Programming Assignment 1 - Information Retrieval

Marcelo Sartori Locatelli
locatellimarcelo@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

ABSTRACT

Nesse trabalho foi implementada uma estrutura de índice invertido. Partindo de um corpus de documentos, cada documento sofre parse e é indexado, gerando índices parciais conforme o limite de memória é alcançado. Para unir esses índices parciais também foi implementada uma operação de mergesort em disco, que une os arquivos com baixo custo de RAM. Esse processo foi paralelizado para garantir maior velocidade. Link para as estruturas geradas: https://drive.google.com/drive/folders/14YGyEzR5wxAMT2xmt0wr6OmMCWsjOFAi?usp=sharing

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Marcelo Sartori Locatelli. 2018. Programming Assignment 1 - Information Retrieval. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 CRIAÇÃO DO INDICE

1.1 Parser

Antes dos documentos serem submetidos ao índice, eles passam por uma etapa de parse, na qual serão descartados símbolos e palavras desnecessárias e o documento será tokenizado. Para isso, foi implementada uma classe parser, que encapsula funções de string nativas de python e da biblioteca nltk para eliminar tudo aquilo que é desnecessário e já retornar dados da forma (token, frequência)

Optou-se por eliminar dígitos, pontuação e caracteres não latinos nessa etapa, visando deixar o índice o mais enxuto possível, evitando ter muitas palavras desnecessárias que nunca serão buscadas (como por exemplo caracteres chineses em um índice para a língua inglesa), além disso, conforme pedido na especificação, também são removidas as stopwords.

A complexidade de tempo dessa etapa é O(d), onde d é o comprimento do maior documento (titulo,texto e keywords concatenados), visto que a complexidade da etapa de parse é dominada pela eliminação de caracteres indesejados, sendo que a checagem de cada caractere individual é O(1) com a operação str.translate() de Python. A complexidade de espaço também é O(d).

1.2 Index Creation

Dado a limitação de memória, o corpus é aberto como um generator de Python, o que garante gasto de espaço O(1) para a leitura do arquivo.

A criação do índice é paralelizada, de modo que cada processo recebe os próximos documentos por meio de uma fila com exclusão mútua e tamanho máximo = n_processos, em que cada posição da fila mantém 1000(valor escolhido empiricamente) para reduzir as tentativas de acesso simultâneo.

O pseudocódigo é praticamente idêntico ao apresentado em sala, com a diferença que os documentos são recebidos de uma fila e há uma checagem de memória ao final de cada iteração para salvar e reiniciar o índice quando o limite está próximo de ser alcançado. Tem-se assim o seguinte pseudocódigo para a criação do índice para cada processo:

Algorithm 1: Execução de um processo

```
Function create_index(processo,document_queue):
    index = map()
    while !EOF(corpus) do
        documents=document_queue.pop()
        for document in documents do
            did+=1 ; // did is shared between threads
            for (term,tf) in tokenize(document) do
                if term not in index.keys() then index[term]
                  = list() ;
                index[term].append((did,tf))
            end
            if process_memory_limit is reached then
              save_and_reset(index);
        end
    end
```

Como ler todos documentos é O(n) e fazer o parse é O(d), essa etapa é O($n * d$), visto que o índice em si é uma hash table, que garante acesso O(1) amortizado e por isso foi escolhido. Como o índice é ordenado antes de ser armazenado, se tem um custo adicional de O($t * \log t$),onde t é o número de listas. Salvar as listas é O($t * |t|$),onde $|t|$ é o tamanho da maior lista invertida, sendo assim o custo total é O($n * d + t * |t| + t * \log t$). A complexidade de espaço é O($t * |t|$).

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1.3 Disk mergesort

Como foi utilizado multiprocessamento em que cada processo salva em arquivos diferentes, é necessário unir esses arquivos posteriormente, isso foi feito por meio de um mergesort implementado para funcionar diretamente no disco. A implementação é relativamente simples, sendo muito similar a um mergesort normal só que com "2 dimensões", o termo do índice e a lista do termo. O pseudocódigo abaixo ilustra o processo. Note que, na realidade, foram lidas/escritas 5000(valor escolhido empiricamente) listas invertidas de cada vez para evitar gastos de tempo muito grandes devido a I/O, esse buffer é um deque, funcionando com uma fila com inserção e remoção O(1).

Algorithm 2: Mergesort em disco

```
Function disk_mergesort(file1,file2):
    while !EOF(file1) and !EOF(file2) do
        (term1,list1),(term2,list2) = file1.read(),file2.read()
        if term1<term2 then write_to_file(term1,list1);
        else if term1>term2 then write_to_file(term2,list2);
        else new_list = merge(list1,list2);
          write_to_file(term1,new_list);
    end
```

Note que isso faz o mergesort entre dois arquivos, mas muito mais do que dois arquivos são criados na criação do indice devido à limitações de memória. Portanto, essa etapa é paralelizada de maneira similar à uma operação de redução paralela, de forma que na etapa 1 se faz o merge dos arquivos originais 2 a 2 (utilizando processos), na etapa 2 se faz o merge daqueles criados na etapa 1, e isso é repetido até que sobre apenas um arquivo. A complexidade de tempo da última chamada de disk_mergesort é $O(t * |t|)$ e ela é a mais custosa, note que não há um fator log, pois só é feita a etapa merge do mergesort, as listas já estão ordenadas. O custo de memória é $O(|t|)$.

1.4 Complexidade final

Complexidade de parser e criação do índice $O(n * d)$, complexidade de salvar o índice $O(|t| * t + t * \log t*)$ e a complexidade de merge é $O(t * |t|)$. Assim a complexidade final é $O(t * \log t + n * d + t * |t|)$. O custo de memória no pior caso é $O(d + t * |t|)$, que ocorreria na última iteração da criação do índice caso a memória não fosse limitada. Com a memória limitada, o custo é constante, sendo no máximo a flag de memória fornecida em MB.

1.5 Codificação

Devido à limitações de python, a codificação bit a bit era muito difícil de se implementar. Por isso, optou-se por uma codificação byte a byte para todos os inteiros do índice invertido. Foi usada a Variable byte encoding, visto que para inteiros pequenos (o que é esperado para a maioria das frequencias da base), ela economiza quantidade significativa de espaço quando comparado com a codificação usual de inteiros. Ela também permite não se usar separadores entre os números, outra economia de espaço. Para evitar ids de documento enormes utilizados juntamente com a codificação, armazenou-se os deltas dos ids dos documentos nas listas invertidas. A codificação é $O(|t|)$ em tempo e $O(|t|)$ em espaço, mas seu uso não muda o comportamento assintótico do algoritmo. Em experimentos menores, o uso desses métodos resultou em um tamanho de arquivo até 60% menor do que só salvar cada lista em uma linha com cada ocorrência separada por vírgula.

1.6 Arquivos extra

Além do indice invertido, ao final da criação do índice, são gerados os arquivos document_index que guarda a relação id/did e o tamanho de cada documento e corpus_statistics que guarda o número de documentos e o comprimento médio. Após a etapa de merge é gerado o lexicon que guarda o offset e número de listas para cada termo, essa escolha foi proposital, para evitar ter que gerar o lexicon durante o merge (o que implicaria em ter que criar uma nova função de merge para o merge final que gerasse o lexicon ou gerar lexicons parciais desnecessários), isso aumentou o tempo de execução em menos de 3 minutos.

2 SPEED-UP DEVIDO A PARALELISMO

Os testes foram feitos utilizando a flag -m 1024. Utilizou-se uma máquina com processador ryzen 5 2600 (6 cores), ssd com leitura 500MB/s e gravação 450MB/s e 32GB de ram(embora só 1GB tenha sido utilizado).

Devido à Global Interpreter Lock de Python, que impede um paralelismo real com threads, optou-se por usar multiprocessos para o paralelismo. Isso vem com um grande overhead de memória, mas acelerou bastante a execução. A tabela 1 mostra o speedup e outras estatísticas relacionadas à performance. Para a entrega final, optou-se em se usar um número de processos igual ao número de cores da máquina.

Threads	**1**	**2**	**4**	**6**
Speedup	1,00	1,43	1,80	2,24
Tempo(s)	7.177	4.991	3.973	3.194
Indexação(s)	4.645	2.692	1.559	1.199
Merge(s)	2.532	2.299	2,414	1.995

Table 1: Speedup observado para diferentes números de processos

3 ESTATÍSTICAS DO ÍNDICE

- Tamanho do índice: 321 MB
- Número de listas: 1.965.885
- Tamanho médio de listas: 64,4
- Número de documentos: 4.641.784
- Média de palavras por documento: 40,3

Link: https://drive.google.com/drive/folders/14YGyEzR5wxAMT2xmt0wr6OmMCWsjOFAi?usp=sharing

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009