Programming Assignment #2
Indexer and Query Processor

Victor Hugo Silva Moura
victorhugomoura@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, BR

ABSTRACT

O objetivo do documento a seguir é detalhar o processo de implementação do Indexer e do Query Processor para a disciplina de Information Retrieval, bem como destacar os desafios enfrentados durante esse processo. Além disso, ao final do documento, serão fornecidas algumas informações e estatísticas relacionadas à indexação realizada para 960.000 páginas e processamento de consultas.

CCS CONCEPTS

• **Information systems** → **Document filtering**; **Search engine indexing**; **Query representation**.

KEYWORDS

information retrieval, document indexing, document filtering, query representation, query processing

ACM Reference Format:
Victor Hugo Silva Moura. 2022. Programming Assignment #2 Indexer and Query Processor. In *Proceedings of (Information Retrieval (UFMG)).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUÇÃO

A tarefa de indexação de páginas web busca extrair e organizar informações contidas nas diversas páginas web, obtidas por meio do processo de crawling, de forma a facilitar o processamento de consultas posteriormente. Essa tarefa é responsável por organizar o conteúdo das páginas em um formato conhecido como índice invertido, no qual é feito um mapeamento de palavra para documento. Ou seja, para cada termo existente no conjunto de documentos, ele contém uma lista de documentos onde a palavra ocorre, bem como possíveis informações adicionais de frequência, posição de ocorrência, etc.

O objetivo principal de um indexador é facilitar a procura por termos de uma consulta e permitir que os documentos relacionados a ela seja mais facilmente mapeados e identificados. Dessa forma, a quantidade de processamento necessária para recuperar documentos para uma consulta é minimizada . Por outro lado, o objetivo principal de um processador de consultas é extrair informações importantes da consulta e do índice criado pelo indexador para retornar um rank de documentos mais relacionados à consulta realizada.

Levando isso em consideração, no contexto da disciplina de Information Retrieval da UFMG, foram desenvolvidos um indexador e um processador de consultas simplificados com o objetivo de tornar mais prático o aprendizado dessas importante ferramentas no processo de indexação e rankeamento das páginas web. Eles são considerados simplificado por não apresentar características mais complexas desses tipos de software, como o expansão de querys, compressão de dados no indexador, correção ortográfica, etc, por exemplo.

Information Retrieval (UFMG), 19 de Junho de 2022, Belo Horizonte, MG

ACM ISBN 978-X-XXXX-XXXX-X/22/06…$0.00
https://doi.org/XXXXXXX.XXXXXXX

2 INDEXADOR

A implementação do indexador seguiu como padrão o modelo geral de um indexador, que foi apresentado em classe, durante as aulas sobre esse assunto. Esse modelo pode ser visto na figura abaixo:

```
1: function index(corpus)
2:   index = map()
3:   did = 0
4:   for document in corpus
5:     did += 1
6:     for (term, tf) in tokenize(document)
7:       if term not in index.keys()
8:          index[term] = list()
9:       index[term].append((did, tf))
10:  return index
```

Figure 1: Modelo Geral de Execução de um Indexador, visto em aula

Algumas modificações foram feitas, no entanto, a fim de cumprir alguns pontos principais do trabalho, como a execução em paralelo e as políticas de gerenciamento de memória e de pré-processamento, por exemplo. Essas modificações e a estrutura final do indexador serão apresentados nas subseções a seguir.

2.1 Pré-Processamento

A fim de garantir que o índice não ficasse muito grande e com muitos termos pouco significativos ou similares, é necessário alguma forma de pré-processar os termos de um documento para unificar termos similares, plurais e flexões de uma palavra, por exemplo. Além disso, alguns termos são pouco significativos para documentos e consultas, por serem termos muito comuns no idioma, conhecidos como stopwords. Esses termos também precisam passar por processamento para serem removidos da página. Para

isso, foram utilizadas funções da biblioteca NLTK.
Inicialmente foi feito a separação de tokens da página, utilizando o "`word_tokenize`" da NLTK, para o idioma Português. Após isso, as palavras passaram pelo processo de stemming, que é responsável por unificar palavras similares, assim como citado no parágrafo anterior. Para esse processo, foi utilizado o "`RSLPStemmer()`", que é um stemmer para português. Dessa forma, os termos já estavam devidamente processados e prontos para serem guardados no índice. No entanto, mesmo após fazer esse pré-processamento, notou-se que ainda haviam diversos termos no índice que eram pouco significativos e/ou nem mesmo representavam palavras reais. Sendo assim, foi criado um dicionário de termos em português e inglês que foi utilizado para definir se um termo seria guardado no índice ou não. Caso o termo esteja presente nesse dicionário, ele é guardado. Caso contrário, ele é descartado. Para a montagem do dicionário, foram utilizadas as coleções de palavras `words`, para palavras em inglês, e `floresta` e `mac_morpho`, para palavras em português, todos da biblioteca NLTK. As palavras desse dicionário foram stemmizadas e as stopwords foram removidas, para promover um dicionário mais próximo do que os termos do documento estariam ao final do pré-processamento.

2.2 Organização no Índice

Com os termos já pre-processados, é necessário guardá-los no índice invertido, seguindo algum dos padrões vistos em sala de aula. O padrão escolhido para esse trabalho foi o modelo que guarda os documentos em que o termo ocorre, bem como a frequência daquele termo dentro do documento, como mostrado no modelo abaixo:

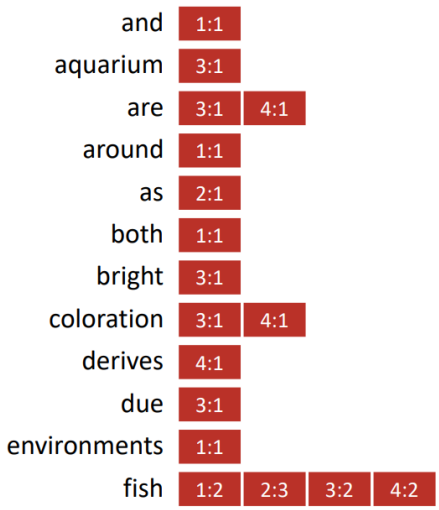

Figure 2: Modelo de Índice Invertido que guarda frequência do termo, visto em aula

Para isso, foi utilizado um dicionário de Python, que contém uma lista para cada termo salvo nele. Essa lista, por sua vez contém diversas tuplas, na qual a primeira posição guarda o id do documento onde o termo ocorreu e a segunda posição guarda quantas vezes o termo ocorreu no documento em questão. Dessa forma, ao inserir um termo no índice, inicialmente é verificado se o termo já existe na estrutura. Caso não exista, o termo é inserido no índice e uma lista é criada para ele. Agora que já garantimos que o termo existe no índice, o id do documento e a frequência do termo são guardados ao fim da lista daquele termo. No caso de termos que já existam no índice, basta executar esse último passo.

2.3 Gerenciamento de Memória

Para garantir que o limite de memória estipulado para o trabalho não fosse ultrapassado, foi utilizado o `setrlimit` da biblioteca resource, de Python, responsável por definir um limite de memória paa o programa. Além disso, foi criada uma regra durante a criação do índice ínvertido no programa. Para garantir que os limites de memória não seriam ultrapassados, criou-se a regra de salvar o índice em disco após o número de elementos no índice ultrapassar 2.000.000 de elementos para cada GB de memória disponível. Sendo assim, caso houvessem apenas 512MB de memória disponíveis, o limite era de 1.000.000 elementos, etc. O número de elementos foi considerado como a soma do tamanho das listas de cada um dos termos contidos no índice.

Apesar de não ocupar completamente a memória inicialmente, esse valor foi definido por considerar que o gasto de memória do programa crescia lentamente ao longo do tempo, por questões de dificuldade de limpeza de memória no Python e por ser uma linguagem que utiliza Garbage Collector, o que não garante que a memória é limpa instântaneamente. Dessa forma, esse valor procura manter um compromisso entre eficiência e gasto de memória ao longo do tempo. Além disso, para valores baixos de memória (1GB), o número de processos criados durante a paralelização foi limitado a 4, para garantir que cada processo tivesse pelo menos 256MB de memória para utilização.

Para salvar o índice em memória, foi utilizado o padrão de salvar um termo por linha em um arquivo `.txt`. Além do termo, a linha contém também a lista de documentos e frequências daquele termo, separados por espaço. Dessa forma, ao ler novamente a linha da memória, o primeiro elemento é sempre o termo, e os próximos elementos são, respectivamente, o id de um documento e a frequência do termo no documento, para cada documento da lista. De forma a facilitar o merge desses arquivos posteriormente, os termos são ordenados alfabeticamente antes de serem salvos no arquivo.

Além de salvar o índice em disco, também é salvo a lista de documentos visitados entre o salvamento anterior e o atual. Nessa lista, para cada linha temos o id, a url e o número de termos do documento, de forma a facilitar o trabalho do processador de queries posteriormente. Ambos arquivos possuem um contador compartilhado que guarda quantos arquivos já foram escritos no disco. Isso busca facilitar o processo de merge ao final, além de manter uma certa organização e prevenir a sobrescrita de arquivos.

2.4 Paralelização

Com a estrutura principal do indexador pronta, o que resta é apenas paralelizar o código. Sendo assim, o loop principal foi inserido em uma função que posteriormente foi transformada em um processo, utilizando a biblioteca multiprocessing de Python. Além disso, o contador de índices guardados no disco, citado na seção anterior,

foi transformado em uma variável compartilhada por todos os processos, de forma que eles pudessem acessar e modificar esse valor à medida que os índices eram salvos em memória.

No entanto, ao compartilhar essa variável o processo de consulta e modificação dela pelos processos pode gerar uma condição de corrida, que é quando processos tentam modificar um recurso simultâneamente, o que pode gerar inconsistências. Nesse caso, foi necessário utilizar um `Lock`, que é uma estrutura presente na biblioteca de Multiprocessing de Python, responsável por garantir exclusão mútua entre processos. Ou seja, na região delimitada por esse mutex, como geralmente são chamados, apenas um processo pode entrar por vez, e somente após ele sair dessa região outro processo pode entrar.

Sendo assim, esse Lock foi utizado para bloquear o acesso de múltiplos processos em todas as regiões que utilizavam variáveis compartilhadas, tornando a paralelização possível. Além disso, como os processos tem id e memória separados dos outros processos, foi necessário aplicar as políticas de gerenciamento de memória para cada um dos processos. Dessa forma, apenas uma fração da memória foi destinada à cada processo, de forma a garantir que eles não iriam ultrapassar a memória total destinada ao indexador.

2.5 Merge de Arquivos

O último passo do indexador é fazer o merge de arquivos criados durante a execução. Para esse merge, aproveitou-se que os arquivos já estavam previamente ordenados para fazer uma ordenação par-a-par. Sendo assim, para cada par de arquivos com mini índices, as linhas deles eram unidas em um novo arquivo seguindo as seguintes regras:

(1) Se um dos arquivos não possui mais linhas, a linha do outro arquivo é escrita no arquivo de merge;
(2) Se os dois possuem linhas, os termos são comparados e a linha correspondente ao menor termo em ordem alfabética é escrita;
(3) Se os dois termos são iguais, a linha deles é únida respeitando a ordenação dos documentos do termo. Ou seja, os documentos são escritos em ordem crescente na memória.

Esse processo é repetido até que o todos os primeiros pares sejam ordenados. Após isso, os novos arquivos criados são ordenados da mesma forma e o processo segue até que reste apenas um arquivo. Esse arquivo, por fim, contém o índice final e é renomeado para corresponder ao nome definido pelos argumentos do programa.

Um processo similar é feito com os arquivos que contém os documentos visitados, com a diferença de que, no segundo passo os ids são comparados, e o terceiro passo não é mais necessário, porque dois processos não visitam o mesmo documento.

Por fim, ao criar o arquivo final de índice, as métricas do índice (tamanho do índice, número de listas e tamanho médio das listas, são calculados). Além disso, é criado um arquivo auxiliar que contém a posição inicial de cada termo no arquivo de índice. Esse arquivo também é utilizado para auxiliar o processador de consultas e tornar o trabalho desse processador mais rápido. Chamaremos ele de arquivo de seeks, para facilitar a referência dele posteriormente.

3 PROCESSADOR DE CONSULTAS

A implementação do processador de consultadas seguiu como padrão o modelo Document-at-a-time (DAAT), que foi apresentado em classe, durante as aulas sobre esse assunto. Esse modelo pode ser visto na figura a seguir.

```
 1: function daat(query, index, k)
 2:   results = heap(k)
 3:   for target = 0 to size(index)
 4:     score = 0
 5:     for term in tokenize(query)
 6:       postings = index[term]
 7:       for (docid, weight) in postings
 8:         if docid == target
 9:           score += weight
10:         postings.after(target)
11:     results.add(target, score)
12:   return results
```

Figure 3: Modelo Geral de um Processador de Consultas DAAT, visto em aula

Algumas modificações foram feitas, no entanto, a fim de cumprir alguns pontos principais do trabalho, como a execução em paralelo e as políticas de scoring e de pré-processamento, por exemplo. Essas modificações e a estrutura final do processador serão apresentados nas subseções a seguir.

3.1 Pré-Processamento

O pré-processamento foi feito de forma similar ao do indexador, de foma a garantir uma consistência entre os termos da consulta e dos documentos salvos no índice. Inicialmente foi feito a separação de tokens da página, utilizando o "`word_tokenize`" da NLTK, para o idioma Português. Após isso, as palavras passaram pelo processo de stemming, utilizando o "`RSLPStemmer()`", que é um stemmer para português. Dessa forma, os termos já estavam devidamente processados e prontos para realização do matching com os documentos.

No entanto, diferentemente do indexador, não foi necessária a criação de um dicionário de termos em português e inglês, pois as consultas já estavam bem formatadas e com termos significativos, ao contrário dos documentos analisados pelo indexador. Sendo assim, essa parte do trabalho foi facilitada.

3.2 Matching de Documentos

Para o matching de documentos, foi utilizado o modelo Document-at-a-time (DAAT), assim como citado no início da seção. Nesse modelo, para cada documento é verificado se os termos da consulta contém ele em suas listas e o score é calculado de acordo com os métodos que serão vistos na próxima seção.

Como o padrão de matching requisitado foi o padrão conjuntivo, todos os termos da consulta necessitam aparecer no documento para que seja feito o casamento da consulta com o documento. Dessa forma, o score é zerado caso um dos termos não apareça no documento que está sendo conferido no momento.

Além disso, de forma a acelerar o processo de matching, para cada consulta um mini índice é construído utilizando o arquivo de seeks, citado no indexador. Por meio desse arquivo é possível recuperar a posição de cada termo no arquivo de índice. Sendo assim, antes mesmo de iniciar o processador, esse arquivo é percorrido para criar um dicionário contendo todos os termos e a posição deles no arquivo de índice. Após isso, para cada consulta, esse dicionário é utilizado para ler o arquivo de índice e recuperar os documentos correspondentes à cada termo da consulta.

3.3 Política de Scoring

Para esse trabalho prático, foram definidas duas funções de pontuação para os documentos: `TFIDF` e `BM25`.

A mais simples de ser implementada foi a TFIDF, pois a fórmula utilizada para ela foi a seguinte:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

onde a frequência do termo é dada por

$$\text{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

e o inverso da frequência do documento (idf) é dada por:

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

Nesse caso, como o tamanho de cada documento estava salvo em um dos arquivos auxiliares criados pelo indexador, o tf foi fácil de calcular. Já para o IDF, foi apenas necessário saber a quantidade total de documentos do corpus `N` e o número de documentos que contém o termo, o que já era dado pelo índice invertido.

Já a fórmula do BM25 utilizada foi a seguinte:

$$\text{BM25}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

onde $f(q_i, D)$ é a frequência do i-ésimo termo no documento D, $|D|$ é o tamanho do documento D, em número de palavras, e *avgdl* é o tamanho médio dos documentos no corpus. Os parâmetros k_1 e b são livres e foram escolhidos como 1.2 e 0.75, respectivamente, seguindo valores padrão para esses parâmetros.

Para calcular essa métrica, foi necessário um processamento adicional antes de começar a processar as consultas. De forma a calcular o avgdl, foi necessário passar pelo arquivo que contém as páginas visitadas e recuperar o tamanho delas. Com isso, foi possível calcular a média de tamanho das páginas. Com essa média calculada, bastou aplicar ela na fórmula para obter o score para o documento que estava sendo analisado.

3.4 Paralelização

Com a estrutura principal do processador de consultas pronta, o que resta é apenas paralelizar o código. Sendo assim, foi criada uma função que recebe um conjunto de consultas e chama o processador para cada uma dessas consultas. Essa função foi posteriormente transformada em um processo, utilizando a biblioteca multiprocessing de Python. Sendo assim, bastou dividir o conjunto de queries de acordo com o número de processos e passar uma fração desse conjunto para cada processo.

Além disso, de forma a manter a impressão do programa organizada, foi necessário utilizar um `Lock` para bloquear a impressão do resultado de múltiplos processos ao mesmo tempo. Dessa forma, foi garantido que a impressão do resultado das queries ocorreriam de forma organizada.

4 EXECUÇÃO

Os dois programas foram executado em um sistema com as seguintes especificações:

- **CPU:** AMD Ryzen 5 3600 (3,9 GHz)
- **RAM:** 16 GB DDR4 (2.800 MHz)
- **Armazenamento:** 2TB HDD (7200 RPM) + 2TB SSD

Inicialmente foram feitos diversos testes pequenos, com no máximo 8.000 documentos, para garantir a funcionalidade do indexador e do processador. Com esse número já foi possível notar alguns padrões e fatores que poderiam prejudicar a execução do processador como a lentidão para recuperar os termos no arquivo de índice e para calcular outros fatores relacionados às métricas.

Para resolver isso, foram criados os arquivos auxiliares citados na implementação do indexador. Após a adição deles, a performance de consulta melhorou significativamente, indo de alguns minutos de processamento para alguns segundos, para uma consulta.

Com esses dois fatores garantidos foram executados novos testes para verificar o número de processos ideal para executar os dois programas, garantindo que haveria melhoras em relação à execução com um processo, porém minimizando o overhead envolvido na troca de contexto entre os processos e/ou gerado pelas zonas de exclusão mútua.

Para o indexador, para cada número de processos, foi feito uma indexação de 120.000 documentos. Os resultados desse teste podem ser vistos à seguir:

Table 1: Tempo de Execução Para 120.000 Documentos

Nº de Threads	Tempo de Execução (s)
1	4956
2	2874
3	2095
4	1589
6	1332
*12	876

* Número de Threads da CPU

Com base nos testes acima, é possível notar que a melhor configuração se deu com 12 processos, que coincidentemente é o mesmo número de threads que a CPU utilizado nos testes possui. Sendo assim, o indexador foi executado utilizando 12 processos, com um tempo estimado de 10512 segundos para terminar, ou seja, 2 horas, 55 minutos e 12 segundos.

Já para o processador foi executado de forma similar para todas as 100 queries fornecidas inicialmente na especificação do trabalho. Abaixo podemos ver os resultados:

Table 2: Tempo de Execução Para 100 Consultas

Nº de Threads	Tempo de Execução (s)
1	1084
2	627
3	562
4	451
6	392
*12	185

Com base nos testes acima, é possível notar novamente que a melhor configuração se deu com 12 processos. Sendo assim, o processador foi executado utilizando 12 processos.

5 RESULTADOS

Após executar o indexador e indexar as 960.000 páginas requisitadas para o trabalho, foi feita uma análise dos dados obtidos por meio dessa indexação. Abaixo temos algumas estatísticas gerais do `corpus` obtido:

- **Tamanho do Índice em MB:** 1.474
- **Número de Listas:** 105800
- **Tamanho Médio De Cada Lista:** 1631.26
- **Tempo Médio Por Arquivo do Corpus:** 69 segundos
- **Tempo Total Gasto:** 1 hora, 50 minutos e 34 segundos

Com isso, podemos ver que no final o crawler acabou gastando menos tempo do que o tempo previsto durante os testes. De acordo com os testes, seria gasto por volta de 3 horas para a execução total.

Considerando agora a complexidade computacional do indexador, temos que, para cada página, são feitas $O(m^2)$ operações, sendo m o número de termos na página. Para cada termo, é verificado se inicialmente se ele aparece no dicionário, o que é $O(1)$. Depois disso, é feita uma contagem daquele termo no documento. Como temos m termos e temos que passar por todos eles para fazer a contagem, temos $O(m^2)$ operações. Considerando que o número de documentos seja n, no total teríamos $O(n * m^2)$ operações para indexar as páginas. Já para o merge, temos uma complexidade $O(nlogk)$, considerando que k é o número de arquivos de mini índice gerados. Como à cada nível de comparação passamos pela coleção inteira de termos e o número de arquivos diminui pela metade à cada passo, chegamos nessa complexidade.

Já para a execução do processador, temos os seguintes histogramas de score para os 10 primeiros resultados de um sample de 10 consultas:

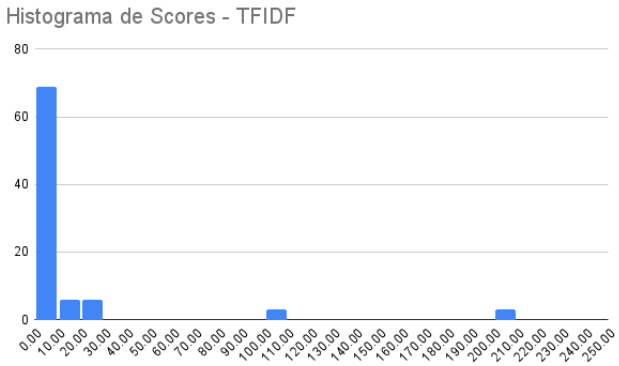

Figure 4: Histograma de Scores para o TFIDF

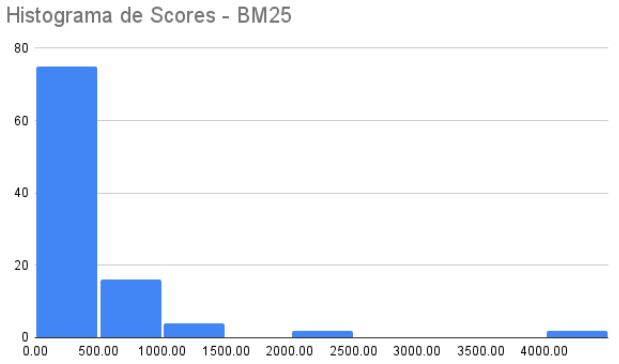

Figure 5: Histograma de Scores para o BM25

Podemos ver que a distribuição de scores é bem similar dos dois métodos, com scores bem próximos de zero para a maioria dos resultados. Porém é possível notar que a escala de scores do BM25 é maior se comparada ao TFIDF, sendo assim os resultados desse método tem valor mais alto de score para documentos com alto valor de matching.

Considerando o processador de queries, temos uma complexidade mais fácil de ser calculada. Para cada query temos que percorrer a lista inteira de documentos e para cada documento é feita uma comparação com os termos da query. Sendo assim, se tivermos n documentos e m termos na query, teremos uma complexidade de $O(n * m)$ no pior caso. Podemos considerar também a criação do dicionário para seeks no arquivo de índice. Dessa forma, se tivermos k termos, teremos $O(k)$ para construção desse dicionário. Por fim, a complexidade pode ser descrita como $O(n * m + k)$.

6 CONCLUSÃO

Por meio da execução desse trabalho, foi possível aprender um pouco mais sobre como funcionam os indexadores e os buscadores web, além de perceber as dificuldades envolvidas durante a implementação de softwares para essa finalidade.

Considero que o trabalho serviu de grande aprendizado para mim e forneceu uma forma prática de aplicar o conteúdo visto em sala de aula.