Programming Assignment #2
Indexer and Query Processor

[Luis Antonio Duarte Sousa]

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

luisduarte.dev@gmail.com

1 Introdução

Este trabalho tem como objetivo implementar os módulos de indexação e processamento de consultas de um motor de busca na web. A proposta envolve o desenvolvimento de dois componentes principais: um indexador, responsável por processar um grande corpus de documentos estruturados da Wikipédia, realizando pré-processamento, tokenização e construção de índices invertidos; e um processador de consultas, que utilizará os índices construídos para recuperar e ranquear documentos relevantes com base em modelos clássicos de ranqueamento, como TF-IDF e BM25. A implementação será realizada em Python 3.13, respeitando restrições rigorosas de ambiente virtual, consumo de memória e paralelização, visando simular condições reais de operação sobre grandes volumes de dados. Ao final, espera-se caracterizar o índice gerado e apresentar os resultados obtidos a partir de um conjunto de consultas, demonstrando a eficiência e a correção das técnicas aplicadas.

2 Implementação

2.1 Indexer

A implementação do indexer segue essencialmente um esquema "SPIMI-like" (Single-Pass In-Memory Indexing), dividido em três fases principais: leitura do corpus e enfileiramento de documentos, construção de índices parciais em múltiplos processos, e mesclagem desses índices parciais em um índice único final. A seguir, descrevo cada etapa:

2.1.1 Leitura e enfileiramento dos documentos

No início, o processo principal (main) cria um Manager do módulo multiprocessing, que disponibiliza uma fila compartilhada (doc_queue) e estruturas de controle (um Value para o contador de arquivos parciais e um Lock para sincronização). Em paralelo ao trabalho de indexação, é iniciado um processo leitor (task_corpus_reader) que abre o arquivo JSONL do corpus e, linha a linha, faz json.loads para converter em dicionário Python e o coloca na doc_queue. Isso garante que os demais processos possam puxar documentos de forma independente, sem concorrer pela leitura do disco. (O uso de processos que são mais caros ao invés de threads é justificado pelo fato de que a memória de um processo é imediatamente liberada quando ele é terminado e portanto permitia um controle maior da quantidade de memória usada na indexação. Caso fosse usado thread, o python não libera a memória imediatamente após apagar uma variável e isso prejudicava a lógica do processo)

2.1.2 Construção de índices parciais em workers

O main calcula um número de "writers" igual à metade do tamanho de CPUS disponíveis (para balancear I/O e CPU), e em cada batch lança esse mesmo número de processos task_writer. Cada writer consome documentos da doc_queue até que o uso de memória (monitorado via psutil.Process(...).memory_info().rss) atinja um limiar definido como 80 % da cota por worker. Para cada documento, extrai campos relevantes (title, text, keywords), concatena-os, pré-processa (tokenização, remoção de stopwords, stemming) via util.preprocess(), e atualiza dois índices em memória:

document_index: um dicionário compartilhado que mapeia doc_id → title, length, usado depois para montar o índice de documentos.

inverted_index: um defaultdict(list) local que agrupa, para cada termo, uma lista de tuplas (doc_id, tf).

Ao ultrapassar a memória, ou ao ter-

minar de consumir a fila, o worker chama util.save˙partial_index_jsonl(), que — sob proteção de um lock — grava o conteúdo de inverted˙index em um arquivo "partial_inversed_k.jsonl" e zera sua estrutura em memória. Esse flush parcial é o cerne do SPIMI: nunca se tenta manter todo o índice em RAM, mas sim segmentos controláveis que são escovados para disco conforme necessário.

2.1.3 Salvamento incremental do índice de documentos

Após cada batch de writers finalizar (isto é, quando todos os processos daquele lote join() retornam), o main chama util.save_partial_document_index(), que ordena e salva o document_index acumulado num arquivo "partial_document_index_n.jsonl". Em seguida o dicionário é reinicializado—novamente, para manter o consumo de memória dentro do orçamento—e o processo repete até a fila se esvaziar e o reader terminar.

2.1.4 Mesclagem dos índices parciais

Com todos os segmentos parciais de índice invertido prontos, o main invoca parallel_merge_partial_indexes(). Primeiro, ele divide uniformemente a lista de arquivos "partial_inversed_*.jsonl" em quatro grupos e lança threads para chamar merge_group() sobre cada grupo. Essa função faz um k-way merge usando um heapq, similar a um external merge-sort: coleta o menor termo disponível de cada arquivo, agrupa listas de postings de arquivos diferentes para o mesmo termo, agrega frequências e escreve em um arquivo intermediário. Depois que todas as threads terminam, faz-se uma última mesclagem de todos os arquivos intermediários em complete_inversed_index.jsonl, computando também o número total de termos e o tamanho médio das listas.

2.1.5 Finalização e estatísticas

Por fim, o código constrói o índice de documentos completo concatenando os arquivos "partial_document_index_*.jsonl" em um único document_index.jsonl, apaga os arquivos parciais e calcula o tamanho do índice em disco, o tempo total de execução e as estatísticas de número de listas e tamanho médio da lista. Esses valores são empacotados em JSON e impressos em stdout, atendendo exatamente ao formato exigido pelo enunciado.

2.2 Processer

A implementação do processor.py segue uma arquitetura multi-thread que pode ser dividida em quatro etapas principais: leitura e pré-processamento das consultas, carregamento seletivo das listas de postagens (inverted lists) necessárias, processamento concorrente das consultas (intersecção DAAT e ranqueamento) e impressão dos resultados.

2.2.1 Leitura e pré-processamento das consultas

O programa inicia em main(), onde lê o arquivo de consultas linha a linha, descartando linhas em branco. Para cada consulta textual, chama util.preprocess(), que aplica tokenização, normalização (remoção de caracteres não alfanuméricos), remoção de stopwords e stemming — exatamente as mesmas etapas usadas na indexação, garantindo compatibilidade entre termos de consulta e documentos. As consultas pré-processadas são armazenadas em queries_tokens, e um conjunto tokens˙set acumula todos os tokens distintos que serão buscados no índice invertido.

2.2.2 Carregamento seletivo do índice invertido

Em vez de ler o índice inteiro, o processor.py abre apenas o arquivo complete_inversed_index.jsonl gerado pelo indexador e o percorre linha a linha. Cada linha contém um objeto JSON com um term e seu doc˙list. Se o term estiver em tokens˙set, sua lista de postagens é salva em query_terms_index[term] e o termo é removido de tokens˙set. Esse filtro prévio evita carregar noções de outros termos que não aparecem em nenhuma consulta, reduzindo consideravelmente o uso de memória e o custo de I/O.

2.2.3 Processamento concorrente: DAAT e ranqueamento

Depois de carregar todas as postagens necessárias, cada consulta (representada por sua lista de tokens e a string original) é enfileirada em `query_queue`. São então criadas cinco threads (`n_threads`) que executam em paralelo a função `process_query()`. Cada thread:

1. Retira uma consulta da fila.
2. Chama `util.naive_daat_and()`, que faz interseção "document-at-a-time" (DAAT) das listas de postagens correspondentes a cada termo da consulta. Esse algoritmo mantém um ponteiro em cada posting list e avança aquele cujo `doc_id` é menor, até encontrar documentos que

contenham todos os termos ou até esgotar alguma lista.

3. Se houver candidatos, invoca `util.rank_documents()`. Primeiro, essa função carrega do `document_index.jsonl` apenas os metadados (comprimento e título) dos documentos candidatos, calculando:

   $N =$ número total de documentos, $\quad avgDL = \frac{1}{N}\sum_{d}|d|$

   Em seguida, calcula o score de cada candidato sob dois modelos:

   TF–IDF com similaridade de cosseno
   BM25 com constantes k = 1,5 e b = 0.75

4. Ordena os resultados pelo score em ordem decrescente e seleciona os top 10.

Por fim, a thread imprime um objeto JSON com a consulta original e sua lista de resultados.

2.2.4 Sincronização e término

Cada thread chama query_queue.task_done() ao finalizar uma consulta, e o main() aguarda query_queue.join() para garantir que todas as consultas sejam processadas. Depois, faz join() em cada thread para limpá-las completamente antes de encerrar o programa. Esse design garante alto throughput, pois enquanto uma thread faz I/O de índice ou calculos pesados, as demais continuam consumindo consultas da fila.

3 Análise de Complexidade

3.1 Indexer

- **Leitura e enfileiramento:** percorre cada um dos D documentos do corpus uma única vez, logo $O(D)$ em tempo e $O(1)$ em espaço adicional (além do buffer de leitura).
- **Construção de índices parciais:** cada doc é tokenizado em T_d termos, pré-processado em $O(T_d)$ e inserido em um dicionário local de tamanho proporcional ao número de termos distintos naquele segmento. Sejam N o número total de tokens no corpus e V o vocabulário final, então o trabalho de inserção em todos os segmentos é $O(N)$ no total, e o uso de memória é limitado a M (orquestrado via flushes parciais).
- **Merge k-way das listas:** para P segmentos parciais e um vocabulário final de tamanho V, a mesclagem paralela faz um merge k-way em cada grupo de segmentos. Cada item (termo) entra e sai do heap uma constante vez por segmento, levando $O(\log P)$ cada operação. O custo total é $O(V\,P\,\log P)$ distribuído em G threads, reduzindo para $\approx O(\frac{V\,P\,\log P}{G})$ em tempo.
- **Espaço em disco:** proporcional ao tamanho do índice final, $O(V + N)$, uma vez que armazenamos para cada termo a lista de pares (doc_id, tf).

3.2 Processor

- **Pré-processamento de consultas:** para Q consultas de tamanho médio L_q, usa-se $O(Q\,L_q)$ para tokenização e stemming.
- **Carregamento seletivo do índice:** percorre o arquivo completo de V termos mas insere apenas T termos relevantes (onde $T \leq \sum L_q$) em $O(V)$ tempo de I/O e $O(T)$ espaço em memória.
- **Interseção DAAT por consulta:** cada interseção de listas de tamanho $n_1, \ldots, n_k$ custa $O(\sum_{i=1}^{k} n_i \log k)$ no pior caso (avançando ponteiros com heap ou comparações lineares), tipicamente $O(N_q)$ onde N_q é soma dos comprimentos das k posting lists.
- **Ranqueamento TF–IDF e BM25:** para cada doc candidato (até C_q por consulta) e k termos, calcula-se pesos em $O(k)$, totalizando $O(Q\,C_q\,k)$ operações aritméticas.
- **Concorrência:** com n_{threads} threads, o throughput tende a $\approx \frac{1}{n_{\text{threads}}}$ do tempo sequencial, limitado por seções de I/O e por conteúdo do CPU (Amdahl's Law).

4 Resultados

A caracterização do índice construído sobre o corpus de 4 641 784 documentos revela um vocabulário final de 1 844 728 termos e um total de 185 528 585 tokens (soma dos comprimentos de todos os documentos). A distribuição de postings por termo evidencia uma forte assimetria: enquanto o valor mediano é 1—ou seja, metade dos termos aparece em apenas um documento—o valor médio é 68,8 e o desvio-padrão supera 2 700, o que indica a existência de um pequeno conjunto de termos extremamente frequentes (o máximo chega a 1 146 955 postings) e uma longa cauda de termos raros.

Em termos de desempenho, a indexação sequencial (1 processo/thread) levou cerca de 2 horas, pois processava um documento por vez e mantinha todos os dados em um único espaço de memória antes de

cada flush. Ao adotar paralelização por processos (com número de workers igual a metade do total de núcleos de CPU), o tempo de indexação caiu para aproximadamente 30 minutos. Cada processo reserva seu próprio heap, permitindo um controle rigoroso de memória e flushes mais frequentes de índices parciais. A etapa de merge k-way também foi paralelizada via threads, sem necessidade de locks, resultando em escalonamento quase linear no número de threads de merge.

Para as consultas de teste, observamos os seguintes resultados:

- **BM25:**
  - "christopher nolan movies": 2 documentos
  - "radiohead albums": 10 documentos
  - "19th century female authors": 5 documentos

  Estatísticas de scores: mínimo $\approx 1{,}79 \times 10^{-5}$, máximo $\approx 4{,}68 \times 10^{-3}$, média $\approx 2{,}52 \times 10^{-3}$, mediana $\approx 4{,}00 \times 10^{-3}$, desvio-padrão $\approx 2{,}08 \times 10^{-3}$. Os valores baixos de BM25 refletem a normalização pesada e frequências de termos relativamente pequenas nos documentos retornados.

- **TF–IDF:**
  - "christopher nolan movies": 2 documentos
  - "radiohead albums": 10 documentos
  - "19th century female authors": 5 documentos

  Estatísticas de scores: mínimo $\approx 0{,}954$, máximo $\approx 0{,}9999$, média $\approx 0{,}9906$, mediana $\approx 0{,}9998$, desvio-padrão $\approx 0{,}0139$. A similaridade de cosseno tende a produzir scores elevados quando há alto grau de coincidência de termos, especialmente em consultas curtas.

No processamento de consultas, utilizamos cinco threads (uma por consulta simultânea), pois não havia restrição de memória e o número de consultas era pequeno. Nesse cenário, a paralelização não reduziu significativamente o tempo total—já que cada consulta é relativamente rápida—mas garantiu que todas as consultas fossem atendidas quase que instantaneamente, maximizando o throughput sem overhead perceptível.‴