Trabalho Prático 1 - Algoritmos 2

Felipe Araujo Melo - 2023027947

Bernardo Borges Machado - 202311111

---

1. Introdução

Este relatório apresenta a concepção e implementação de um sistema de Recuperação de Informação (RI) projetado para indexar coleções de documentos textuais e processar consultas complexas. O sistema é fundamentado em dois módulos principais, um indexador e um recuperador. O módulo de indexação é responsável por processar o corpus, tokenizar o conteúdo, e construir o índice invertido, que é armazenado de forma eficiente em uma Árvore Trie Compacta. Concomitantemente à indexação, o sistema calcula estatísticas globais (média μ, desvio padrão σ, e frequência documental df) para cada termo. O módulo de recuperação, por sua vez, implementa a lógica de busca, suportando consultas booleanas completas (AND, OR, parênteses) através da conversão para Notação Polonesa Reversa (RPN). Após a filtragem booleana, os documentos resultantes são ordenados por relevância utilizando um modelo de Z-score, que calcula a média das pontuações Z dos termos da consulta para cada documento, entregando assim uma lista de resultados ordenada por relevância.

---

2. Como Executar o Projeto

2.1. Dependências

Para garantir a praticidade na avaliação, toda a lógica de execução foi unificada no servidor web. O projeto utiliza apenas uma biblioteca externa principal, o Flask.

Todas as demais bibliotecas (`os`, `re`, `json`, `math`, `string`, `collections.deque`) são parte da biblioteca padrão do Python.

Para instalar a dependência, execute o seguinte comando no terminal:

```
pip install Flask
```

2.2. Execução Unificada

Para rodar o projeto, basta executar um único comando no terminal, a partir da pasta raiz do projeto:

```
python app.py
```

Nós implementamos uma lógica de execução automática, assim, ao ser iniciado, o `app.py` verifica automaticamente se os arquivos de índice (`inverted_index.txt`, `global_stats.json`, `doc_id_map.json`) existem. Se existirem, o servidor Flask é iniciado imediatamente, carregando os dados do índice. Se não existirem (como na primeira execução), o `app.py` automaticamente chama o módulo `indexer.py` para executar o processo de indexação completo. O terminal exibirá o progresso da indexação e, ao final, o servidor Flask será iniciado e a aplicação estará pronta para uso.

2.3. Arquivo de entrada

Na nossa implementação, definimos o caminho para a pasta bbc, ou seja, o lugar onde os documentos estarão disponíveis dentro do código app.py na variável global

CORPUS_PATH e definimos ela como “bbc”. Sendo assim, para o código funcionar, deve ser adicionada a pasta de nome “bbc” contendo dentro dela as 5 pastas com os documentos no mesmo repositório do código.

---

3. Arquitetura da Solução

O sistema foi dividido em quatro módulos principais em Python e três templates HTML. Cada um deles possui uma função específica, a qual será abordada a seguir e mais bem detalhada na sessão de implementação:

- **`indexer.py`:** Atua como o construtor do índice. É responsável por ler o corpus, processar (tokenizar) os textos, construir a Trie e calcular as estatísticas de Z-score. Seu trabalho é criar os três arquivos de índice no disco.
- **`compact_trie.py`:** Define as classes `TrieNode` e `CompactTrie`. Contém toda a lógica de inserção, busca e, crucialmente, a persistência (serialização/desserialização) da árvore em um formato próprio.
- **`RI.py`:** Define a classe `InformationRetriever` e atua como o cérebro da busca. Ele carrega a Trie e as estatísticas, recebe a string de consulta, processa-a (Shunting-Yard) e retorna a lista final de IDs de documentos, ordenados por relevância (Média de Z-scores).
- **`app.py`:** Utiliza Flask para servir a aplicação. Gerencia as rotas, lida com as requisições do usuário, chama o `RI.py` para obter os resultados e, em seguida, aplica a lógica de apresentação (extração de título, geração de snippet, filtros de relevância e paginação) antes de enviar os dados para os templates.

---

4. Implementação Detalhada e Decisões de Design

Essa seção detalha as principais decisões de implementação tomadas durante o desenvolvimento de cada módulo (que tiveram suas funções introduzidas na seção anterior)

4.1. Módulo de Indexação e Tokenização (`indexer.py`)

A decisão mais crítica no módulo de indexação foi a sobre como definiríamos exatamente uma palavra. Após testes, observamos que a regex padrão "`[a-z]+`" era insuficiente, pois quebrava termos comuns e importantes. Adotamos a expressão regular "`re.findall(r'[a-z0-9&-]+', text)`". Essa regra permite que letras, números, o caractere "`&`" e o hífen façam parte de um token. Isso resolveu um problema de inconsistência onde a busca por "R&B" falhava, pois o indexador antigo (com `[a-z]+`) salvava "r" e "b" como palavras separadas. Com a nova regra, "`r&b`, `sci-fi` e `mp3`" são tratados como termos únicos. Essa abordagem se mostrou mais robusta porque, em vez de tentarmos limpar a pontuação indesejada do texto, nós definimos exatamente quais caracteres são permitidos *dentro* de uma palavra. Isso evitou a quebra de termos importantes e garantiu que o índice refletisse as palavras como o usuário esperaria buscá-las.

4.2. Estrutura da Trie e Persistência (`compact_trie.py`)

Para atender ao requisito de armazenamento em "formato próprio" e evitar `pickle` ou serializadores de objetos, implementamos uma lógica de persistência personalizada para a `CompactTrie`. O `TrieNode` armazena o `label` (prefixo), o dicionário `children` e o `inverted_index` (lista de tuplas `(DocID, Frequência)`). Crucialmente, ele armazena o booleano `is_terminal`, que marca o fim de uma palavra válida, permitindo à árvore distinguir prefixos (ex: "th") de palavras reais (ex: "the").

Para o "Formato Próprio" (`inverted_index.txt`), criamos um formato de texto onde cada linha representa um nó da árvore. As informações são salvas usando um

separador `|`, no formato:
“`label|is_terminal(1/0)|num_children|inverted_index_string`”. A serialização é feita com uma travessia recursiva em pré-ordem (`pre_order_serialize`). A desserialização (`load_from_file`) lê o arquivo linha por linha e usa uma pilha para reconstruir a hierarquia da árvore, usando o `num_children` de cada nó para saber quantos filhos deve esperar antes de retornar ao pai. Para os arquivos auxiliares (`doc_id_map.json` e `global_stats.json`), que armazenam dicionários simples, optamos pelo formato JSON. Esta decisão foi tomada pois a restrição principal do formato próprio foi aplicada à nossa estrutura de dados complexa (a Trie), e o JSON foi escolhido para os dados auxiliares por considerarmos um padrão legível e eficiente.

4.3. Módulo de Recuperação e Ranking (`RI.py`)

O `RI.py` implementa o algoritmo **Shunting-Yard** (`_to_rpn`) para converter a consulta infixa do usuário (com `AND`, `OR` e `()`) em Notação Polonesa Reversa (RPN). A expressão RPN é então avaliada (`_evaluate_rpn`) usando uma pilha, onde os operadores lógicos executam operações de interseção (`AND`) e união (`OR`) nos conjuntos de DocIDs.

Conforme solicitado pelo enunciado, o cálculo de ranking é feito pela média simples dos z-scores. A função `_rank_results` soma o z-score de cada termo da consulta encontrado no documento e divide pelo número de termos encontrados (`total_z_score / term_count`). É importante salientar que nossa fórmula usada para calcular o Z-score foi simples e usa a frequência absoluta da palavra no documento, não considerando o tamanho do documento para fazer uma espécie de frequência ponderada (quantas vezes a palavra aparece com base em quantas palavras tem naquele documento). Acreditamos que a abordagem mais simples de frequência absoluta funcionou bem para os documentos que estávamos trabalhando e que conseguiu organizar eles de uma forma adequada.

4.4. Aplicação Web e Interface (app.py)

A maior parte da lógica de qualidade e usabilidade reside no `app.py`. A relevância de um resultado não depende apenas de o termo existir (o que é garantido pelo `RI.py`), mas de ele ser apresentado de forma correta e útil.

Para isso, implementamos um sistema de validação em duas etapas. Primeiro, o `RI.py` (usando a CompactTrie) realiza a busca exata e retorna a lista de documentos que contêm os termos da consulta (ex: a busca por "d" retorna apenas documentos que contêm a palavra "d"). Segundo, o `app.py` aplica uma camada de validação contextual ao tentar gerar o snippet.

O desafio é que uma busca de texto simples para *destacar* a palavra "d" no documento poderia, por engano, destacar o "d" dentro de outra palavra (como em "media"). Para evitar isso, a função `generate_snippet` implementa uma verificação de "fronteira de palavra" (conhecida como *word boundary*).

Nossa implementação usa uma expressão regular que ancora o termo de busca. Por exemplo, ao buscar "d", o padrão se torna `\bd\b`. Isso garante que a busca encontre "d" como uma palavra isolada, mas ignore corretamente o "d" dentro de "media".

Essa abordagem também resolve o problema da pontuação. Como o `indexer.py` salva o termo 'palavra' (sem o ponto final, graças à regex de indexação), e o `app.py` busca por `\bpalavra\b`, a busca funciona perfeitamente mesmo se a palavra estiver no final de uma frase (como "...uma palavra." ou "...outra palavra!"). Isso acontece porque a pontuação (como '.', '!', '?') age como uma "fronteira de palavra" natural, validando a correspondência sem que a pontuação seja incluída no termo destacado. Para garantir a robustez, caso essa busca falhe, o sistema ainda tenta uma verificação manual, inspecionando se os caracteres ao redor do termo são espaços ou pontuações.

Se, mesmo com essas verificações, a `generate_snippet` não encontrar nenhuma ocorrência válida, ela retorna `None`. A rota `search` no `app.py` então veta esse

documento da lista final, garantindo que apenas resultados relevantes sejam exibidos.

Essa verificação de "fronteira de palavra" foi então refinada para lidar com exceções de apóstrofo. Para refinar buscas de letras únicas, implementamos regras específicas: a busca por "t" ignora contrações de negação (como "don't"). A busca por "d" é mais complexa: ela é refinada para barrar nomes (como "D'Arcy", onde o apóstrofo vem depois), mas permitir contrações semanticamente válidas (como "they'd", onde o apóstrofo vem *antes*). A ideia geral foi tentar permitir a ocorrência de letras únicas quando elas representassem uma palavra abreviada.

Para a geração do snippet e título, identificamos que a primeira linha de cada documento .txt era o seu título. A função `generate_snippet` lê o título e o conteúdo separadamente, mas realiza a busca pela "palavra isolada" no conteúdo completo (título + resto) para que os termos no título também sejam encontrados.

O snippet em si não é centrado na primeira ocorrência, mas sim no termo mais relevante (aquele com o maior z-score) naquele documento (para o caso de uma consulta que use `AND`, na qual teremos as duas palavras no documento). Além disso, a lógica prioriza encontrar uma ocorrência com 80 caracteres de contexto prévio e 80 caracteres depois, mas caso não consiga fazer isso , no caso de só haver uma palavra no início ou no final, ele considera essa palavra como válida e retorna ela com quantos caracteres tiver antes ou depois.

4.5. Interatividade e Design da Interface

Para a interatividade, os títulos nos resultados são clicáveis, abrindo uma nova aba que leva a uma nova rota, exibindo o documento completo em uma página dedicada, com formatação e design próprios. Também implementamos um componente de paginação avançada que, além dos botões "Anterior" e "Próxima", exibe os números das páginas para facilitar a navegação em grandes volumes de resultados.

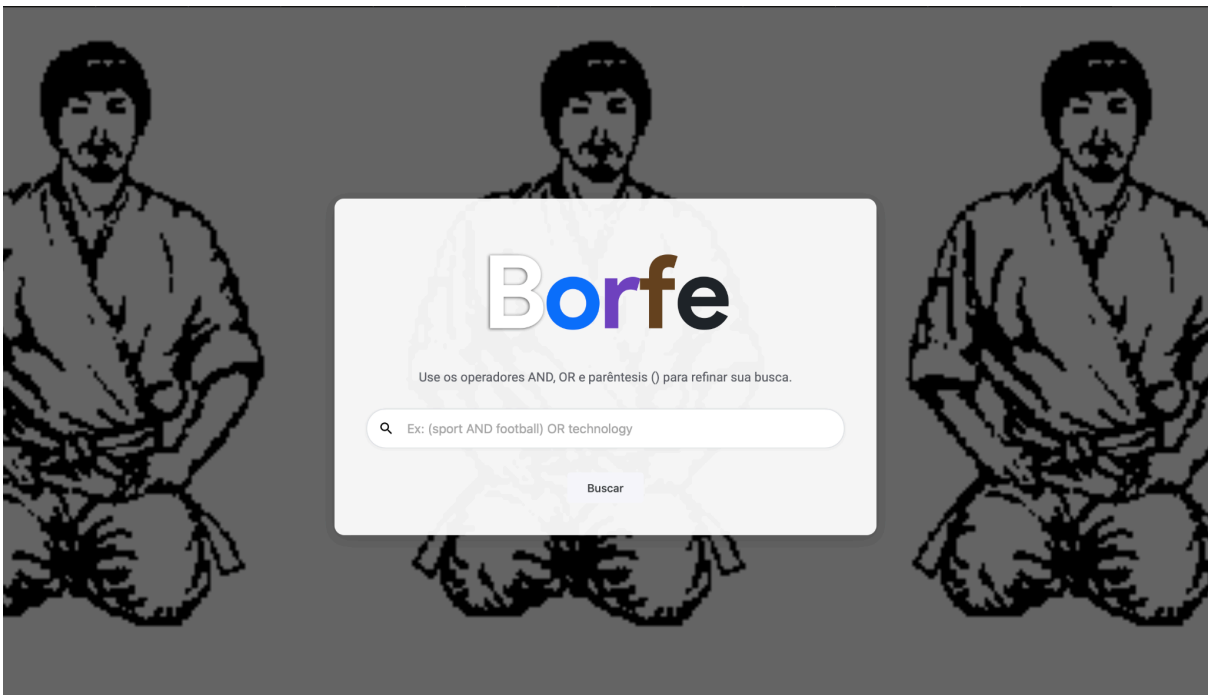
---

5. Exemplos de Funcionamento

Abaixo estão os screenshots do sistema final em operação, ilustrando tanto o fluxo de uso quanto os mecanismos específicos de busca implementados.

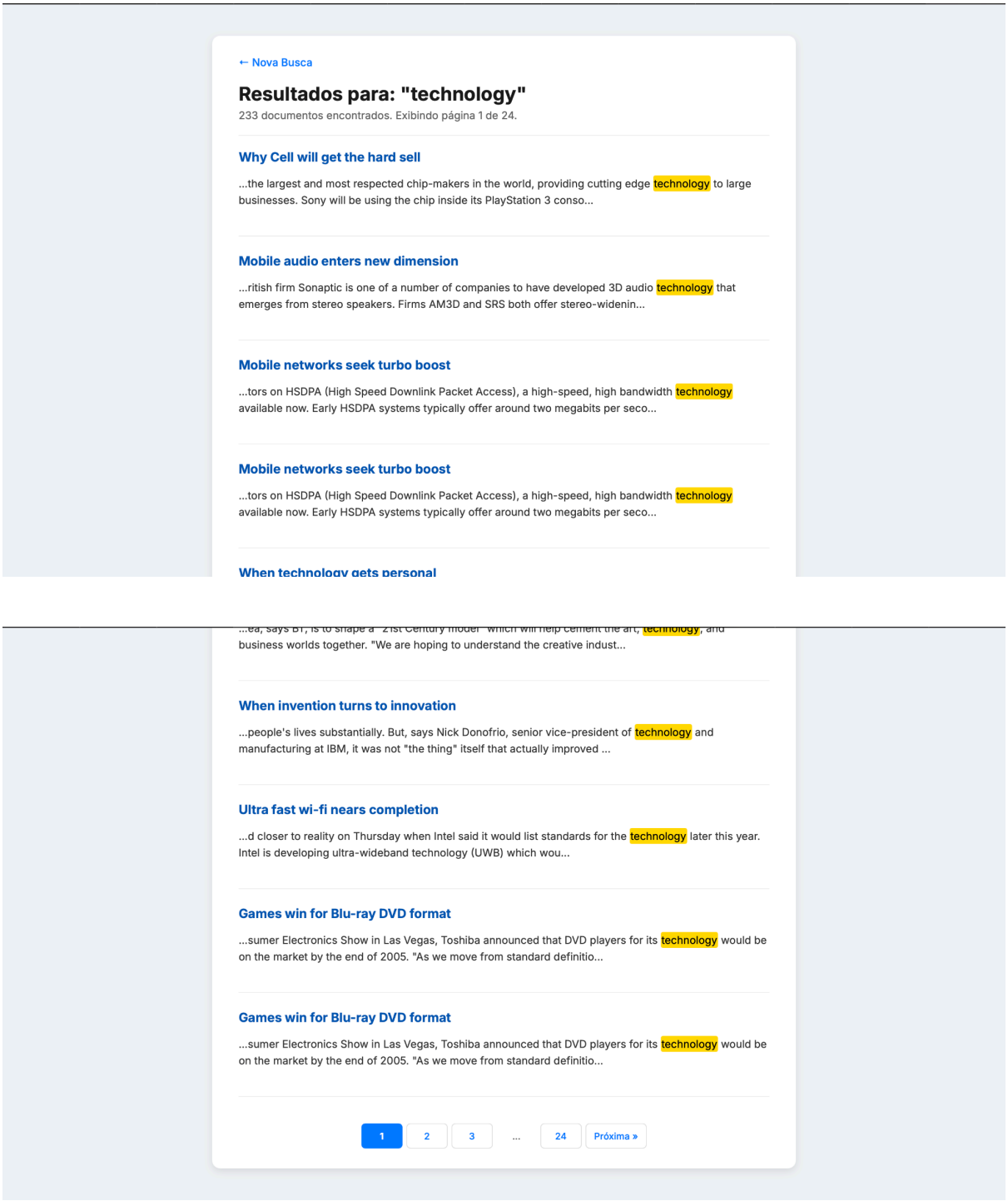
5.1. Página Inicial "Borfe"

Falando sobre o design, a identidade visual foi uma escolha pessoal. Uma vez que os dois autores do trabalhos são praticantes de artes marciais (judô e jiu-jitsu), optamos por chamar o nosso buscador de **"Borfe"**, vindo de **Bor**ges e **Fe**lipe. Usamos como cor para os caracteres do título, as cores das faixas do jiu-jitsu (branca, azul, roxa, marrom e preta) em ordem da esquerda para direita, além do gif de saudação japonesa como plano de fundo para a página principal.

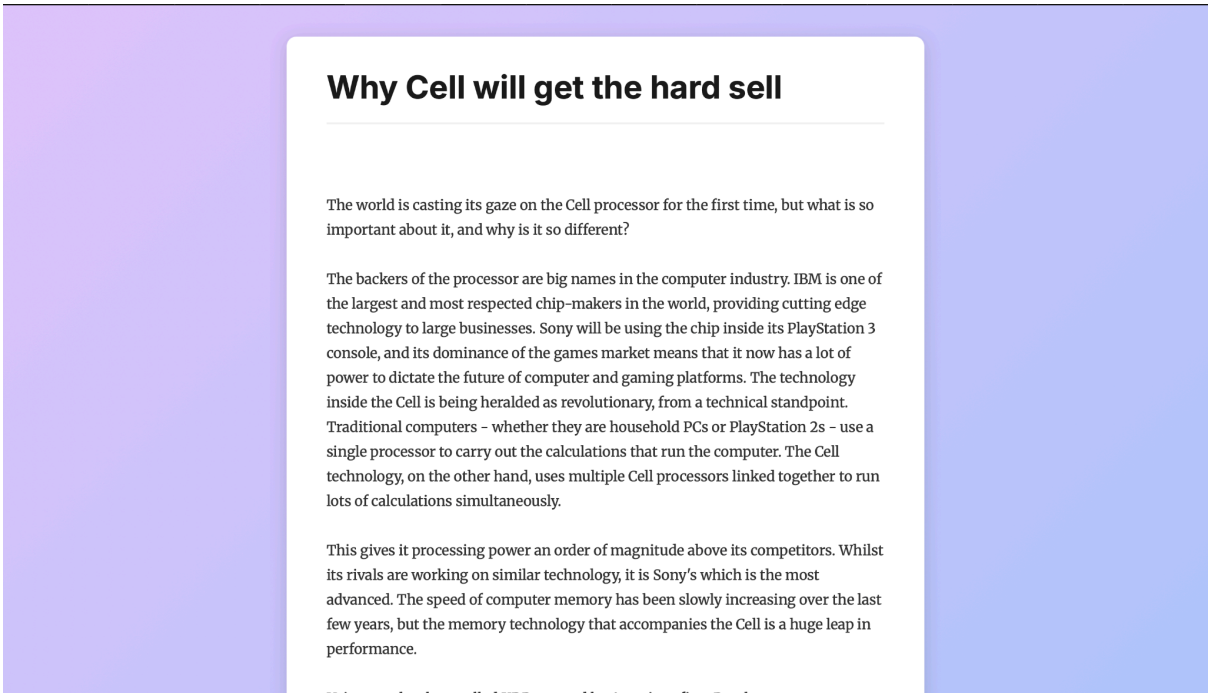


5.2. Página de Resultados

A página de resultados exibe os títulos clicáveis (que abrem em nova aba), o snippet com o termo mais relevante destacado (calculado pelo z-score) e o componente de paginação avançada na parte inferior.

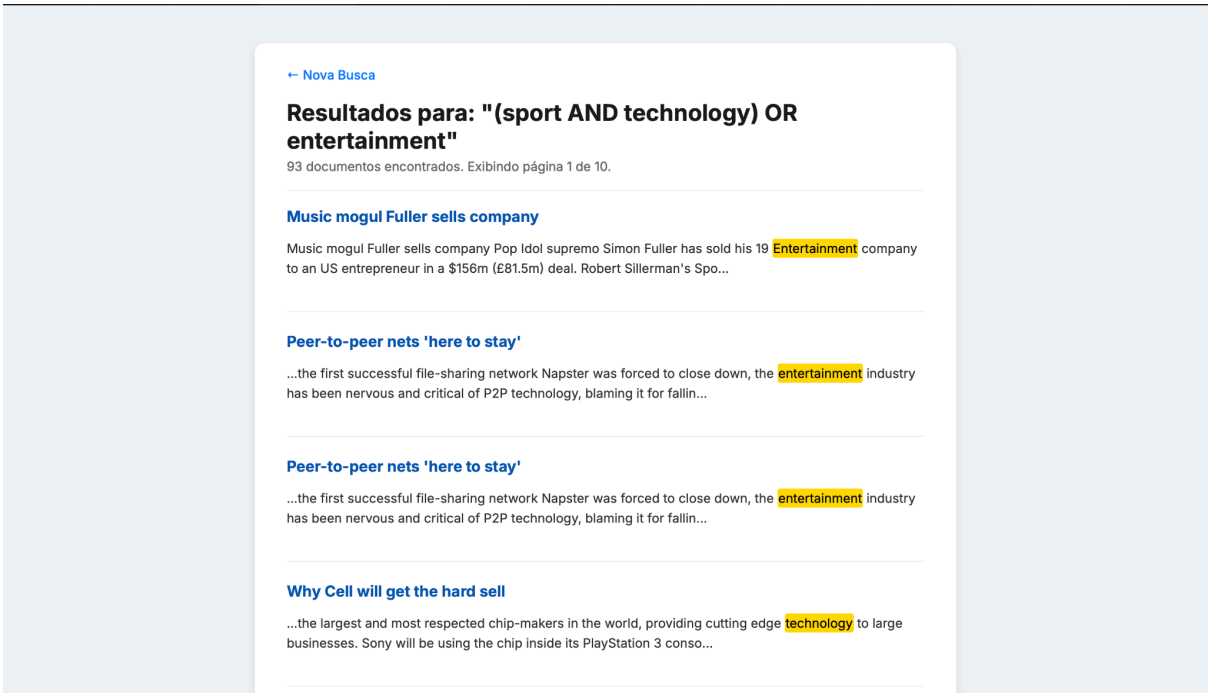


5.3. Página de Leitura do Documento Ao clicar em um título, o usuário é levado a uma nova aba, a qual renderiza o documento completo com uma formatação otimizada para leitura.



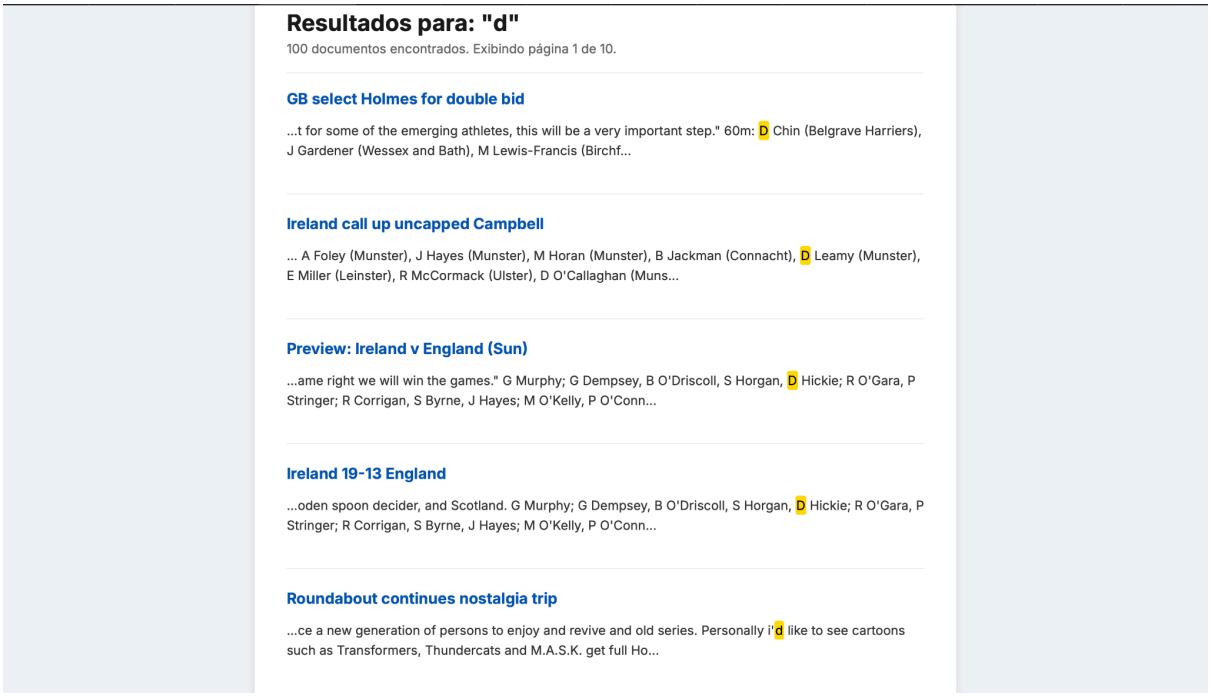
5.4. Exemplos de Mecanismos de Consulta Abaixo, demonstramos o funcionamento de mecanismos específicos que foram detalhados na Seção 4.

- **Exemplo 1: Lógica Booleana e Parênteses**



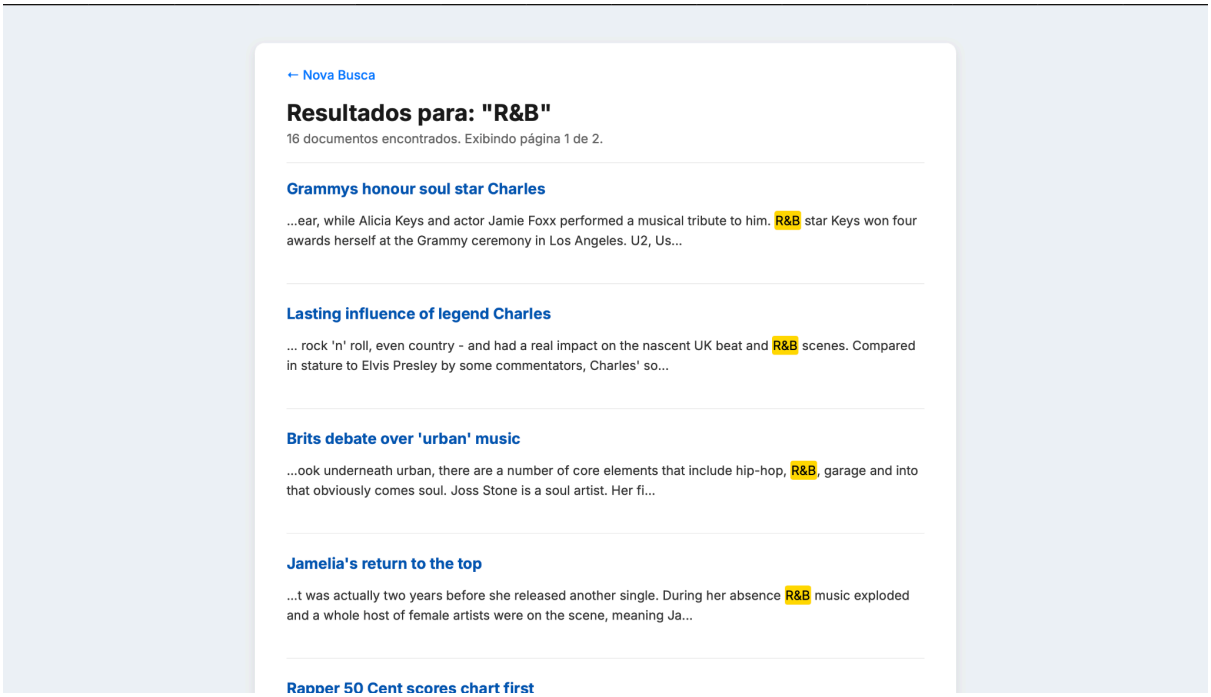
Nesse caso, o sistema processa corretamente a precedência dos operadores, retornando documentos que são simultaneamente sobre esporte e tecnologia, ou documentos que são sobre entretenimento, validando o parser Shunting-Yard.

- **Exemplo 2: Verificação de "Word Boundary" e Apóstrofo**



Nesse caso buscamos pela letra D para mostrar uma ocorrência do apóstrofo (último caso da foto), mostrando que a lógica de permitir o apóstrofo antes do D e proibir depois está funcionando corretamente.

- **Exemplo 3: Tokenização e Ranking de Relevância**



O resultado valida a decisão de tratar R&B como um token único. Além disso, o documento exibido no topo é o que possui o maior Z-score para este termo, demonstrando que o ranking de relevância está funcionando corretamente.

---

6. Conclusão

Este trabalho permitiu aplicar conceitos teóricos fundamentais de Algoritmos 2 em um cenário prático e complexo. A implementação da `CompactTrie` e seu formato de persistência personalizado foram os maiores desafios técnicos, exigindo um entendimento profundo de estruturas de dados e recursão. A implementação do `RI.py` solidificou o conhecimento sobre algoritmos de processamento de consulta, como o Shunting-Yard. Finalmente, a camada de apresentação em Flask (`app.py`) demonstrou a importância de refinar a lógica de busca com heurísticas (como o filtro de veto e as regras de apóstrofo) para traduzir dados brutos em resultados semanticamente relevantes e úteis para o usuário. O resultado é um sistema de busca funcional e robusto que busca atender os requisitos descritos na especificação do trabalho.