulo 4 apresenta o desenvolvimento do ambiente criado para a realização dos testes, mostrando os conjuntos de dados utilizados e as adaptações realizadas.
- O capítulo 5 apresenta os resultados obtidos nos testes realizados, bem como a análise comparativa entre as estruturas de indexação.
- Por fim, o capítulo 6 traz as conclusões obtidas neste trabalho de conclusão de curso.

2 BUSCAS POR SIMILARIDADE

Este capítulo aborda os conceitos fundamentais para o desenvolvimento e entendimento deste trabalho. Começamos explicando o significado de similaridade para o ser humano, com a abstração necessária para transferência ao ambiente computacional. Para realizar isso, uma análise detalhada da representação dos dados complexos, com a criação dos vetores de características, e suas técnicas de preparo é apresentada ao leitor. Outros conceitos importantes como a definição do espaço métrico e funções de distâncias são brevemente discutidos. Por fim, realizamos a apresentação dos 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 [13]. 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, [14]. A figura 1 demonstra uma situação em que 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 [15]. 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 [8, 2].

A principal maneira de representar esses dados complexos para realizar comparações por similaridade consiste em duas 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 dois 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) [16]. 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 [17].

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 [18]. É de suma importância destacar que por combinar várias características simultaneamente, o problema da maldição da dimensionalidade ocorre conforme comentado em [18, 19].

Os autores em [18] 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 [20],*Mel-Frequency Cepstral Coefficients* (MFCC) e *Short Time Fourier Transform* (STFT) para faixas de áudio [21, 22].

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* são encontradas em várias áreas do conhecimento como medicina, educação, negócios e redes sociais em diversos formatos, como textos, imagens e vídeos [23]. Como comentado por [19], 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 [24] é o aumento da qualidade dos dados, melhoria 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 [25]. 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

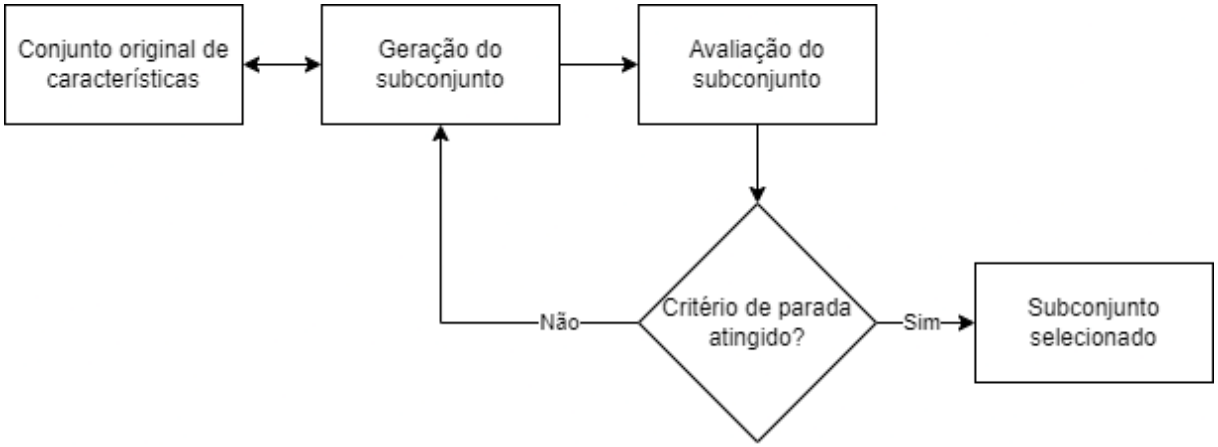


Figura 2 – Processo de seleção de características. Adaptado de [1]

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 [25]:

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.

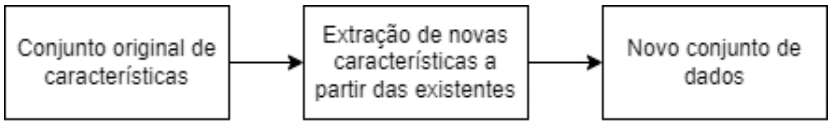


Figura 3 – Processo de extração de características. Adaptado de [1]

Os métodos mais comuns de realizar esse método de acordo com os autores em [2, 1, 24, 26] 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 [17]. 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 [27]. 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 [28].

2.2 Tipos de busca por similaridade

Os principais tipos de busca por similaridade podem ser divididos em alguns grupos como exposto por [17]:

- *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 [29] 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 [30], 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 5.

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.

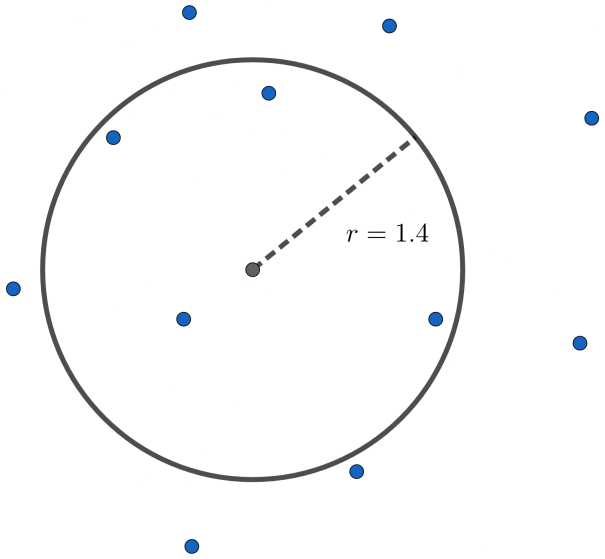


Figura 4 – Exemplo da Rq com $r = 1.4$

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.

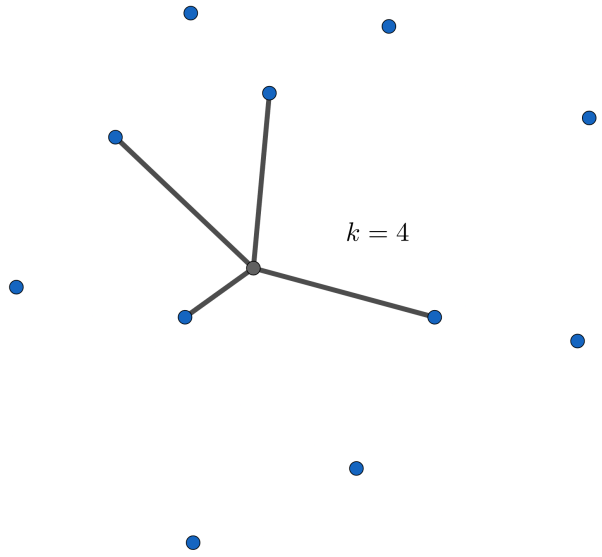


Figura 5 – Exemplo de $kNNq$ com $k = 4$

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 [31] 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 [32], 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: ${}_ck$-*NNq* (*condition-extended k-NN query*). A definição desse operador é descrita pela notação 2.12:

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

Os termos presentes na notação são: $\ddot{\sigma}$ caracterizando uma *k-NNq*, uma coluna da relação de entrada S_j, o operador ${}_ckNN$, 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 em que 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 [33], 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 expressa 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 um conjunto 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.

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

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 [11]. 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 [11], 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 8 abordagens distintas para solucionar problemas encontrados durante o processo de tratar e armazenar os dados complexos nos SGBDs. As 7 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* [11], a *DBM-tree* [34] e a *M-tree* [35], estruturas baseadas em técnicas de *hashing* como o LSH (*Locality Sensitive Hashing*) [36] e estruturas baseadas em grafos de proximidade [37, 38]. 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 [11] 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 em que 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

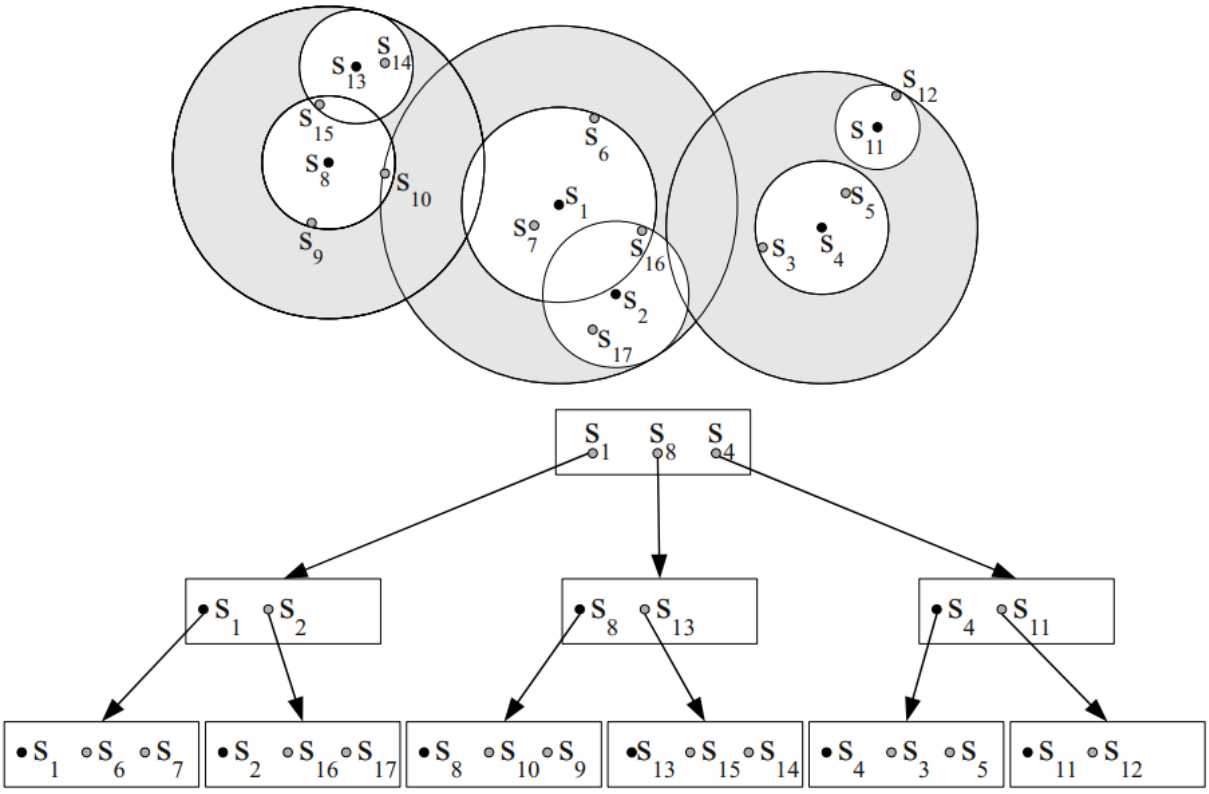


Figura 6 – Representação gráfica da *Slim-tree*, extraído de [2]

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 criadas 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 [11] 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 menos *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.

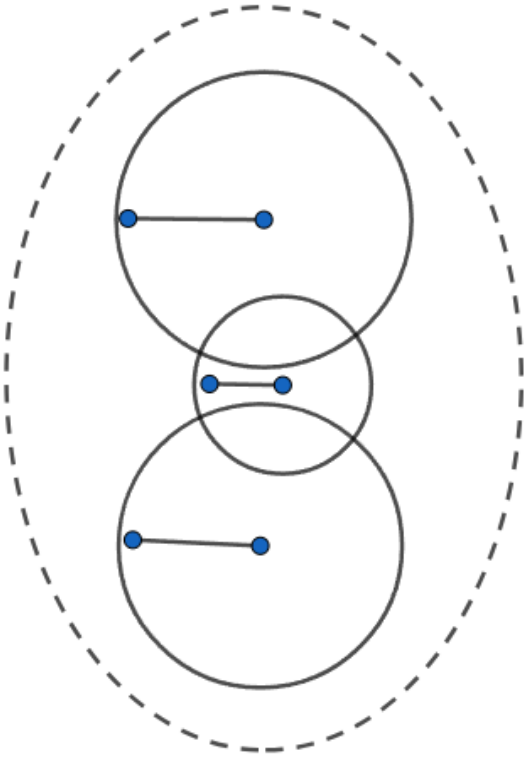


Figura 7 – *Slim-tree* sem *overlap*

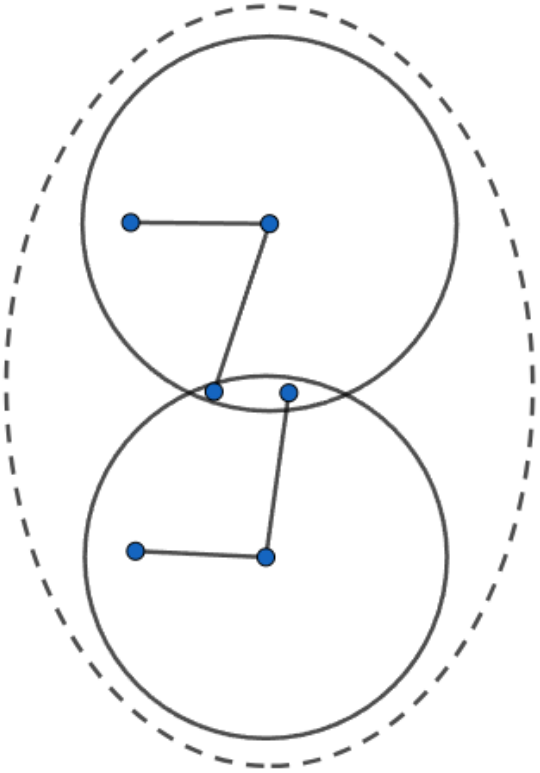


Figura 8 – *Slim-tree* com *overlap*

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, 35]. 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, 39]. 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.

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* [11]. 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 processadas somente após filtragem dos dados em potencial, ademais, esse processo é executado somente se a condição de busca for satisfeita. 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.

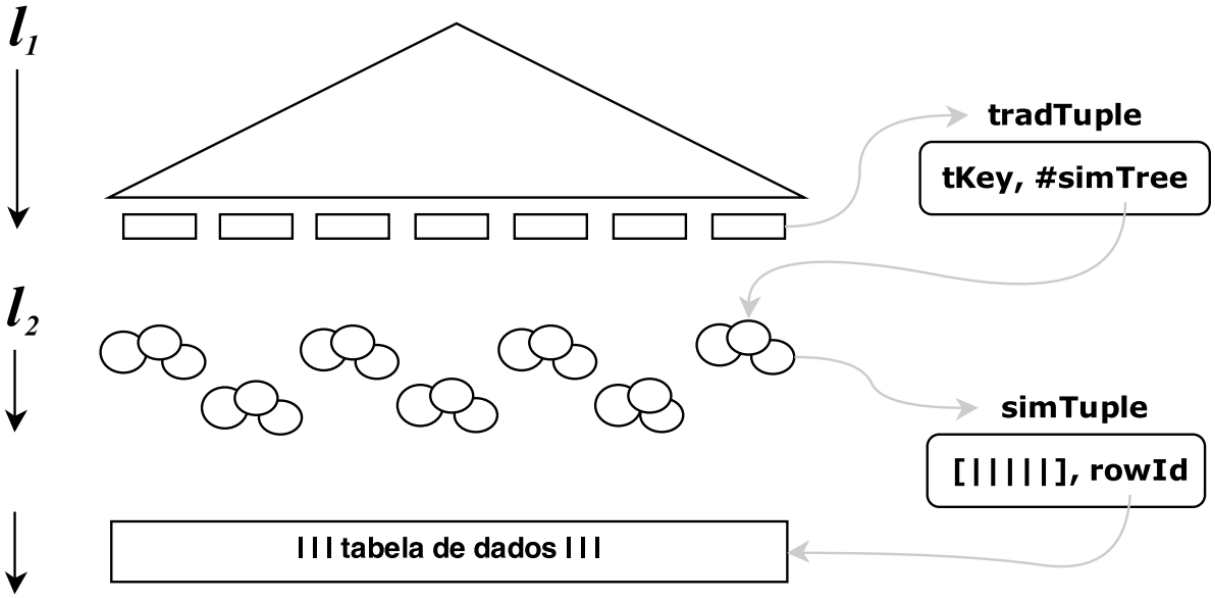


Figura 9 – Representação gráfica da *cx-Sim* Simple Tree*, extraído de [3]

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).

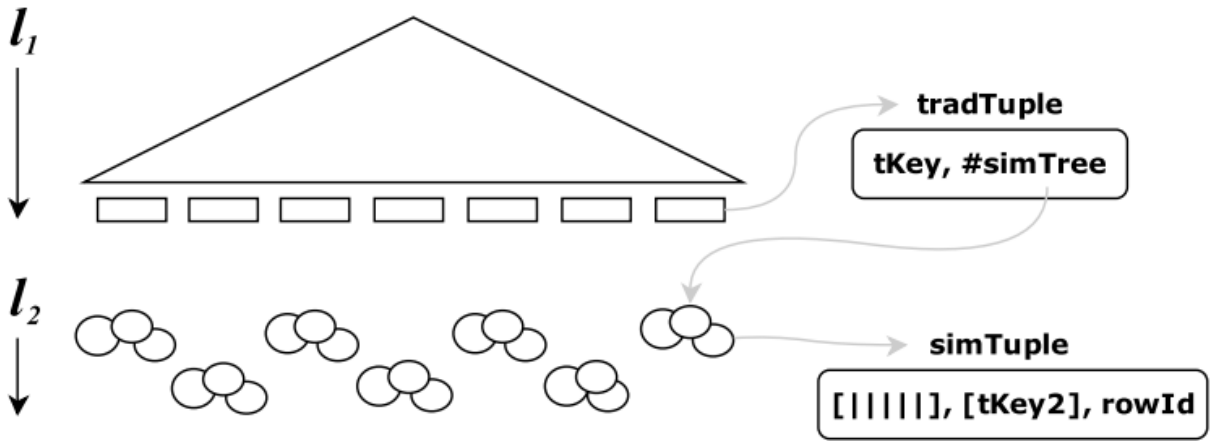


Figura 10 – Representação gráfica da *cx-Sim* Covering Tree*, extraído de [3]

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.

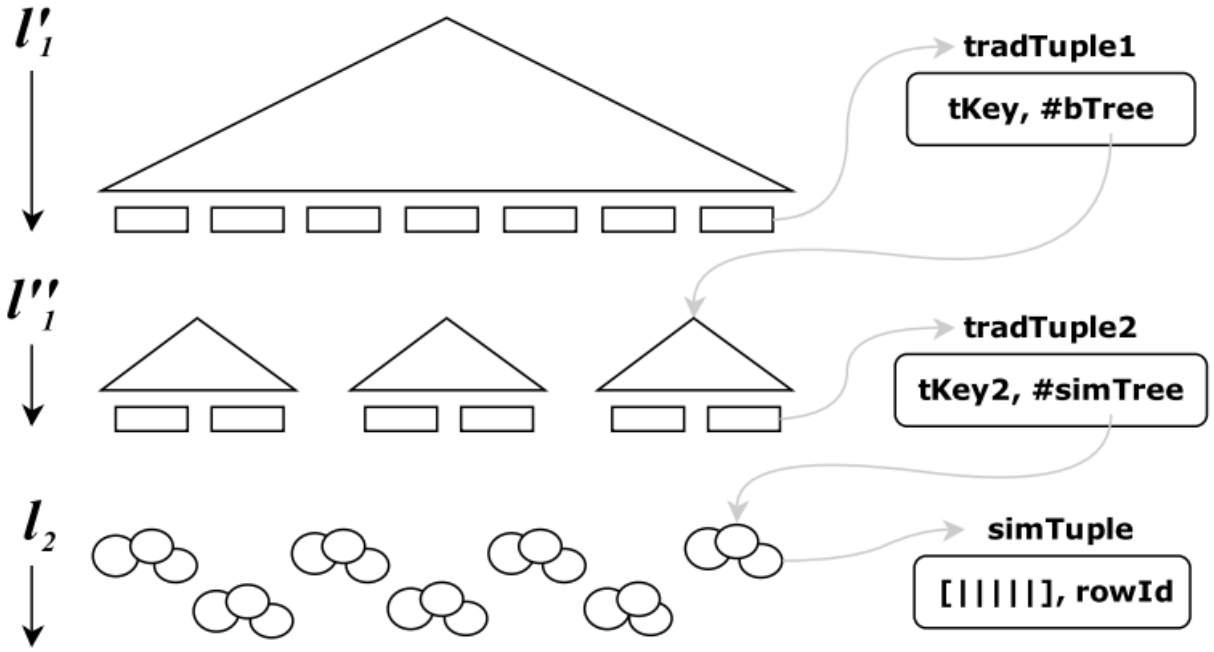


Figura 11 – Representação gráfica da *cx-Sim* Chained Tree*, extraído de [3]

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

A busca nessa 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.

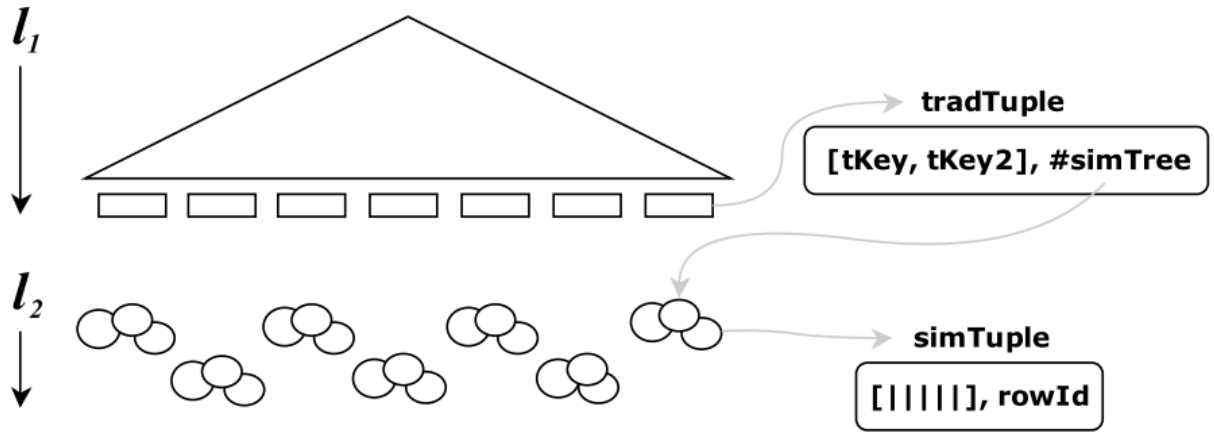


Figura 12 – Representação gráfica da *cx-Sim* Composite Tree*, extraído de [3]

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 *BSlim*

O algoritmo BSlim 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 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 verificação 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 [40, 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 ao menos uma palavra-chave da consulta [12, 43].

Existem 2 principais grupos de índices para resolver esse tipo de consulta, assim como mostrado em [40]:

- *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* [12]

- *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 $IR^2 - tree$ [46], *KR*-tree* [47], *IR-tree* [48, 49, 50], *WIBR-tree* [51], *SKI* [52] e a *IL-Quadtree* [40].

A seguir, será apresentado uma breve descrição sobre o *S2I* proposta por [12]. A escolha dessa estrutura foi feita a partir de análises de resultados de buscas do tipo k vizinhos mais próximos, considerando que o *S2I* consegue obter bons resultados com buscas onde o número de *keywords* é baixo, assim como mostrado por [12] e [40].

3.3.1 *S2I*

O *S2I (Spatial Inverted Index)* proposto por [12] foi criado visando otimizar a busca de palavras-chaves em um conjunto de dados, utilizando um índice invertido. Essa estrutura está no cerne de motores de busca em larga escala [53], e consiste em uma técnica de armazenamento de textos na qual para cada termo presente, existe uma lista de documentos onde é possível encontrá-lo [54]. Outra informação importante armazenada nessa estrutura, é a frequência da presença dos termos nos textos, o que possibilita acelerar as buscas, ao passo que é feito um ranqueamento das palavras mais comuns [54].

O principal motivo da criação do *S2I* foi devido à maneira como a *IR-Tree* [49, 50] (e suas variações) [48] e a IR^2*-tree* [46] eram construídas. Essas estruturas utilizam uma *R-tree* com um índice invertido em cada nó, fato que implica na necessidade de acesso de cada índice para verificar a similaridade entre as palavras-chave da busca com o conteúdo do nó da árvore, o que pode ocasionar um aumento do custo computacional e tempo de execução. Isso é obtido através da utilização de diferentes tipos de armazenamento, guiados pela relevância dos termos, como mostrado pelo autor.

Considerando um conjunto de objetos espaço-textuais |P|, onde cada objeto $p \in P$ está na forma $p = \langle p.id, p.l, p.d \rangle$, onde $p.id$ é um identificador único, $p.l$ é a localização geográfica (latitude e longitude) e $p.d$ é a descrição textual. Além disso, considere q uma consulta *TOPK-SK*, onde $q = \langle q.l, q.d, q.k \rangle$, onde $q.l$ é a localização geográfica da consulta, $q.d$ é a descrição textual da consulta (*keywords*) e $q.k$ é o número esperado de objetos a serem retornados. A execução de uma consulta nessa estrutura leva em consideração a pontuação de cada objeto $\tau(p, q)$ que é calculada da seguinte maneira:

$$\tau(p, q) = \alpha \cdot \delta(p.l, q.l) + (1 - \alpha) \cdot \theta(p.d, q.d)$$

Onde α é um parâmetro de ajuste que controla a importância da localização geográfica em relação à descrição textual, $\delta(p.l, q.l)$ é a distância entre a localização geográfica do objeto p e a localização geográfica da consulta q, e $\theta(p.d, q.d)$ é a relevância textual entre a descrição do objeto p e a consulta q. O autor utiliza a distância euclidiana para calcular a distância entre as localizações geográficas, enquanto a similaridade textual foi calculada

utilizando a similaridade por cosseno utilizando a abordagem proposta por Zobel e Moffat [55].

O *S2I* mapeia cada uma das palavras-chave em 2 estruturas dependendo do número de ocorrências. Termos mais frequentes são armazenados em *aR-Trees* (*Aggregated R-tree*)[56], enquanto os menos frequentes são colocados em blocos [12].

Os componentes presentes no *S2I* são 3:

- Vocabulário: Armazena para cada termo o número de objetos/textos que possuem o termo, uma flag indicando qual o tipo de estrutura que está armazenando o termo (bloco ou *aR-Tree*) e um ponteiro para a localização da estrutura
- Blocos: Cada bloco é um conjunto de objetos, onde cada objeto possui um identificador único, a localização (geográfica, por exemplo) e o impacto do termo na descrição do objeto.
- Árvores: *aR-trees* de cada termo se assemelham com as *R-trees*, onde cada nó intermediário guarda o MBR (*Minimum Bounding Rectangle*) de cada um dos nós filhos. O que diferencia as *aR-trees* das *R-trees* reside no fato que os nós da *aR-tree* armazenam também valores agregados não espaciais (não complexos).

A tabela 4 demonstra a organização feita no *S2I*.

Termo	ID	Freq	Tipo armazenamento	Ponteiro	Armazenamento
Automóvel	id_1	5	Árvore	0x3A1F5678	aR^{id_1}
Volante	id_2	3	Bloco	0x8EBC2A95	$< p_1, p_3, p_5 >$
Rodas	id_3	7	Árvore	0x17D49FBE	aR^{id_3}
Capacete	id_4	2	Bloco	0xA0EF834D	$< p_2, p_4 >$

Tabela 4 – Representação visual do *S2I*

O autor destaca diversas vantagens relacionadas ao uso do *S2I*. Entre elas, a frequência dos termos é utilizada para decidir onde armazenar os termos, o que ocasiona em uma redução do número de acessos necessários para a busca. Outro ponto importante é a capacidade do *S2I* de suportar a distribuição de grandes conjuntos de dados, fato que possibilita a utilização de técnicas de motores de busca como o particionamento de termos [55]. Por fim, para consultas com apenas uma palavra-chave, o *S2I* acessa apenas uma pequena *aR-tree* ou um bloco, dependendo da frequência do termo no documento, o que ocasiona em um menor tempo de execução.

Consultas do tipo *TOPK-SK* com apenas uma palavra-chave t são realizadas de maneira eficiente em virtude de acessarem somente um único bloco ou árvore. Se os objetos estão armazenados em um bloco, a busca segue 2 etapas simples: primeiro, recupere todos os objetos p presentes no bloco, e em seguida, insira todos os objetos p em um max-heap considerando a ordem decrescente de $\tau(p, q)$. Para objetos que possuem uma frequência maior, por conseguinte estão armazenados em uma *aR-tree*, a busca segue o algoritmo *Single Keyword Algorithm* (3). É importante ressaltar que o *S2I* também suporta consultas com múltiplas palavras-chave, porém é necessário que pontuações parciais sejam calculadas, assim como mostrado pelo autor.

Algoritmo 3: SKA(*MaxHeap* H^t, *Query* q)

```
1  início
       Entrada: MaxHeap H^t com entradas em ordem decrescente do valor
                τ(p, q), Query q
       Saída: O próximo objeto p com maior valor de τ(p, q)
2      Entrada e ← H^t.pop()
3      enquanto e não é um objeto faça
4          se e é um nó-intermediário então
5              para cada nó-filho e' de e faça
6                  H^t.insert(e', τ(e', q));
7          senão
8              para cada objeto espaco-textual p contido em e faça
9                  H^t.insert(p, τ(p, q));
10         e ← H^t.pop()
11     retorna e;
```

4 DESENVOLVIMENTO DO AMBIENTE EXPERIMENTAL

O desenvolvimento do ambiente experimental foi divido em algumas etapas primordiais para a realização dos experimentos e análises dos resultados obtidos. A primeira parte consistiu na análise e entendimento do funcionamento do código-fonte das estruturas apresentadas, bem como conduzir adaptações permitindo a execução dos experimentos. A segunda parte foi a escolha e ajuste de conjuntos de dados compatíveis com as categorias de busca que as estruturas conseguem processar. A terceira parte envolveu a criação de um ambiente de execução que viabilizasse a condução eficiente e organizada dos experimentos, bem como a elaboração de relatórios de desempenho. Este capítulo apresenta as etapas realizadas para a criação do ambiente experimental, bem como as adaptações feitas nos códigos-fonte das estruturas e a escolha dos conjuntos de dados utilizados.

4.1 Análise e adaptação do código-fonte

A estrutura *Slim-tree* (nas abordagens *Table-Slim* e *Covering-Slim*) de propriedade do GBDI (Grupo de Banco de Dados e Imagens) do ICMC-USP (Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo) está disponível na biblioteca *Arboretum*[1] e foi implementada utilizando a linguagem de programação C++. As estruturas *cx-Sim** e o algoritmo *BSlim* foram criadas em C++ utilizando as implementações disponíveis nas bibliotecas do *GbdiLibs* e estão disponíveis no repositório *Gen-Knn*[2]. Por fim, a estrutura *S2I* foi implementada em Java utilizando a biblioteca *XXL* (*eXtensible and fleXible Library*)[3] feita especificamente para o processamento de consultas, permitindo acesso a componentes de baixo nível como os acessos a disco sendo disponibilizada diretamente pelo autor [12]. A utilização da *Slim-tree* foi feita através do *Gen-Knn*. As próximas seções apresentam as adaptações feitas em cada um dos repositórios para que fosse possível realizar os experimentos.

4.1.1 Adaptações realizadas no repositório *Gen-Knn*

Sobre as adaptações feitas nos códigos do repositório *Gen-Knn*, foi necessário alterar alguns métodos para que fossem aceitos tanto valores inteiros, quanto valores de ponto flutuante, em virtude de vários conjuntos de dados utilizarem ambos os tipos de dado. Uma das características do *Gen-Knn* é que originalmente ele era executado mais de uma vez passando um número de argumento durante sua invocação para que uma função

[1] http://gbdi.icmc.usp.br/en/projects/#/projects/2000-arboretum-caetano
[2] https://bitbucket.org/cross-uel/gen-knn/src/master/
[3] https://github.com/umr-dbs/xxl

por vez fosse realizada, assim sendo, foi conduzida a criação de funções executem todos os passos necessários para a realização dos testes com apenas uma invocação.

Além disso, para que essas estruturas sejam comparáveis com o *S2I*, foi necessário retirar o tempo para recuperar uma tupla completa utilizando seu *rowId* em algumas estruturas. Essa parte será explicada com mais detalhes na seção dos experimentos. Por fim, foi feito a escolha de valores nos atributos que conferem determinadas seletividades nos experimentos.

4.1.2 Adaptações realizadas na estrutura *S2I*

No código do S2I, foi necessário realizar mais alterações no código-fonte. Esse fato se decorreu em virtude do código original armazenar em arquivos objetos com o tipo *Double* do Java ao invés de *Float* como é utilizado no *Gen-Knn*. A biblioteca *XXL* original possuia algumas classes apenas do tipo *Double*, o que ocasionou a necessidade de criar classes para o tipo *Float* e adaptar o código para que ele aceitasse o uso do *Float*. Dessa maneira, a escrita e leitura de arquivos no *S2I* foram adaptadas para *Float*. Também fez-se necessário limitar a execução do programa a uma *thread* apenas, já que o S2I utilizava múltiplas *threads* em contraste ao *Gen-Knn* que utiliza apenas uma.

Originalmente, o S2I separa um texto em tokens e indexa cada um deles no seu índice, ou seja, ele consegue utilizar apenas uma coluna. Para que ele conseguisse aceitar consultas com condições compostas com múltiplas colunas foi necessário acresentar marcadores de coluna do tipo *a_* e *b_* para diferenciar as colunas. É importante dizer que o S2I considera apenas *keywords* para realizar suas buscas por utilizar um *HashMap* em seu índice, e não suporta intervalos de buscas, como é o caso das estruturas *cx-Sim**. Para tornar possível comparar o desempenho do S2I com as estruturas *cx-Sim**, foi necessário adaptar o código colocando todos os valores discretos do invervalo fornecido, o que aumentou significativamente a quantidade de *keywords* a serem buscadas. Ademais, a introdução dos marcadores de coluna possibilitou utilizar o $\alpha = 1$, dando total prioridade para a busca por localização espacial, já que apenas tuplas que contém as *keywords* da query são inseridas no *MaxHeap*. Além disso, no caso das consultas compostas, foi realizado manualmente a operação AND entre as tuplas retornadas de cada consulta, uma vez que o S2I não suporta consultas compostas.

Por fim, a sequência das melhores tuplas que o *S2I* retorna ao usuário utilizava uma *Lattice* organizando em ordem decrescente do valor de $\tau(p, q)$, porém, fez-se necessário remover a *Lattice* e ordenar as tuplas de acordo com o valor de $\tau(p, q)$ com o *sort* disponível na *Collections* do Java (Timsort ou Quicksort), devido à baixa performance quando muitas *keywords* são utilizadas. Para retornar as k melhores tuplas, por já estarem ordenadas, foi necessário apenas retornar as k primeiras tuplas realizando um corte da lista.

4.2 Escolha e ajuste dos conjuntos de dados

Os conjuntos de dados utilizados nos experimentos foram escolhidos respeitando as características das estruturas *cx-Sim** e o *S2I*, considerando a capacidade de processamento dos dados. O primeiro conjunto de dados escolhido foi o *USCities* que contém informações sobre cidades dos Estados Unidos do ano 2000, com atributos como nome, estado, população, tamanho médio de família, idade média da população, latitude e longitude, sendo esses dois últimos os atributos complexos. O segundo conjunto de dados é o *HCImages* que contém informações sobre imagens médicas no formato DICOM como idade, peso e data de nascimento dos pacientes, realizados no Hospital das Clínicas da Faculdade de Medicina de Ribeirão Preto da Universidade de São Paulo (HCFMRP-USP), sendo que um histograma de 256 níveis de cinza extraído da imagem os atributos complexos. Por fim, o último conjunto de dados é uma adaptação do *HCImages* chamado *HCImages Transformed* que remove algumas colunas não utilizadas nos testes, bem como a transformação das 256 colunas do histograma em 2 colunas através da técnica de *Principal Component Analysis* (PCA). Este último conjunto de dados foi necessário a fim de se utilizar o *S2I* nos experimentos, uma vez que o *S2I* opera apenas com dados bidimensionais.

	Conjuntos de dados		
Atributos	USCities	HCImages	HCImages Transformed
Descrição	Informações populacionais de cidades dos Estados Unidos no ano 2000	Informações de exames de pacientes do HCFMRP-USP	Informações de exames de pacientes do HCFMRP-USP, com PCA aplicado
Número de tuplas	25374	501100	501100
Número de colunas	101	269	5
Quantidade de atributos tradicionais	99	13	3
Quantidade de atributos complexos	2	256	2
Tipo de atributo complexo	Localização geográfica (Latitude, Longitude)	Histograma de 256 tons de cinza	PCA, redução de 256 dimensões para 2
Distância utilizada	Euclidiana (L_2)	Manhattan (L_1)	Euclidiana (L_2)

Tabela 5 – Informações sobre os conjuntos de dados utilizados nos experimentos.

A tabela 5 mostra informações mais detalhadas sobre os conjuntos de dados utilizados, e é importante ressaltar que a escolha desses conjuntos de dados foi feita com o intuito de se obter resultados que possam ser comparáveis entre as estruturas, uma vez que as estruturas *cx-Sim** e o *S2I* possuem características distintas. É importante ressaltar que por questões de privacidade, os dados dos conjuntos apresentados não permitem a identificação de particularidades das populações ou pacientes.

A utilização do *USCities* é em virtude de possuir grande facilidade de visualização

da distância de similaridade entre os elementos por considerar um espaço bidimensional. A adoção do *HCImages* deu-se pelo fato do tipo de atributo complexo ser diferente do *USCities*, sendo um histograma de 256 tons de cinza, o que possibilita a análise de desempenho das estruturas com mais atributos complexos utilizando-se outras distâncias. Por fim, a criação do *HCImages Transformed* através do PCA foi feita para que fosse possível a utilização do *S2I* nos experimentos com o *HCImages*, uma vez que o *S2I* pressupõe a existência de um espaço bidimensional para realizar operações de busca e utiliza a distância euclidiana no cálculo de similaridade entre os elementos.

4.3 Criação de uma plataforma de execução de testes

Para a execução dos testes, foi criada uma plataforma para a execução dos testes, manipulação dos dados e geração de relatórios e gráficos. As funcionalidades incluem configurações de testes, automatização e controle sobre as execuções independente da linguagem do código a ser executado, e está disponível em um repositório do Github[4]. A plataforma conta com um logger que armazena informações sobre a execução dos testes, como tempo de execução e código retornado pelos programas (utilizados para verificar se a execução foi bem-sucedida). Além disso, ela produz automaticamente os gráficos de desempenho para cada teste realizado. A fim de possibilitar análises mais profundas sobre cada conjunto de dados, também foram desenvolvidos scripts SQL para a criação de tabelas e inserção de dados dos conjuntos utilizados nos testes em um banco de dados PostgreSQL. As inserções são feitas de forma automática a partir de um script Python no mesmo diretório dos arquivos de dados.

Neste trabalho, foram mapeados todos os testes a serem realizados, bem como os parâmetros a serem utilizados em cada um deles. Foram consideradas métricas do tipo: tempo médio de consulta, quantidade de acessos a disco e número de cálculos de distância. A execução foi feita de maneira automatizada utilizando a plataforma de testes criada. O trabalho exigiu a codificação nas linguagens C++, Java e Python e um esforço considerável de análise e interpretação dos resultados tanto na etapa de adaptação das estruturas, quanto no estudo comparativo conduzido.

O próximo capítulo apresenta os experimentos realizados e os resultados obtidos, bem como a análise dos resultados e a comparação entre as estruturas *cx-Sim** e o *S2I*. O capítulo detalha todos os testes realizados, com os parâmetros e conjuntos de dados utilizados.

[4] https://github.com/rmshimomura/Internship-UEL

5 EXPERIMENTOS E RESULTADOS

Os experimentos foram realizados utilizando os conjuntos de dados *USCities*, *HCImages* e *HCImages Transformed* e as estruturas *cx-Sim** e o *S2I*. Os testes foram divididos em 2 partes: a primeira parte consistiu em testes com consultas do tipo k vizinhos mais próximos variando-se o valor k e a segunda parte consistiu em testes com consultas do tipo k vizinhos mais próximos, dessa vez variando-se a seletividade da consulta. Os testes foram realizados em um computador do laboratório do CIA-AGRO [1] no departamento de computação da Universidade Estadual de Londrina com as seguintes configurações: processador AMD Ryzen 5 PRO 5650G (6 núcleos/12 *threads* em 3.9GHz), 32GB de memória RAM em 3200MHz, NVME SAMSUNG MZVL4512HBLU-00BL7 (3500MB/s de escrita e 2500MB/s de leitura) e sistema operacional Linux Ubuntu 22.04.4 LTS.

Os tipos de consultas realizadas em cada um dos conjuntos de dados foram os seguintes.

- Condição simples:
  - *USCities*: retorne as k cidades mais próximas a cidade c onde a idade média da população está entre w e x anos;
  - *HCImages* e *HCImages Transformed*: retorne as k imagens mais próximas a imagem i onde a idade do paciente está entre w e x anos.
- Condições compostas:
  - *USCities*: retorne as k cidades mais próximas a cidade c onde o tamanho médio da família está entre w e x anos e a idade média da população está entre y e z anos;
  - *HCImages* e *HCImages Transformed*: retorne as k imagens mais próximas a imagem i onde a idade do paciente está entre w e x anos e o peso do paciente está entre y e z quilos.

As métricas coletadas nos experimentos são:

- tempo médio de consulta;
- número de acessos ao arquivo de índice;
- número de cálculos de distância;

[1] http://ciaagro.uel.br/

- número total de acessos a disco (arquivos de índice e arquivos de dados);
- quantidade de comparações realizadas entre os atributos simples.

Destas métricas, três são comparáveis entre as estruturas *cx-Sim** e o *S2I* (tempo médio de consulta, número total de acessos ao índice e número de cálculos de distância), e as demais são específicas das estruturas *cx-Sim**.

Assim como foi comentado no capítulo anterior, foi necessário subtrair o tempo de recuperação de tuplas através do *rowId* em algumas estruturas do repositório *Gen-Knn*, uma vez que o *S2I* não armazena a tupla completa no seu índice (somente a(s) coluna(s) escolhida(s)), enquanto as outras estruturas realizam o processo de recuperação da tupla completa após a busca. Caso contrário, o tempo de execução do *S2I* seria muito menor que o das outras estruturas, o que não seria justo para a comparação.

No caso do *S2I*, não foi possível coletar o número de comparações realizadas entre os atributos simples, em virtude do uso de uma classe do *HashMap* já compilada do Java em um .class que não permitia a modificação do código para a coleta dessa métrica em caso de colisão de *hash* dos termos no índice.

Objetivando obter resultados mais precisos e justos, no conjunto de dados *USCities* foram utilizados 500 centros de consultas, enquanto nos conjuntos *HCImages* e *HCImages Transformed* foram utilizados 100 centros de consulta, fazendo-se a média aritmética das métricas apresentadas anteriormente.

É importante ressaltar que nos testes do conjunto de dados *HCImages* não foi possível utilizar o *S2I* devido à sua limitação de operar somente sobre dados bidimensionais.

Nos gráficos com condições simples, a legenda "CXSim" refere-se à estrutura *cx-Sim* simple*.

5.1 Testes com consultas do tipo k vizinhos mais próximos variando k

Os testes com consultas do tipo k vizinhos mais próximos foram realizados com o intuito de analisar o desempenho das estruturas *cx-Sim** e do *S2I* em consultas com condições simples e compostas. Foram realizados testes com valores de k variando nos valores: 1, 5, 10, 25, 50, 75, 100, 150, 200, 300, 400, 500 A seletividade foi calculada utilizando a fórmula:

$$\text{Seletividade}(\%) = \left(1 - \frac{\text{Número de tuplas retornadas}}{\text{Número total de tuplas}}\right) * 100$$

5.1.1 Condições simples

Nesta subseção serão apresentados os resultados dos testes realizados com consultas do tipo k vizinhos mais próximos variando k com condições simples para os três conjuntos de dados utilizados. Para cada conjunto de dados, temos as seletividades escolhidas mostradas na tabela 6 para condições simples.

Base de dados	Seletividade	Consulta
USCities	90.00%	$44.8 \leq$ POPULATION_MEDIAN_AGE ≤ 70.1
HCImages	90.90%	$53 \leq PATIENT\_AGE \leq 58$
HCImages Transformed	90.90%	$53 \leq PATIENT\_AGE \leq 58$

Tabela 6 – Seletividades utilizadas nos testes com condição simples

As figuras 13, 14 e 15 mostram as três métricas comparáveis entre as estruturas *cx-Sim** e o *S2I*, variando a quantidade de vizinhos mais próximos k.

Em relação ao tempo médio de consulta, é importante dizer que o comportamento do *S2I* é praticamente constante para os outros valores de k devido ao fato da única diferença de execução ser a quantidade de tuplas que são retornadas, que implica em um corte na lista de candidatos, o que ocasiona em um tempo de execução praticamente constante.

Para a estrutura *cx-Sim* simple*, é possível notar que para conjuntos pequenos como o *USCities*, tem-se um desempenho mediano no tempo médio de consulta, porém para conjuntos maiores como o *HCImages* e o *HCImages Transformed*, o desempenho é muito superior, demonstrando assim sua escalabilidade superior em relação aos demais.

Sobre a quantidade de acessos ao índice (figura 14), é possível perceber que o *S2I* possui um número de acessos muito maior que o *cx-Sim** para conjuntos grandes (*HCImages Transformed*), o que é esperado, uma vez que o *S2I* realiza a busca de todas as tuplas que possuem as *keywords* da consulta para somente depois realizar a ordenação por *score* e corte da lista, enquanto o *cx-Sim** busca a tuplas que satisfazem a condição passada e vai adicionando em uma fila de prioridade, o que ocasiona em um número menor de acessos. Esse comportamento ocorre devido a vários *caches miss* que ocorrem no *S2I* devido à quantidade de tuplas que são buscadas.

Por fim, é possível notar que na quantidade de cálculos de distância (figura 15), o *S2I* sempre realiza muito mais cálculos de distância que qualquer outro método (na ordem de 100 a 1000 vezes mais) independente do conjunto utilizado, devido ao seu funcionamento mostrado anteriormente.

As outras duas métricas avaliadas das estruturas do repositório *Gen-Knn* são mostradas nas figuras 16, 17. Os acessos totais ao disco para os métodos *BSlim*, *Covering Slim* e *cx-Sim* simple* são muito menores se comparados ao método *Table Slim* já que neces-

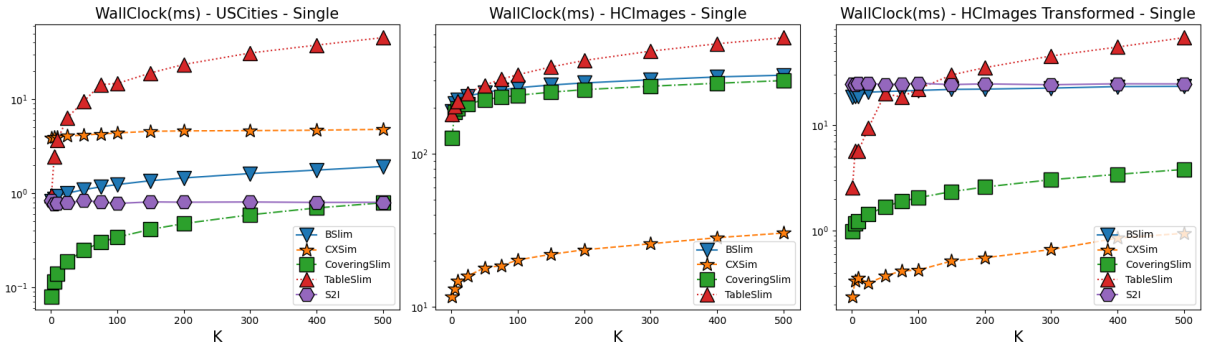


Figura 13 – Tempo médio de consulta para consultas do tipo k vizinhos mais próximos variando k com condições simples

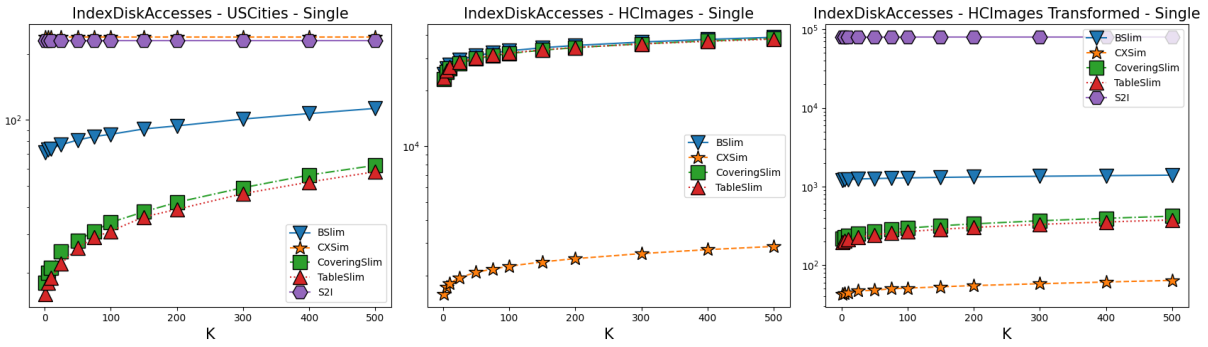


Figura 14 – Número total de acessos ao índice para consultas do tipo k vizinhos mais próximos variando k com condições simples

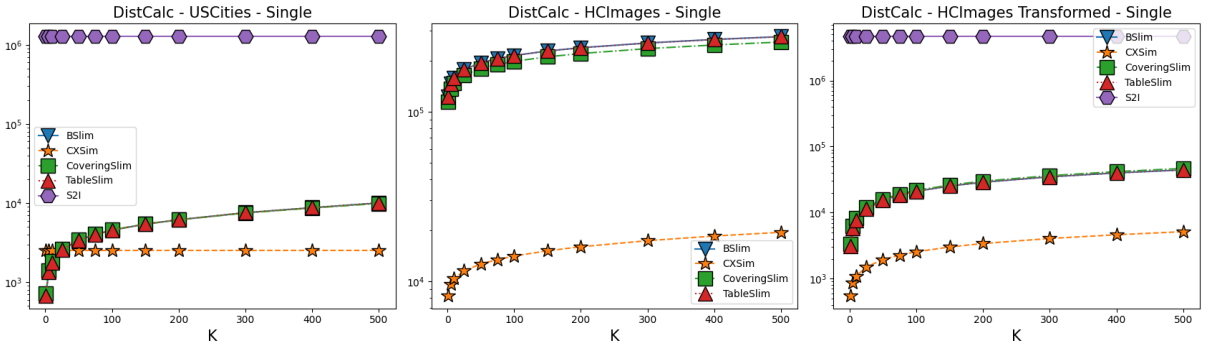


Figura 15 – Número de cálculos de distância para consultas do tipo k vizinhos mais próximos variando k com condições simples

sitam somente k acessos a disco por execução da consulta, como mostrado na figura 16. Isso ocorre porque o *Table Slim* necessita acessar uma vez o disco para recuperar o dado complexo e outra vez para recuperar o dado tradicional, enquanto os outros métodos necessitam somente de um acesso ao disco para analisar os dois.

A estrutura *cx-Sim* simple* também se destaca na quantidade de comparações realizadas entre os atributos simples (figura 17) devido a sua arquitetura em dois níveis, onde esse método demonstra uma quantidade constante de comparações feitas na B^+-

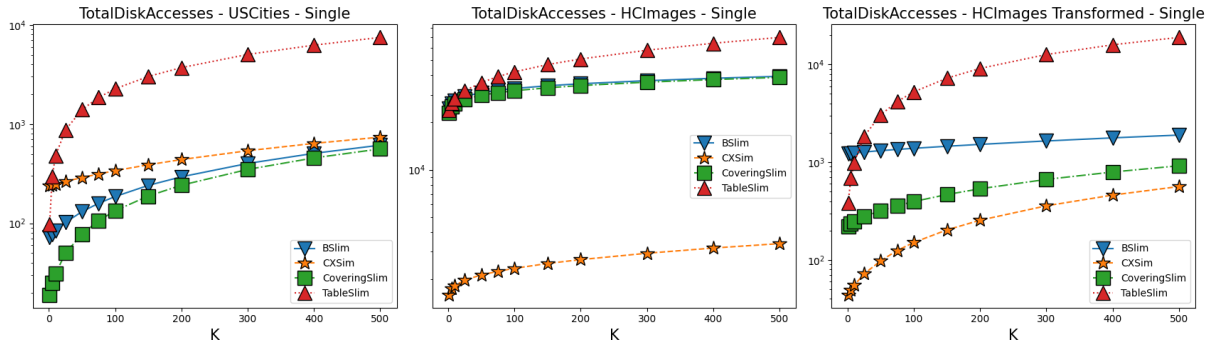


Figura 16 – Número total de acessos ao disco para consultas do tipo k vizinhos mais próximos variando k com condições simples

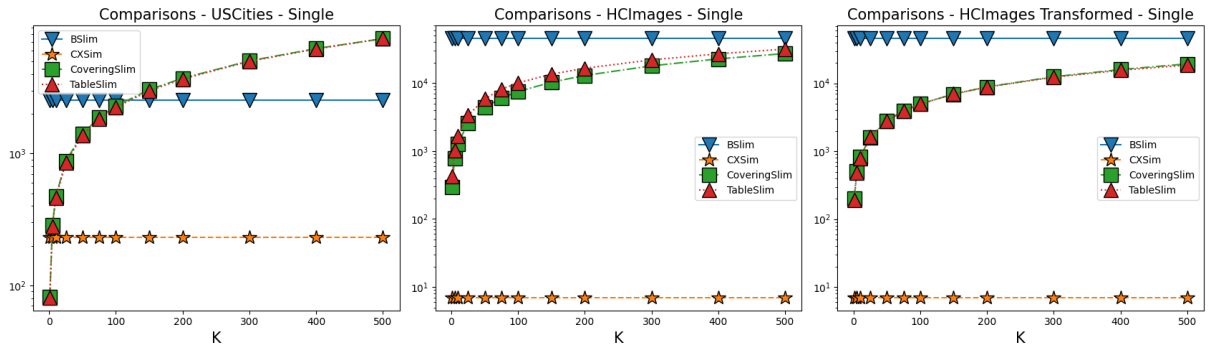


Figura 17 – Número de comparações realizadas entre os atributos simples para consultas do tipo k vizinhos mais próximos variando k com condições simples

tree. O elevado número de comparações mostrado pelos outros métodos se dá pelo fato de que eles primeiro realizam o cálculo de distância e depois verificam se a tupla satisfaz a condição passada, o que ocasiona em um número maior de comparações, já que não necessariamente a tupla que possui a menor distância é a que satisfaz a condição.

5.1.2 Condições compostas

Nesta subseção, serão apresentados os resultados dos testes realizados com consultas do tipo k vizinhos mais próximos variando k com condições compostas para os três conjuntos de dados utilizadas. As estruturas *cx-Sim Simple*, *cx-Sim Covering tree*, *cx-Sim Chained tree*, *cx-Sim Composite tree* e o *S2I* foram utilizados nos testes, sendo que este último utiliza a adaptação de nomear as colunas com *a_* e *b_* para realizar a busca por múltiplas colunas. A tabela 7 mostra as seletividades escolhidas para as consultas com condições compostas.

As figuras 18, 19 e 20 mostram as três métricas comparáveis entre as estruturas *cx-Sim** e o *S2I* para consultas com condições compostas variando a quantidade de vizinhos mais próximos k.

Em relação ao tempo médio de consulta (figura 18), em *USCities*, o *S2I* e *cx-Sim*

Base de dados	Seletividade	Consulta
USCities	95.00%	$40 \leq$ POPULATION_MEDIAN_AGE ≤ 48 AND $2.9 \leq$ AVERAGE_FAMILY_SIZE ≤ 2.97
HCImages	95.00%	$36 \leq$ PATIENT_AGE ≤ 51 AND $110 \leq$ PATIENT_WEIGHT ≤ 150
HCImages Transformed	95.00%	$36 \leq$ PATIENT_AGE ≤ 51 AND $110 \leq$ PATIENT_WEIGHT ≤ 150

Tabela 7 – Seletividades utilizadas nos testes com condição composta

Covering tree demonstraram bons resultados em comparação com as outras três abordagens. O *S2I* possui bom desempenho para conjuntos pequenos, assim como mostrado nos testes com condição simples na figura 13, porém seu desempenho é muito pior em conjuntos maiores como o *HCImages Transformed* por envolver a interseção das listas (AND lógico comentado na seção de adaptações). Já a abordagem *cx-Sim Covering tree* se beneficia do particionamento do conjunto de dados em florestas de árvores *Slim-tree* relativamente pequenas.

Os acessos ao índice (figura 19) seguem o mesmo padrão de comportamento dos testes com condições simples, onde o *S2I*, para conjuntos grandes, realiza um número muito maior de acessos a disco que os demais métodos, enquanto as estruturas *cx-Sim** se beneficiam da menor quantidade de acessos ao índice para conjuntos maiores. Em particular, pode-se observar que para o conjunto de dados *HCImages*, o particionamento realizado pelo *cx-Sim Chained tree* e *cx-Sim Composite tree* se mostra eficaz tanto no número de acessos ao índice quanto no tempo de consulta, o que é explicado pela redução do tamanho da árvore B^+*-tree*.

Por fim, a partir do número de cálculos de distância (figura 20) é possível perceber que os métodos que utilizam os dois atributos simples antes de avaliar a similaridade entre os atributos complexos (*cx-Sim Chained tree* e *cx-Sim Composite tree*) realizam menos cálculos de distância por construírem florestas de árvores *Slim-tree* menores, independente do conjunto.

As outras duas métricas exclusivas das estruturas do repositório *Gen-Knn* são mostradas nas figuras 21, 22.

O método *cx-Sim Simple* precisa acessar o arquivo de dados para testar a segunda condição, o que ocasiona em um tempo maior de consulta e pode ser verificado na figura 21, onde seu acesso cresce de maneira mais acelerada em relação aos demais métodos. Para o *HCImages*, a situação se inverte de maneira que o *cx-Sim Chained tree* e o *cx-Sim Composite tree* se sobressaem em relação aos demais métodos, fruto da partição dos

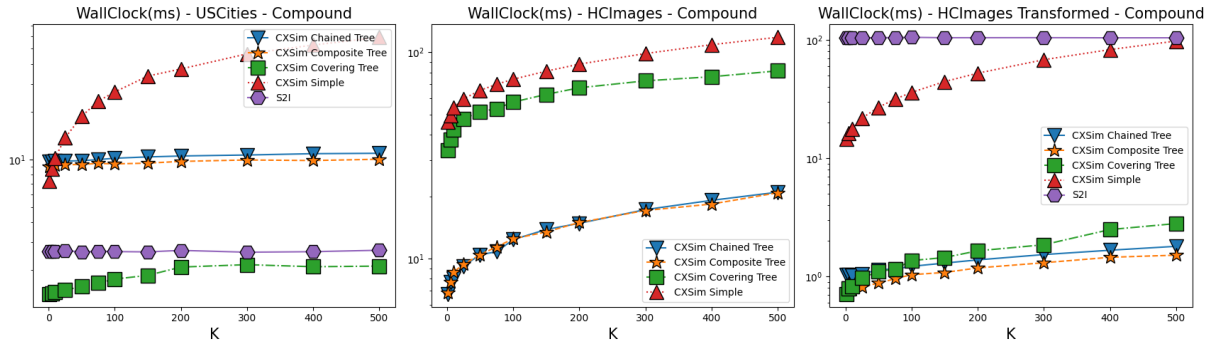


Figura 18 – Tempo médio de consulta para consultas do tipo k vizinhos mais próximos variando k com condições compostas

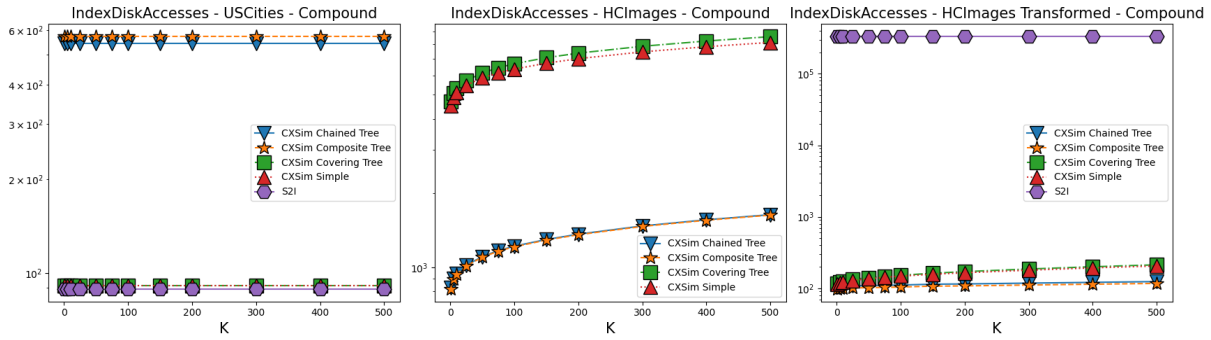


Figura 19 – Número total de acessos ao índice para consultas do tipo do tipo k vizinhos mais próximos variando k com condições compostas

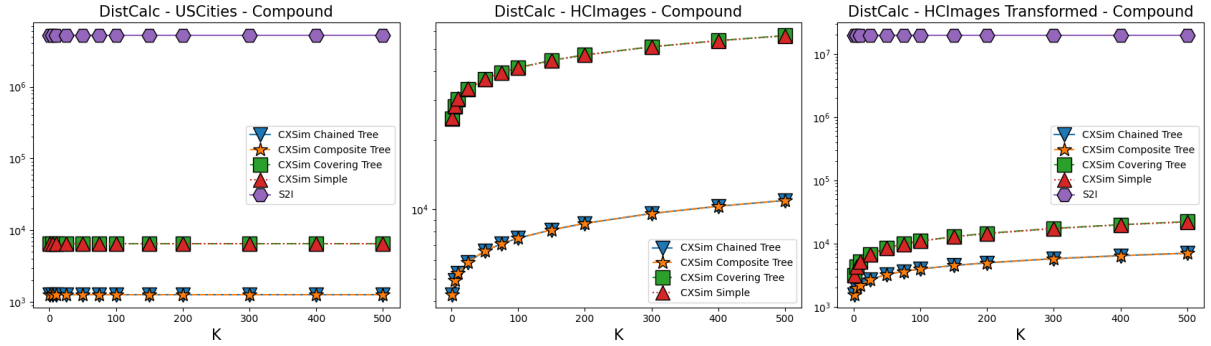


Figura 20 – Número de cálculos de distância para consultas do tipo k vizinhos mais próximos variando k com condições compostas

dados em florestas de árvores *Slim-tree* menores por utilizarem ou a abordagem de árvores encadeadas ou a abordagem de árvores com chave composta como mostrado nas seções 3.2.3 e 3.2.4. Foi interessante também notar que para o *HCImages Transformed*, o *cx-Sim Covering tree* se sobressaiu em relação aos demais métodos, o que pode ser explicado pela redução da dimensionalidade dos dados, o que ocasiona em uma menor quantidade de acessos a disco por reduzir o tamanho da tupla.

Por fim, a quantidade de comparações realizadas entre os atributos simples (figura

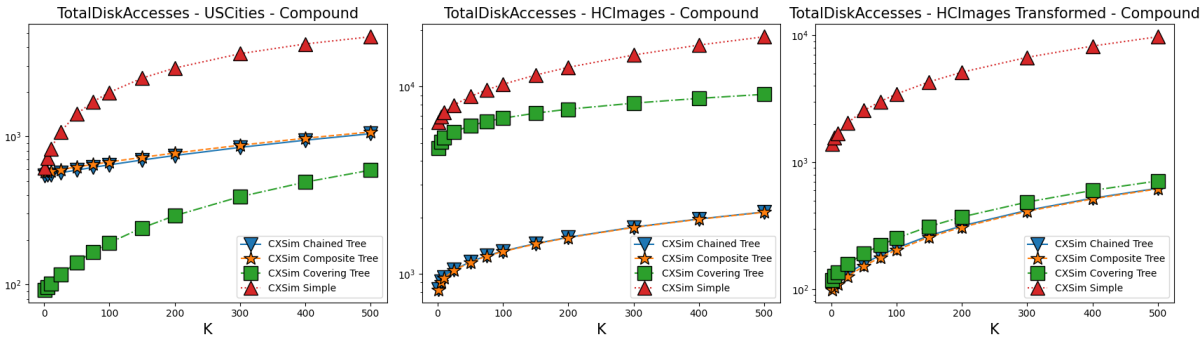


Figura 21 – Número total de acessos ao disco para consultas do tipo k vizinhos mais próximos variando k com condições compostas

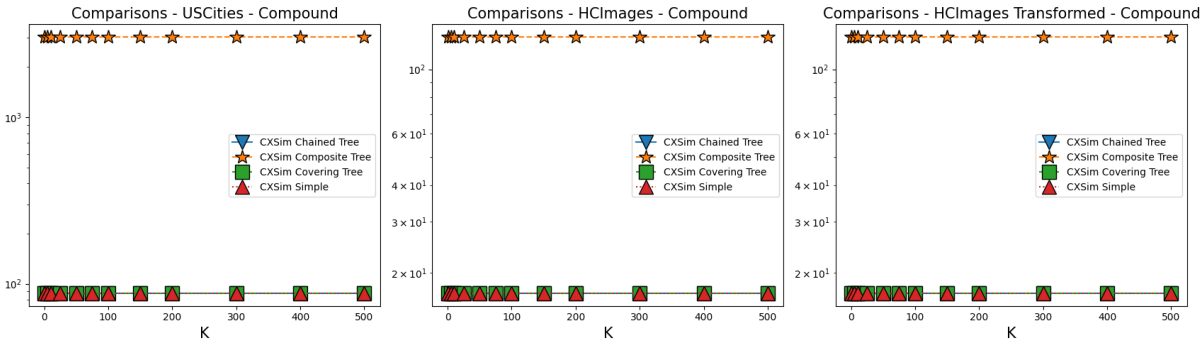


Figura 22 – Número de comparações realizadas entre os atributos simples para consultas do tipo k vizinhos mais próximos variando k com condições compostas

22) elucidam que o método *cx-Sim Composite tree* sempre realiza mais comparações. Isso ocorre porque os dois atributos simples indexados estão na mesma estrutura (por estarem na chave composta de cada nó), o que ocasiona uma árvore B^+-*tree* com maior altura em relação as demais abordagens, o que ocasiona em mais comparações.

5.2 Testes com consultas do tipo k vizinhos mais próximos variando a seletividade

Os testes com consultas do tipo k vizinhos mais próximos variando a seletividade foram realizados com o intuito de analisar o desempenho das estruturas *cx-Sim** e do *S2I* em consultas com condições simples e compostas. Foram realizados testes com diferentes valores de seletividade variando no intervalo de 50% até 99.9%. O valor de k foi fixado em 100 para todos os testes. Os valores de seletividade variam de acordo com cada conjunto de dados, respeitando o intervalo mencionado anteriormente.

5.2.1 Condições simples

Nesta subseção serão apresentados os resultados dos testes realizados com consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples para os três conjuntos de dados utilizadas. As consultas com condições simples, como mencionado no início deste capítulo seguem o formato mostrado abaixo e os valores de w e x que conferem as seletividades utilizadas nos testes são mostrados nas tabelas 8 e 9.

- *USCities*: retorne as k cidades mais próximas a cidade c onde a idade média da população está entre w e x anos.
- *HCImages* e *HCImages Transformed*: retorne as k imagens mais próximas a imagem i onde a idade do paciente está entre w e x anos.

Seletividade	w	x
50%	37.4	69
70%	40	62.2
90%	44.8	70.1
94,99%	48.2	77.4
98,99%	56.7	66
99,90%	66.1	67.8

Tabela 8 – Seletividades utilizadas nos testes com condições simples - USCities

Seletividade	w	x
50%	8	49
70%	53	78
89,99%	61	70
94,99%	52	54
98,97%	83	89
99,88%	87	89

Tabela 9 – Seletividades utilizadas nos testes com condições simples - HCImages e HCImages Transformed

As figuras 23, 24 e 25 mostram as três métricas comparáveis entre as estruturas *cx-Sim** e o *S2I* para consultas com condições simples variando a seletividade da consulta.

O tempo médio de consulta (figura 23) para o *Covering slim* e *Table slim* foi sempre aumentando, em consequência do aumento da filtragem de tuplas, o que requer a varredura de mais tuplas para encontrar as k mais próximas que satisfazem as condições passadas. É importante saber que isso aconteceu porque nesses dois métodos, a checagem da condição é feita após o cálculo de distância, o que ocasiona em um aumento do tempo de consulta. Já para o *cx-Sim* simple* e o *S2I*, o aumento da seletividade causou a diminuição do tempo

de consulta, o que é esperado, uma vez que no caso do *cx-Sim* simple*, a quantidade de *Slim-trees* que são acessadas é menor, e no caso do *S2I*, a quantidade de keywords que são buscadas é menor, o que ocasiona em um menor tempo de consulta.

Para o número total de acessos ao índice (figura 24), pode-se observar o mesmo comportamento do tempo médio de busca, onde o *Covering slim* e o *Table slim* aumentam a quantidade de acessos ao índice, enquanto o *cx-Sim* simple* e o *S2I* diminuem a quantidade de acessos ao índice. Novamente, isso acontece porque o *Covering slim* e o *Table slim* precisam acessar mais tuplas para verificar se satisfazem a condição, enquanto o *cx-Sim* simple* garante que apenas tuplas que satisfazem a condição sejam acessadas e o *S2I* garante que apenas tuplas que possuem as keywords sejam acessadas.

Por fim, a quantidade de cálculos de distância é influenciada diretamente pela presença de uma estrutura que faça a pré-validação das tuplas que satisfazem a condição, como mostrado na figura 25. O *cx-Sim* simple* e o *S2I* possuem tendências semelhantes, onde a quantidade de cálculos de distância diminui com o aumento da seletividade, enquanto o *Covering slim* e o *Table slim* possuem tendências opostas, onde a quantidade de cálculos de distância aumenta com o aumento da seletividade. Comparando o *cx-Sim* simple* e o *S2I*, a diferença de comparações chega a ser da ordem de 1000 vezes menor para o *cx-Sim* simple*, o que é esperado, uma vez que o *S2I* realiza a busca de todas as tuplas que possuem as *keywords* da consulta.

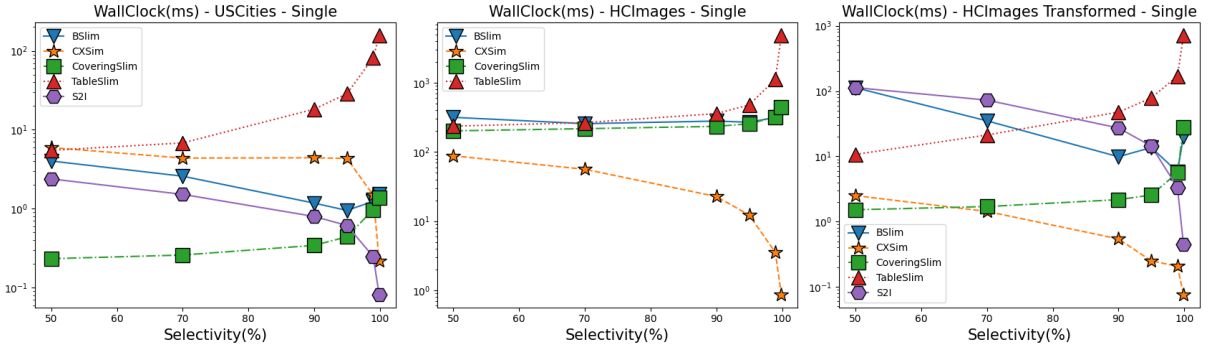


Figura 23 – Tempo médio de consulta para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

As outras duas métricas avaliadas das estruturas do repositório *Gen-Knn* são mostradas nas figuras 26, 27. Tanto o número de acessos a disco (figura 26), quanto o número de comparações realizadas (figura 27), seguem as mesmas tendências das métricas anteriores, onde o *Covering slim* e o *Table slim* aumentam a quantidade de acessos a disco e comparações realizadas, enquanto o *cx-Sim* simple* diminui a quantidade de acessos a disco e comparações realizadas, fruto da presença da B^+*-tree* que faz a pré-validação das tuplas que satisfazem a condição.

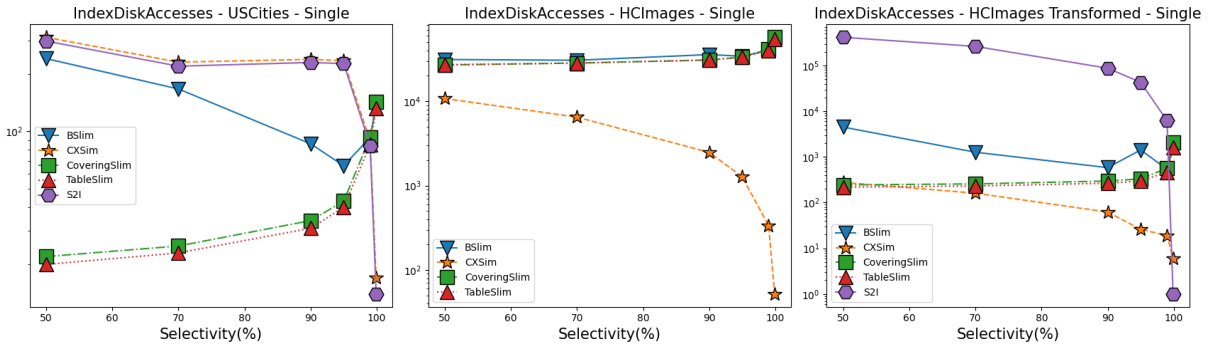


Figura 24 – Número total de acessos ao índice para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

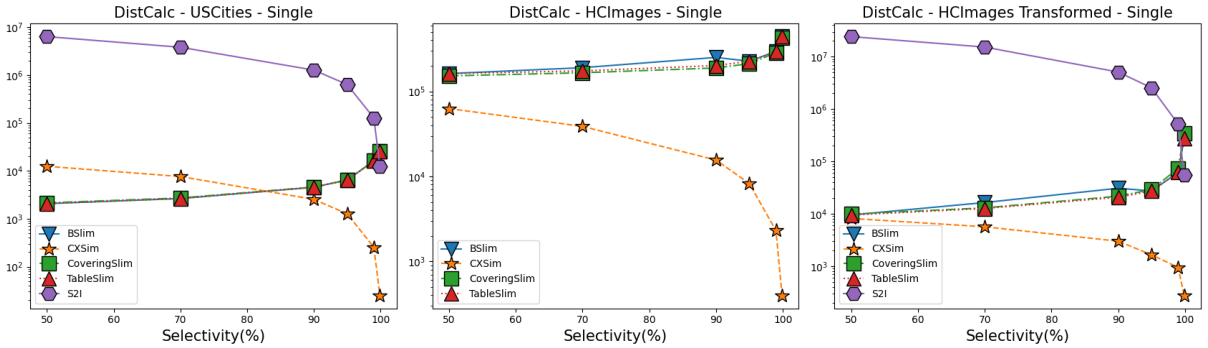


Figura 25 – Número de cálculos de distância para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

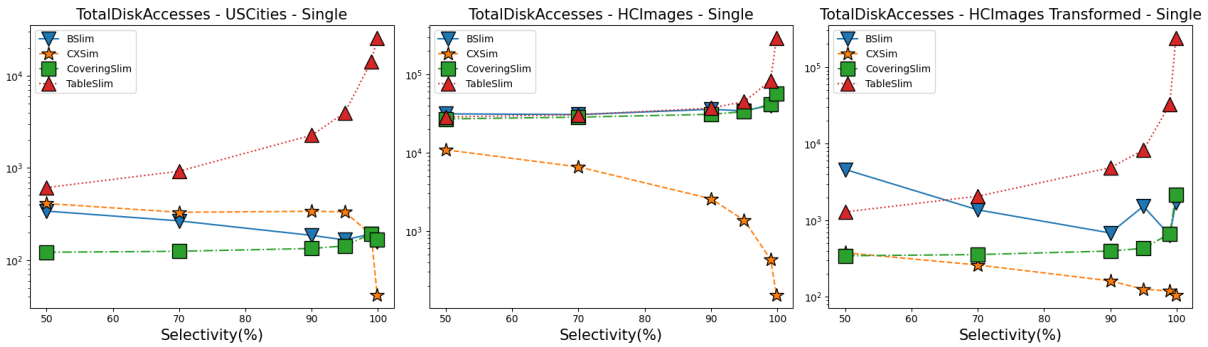


Figura 26 – Número total de acessos ao disco para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

5.2.2 Condições compostas

Nesta subseção, serão apresentados os resultados dos testes realizados com consultas do tipo k vizinhos mais próximos variando a seletividade com condições compostas para os três conjuntos de dados utilizadas. As consultas com condições compostas, como mencionado no início deste capítulo, seguem o formato mostrado a seguir e os valores de w, x, y e z que conferem as seletividades utilizadas nos testes são mostrados nas tabelas 10 e 11.

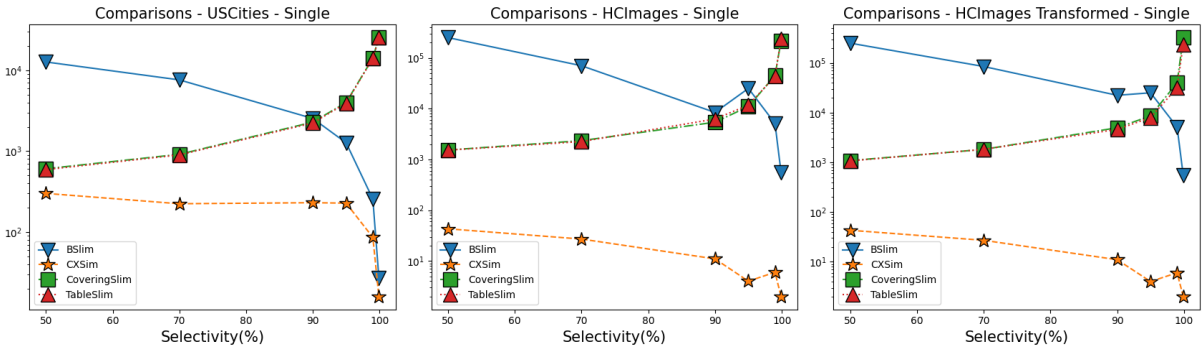


Figura 27 – Número de comparações realizadas entre os atributos simples para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

- *USCities*: retorne as k cidades mais próximas a cidade c onde a idade média da população está entre w e x anos e a média do tamanho da família está entre y e z.
- *HCImages e HCImages Transformed*: retorne as k imagens mais próximas a imagem i onde a idade do paciente está entre w e x anos e o peso do paciente está entre y e z.

Seletividade	w	x	y	z
75,16%	40	48	2.6	8.1
76,67%	40	48	2.7	8.1
80,75%	40	48	2.8	8.1
84,45%	40	48	2.85	8.1
88,59%	40	48	2.9	8.1
94,42%	40	48	3.0	8.1
97,55%	40	48	3.1	8.1
98,98%	40	48	3.2	8.1
99,60%	40	48	3.3	8.1

Tabela 10 – Seletividades utilizadas nos testes com condições compostas - USCities

Seletividade	w	x	y	z
74.33%	39	55	50	150
75,80%	39	55	70	150
79,53%	39	55	72	150
83,99%	39	55	90	150
88,15%	39	55	95	150
94,49%	39	55	110	150
98,48%	39	55	120	150
99,66%	39	55	130	150

Tabela 11 – Seletividades utilizadas nos testes com condições compostas - HCImages e HCImages Transformed

As figuras 28, 29 e 30 mostram as três métricas comparáveis entre as estruturas *cx-Sim** e o *S2I* para consultas com condições compostas variando a seletividade da consulta. Foi possível observar o bom desempenho que o método *cx-Sim Covering tree* obteve no conjunto de dados *USCities*, mantendo-se com um tempo de consulta baixo e um número de acessos ao índice baixo. Os métodos *cx-Sim Chained tree* e *cx-Sim Composite tree* obtiveram também resultados satisfatórios na medida em que a seletividade aumentava, o que é explicado pela redução do tamanho da árvore B^+*-tree*, chegando até mesmo a superar o *S2I* em *USCities*. Para conjuntos de dados maiores, como o *HCImages* e *HCImages Transformed*, o uso dos dois atributos simples nas estruturas da primeira camada (*cx-Sim Chained tree* e *cx-Sim Composite tree*) mostrou-se ainda mais eficaz, reduzindo o tamanho das árvores *Slim-tree* e consequentemente o tempo de consulta, número de acessos ao índice e quantidade de cálculos de distância realizados.

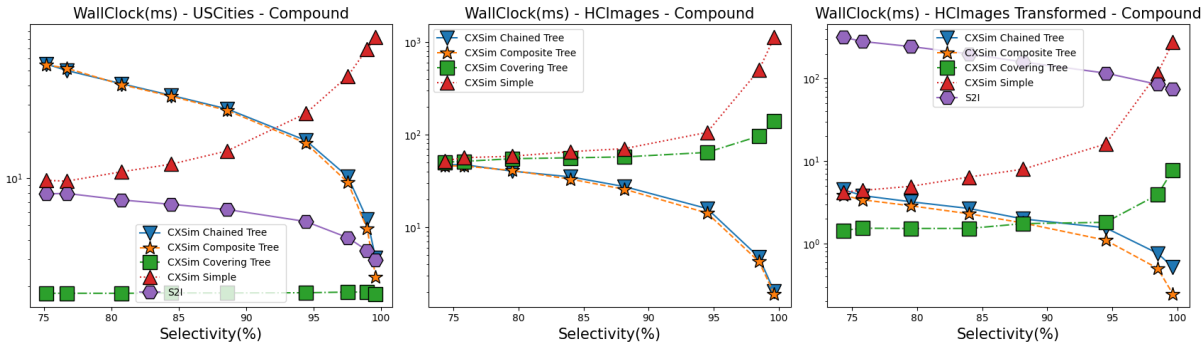


Figura 28 – Tempo médio de consulta para consultas do tipo k vizinhos mais próximos variando a seletividade com condições compostas

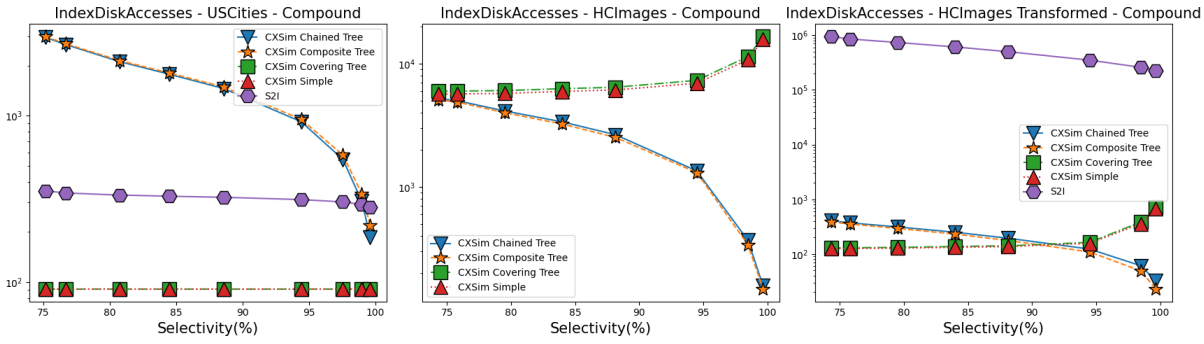


Figura 29 – Número total de acessos ao índice para consultas do tipo k vizinhos mais próximos variando a seletividade com condições compostas

Por fim, as outras duas métricas avaliadas das estruturas do repositório *Gen-Knn* são mostradas nas figuras 31, 32.

A quantidade de acessos a disco (figura 31) do método *cx-Sim simple* é crescente pois o método precisa acessar o arquivo de dados para testar a segunda condição, o que ocasiona em um tempo maior de consulta e pode ser verificado na figura 28. Para

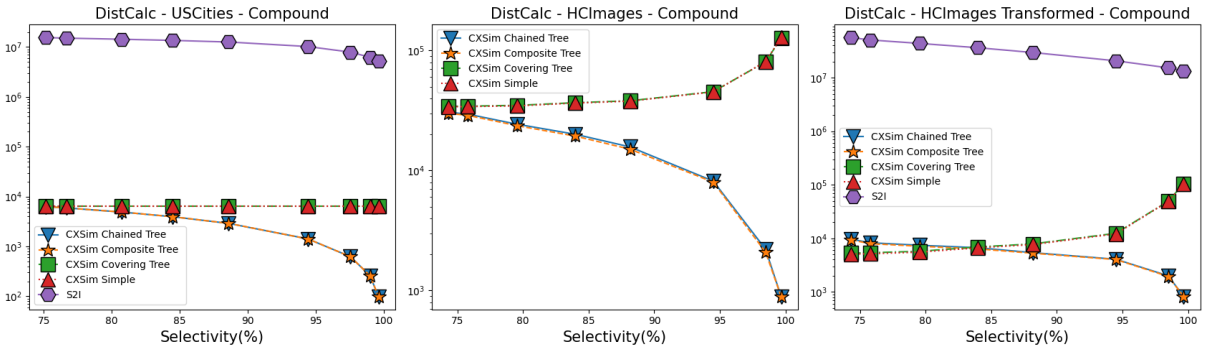


Figura 30 – Número de cálculos de distância para consultas do tipo k vizinhos mais próximos variando a seletividade com condições compostas

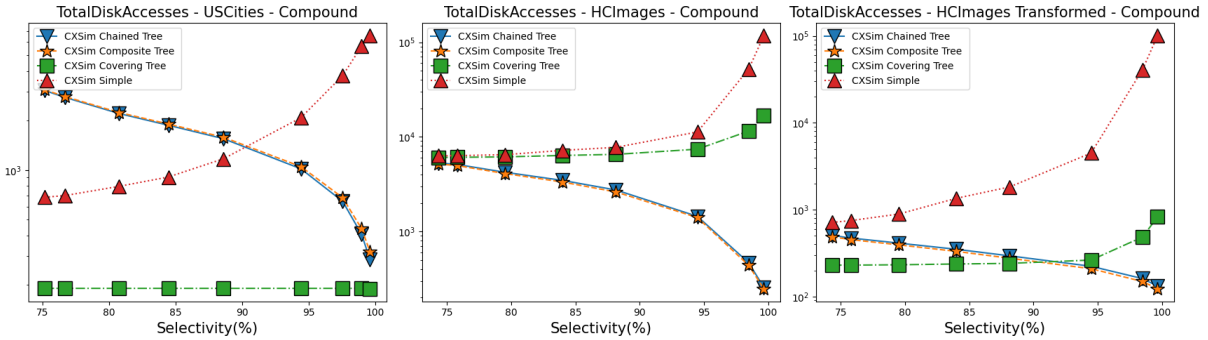


Figura 31 – Número total de acessos ao disco para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

os conjuntos maiores (*HCImages* e *HCImages Transformed*), a divisão do arquivo de dados proporcionado pela primeira camada das estruturas *cx-Sim Chained tree* e *cx-Sim Composite tree* em florestas de árvores *Slim-tree* menores enaltece a eficácia desses métodos, reduzindo o número de acessos a disco.

Por fim, a quantidade de comparações realizadas entre os atributos simples (figura 32) elucida que o método *cx-Sim Composite tree* sempre realiza mais comparações, em função da altura da árvore B^+*-tree* ser maior em relação as demais abordagens, porém conferindo maior performance para a estrutura.

5.3 Sumarização dos resultados obtidos

Em síntese, pode-se fazer as seguintes considerações sobre os resultados obtidos.

- O tamanho do conjunto de dados é um fator determinante para a escolha da estrutura de indexação, uma vez que o *S2I* se mostrou eficaz para conjuntos de dados pequenos, enquanto as estruturas *cx-Sim** se mostraram eficazes para conjuntos de dados maiores.

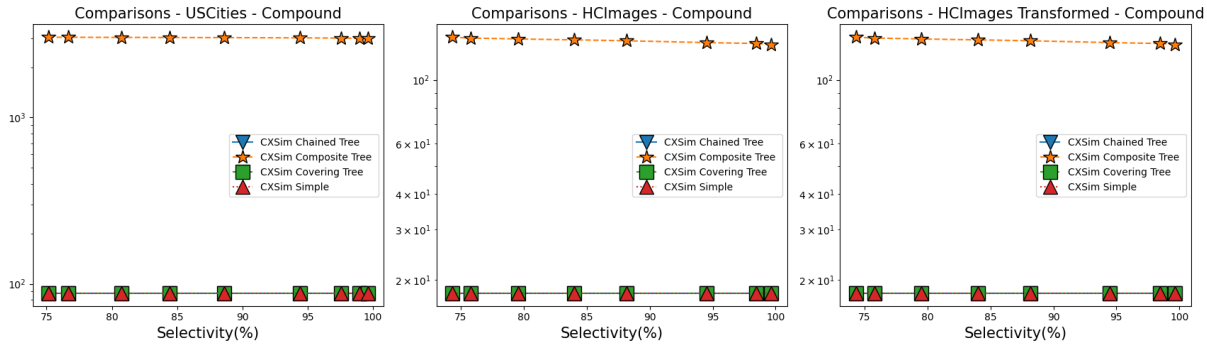
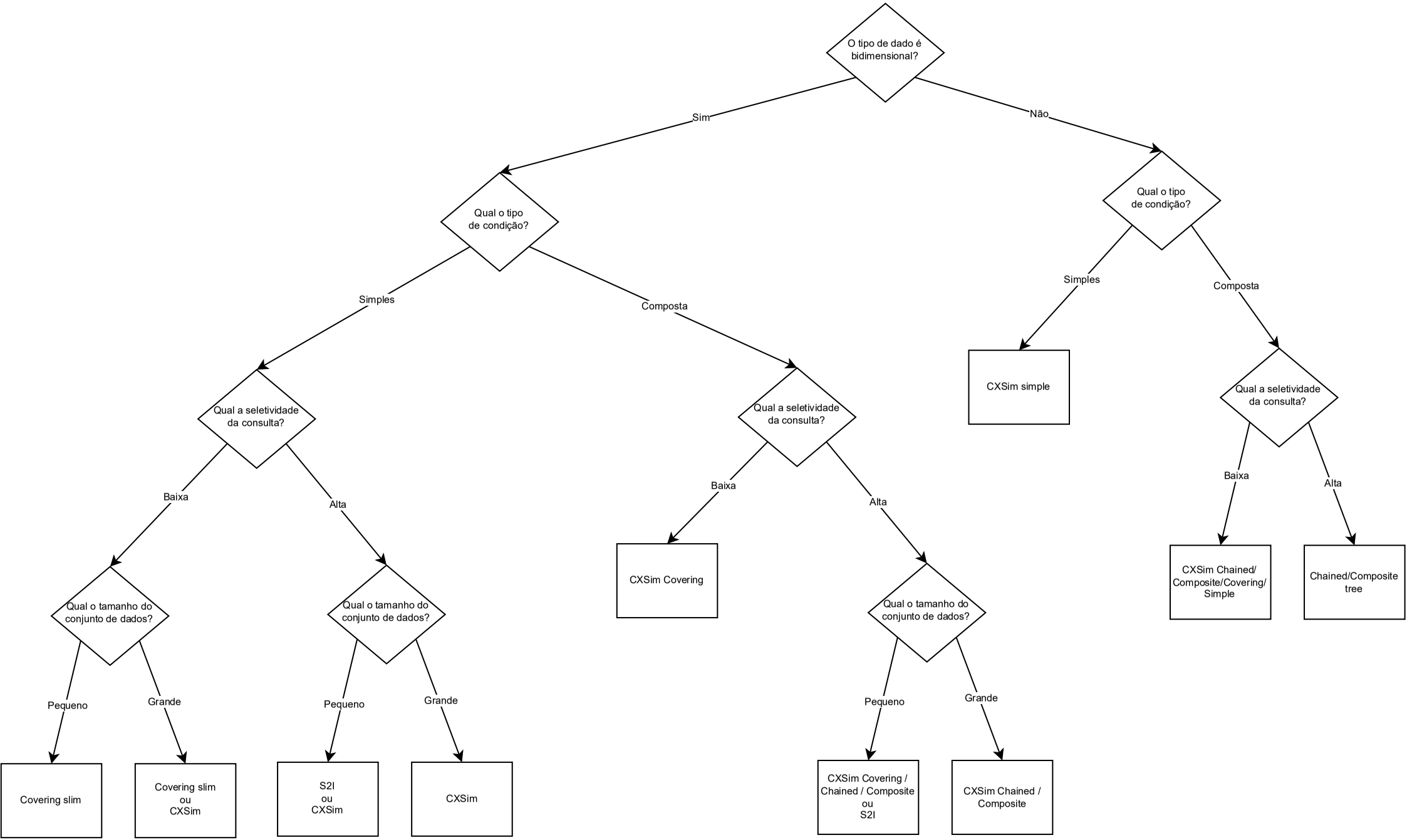


Figura 32 – Número de comparações realizadas entre os atributos simples para consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples

- O particionamento do conjunto de dados proporcionado pelo uso dos atributos simples reduz o tamanho das árvores métricas e consequentemente o tempo de consulta, número de acessos ao índice e quantidade de cálculos de distância realizados.
- A seletividade impacta na escolha do método de indexação, onde os métodos que utilizam *Covering-Slim-trees* em sua composição se mostraram eficazes para consultas com baixa seletividade. Para altas seletividades, os métodos com árvores *Slim-tree* menores se sobressaíram.

Concluindo a análise realizada, a figura a seguir mostra um diagrama de escolha da estrutura mais adequada para cada caso, considerando-se a natureza do dado, uma vez que o *S2I* é limitado a dados bidimensionais, bem como o tamanho do dataset, a seletividade da consulta e o tipo de condição. Foram analisados principalmente os gráficos de variação de seletividade durante a construção do diagrama.


O tipo de dado é bidimensional?
Sim
Não
Qual o tipo de condição?
Qual o tipo de condição?
Simples
Composta
Simples
Composta
Qual a seletividade da consulta?
Qual a seletividade da consulta?
CXSim simple
Qual a seletividade da consulta?
Baixa
Alta
Baixa
Alta
Baixa
Alta
Qual o tamanho do conjunto de dados?
Qual o tamanho do conjunto de dados?
CXSim Covering
Qual o tamanho do conjunto de dados?
CXSim Chained/ Composite/Covering/ Simple
Chained/Composite tree
Pequeno
Grande
Pequeno
Grande
Pequeno
Grande
Covering slim
Covering slim ou CXSim
S2I ou CXSim
CXSim
CXSim Covering / Chained / Composite ou S2I
CXSim Chained / Composite

6 CONCLUSÃO

Em virtude da enorme quantidade de dados gerados diariamente, a busca por métodos eficazes de recuperação de informações se torna cada vez mais necessária. Os SGBDs tradicionais enfrentam dificuldades para padronizar e recuperar informações de dados complexos, como imagens e vídeos, o que motiva a busca por novas abordagens de indexação e recuperação de informações, considerando a presença de atributos simples ligados a atributos complexos.

O presente trabalho explorou o conceito de similaridade e expôs os principais tipos de busca por similaridade, com enfoque no grupo de consultas por abrangência e k vizinhos mais próximos, sendo que neste último, foi explorado a introdução de condições adicionais na busca, mostrando suas peculiaridades. Em especial, o trabalho focou na exploração no tipo de busca pelos k vizinhos mais próximos, que é uma das operações mais comuns em sistemas de recuperação de informações, com a adição de condições sobre os dados complexos. Foram apresentadas estruturas de indexação próprias para a busca por similaridade, como a *Slim-tree* (*Table Slim*, *Covering Slim*), *cx-Sim* tree* (*Single*, *Covering*, *Chained*, *Composite*), *BSlim*. A primeira foi descrita com mais detalhes em virtude de sua importância para a compreensão das estruturas *cx-Sim** que utilizam a *Slim-tree* em sua arquitetura. Também foi incluso o método *S2I*, que é um método de busca por palavras-chave em documentos textuais, que pode ser considerado um caso especial de ${}_cKNNq$, onde a condição é de igualdade em um atributo simples da tupla.

A principal contribuição deste trabalho foi a avaliação experimental das estruturas de indexação *cx-Sim** e *S2I* em consultas do tipo k vizinhos mais próximos com condições simples e compostas, variando o valor de k e a seletividade da consulta. Destaca-se que esse processo envolveu o entendimento e adaptação dos códigos-fonte das estruturas de indexação para a execução dos testes, a escolha e preparação dos conjuntos de dados e a execução dos testes. A fim de organizar e monitorar esses processos, foi desenvolvida uma plataforma que permitiu a execução dos testes de forma automatizada, facilitando a comparação dos resultados obtidos e geração de relatórios de desempenho. Esta plataforma conta também com scripts auxiliares para a inserção dos dados em um banco de dados PostgreSQL, permitindo acelerar o processo de análise de cada conjunto de dados utilizado.

Por fim foram apresentados os resultados obtidos, que foram divididos em 4 partes: testes com consultas do tipo k vizinhos mais próximos variando k com condições simples e compostas, e testes com consultas do tipo k vizinhos mais próximos variando a seletividade com condições simples e compostas. Em síntese, existem algumas considerações que podem ser feitas a partir dos resultados obtidos, onde o tamanho do dataset, a

seletividade da consulta e o tipo de condição são fatores determinantes para a escolha da estrutura de indexação a ser utilizada, como mostrado no diagrama no final do capítulo 5. Em especial, a natureza dos dados, tamanho e quantidade de tuplas influenciam diretamente no desempenho das estruturas nos quesitos de acessos a disco, tempo de consulta e quantidade de cálculos de distância realizados.

Para trabalhos futuros, sugere-se a adição de novas estruturas de indexação, aumentando a gama de opções para a escolha da estrutura mais adequada para cada situação. Além disso, a adição de novos conjuntos de dados com diferentes tipos de dados complexos, inserção de novas métricas e a realização de testes com diferentes tipos de consultas e condições podem enriquecer ainda mais os resultados obtidos. Seria interessante também a execução dos testes em um disco rígido, para avaliar o impacto da latência de acesso a disco nos resultados obtidos.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[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] LI, Z. et al. Ir-tree: An efficient index for geographic document search. *IEEE transactions on knowledge and data engineering*, IEEE, v. 23, n. 4, p. 585–599, 2010.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] PIBIRI, G. E.; VENTURINI, R. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 53, n. 6, p. 1–36, 2020.

[54] MAHAPATRA, A. K.; BISWAS, S. Inverted indexes: Types and techniques. *International Journal of Computer Science Issues (IJCSI)*, Citeseer, v. 8, n. 4, p. 384, 2011.

[55] ZOBEL, J.; MOFFAT, A. Inverted files for text search engines. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 38, n. 2, p. 6–es, 2006.

[56] PAPADIAS, D. et al. Efficient olap operations in spatial data warehouses. In: SPRINGER. *Advances in Spatial and Temporal Databases: 7th International Symposium, SSTD 2001 Redondo Beach, CA, USA, July 12–15, 2001 Proceedings 7*. [S.l.], 2001. p. 443–459.