Título do Documento

Seu Nome

January 6, 2025

Teste

1. INTRODUÇÃO À INDEXAÇÃO

1.1 O que é indexação?

Um índice é uma estrutura de dados que melhora a velocidade das operações de query de dados em uma tabela, ao custo de escritas adicionais e espaço de armazenamento para manter a estrutura do índice. Índices permitem localizar dados rapidamente sem precisar buscar sequencialmente em cada linha de uma tabela.

A maioria dos softwares de banco de dados inclui tecnologia de indexação que permite buscas em tempo sub-linear para melhorar o desempenho, já que a busca linear é ineficiente para grandes bancos de dados.

[1]

1.2 Implementações de índices

1.2.1 B Tree

Uma **B-tree** é uma estrutura de dados em árvore auto-balanceada que mantém dados ordenados e permite buscas, acessos sequenciais, inserções e deleções em tempo logarítmico. A B-tree generaliza a árvore binária de busca, permitindo nós com mais de dois filhos.

É amplamente utilizada em sistemas de arquivos e bancos de dados. É uma estrutura que se beneficia da leitura e escrita em bloco, levando vantagem em um aspecto historicamente relevante, uma vez que o número de operações de I/O (em discos magnéticos) era igualmente relevante para o desempenho quanto o número de operações de comparação.

Foi inventada por Rudolf Bayer e Edward M. McCreight em 1972 [2] (o B não foi explicado por eles).

> *What Rudy (Bayer) likes to say is, the more you think about what the B in B-Tree means, the better you understand B-Trees!*

Os principais algoritmos associados a B-trees são: busca (algorithm 1) e inserção (algorithm 2) (existem variações para a operação de deleção).

São necessárias duas funções auxiliares para a inserção: splitChild, que divide um nó cheio em dois, e insertNonFull, que insere uma chave em um nó não cheio.

Bulk loading?

Algorithm 1 Algoritmo de busca na B Tree, assumindo que a chave k é o valor a ser buscado e x é o nó onde a busca começa.

```
1:  procedure BTreeSearch(x, k)
2:      i ← 0
3:      while i < x.n and k > x.key[i] do
4:          i ← i + 1
5:      end while
6:      if i < x.n and k = x.key[i] then
7:          return x
8:      end if
9:      if x.leaf then
10:         return None
11:     end if
12:     return BTreeSearch(x.child[i], k)
13: end procedure
```

Algorithm 2 Algoritmo de inserção na B Tree, assumindo que a chave k é o valor a ser inserido.

```
1:  procedure BTreeInsert(T, k)
2:      r ← T.root
3:      if r.n = 2(T.d) − 1 then
4:          s ← new Node
5:          T.root ← s
6:          s.child[1] ← r
7:          splitChild(s, 1)
8:          insertNonFull(s, k)
9:      else
10:         insertNonFull(r, k)
11:     end if
12: end procedure
```

[2], [3], [4]

1.2.2 B+ Tree

Uma B+ tree pode ser vista como uma B-tree onde cada nó contém apenas chaves (não pares chave-valor), com um nível adicional de folhas ligadas na parte inferior.

O principal valor de uma B+ tree está no armazenamento de dados para recuperação eficiente em um contexto de armazenamento orientado a blocos, como sistemas de arquivos. Diferente das árvores binárias de busca, as B+ trees têm um fanout muito alto (número de ponteiros para nós filhos em um nó, tipicamente na ordem de 100 ou mais), o que reduz o número de operações de I/O necessárias para encontrar um elemento na árvore.

Aplicações: iDistance

[5]

1.3 Indexação Multidimensional

Exemplo de como estava sendo realizada a indexação multidimensional em multimidia(imagens)

Efficient and Effective Querying by Image Content

1.3.1 R-Tree

Usada para multidimensional

1.3.2 KD-Tree

1.3.3 M-Tree

Usada para espaços métricos

1.3.4

Survey [6]

BIBLIOGRAPHY

[1] Wikipedia, *Database index*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Database_index.

[2] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970, pp. 107–141.

[3] Wikipedia, *B tree*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/B-tree.

[4] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[5] Wikipedia, *B+ tree*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/B%2B_tree.

[6] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 170–231, 1998.

2. INDEXAÇÃO PARA FINGERPRINTS

2.1 Introdução

- Matching local baseado em minúcias
  - Abordagens antigas [1], [2]
  - Associa cada minúcia as suas vizinhas em estruturas invariantes a rotação e distâncias[3], [4]
  - Baseada em cilindros, veja section 2.2
  - Outros métodos que incluem mais características: local orientation field, local frequency, ridge shapes

Estruturas locais de uma minúcia central podem ser baseadas em:

- *Vizinhos mais próximos*, que consideram as k minúcias mais próximas [3].

  A vantagem dessa representação é com relação ao tamanho fixo, facilitando no procedimento de comparação.

- *Raio fixo*, que considera todas as minúcias dentro de um raio fixo, usada em [4].

  A vantagem dessa representação é a tolerância com relação a ruído (minúcias extras ou faltantes).

2.2 Minutia Cylinder-Code

Baseada em [5], [6]

2.2.1 Representação

Uma representação tridimensional de minúcias baseada em distâncias entre minúcias e ângulos relativos. A representação recebe o nome de Minutia Cylinder-Code (MCC) e tem como características principais: invariância de rotação, tamanho fixo e orientada a codificação binária.

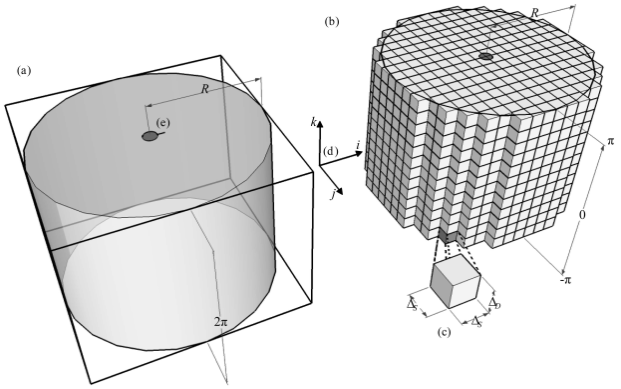

Fig. 2.1: Representação de um MCC.

O esquema de representação é um cilindro segmentado, como na fig. 2.1. O cilindro é um recorte de um cubo dividido em células, onde cada uma possui um valor indexado por $C_m[i, j, k]$.

O cálculo dos valores de $C_m[i, j, k]$ é complicado, mas essencialmente envolve as seguintes ideias:

- Verifica se está em uma região válida: dentro do cilindro e dentro do *convex hull* da fingerprint.
- Calcula a contribuição de cada minúcia vizinha usando uma Gaussiana, fig. 2.2.
- Calcula a contribuição de cada minúcia usando a diferença entre a orientação.

2.2.2 Similaridade

A similaridade entre dois cilindros é obtida a partir do seguinte procedimento:

1. Lineariza o cilindro em um vetor, similar a operação de `reshape`. Por exemplo, o cilindro de uma minúcia a, $C_a[i, j, k]$, é linearizado em $\mathbf{c}_a$.
2. Seleciona todas as entradas comparáveis desses vetores (células que são *válidas* em ambos) que dão origem aos vetores $\tilde{\mathbf{v}}_a$ e $\tilde{\mathbf{v}}_b$.
3. Na implementação binária, é realizado um XOR bit a bit entre os vetores.

$$\gamma(\tilde{\mathbf{v}}_a, \tilde{\mathbf{v}}_b) = 1 - \frac{\|\tilde{\mathbf{v}}_a \oplus \tilde{\mathbf{v}}_b\|}{\|\tilde{\mathbf{v}}_a\| + \|\tilde{\mathbf{v}}_b\|} \tag{2.1}$$

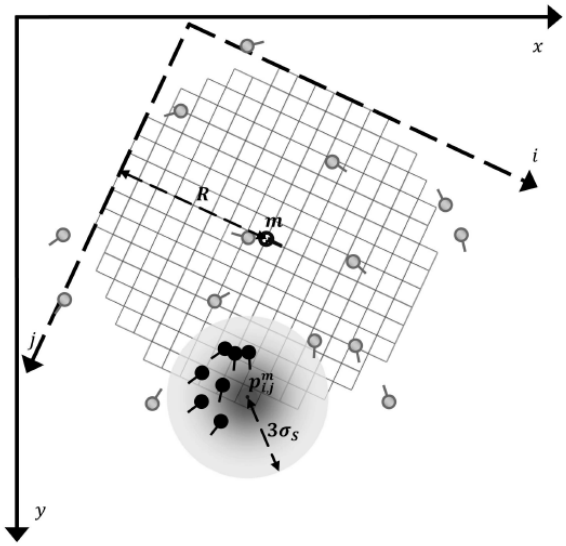

Fig. 2.2: Contribuição da distância de uma minúcia vizinha. Cores mais escuras refletem uma maior contribuição.

2.2.3 Indexação

Para a indexação de um MCC é empregado um esquema de *locality-sensitive hashing* (LSH), [7], [8].

O procedimento consiste em:

1. Seja $\mathbf{v}_m$ o vetor obtido da linearização de um cilindro.

2. É feita a projeção de $\mathbf{v}_m$ em um subespaço $\mathbf{h}_m$. A projeção é obtida simplesmente ao escolher um subconjunto dos índices H do vetor original.
$$\mathbf{h}_m = \mathbf{v}_m[H] \tag{2.2}$$

3. O conjunto H define uma função que mapeia um vetor binário $\mathbf{v}_m$ em um número natural obtido ao interpretar o vetor binário $\mathbf{h}_m$ como um número natural.
$$h_H : \{0,1\}^n \to \mathbb{N} \tag{2.3}$$

4. São definidos ℓ conjuntos $H_1, H_2, \ldots, H_\ell$, cada um com uma função h_{H_i}.

5. O índice por sua vez é um conjunto de *hash tables*, $\mathbb{H}_1, \mathbb{H}_2, \ldots, \mathbb{H}_\ell$, onde cada hash table tem os seus buckets definidos pela função h_{H_i}.

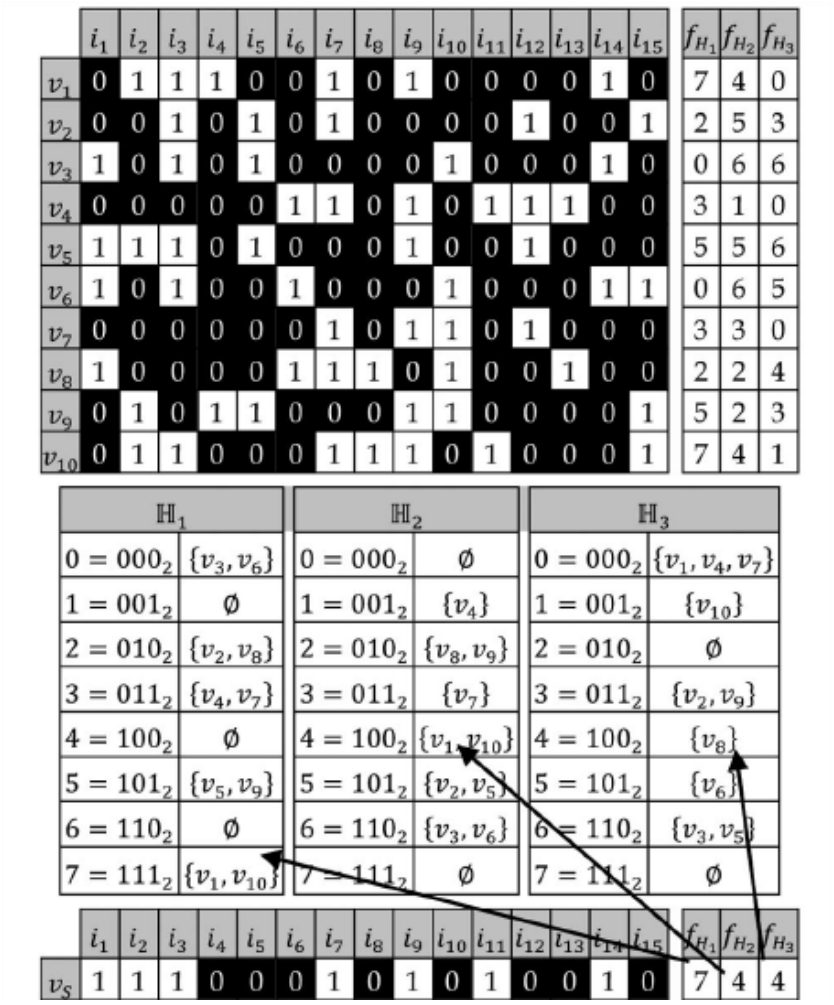

Fig. 2.3: Ilustração do procedimento de indexação usando LSH de um MCC.

6. A indexação segue fazendo a consulta do vetor desejado em cada hash table, retornando os candidatos.

7. Por fim, os candidatos são ranqueados usando a *distância de Hamming* entre os vetores.

O procedimento enumerado acima é ilustrado na fig. 2.3.

BIBLIOGRAPHY

[1] A. K. Hrechak and J. A. McHugh, "Automated fingerprint recognition using structural matching," *Pattern recognition*, vol. 23, no. 8, pp. 893–904, 1990.

[2] A. J. Willis and L. Myers, "A cost-effective fingerprint recognition system for use with low-quality prints and damaged fingertips," *Pattern recognition*, vol. 34, no. 2, pp. 255–270, 2001.

[3] X. Jiang and W.-Y. Yau, "Fingerprint minutiae matching based on the local and global structures," in *Proceedings 15th international conference on pattern recognition. ICPR-2000*, IEEE, vol. 2, 2000, pp. 1038–1041.

[4] N. K. Ratha, R. M. Bolle, V. D. Pandit, and V. Vaish, "Robust fingerprint authentication using local structural similarity," in *Proceedings Fifth IEEE Workshop on Applications of Computer Vision*, IEEE, 2000, pp. 29–34.

[5] R. Cappelli, M. Ferrara, and D. Maltoni, "Minutia cylinder-code: A new representation and matching technique for fingerprint recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 12, pp. 2128–2141, 2010.

[6] R. Cappelli, M. Ferrara, and D. Maltoni, "Fingerprint indexing based on minutia cylinder-code," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 5, pp. 1051–1057, 2010.

[7] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, 1999, pp. 518–529.

[8] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

3. VECTOR DATABASES

3.1 Introdução

Os chamados *Vector Databases* (VDB) surgem em um cenário no qual *dados não estruturados* são dominantes em praticamente todas as esferas e as previsões apontam para um crescimento contínuo e acelerado. Exemplos de dados dessa forma são aqueles que não podem ser armazenados de forma padronizada em tabelas, como imagens, vídeos, áudios e textos.

Para contrapor a definição de dados não estruturados, entende-se por *dados estruturados* aqueles que podem ser colocados em forma de tabelas, como aqueles armazenados em bancos de dados relacionais. Define-se também os *dados semi-estruturados*, que são aqueles que possuem uma estrutura menos rígida, como arquivos XML e JSON.

A mudança de paradigmas apresentadas pelos dados não estruturados diz respeito a como deve ser feita a interpretação desses dados. Diferentes imagens de cachorros são objetivamente diferentes analisando o seu conteúdo bruto (valor dos pixels), mas todas as imagens apresentam uma *similaridade semântica*. O desafio passa a ser então como representar, armazenar e realizar busca nesses dados, de forma que eles reflitam esse aspecto.

A abordagem que segue é a de representar os dados por meio de métodos de *aprendizado profundo*, gerando a partir de um pedaço de dado não estruturado um vetor multidimensional, chamado de *embedding*. Observa-se na literatura que uma rede neural bem treinada é capaz de capturar a semântica desses dados e, portanto, a similaridade entre eles é refletida por uma medida de distância entre os vetores.

3.1.1 Aplicações

...

3.2 Divisão dos métodos

- Busca exata

  - Linear Scan
  - Space Partitioning (KD-Tree, Ball-Tree, Inverted File Index)
- Busca aproximada
  - Cluster-based
  - Graph-based
  - Tree-based
  - Hash-based
- Quantização
  - Scalar Quantization
  - Product Quantization

3.3 Métodos exatos

3.3.1 Linear Scan

O algoritmo de busca linear, *flat indexing*, calcula a distância entre a query Q e todos os elementos do conjunto de dados $\mathcal{D}$, retornando o elemento mais próximo. A complexidade desse algoritmo é $O(n)$, onde n é o número de elementos em $\mathcal{D}$, tornando-o muitas vezes impraticável em cenários reais. O algoritmo é apresentado no algorithm 3.

Algorithm 3 Algoritmo de busca linear de uma query Q em um conjunto de dados $\mathcal{D}$.

```
1:  procedure LinearScan(Q, D)
2:      best_dist ← ∞
3:      best_match ← None
4:      for d ∈ D do
5:          dist ← dist(Q, d)
6:          if dist < best_dist then
7:              best_dist ← dist
8:              best_match ← d
9:          end if
10:     end for
11:     return best_match
12: end procedure
```

3.3.2 Particionamento de Espaço

Métodos exatos de busca podem ser vistos em section 1.3.

O que acontece nesses caso é a maldição da dimensionalidade. Sendo assim, métodos de busca aproximados devem ser utilizados, trocando um pouco de precisão por eficiência.

3.4 Métodos baseados em clusterização

3.4.1 Inverted File Index (IVF)

A estratégia de IVF é baseada em dividir o espaço em partições, representadas a partir do centroide de todos os vetores que pertencem a essa partição.

Os centroides de cada partição, ou *clusters*, são determinados pelo algoritmo de clusterização *k-means*. O funcionamento desse algoritmo consiste em: inicialmente selecionar aleatoriamente k vetores do conjunto de dados e atribuir a cada um deles um cluster; em seguida adicionar a cada cluster os vetores mais próximos a ele; atualizar o novo centroide de cada cluster; repetir o processo até a convergência [1], [2].

A busca, dessa forma, pode ser limitada somente aos vetores que pertencem a uma partição ou conjunto de partições, reduzindo o espaço de busca, como ilustra algorithm 4.

3.5 Métodos baseados em grafos

3.5.1 Navigable Small World

[3]

3.5.2 Hierarchical Navigable Small World (HNSW)

Skip List + Navigable Small World

Algorithm 4 Algoritmo de busca usando o IVF de uma query Q em um índice $\mathcal{I}$.

procedure IVFSEARCH($\mathcal{I}, query, top\_k, nprobe$)
    $distances \leftarrow []$
    for all $c \in \mathcal{I}.centroids$ **do**
        $d \leftarrow \|query - c\|$
        $distances \leftarrow distances \cup \{(c, d)\}$
    end for
    $candidateCentroids \leftarrow$ **selecionar os** $nprobe$ **menores** d
    $candidateVectors \leftarrow \{\}$
    for all $ctr \in candidateCentroids$ **do**
        $candidateVectors \leftarrow candidateVectors \cup \mathcal{I}[ctr]$
    end for
    $scored \leftarrow []$
    for all $v \in candidateVectors$ **do**
        $dist \leftarrow \|query - v\|$
        $scored \leftarrow scored \cup \{(v, dist)\}$
    end for
    ordenar $scored$ **por** $dist$ **ascendente**
    return primeiros top_k **de** $scored$
end procedure

3.6 Métodos baseados em árvores

3.6.1 Approximate Nearest Neighbors Oh Yeah (ANNOY)

3.7 Métodos baseados em hashing

3.7.1 LSH

3.8 Quantização

A quantização busca diminuir o tamanho do banco de dados, representando os vetores por suas representações quantizadas. Note que isso é diferente do método de redução de dimensionalidade, como (PCA, t-SNE, UMAP).

3.8.1 Scalar Quantization

Transforma vetores de floats em vetores de inteiros. Para cada dimensão, o método busca todo o alcance e divide essa faixa uniformemente em bins. Se queremos armazenar o inteiro em um `uint8`, então temos $2^8 = 256$ bins.

3.8.2 Product Quantization

Dado um vetor que possui d bits. Cada vetor é dividido em m subvetores, cada um com d/m bits. Em seguida, para todos os subvetores, é feita uma clusterização por k-means. Substituímos todos os subvetores por seus respectivos centroides.

3.9 Opções comerciais

BIBLIOGRAPHY

[1] Wikipedia, *k-means clustering*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering.

[2] Wikipedia, *Lloyd's algorithm*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Lloyd%27s_algorithm.

[3] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.