CENTRO UNIVERSITÁRIO UNIFECAF

GRADUAÇÃO EAD

DATA STRUCTURE STRATEGY AND IMPLEMENTATION

PROFESSOR ROBINSON LEMOS BEZERRA

TUTOR PROFESSOR MARCEL TEIXEIRA

Implementação de um Sistema de Recomendação de Produtos Utilizando Grafos

William Gama Kerpen

RA 2903

Preetz

2025

Implementação de um Sistema de Recomendação de Produtos Utilizando Grafos

Este trabalho faz parte da avaliação da disciplina de Redes de Computadores, do curso de graduação em Analise e Desenvolvimento de Sistemas, ministrado pelo Centro Universitário UniFecaf. Sob orientação do Professor Conteudista Afonso Brandão e Tutor Professor Fernando Leonid.

William Gama Kerpen

RA 2903

Preetz

2025

SUMÁRIO

1. Introdução...............................................................................................................................1

2. Objetivos.................................................................................................................................2

3. Apresentação do Projeto.........................................................................................................3

3.1. Apresentação do Projeto em Diagrama de rede.......................................................3

3.2. Apresentação do Projeto em Vídeo..........................................................................4

4. Conclusão................................................................................................................................5

**1.Introdução**

A crescente demanda por eficiência em plataformas de e-commerce tem evidenciado a importância de sistemas de busca rápidos e precisos. No caso passado a MegaStore, empresa fictícia de grande porte, enfrenta desafios significativos relacionados à lentidão e imprecisão de seu sistema de busca, comprometendo a experiência do usuário e a conversão de vendas.

Na vida real empresas como Mercado Livre possui mais de 60 Milhões de Ofertas, a Amazon 100 milhões de produtos, Shopee e Magazine Luiza não divulgam a quantidade de produtos, mas são milhões de vendedores que resultam em números tão grandes quanto a de seus concorrentes.

Com uma quantidade tão grande de dados é essencial para as empresas buscarem por algoritmos que possibilitem o rapidez na busca para seus usuários.

Este trabalho tem como objetivo explorar as causas do problema, os impactos sobre os usuários, propor soluções viáveis e discutir a implementação de estruturas de dados eficientes em Rust como alternativa de otimização para as buscas.

Este trabalho não tem foco em segurança e códigos dele não são seguros, não aconselho a utilização dos mesmos sem a devida revisão para utilização em projetos reais.

**2. Causas da Lentidão e Imprecisão**

As principais causas da lentidão e imprecisão do sistema de busca da MegaStore estão relacionadas ao uso de estruturas de dados inadequadas, algoritmos de busca não otimizados e à ausência de indexação eficiente. Sistemas que operam com listas lineares ou bancos de dados relacionais sem mecanismos de cache ou pré-processamento tendem a apresentar alta latência e baixa relevância nos resultados (MANNINO; RAHMAN, 2015). Além disso, a falta de personalização e de algoritmos de recomendação contribui para a imprecisão das buscas.

**3. Impacto nos Usuários**

Os principais afetados pela baixa performance do sistema de busca são os consumidores e os administradores da plataforma. Os consumidores enfrentam frustrações ao não encontrar produtos desejados, o que pode levá-los a abandonar o site e buscar alternativas em concorrentes. Já os administradores observam queda nas taxas de conversão, aumento no custo de aquisição de clientes e perda de competitividade (LAUDON; TRAVER, 2021).

**4. Soluções Possíveis**

Entre as soluções possíveis para otimizar o sistema de busca da MegaStore, destacam-se:

* **Implementação de tabelas hash**: permite acesso rápido a dados indexados por chave.
  + *Prós*: alta velocidade de busca; baixo consumo de memória.
  + *Contras*: complexidade na resolução de colisões; dificuldade em buscas por aproximação.
* **Uso de grafos para recomendação**: modela relações entre produtos e usuários.
  + *Prós*: permite sugestões personalizadas; melhora a relevância dos resultados.
  + *Contras*: maior complexidade computacional; necessidade de algoritmos especializados.
* **Tries (árvores de prefixo)**: otimizam buscas por prefixo, como autocompletar.
  + *Prós*: excelente desempenho em buscas textuais; útil para grandes volumes de dados.
  + *Contras*: alto consumo de memória; difícil manutenção.

**4. Barreiras à Implementação**

A implementação dessas soluções enfrenta barreiras como a necessidade de reestruturação do backend, capacitação da equipe técnica em Rust, e integração com sistemas legados. Além disso, a migração de dados e a adaptação dos algoritmos existentes exigem planejamento cuidadoso e testes extensivos (SEDGEWICK; WAYNE, 2011).

**5. Contribuição das Tabelas Hash em Rust**

Rust é uma linguagem de programação moderna que oferece segurança de memória e alto desempenho. A utilização de tabelas hash em Rust pode reduzir significativamente o tempo de busca por produtos, graças à eficiência da estrutura HashMap, que permite acesso constante (O(1)) a elementos indexados (KLABNIK; NICHOLS, 2018). Além disso, Rust evita erros comuns como vazamentos de memória e condições de corrida, tornando o sistema mais robusto.

**6. Requisitos de Hardware e Software**

Para implementar a solução proposta, são necessários servidores com boa capacidade de processamento paralelo (multi-core), memória RAM suficiente para armazenar as estruturas de dados em tempo real e SSDs para acesso rápido ao banco de dados. No aspecto de software, é necessário o compilador Rust, bibliotecas como serde para serialização de dados, e ferramentas de integração contínua como GitHub Actions ou Jenkins.

**7. Integração com a Plataforma**

A solução pode ser integrada à plataforma de e-commerce da MegaStore por meio de APIs RESTful escritas em Rust, que se comunicam com o frontend e o banco de dados. Essa abordagem modular permite que o novo sistema de busca funcione como um microserviço, facilitando a escalabilidade e a manutenção (NEWMAN, 2015).

**8. Indicadores de Desempenho**

Os principais indicadores de desempenho para avaliar a eficácia da solução incluem:

* Tempo médio de resposta das buscas.
* Taxa de conversão de usuários que utilizam a busca.
* Precisão e recall dos resultados apresentados.
* Uso de CPU e memória durante picos de acesso.
* Tempo de uptime e número de falhas.

**9. Implicações a Longo Prazo**

Caso o problema do sistema de busca não seja resolvido, a MegaStore poderá enfrentar queda progressiva na satisfação dos clientes, aumento da taxa de abandono do carrinho, e perda de market share. Além disso, a reputação da marca pode ser prejudicada, dificultando parcerias comerciais e expansão internacional (KOTLER; KELLER, 2012).

**10. Sistema de Buscas**

Para o caso fictício criei uma página de buscas simulando a página inicial de um site de compras online, para isso foi utilizado a linguagem Python com o framework Flask, foi utilizado também JavaScript, HTML, CSS com Bootstrap.

O banco de dados foi carregado aproximadamente 2 Mil produtos fictícios com a biblioteca Faker e o uso de IA para criação de listas com nomes reais.

Com isso o site já estava funcionado:

Ein Bild, das Text, Screenshot, Website, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Kleidung, Website enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Kleidung, Text, Screenshot, Collage enthält.

KI-generierte Inhalte können fehlerhaft sein.

Porém as recomendações e as buscas não estavam otimizadas, exemplo ao buscar a palavra calca, sem o ç, não aparece resultado algum.

Ein Bild, das Text, Screenshot, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

**10. Analise de Complexidade Big O**

A análise de complexidade Big O é uma ferramenta fundamental na ciência da computação utilizada para descrever o comportamento assintótico de algoritmos, ou seja, como o tempo de execução ou o uso de memória de um algoritmo cresce à medida que o tamanho da entrada aumenta. Essa notação permite classificar algoritmos segundo sua eficiência, considerando principalmente o pior caso, o que é essencial para prever o desempenho em situações com grandes volumes de dados. A notação Big O expressa a complexidade em termos matemáticos, destacando as funções que dominam o crescimento do custo computacional, como O(1) para tempo constante, O(n) para tempo linear, O(n log n) para algoritmos eficientes de ordenação, até complexidades mais elevadas como O(n²), O(2ⁿ) e O(n!), que indicam crescimento rápido e, muitas vezes, inviabilidade prática para entradas grandes. Além do tempo, a notação também pode ser aplicada para medir a complexidade espacial, ou seja, a quantidade de memória utilizada pelo algoritmo. Compreender a análise Big O é crucial para a escolha e otimização de algoritmos, garantindo que sistemas sejam eficientes e escaláveis. Essa análise é amplamente adotada em cursos e práticas de engenharia de software para garantir a qualidade e a performance do código desenvolvido (camillacorreia, s.d.; freeCodeCamp.org, 2021).

Ao realizarmos a analise de complexidade no algoritmo atual temos então:

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

No arquivo views.py, a função index() realiza uma busca condicional no banco de dados. Quando há um termo de pesquisa, o algoritmo executa um filtro com cláusulas ilike, que são operações de busca textual. Essas buscas, dependendo da indexação do banco, podem ter complexidade **O(n)**, onde *n* representa o número de registros na tabela Produtos. Caso não haja termo, o sistema realiza uma ordenação por data de criação e limita o resultado a oito produtos, o que, em termos de complexidade, pode ser considerado **O(log n)** para a ordenação (dependendo do algoritmo interno do banco) e **O(1)** para a limitação da quantidade de resultados.

Na função produto\_details(id), o método get\_or\_404 realiza uma busca por chave primária, que é indexada e, portanto, possui complexidade **O(1)**. A busca pela imagem principal (filter\_by(is\_primary=True).first()) depende da quantidade de imagens associadas ao produto, o que pode ser considerado **O(m)**, sendo *m* o número de imagens vinculadas ao produto específico. A consulta de produtos recomendados exclui o produto atual e ordena os demais por data de criação, limitando a quatro resultados. Essa operação, similar à anterior, pode ser considerada **O(log n)** para ordenação e **O(1)** para a limitação.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

No modelo models.py, as classes Produtos e ProdutoImagem representam estruturas de dados persistentes. A complexidade associada a essas classes está mais relacionada às operações que as utilizam, como inserções, atualizações e consultas. Como o projeto utiliza SQLAlchemy, muitas dessas operações são otimizadas por meio de índices e relacionamentos, o que contribui para uma complexidade eficiente na maioria dos casos.

Portanto, de forma geral, o sistema apresentado possui complexidade predominantemente linear (O(n)) nas buscas textuais e constante (O(1)) nas operações por chave primária e nas limitações de resultados. Essa análise é válida sob a suposição de que o banco de dados está devidamente indexado e que as operações são realizadas sobre volumes moderados de dados, porém com o crescimento da base de dados para um volume real com cerca de milhões de produtos cadastrados, as buscas apresentariam produtos indesejados ou lentidão do banco.

**11. Melhoramento com RUST**

Melhorar a busca e os resultados de recomendação para clientes utilizando **Rust** com tabela hash ou árvore binária envolve aplicar estruturas de dados eficientes para acelerar o acesso e a análise de padrões de consumo. Rust, por ser uma linguagem de baixo nível com alto desempenho e segurança de memória, é ideal para sistemas de recomendação que exigem rapidez e escalabilidade.

Atualmente já está sendo utilizado o consultas SQL com cláusulas LIKE, WHERE, e LIMIT, que são processadas pelo SQLite.

O SQLite pode usar índices baseados em B-tree para acelerar essas consultas, mas isso é feito automaticamente.

Embora o SQLite já organize os dados internamente com B-trees, podemos usar estruturas explícitas no RUST para:

* Acelerar buscas repetidas (cache)
* Organizar dados em memória para lógica mais complexa
* Evitar múltiplas consultas ao banco
* Criar recomendações dinâmicas com base em padrões

**11.1. Tabela hash (HashMap)**

A utilização de estruturas de dados como tabelas hash e árvores binárias é amplamente reconhecida como uma estratégia eficiente para otimizar sistemas de recomendação, especialmente em aplicações que exigem alto desempenho e escalabilidade. A tabela hash, também conhecida como HashMap, é uma estrutura que permite acesso direto aos dados por meio de chaves únicas, com complexidade média de tempo de acesso constante, ou seja, O(1). Essa característica torna a tabela hash ideal para armazenar relações de coocorrência entre produtos, permitindo que o sistema recomende rapidamente itens frequentemente comprados em conjunto. Por exemplo, ao registrar que o produto A foi comprado com o produto B três vezes, o sistema pode sugerir B sempre que A for consultado, utilizando uma estrutura como {"produtoA": {"produtoB": 3}}.

Isso permite:

* Buscar rapidamente produtos com nome semelhante
* Criar um cache para evitar consultas repetidas
* Agrupar produtos por categoria ou popularidade

**11.2 Arvore binária (ex: BTreeMap)**

Por outro lado, a árvore binária de busca, especialmente em sua forma balanceada como a B+ Tree, oferece uma complexidade de acesso logarítmica, O(log n), e é particularmente útil quando se deseja manter os dados ordenados ou realizar buscas por intervalo. Essa estrutura é vantajosa em cenários onde a recomendação depende de atributos como faixa de preço, popularidade ou data de compra, permitindo uma navegação eficiente entre os nós da árvore. Embora o acesso seja mais lento que na tabela hash, a árvore binária proporciona maior flexibilidade na organização dos dados e na execução de consultas complexas.

A escolha entre essas estruturas depende dos requisitos específicos do sistema. Em geral, a tabela hash é preferida para consultas rápidas e diretas, enquanto a árvore binária é mais adequada para operações que exigem ordenação ou busca por intervalo. Ambas podem ser implementadas com alto desempenho na linguagem Rust, que oferece segurança de memória, concorrência eficiente e velocidade comparável ao C++, sendo ideal para aplicações críticas como sistemas de recomendação em tempo real (UlissesZorzan, 2023)

Por esse motivo priorizei a utilização da tabela hash, a recomendação dos produtos poderiam ser por: produtos comprados juntos, ou por popularidade, como não temos essas funcionalidades no site ainda, optei por fazer a a recomendação deve ser feita pelo nome do produto, assim quando a pessoa procurar por saia, a recomendação seja outras saias, na ausência de produtos com nomes similares deve aparecer produtos da mesma categoria.

Isso permite:

* Buscar produtos dentro de uma faixa de preço
* Navegar ordenadamente por produtos
* Criar recomendações “do mais barato ao mais caro” ou vice-versa

**11.3 Implementação do código**

Definir a estrutura Produto

Ein Bild, das Text, Schrift, Screenshot, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Carregar todos os produtos do banco

Ein Bild, das Text, Screenshot, Display, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Buscar recomendação por nome

Ein Bild, das Text, Screenshot, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Recomendação por categoria

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Integração na main

Ein Bild, das Text, Screenshot, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

**12. Referências**

KLABNIK, S.; NICHOLS, C. *The Rust Programming Language*. No Starch Press, 2018.

KOTLER, P.; KELLER, K. L. *Administração de Marketing*. 14. ed. São Paulo: Pearson, 2012.

LAUDON, K. C.; TRAVER, C. G. *E-commerce 2021: business, technology, society*. Pearson, 2021.

MANNINO, M.; RAHMAN, M. *Database Design, Application Development, and Administration*. McGraw-Hill, 2015.

NEWMAN, S. *Building Microservices*. O'Reilly Media, 2015.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Addison-Wesley, 2011.

CAMILLA CORREIA. Análise de complexidade de algoritmos - Big O. Disponível em: GitHub - camillacorreia/big-o-notation. Acesso em: 18 set. 2025.

freeCodeCamp.org. . O que é a notação Big O: complexidade de tempo e de espaço. 15 dez. 2021. Disponível em: freeCodeCamp.org artigo sobre notação Big O. Acesso em: 18 set. 2025.