Trabalho de Recuperação de Informação - Indexer e Query Processor

DANIEL SOUZA DE CAMPOS, Departamento de Ciência da Computação - UFMG, Brasil

ACM Reference Format:
Daniel Souza de Campos. 2022. Trabalho de Recuperação de Informação - Indexer e Query Processor. 1, 1 (June 2022), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Um sistema de recuperação de informação é formado, essencialmente, por três partes: *Crawler* (que pode ser um *Web Crawler*), Indexador e o Processador de Consultas.

O *Crawler* deve ser capaz de gerar um *corpus* de documentos sobre os quais o sistema atuará. O Indexador é responsável por salvar os documentos do *corpus* de forma que possam ser recuperados de forma eficiente via uma consulta. O Processador de Consultas recebe uma consulta e retorna documentos relevantes para o usuário de acordo com uma função de ranqueamento.

No presente trabalho, será apresentada a implementação de um Indexador e um Processador de Consultas. Ambos devem atuar sobre um *corpus* já fornecido de cerca de 1 milhão de documentos já tokenizados. Fatores como paralelização, armazenamento e limitação de memória principal também são discutidos.

2 FERRAMENTAS PARA O TRABALHO

O trabalho foi desenvolvido na linguagem Python 3.8.3 no editor de códigos *Visual Studio Code*. Para controle de versionamento e como repositório do trabalho foi utilizado o *GitHub* e o trabalho ficará disponível publicamente em https://github.com/Pendulun/IndexerQueryProcessor. Lá será possível ver todos os *commits* realizados ao longo do desenvolvimento do programa. A máquina de desenvolvimento é um notebook DELL, 8GB RAM, 1TB HDD com processador Intel Core de sétima geração que possui 4 processadores lógicos. Foi utilizado o WSL (*Windows Subsystem for Linux*) o que permitiu o uso da biblioteca *resource*.

3 INDEXADOR

Nesta seção, serão explicadas as estratégias, decisões e dificuldades de implementação do Indexador.

3.1 Descrição

Um Indexador deve salvar os documentos do *corpus* de forma que a sua futura recuperação por meio de uma consulta seja o mais eficiente possível. Uma possível estrutura de dados seria uma matriz onde cada linha representa um termo do vocabulário e as colunas são os documentos. Essa matriz possuiria uma entrada 1 na posição *i,j* se o termo *i* estiver presente no documento *j* ou 0 caso contrário. Para alguns termos mais comuns em um vocabulário, pode até ser que eles possuiriam uma linha cheia de 1's, entretanto, a grande maioria dos termos possuiria uma linha com muito mais 0's do que 1's. Dessa forma, essa matriz seria extremamente esparsa, o que acarretaria em um desperdício de memória.

Como uma consulta é um conjunto de termos, faz mais sentido ligar documentos a palavras do que o contrário. Dessa forma, a estrutura de dados utilizada para salvar os documentos do *corpus* se chama Lista Invertida. Na Lista Invertida, cada termo de um vocabulário possui associado a ele uma lista de documentos cujo o termo aparece pelo menos uma vez. Dessa forma, evita-se criar a matriz esparsa descrita anteriormente.

Cada entrada em uma lista de documentos associados a um termo (ou *token*) é chamado de *posting*. Um *posting* pode conter mais elementos do que apenas o identificador de um documento. A informação mais básica que um documento pode ter é a quantidade de vezes em que o termo relativo da lista invertida atual aparece nesse documento. Essa informação também é chamada frequência do termo (*term frequency-tf*).

Esse outro tipo de informação associado a um documento na lista de um *token* será importante no ranqueamento dos documentos em uma consulta. Isso será visto em uma seção mais a frente.

3.2 Pré-processamento de Documentos

Como já dito na Introdução, os documentos do *corpus* já estavam tokenizados. Para criar a estrutura do Index de acordo com os *tokens* presentes nos documentos, deve-se aplicar uma técnica que mapeie *tokens* para uma forma que capture sua essência. Essa técnica é chamada de *stemming*. O que o *stemming* faz é reduzir uma palavra a outra, não necessariamente correta, por meio de operações no seu sufixo. Dessa forma, consegue-se mapear palavras derivadas de uma mesma palavra para sua "forma original". Cada língua possui suas próprias regras de *stemming* e, provavelmente, elas não fazem sentido quando aplicadas em uma língua que não seja do seu objetivo original.

Nesse trabalho, foi percebido que o *corpus* continha documentos em várias línguas diferentes, entretanto, sempre foi aplicado o *stemming* com regras para o português. Tentou-se filtrar apenas os documentos em português para gerar o index, mas foi percebido que isso aumentaria muito o custo de indexação e, portanto, não foi aplicado.

Alguns outros filtros no pré-processamento de documentos foram:

- Palavras deveriam ter, no máximo, 23 letras ou seriam descartadas
- Palavras serão transformadas em minúsculas
- *Stopwords* da língua portuguesa serão removidas
- Uma palavra não pode ser formada somente por pontuações

Essas outras regras foram aplicadas de acordo com as observações empíricas dos *tokens* dos documentos.

Author's address: Daniel Souza de Campos, danielcampos@dcc.ufmg.br, Departamento de Ciência da Computação - UFMG, Belo Horizonte, Minas Gerais, Brasil.

XXXX-XXXX/2022/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A implementação desses filtros está no arquivo *parserClasses/myparser.py* a partir da pasta raiz do projeto.

3.3 Implementação do Index

Como descrito anteriormente, o Index é caracterizado pela Lista Invertida. Para facilitar a inserção de novos tokens, o Index é, simplesmente, um dicionário onde a chave é o termo e o valor é outro dicionário de *postings* onde a chave é o identificador do documento e o valor é outro dicionário que contém informações sobre o *posting*. No caso, a única informação é a frequência do termo no documento. Dada a implementação como *Hash Tables* de um dicionário em *Python*, o Index permite acesso em $O(1)$ tanto para o token quanto para o documento associado a um token.

Como está sendo utilizada a função *setdefault* no Index, ele permite adicionar documentos a tokens que ainda não estavam presentes no Index em $O(1)$ também. Logo, tudo é $O(1)$ na hora de inserir, atualizar ou recuperar postings do Index. Entretanto, essa facilidade vem com o custo de manter uma tabela *Hash* para cada dicionário. De acordo com a implementação de um dicionário em *Python*, a tabela *Hash* é ocupada em dois terços do seu tamanho antes de ser alocada mais espaço.

A implementação do Index está no arquivo *indexClasses/index.py* a partir da pasta raiz do projeto.

3.4 Implementação do Indexer

Com o Index em si já implementado, falta quem, de fato, irá populá-lo. Dada a natureza do *corpus* de quase 1 milhão de arquivos e a opção de o usuário limitar o uso de memória principal utilizada pelo programa, deveria ser empregada a paralelização e criação de sub-índices em memória secundária para posterior junção em um único arquivo Index.

3.4.1 Paralelização. O *corpus* estava dividido em 96 arquivos *warc* em que cada um contém 10 mil documentos a serem indexados. Realizar essa indexação de forma sequencial seria um desperdício de capacidade computacional das máquinas modernas e de tempo. Dessa forma, foi implementada uma técnica de paralelização de indexação. Na verdade, foram testadas várias formas de paralelização. São elas:

(1) Divisão em três tipos de processos diferentes: Leitor, Trabalhador e Indexador
   - Nesse caso, existe um processo responsável por ler os arquivos warc e popular uma fila de onde os processos Trabalhadores retiram documentos, os pré-processam e formam a distribuição de frequências de tokens desse documento. Um quarto processo recebe essas distribuições e as adiciona/atualiza no index. Dessa forma, apenas um processo gerenciaria o Index.

(2) Cada processo recebe uma lista de arquivos warc a indexar. Além disso, todo processo possui seu próprio Index e é responsável por, ler, pre-processar e indexar os documentos lidos.

(3) Divisão em dois tipos de processos diferentes: Leitor e Trabalhador.
   - Nesse caso, também existe um processo responsável por ler documentos contidos em arquivos warc e popular uma fila de onde os processos Trabalhadores retiram esses documentos. Entretanto, cada Trabalhador possui seu próprio Index e deve pré-processar e indexar o documentos recebidos.

Método	Estimativa (horas) para indexar 1 milhão
Sequencial	22.57
1 Leitura, 2 Trab, 1 Indexador	13.61
4 Trabalhadores	8.4
1 Leitura, 3 Trabalhadores	5.68

Table 1. Estimativa Indexação usando vários métodos

Foi preferível a utilização de processos e não *threads* pois a maior parte do tempo gasto seria de processamento e indexação do texto de um documento, e não a leitura em si do documento a partir do arquivo *warc*.

A Tabela-1 mostra vários tempos estimados de execução da indexação para apenas gerar os sub-índices, ou seja, sem a parte de sua junção. Esses testes foram feitos levando em consideração o tempo para indexar cerca de 10 mil documentos. Dessa forma, foi escolhido seguir com a técnica de paralelização proposta de número 3 como descrito anteriormente. Esse método foi baseado no problema do Jantar dos Selvagens (*Dining Savages Problem*) descrito no livro *The Little Book of Semaphores*.

Independente da implementação de paralelização escolhida, supondo que existam N tokens em um documento sendo desses M únicos, a distribuição de tokens do documento é igual à um dicionário onde a chave é o token M_i e o valor é a sua frequência tf_i no documento. Dessa forma, adicionar a distribuição de um único documento no Index implementado é igual a $M * O(1) = O(M)$.

3.4.2 Geração de sub-índices. Como existe a opção de limitação de memória principal utilizada e cada processo trabalhador possui o seu próprio index, a ideia para forçar essa limitação foi a seguinte:

(1) Define o máximo de memória como 90% do que foi indicado pelo usuário
(2) O processo de leitura de documentos tem limite de uso de memória igual a 30 MB
(3) Os processos trabalhadores recebem cada um valor igual dividido de acordo com a quantidade de memória que sobrou.

Durante o desenvolvimento do trabalho, não consegui utilizar corretamente a biblioteca *resource* e sua restrição de memória. Dessa forma, utilizei a biblioteca *psutil* para recuperar o uso de memória principal de cada processo. No caso dos processos trabalhadores, quando era detectado que o seu limite de memória passou do limite, ele gravava o seu index gerado ordenado por tokens até o momento em um arquivo e o limpava em memória principal.

Apesar de só ser possível detectar que o processo atual passou de seu uso de memória máximo após ele efetivamente fazer isso, o não uso de 10% da memória inicial permitida pelo usuário na divisão de memória entre os processos permitiu que o valor total original nunca fosse ultrapassado durante as execuções de teste e de indexação dos 1 milhão de arquivos. Isso foi testado para o valor máximo de 1024 MB.

Os arquivos salvos pelo sistema estão todos no formato *pickle* pois facilitam o armazenamento e posterior carregamento de estruturas de dados simples em *Python*.

Como, para gerar o Index dos 1 milhão de arquivos, foi definido o limite de uso de memória em 1024 MB, os processos trabalhadores tiveram que gerar muitos sub-índices e gravá-los em memória. Foram 803431 arquivos de sub-índices gerados totalizando 9,7GB de dados. Isso dá, em média 10KB por arquivo. Essa quantidade enorme de arquivos gerados não era esperada. Os arquivos de sub-índices são gerados em uma pasta chama *index* a partir da pasta raiz do projeto.

Apesar das estimativas de 5,68 horas para gerar todos os sub-índices, esse processo sozinho levou cerca de 28140 segundos (469 minutos ou 7 horas e 49 minutos). Acredito que o tempo de processamento foi pior, além de outros motivos, ao fato de que um serviço de Segurança do *Windows* chamado *Antimalware Service Executable* ficava verificando os processos *Python* e tomava, sozinho, cerca de 80% do processamento da CPU. Quando isso foi identificado, tentei desabilitar a Verificação em Tempo Real e esse processo sumiu. Entretanto, como o próprio *Windows* deixou claro, ele deveria ser ativado automaticamente depois de um tempo e isso realmente ocorreu. Como a indexação foi realizada de madrugada, não foi possível ficar verificando continuamente essa questão e, portanto, o processo levou mais do que o esperado.

Importante notar que o processo de leitura de arquivos *warc* também gerou um arquivo de mapeamento de identificador de documento (*id*) para sua URL. Esse arquivo foi gerado no arquivo *id_to_doc/id_to_doc_map.pickle* a partir da pasta raiz do projeto.

3.4.3 Junção de sub-índices. Nesse ponto, existiam 803 mil arquivos de sub-índices que deveriam ser juntados em um só ainda levando em consideração a restrição de memória imposta pelo usuário. Dessa forma, uma espécie de *Merge Sort* Externo foi implementado.

A ideia foi a seguinte:

(1) Crie blocos de 100 arquivos de sub-índices
(2) Para todos os arquivos em um bloco, junte os sub-índices em um único arquivo
(3) Realize isso *N* vezes até que seja gerado um último arquivo com o Index completo

Ao dividir os 803 mil arquivos em blocos de 100, ficou fácil paralelizar esse processo. A junção em si dos sub-índices acontece da seguinte forma:

(1) Leia uma linha de cada arquivo
(2) Mantenha um dicionário em que as chaves são tokens e os valores são listas de índices de documentos cuja linha atual se refere à lista invertida do token da chave
(3) Pegue os arquivos associados ao "menor token" ordenado alfabeticamente
(4) Junte os *postings* atuais relativos a esses arquivos de sub-índices em uma única lista ordenada pelo identificador dos documentos dos *postings*
(5) Leia uma nova linha de cada arquivo que participou das operações passadas
(6) Realize esse processo até que acabe as linhas de todos os arquivos de sub-índices

A Figura-1 exemplifica o *merging* de 4 arquivos de sub-índices. Cada quadrado representa que a linha lida do arquivo é relativa ao *token* de dentro do quadrado. Ao longo do tempo, apenas os *postings* relativos ao "menor token" são juntados. Para tal, foi importante salvar os sub-índices ordenados pelo token. Além disso, como é lido apenas uma linha por vez de cada arquivo, há um controle da memória utilizada pelo programa.

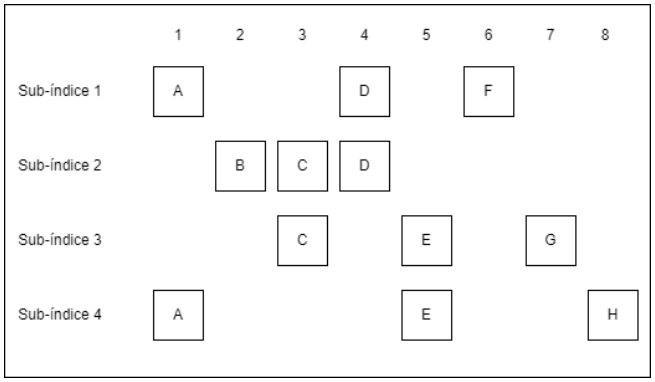

Fig. 1. Processo de *merging* de sub-índices

A cada *merge* de postings, ele eram adicionados à uma mesma estrutura de Index já descrita em uma seção passada. Da mesma forma que no processo de geração de sub-índices, cada processo tinha um limite baseado na divisão de 90% da memória disponibilizada pelo usuário. Quando era detectado que um processo passou do seu limite, de acordo com o *psutil*, ele gravava o conteúdo do seu Index atual sempre em um mesmo arquivo. Dessa forma, ao final de todo o processo, um sub-índice final seria produzido.

A figura-2 mostra o processo de *merging* dos 803 mil sub-índices em um só.

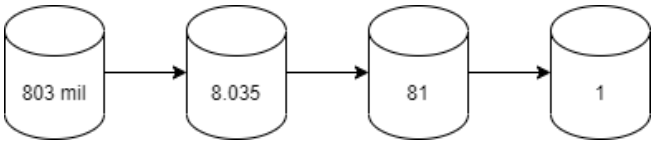

Fig. 2. Processo de *merging* de sub-índices

Esse processo, ao todo, levou mais de 17 horas e também foi afetado pelo Serviço de Segurança do *Windows* descrito anteriormente. A maior parte do tempo foi gasto no primeiro nível de *merging* (15 horas). Ao final, foi possível extrair as seguintes características sobre o último Index:

- *Número de tokens*: 14670956 *tokens*
- *Tamanho médio da lista invertida*: 9.833989754996198
- Espaço total ocupado: 1.732.797 KB

O arquivo com o Index final foi gerado em *final_index/index.pickle*. O index final e o mapeamento de identificador de documentos para sua URL estarão disponíveis em: https://drive.google.com/drive/folders/1pSgbBdz_8PoOnqYLJrM6f_w4ZVaRuaWy?usp=sharing.

O indexador está disponível no arquivo *indexClasses/indexer.py* a partir da pasta raiz do projeto.

4 PROCESSADOR DE CONSULTAS

Nessa seção, serão descritas as decisões e dificuldades de implementação do Processador de Consultas.

4.1 Descrição

Uma vez que o Index esteja pronto, o usuário poderá realizar consultas. É trabalho do Processador de Consultas receber a consulta do usuário, pré-processá-la e retornar os documentos relevantes para o usuário. Entretanto, não basta retornar um conjunto com todos os documentos que o sistema ache que sejam relevantes, eles devem vir em uma certa ordem de relevância.

Para tal, é necessária uma função de ranqueamento que, a partir de características de um documento, produz um *score* que represente o quão relevante ele é para a consulta. No caso do trabalho, existem a informação de frequência do termo em um documento, o tamanho do documento, em quantos documentos um termo aparece e a quantidade de documentos total indexados. Todas essas informações podem ser utilizadas para ranquear um documento dada uma consulta do usuário.

4.1.1 TFIDF. A primeira função de ranqueamento é conhecida como TF-IDF.

$$TFIDF_{y \in C,Q} = \sum_{x \in Q} tf_{x \in y} \times log \frac{N}{df_x}$$

Ela simplesmente diz que o *score* de um documento y no *corpus* C é igual à soma da frequência do termo x em y vezes o log do número total de documentos no *corpus* N sobre a quantidade de documentos que contém o termo x df_x para todo termo x na consulta Q.

4.1.2 BM25. A segunda função de ranqueamento é conhecida como BM25.

$$BM25_{y \in C,Q} = \sum_{x \in Q} IDF(x) \times \frac{tf_{x \in y} \times (k_1 + 1)}{tf_{x \in y} + k_1 \times (1 - b + b \times \frac{|D|}{avgdl})}$$

Onde $|D|$ é o tamanho do documento y, $avgdl$ é o tamanho médio de documentos no *corpus* e k_1 e b são "hiperparâmetros" da função.

Como seria necessário o tamanho médio dos documentos no *corpus* e o tamanho de cada documento, mas essas informações não foram gravadas ao se gerar o Index, o BM25 não foi implementado e, se escolhido como forma de ranqueamento do Processador de Queries, o TF-IDF será usado no seu lugar.

4.2 Implementação

4.2.1 Paralelização. Já que a entrada para o Processador de Consultas é um arquivo que pode conter várias consultas a serem respondidas, foi aplicada um paralelismo de dados simples onde existe uma *Pool* de *Threads* e, ao longo do tempo, cada *thread* fica responsável por processar uma consulta.

Ao final, os resultados são unidos e apresentados para o usuário não necessariamente na mesma ordem das consultas no arquivo de entrada.

4.2.2 Pré-processamento da Consulta. O primeiro passo para processar a consulta é aplicar o mesmo pré-processamento de textos dos documentos de forma que os *tokens* da consulta fiquem iguais aos seus semelhantes no Index.

4.2.3 Encontrar as listas invertidas. O segundo passo após pré-processar o *tokens* da consulta é encontrar, no Index, as Listas Invertidas de cada *token*. Para tal, os tokens pré-processados da consulta são ordenados alfabeticamente e, sequencialmente, procurados no arquivo do Index final. Dessa forma, se o *token* menos bem ordenado alfabeticamente for o M-ésimo no Index, a procura será O(M).

4.2.4 Scoring. Uma vez em posse das Listas Invertidas, é possível calcular um *score* para os documentos. O cálculo dos scores é semelhante ao processo de *merging* de sub-índices. Dadas as listas invertidas, lemos o primeiro documento de cada uma delas. Pegamos todos os *postings* associados ao menor documento atual. Calculamos o seu *score* com base nos valores de frequência em cada lista invertida. A Figura-3 exemplifica esse processo. No caso, os quadrados são formados por pares de identificadores e frequência do *token* no documento.

Após calcular o *score* do documento, ele é adicionado a uma lista ordenada de forma decrescente pelo *score* dos documentos. Essa lista possui um tamanho máximo de 10 elementos. Dessa forma, a partir do 11° documento, o que possuir o menor *score* sempre será excluído da lista.

O algoritmo deve ler todos os documentos associados à uma lista invertida. Logo, sendo M o tamanho da maior lista invertida relativa à consulta atual, temos custo O(M). Além disso, existe a ordenação da lista com os *scores*. Sendo essa ordenação via a função *sort* do Python e, já que o tamanho máximo dessa lista é 10, temos complexidade $O(10log10) = O(log10) = O(1)$. Assim, como essa ordenação é realizada à cada leitura das listas invertidas, temos complexidade total de $O(M) \times O(1) = O(M \times 1) = O(M)$.

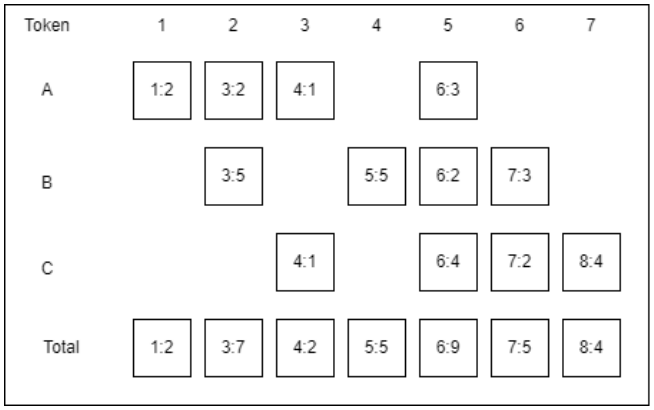

Fig. 3. Processo de *scoring* de documentos

4.2.5 Mapeamento de identificadores para URL's. O último passo do Processador de Consultas é mapear os identificadores no *ranking* dos 10 documentos mais relevantes encontrados para suas respectivas URL's.

Como já dito na seção que explica a indexação, foi gerado um arquivo com pares de identificador para URL. Do jeito que foi

gravado, os identificadores não estão ordenados nesse arquivo. Dessa forma, a procura de URL para um identificador é $O(N)$, sendo N o número total de documentos no *corpus*. Como isso é feito para cada um dos 10 documentos retornados, a complexidade total será de $10 \times O(N) = O(N)$.

A implementação do Processador de Consultas está no arquivo *queryProcessingClasses/queryProcess.py* a partir da pasta raiz do projeto.

5 CONCLUSÃO

O objetivo inicial do trabalho era implementar um Indexador e um Processador de Consultas para um Sistema de Recuperação de Informação. Um Indexador deve salvar os documentos presentes no *corpus* de forma que sua posterior procura seja a mais eficiente possível. Além disso, a implementação deve lidar com aspectos de memória e uso eficiente da capacidade de processamento paralelo das máquinas atuais. Já o Processador de Consultas deve procurar no Index gerado pelo Indexador os documentos relevantes para a consulta e os retornar em lista ordenada decrescentemente pelas suas relevâncias via aplicação de uma função de ranqueamento.

Pode-se dizer que os objetivos foram alcançados apesar das dificuldades, performance geral e não implementação do BM25. Como já mencionado, o Indexador levou muito tempo para gerar o Index final além de produzir 803 mil arquivos de sub-índices. Com certeza, deve existir uma forma melhor de gerar o Index final, entretanto, a implementação atual funciona.

Já o Processador de Consultas poderia ser melhorado se o arquivo de mapeamento de identificadores de documentos para suas URL's fosse ordenado pelos identificadores. Além disso, também devem existir técnicas melhores de detectar e aplicar a função de *scoring* nos documentos das listas invertidas que não seja percorrer completamente todas as listas invertidas relativas aos *tokens* da consulta.

Finalmente, com esse trabalho, foi possível praticar conceitos aprendidos ao longo da matéria de Recuperação de Informação e ter uma ideia do real esforço que empresas donas de sistemas de recuperação devem lidar diariamente. O trabalho pode ser considerado massante dado os inúmeros pontos de dificuldade encontrados durante a implementação, principalmente, escolha de um método de paralelismo para a Indexação e verificação de memória utilizada. Entretanto, ao terminar a implementação, e ver o sistema funcionando, ele gerou um certo orgulho por completar essa tarefa.