Desafios na Implementação da Ordenação Externa

Introdução

O presente texto trata-se das dificuldades de implementação e entendimento algoritmo no contexto do objetivo proposto.
Neste trabalho foi utilizado a seguinte ferramenta de LLM:

- aistudio.google.com (Gemini 2.5)
- github.com/features/copilot

0. Entendimento do Merge Sort

Inicialmente foram necessárias visualizações mentais para o melhor entendimento do Algoritmo:

- joaoarthurbm.github.io

O uso da IA como geradora de exemplos tangíveis não foi satisfatório. Porém a partir deles foi possível elaborar e destrinchar o algoritmo *na mão*, com papel e caneta, a fim de se entender uma possível abordagem. Ver exemplo:

Havendo um *buffer* de tamanho **3**, realizamos as seguintes operações

[8,3,9,2,5,1,7,4,6]

Estado Inicial da Mesclagem:

- **Runs:** run_0:[3,8,9], run_1:[1,2,5], run_2:[4,6,7]
- **Arquivo Final:** []
- **Candidatos (topo de cada run):** 3 (do run_0), 1 (do run_1), 4 (do run_2)

[1]

- **Comparação:** min(3, 1, 4) é 1.
- **Ação:** Mover 1 para o arquivo final.
- **Reposição:** O 1 veio do run_1. O próximo elemento no run_1 é 2.
- **Candidatos Atuais:** 3 (run_0), 2 (run_1), 4 (run_2)
- **Arquivo Final:** [1]

[1, 2]

- **Comparação:** min(3, 2, 4) é 2.
- **Ação:** Mover 2 para o arquivo final.
- **Reposição:** O 2 veio do run_1. O próximo elemento no run_1 é 5.

- **Candidatos Atuais:** 3 (run_0), 5 (run_1), 4 (run_2)
- **Arquivo Final:** [1, 2]

…

[1, 2, 3, 4, 5, 6, 7, 8]

- **Comparação:** Só resta o 8.
- **Ação:** Mover 8 para o arquivo final.
- **Reposição:** O 8 veio do run_0. O próximo elemento no run_0 é 9.
- **Candidatos Atuais:** Apenas 9 (run_0).
- **Arquivo Final:** [1, 2, 3, 4, 5, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8, 9]

- **Comparação:** Só resta o 9.
- **Ação:** Mover 9 para o arquivo final.
- **Reposição:** O 9 veio do run_0. O run_0 agora está esgotado.
- **Candidatos Atuais:** Nenhum. Todos os runs estão vazios.
- **Arquivo Final:** [1, 2, 3, 4, 5, 6, 7, 8, 9]

Os mesmos passos lógicos são realizados para os arquivos.

1. Gerenciamento de Memória vs. Performance (O Dilema do Buffer)

O coração do problema é o equilíbrio entre o uso de memória e a velocidade do processo.

- **Buffer Pequeno:** Se você alocar um buffer de memória muito pequeno, a Fase 1 (divisão) gerará um número enorme de arquivos temporários ("runs"). Isso torna a Fase 2 (mesclagem) extremamente lenta, pois o sistema operacional terá que abrir, ler e gerenciar centenas ou milhares de arquivos simultaneamente, causando uma sobrecarga de I/O (leitura/escrita em disco).
- **Buffer Grande:** Um buffer grande é ideal, pois cria menos runs, tornando a mesclagem muito mais rápida. No entanto, o objetivo é justamente lidar com a restrição de memória. Definir um buffer que seja "grande o suficiente" para ser eficiente, mas "pequeno o suficiente" para não estourar a RAM disponível, é um desafio de otimização crucial.

2. Complexidade do Gerenciamento de Arquivos e I/O

Diferente de um algoritmo em memória, a ordenação externa é dominada por operações de disco, que são ordens de magnitude mais lentas que operações de RAM.

- **Criação e Limpeza:** O código precisa criar, nomear, rastrear e, ao final, apagar de forma confiável um grande número de arquivos temporários. Uma falha no meio do processo pode deixar o disco cheio de "lixo". É preciso implementar um tratamento de erros robusto (como blocos try...finally) para garantir que a limpeza ocorra mesmo se a ordenação falhar.
- **Sobrecarga de Abertura/Fechamento:** Abrir e fechar arquivos repetidamente consome recursos do sistema operacional. Uma implementação ingênua que mescla apenas dois arquivos por vez (A e B geram C; C e D geram E; etc.) é muito ineficiente. A abordagem de mesclagem de k-vias (*k-way merge*), que mescla todos os k runs de uma só vez usando uma heap, é muito superior, mas também mais complexa de implementar.

3. Manipulação de Dados e Tipagem da Chave de Ordenação

Um arquivo CSV não tem tipos de dados intrínsecos; tudo é, a princípio, uma string.

- **Comparação Correta:** Se a chave de ordenação for um número (como um ID ou um preço), ele precisa ser convertido de string para um tipo numérico (inteiro ou float) antes de cada comparação. Caso contrário, a ordenação lexicográfica de strings resultará em erros (ex: "10" virá antes de "2"). O código precisa lidar com essa conversão de forma consistente.
- **Dados Mistos ou Ausentes:** E se uma linha tiver um valor não numérico na coluna que deveria ser numérica? Ou se o valor estiver vazio? O programa precisa ter uma lógica para lidar com esses casos de borda sem quebrar, seja tratando-os como um valor mínimo/máximo ou lançando um erro controlado.

4. Escalabilidade da Fase de Mesclagem

A performance da Fase 2 (mesclagem) depende criticamente de como se encontra o "próximo menor elemento" entre todos os runs.

- **Comparação Linear vs. Heap:** Se você tiver 1.000 runs, uma abordagem simples seria ler a primeira linha de cada um e iterar por essa lista de 1.000 linhas para encontrar a menor. Isso teria que ser repetido para cada linha do arquivo final, tornando o processo extremamente lento (complexidade O(k) para cada linha, onde k é o número de runs).
- **Implementação da Heap:** O uso de uma min-heap (como o módulo heapq do Python) resolve isso, encontrando o menor elemento em tempo logarítmico (O(log k)). No entanto, a implementação correta exige o armazenamento de tuplas na heap (chave, indice_do_run, linha_completa) para saber de qual arquivo repor a próxima linha, adicionando uma camada de complexidade ao código.

5. Manutenção do Formato e Metadados

O processo não pode corromper a estrutura do arquivo.

- **Cabeçalho:** O cabeçalho do arquivo original precisa ser lido e escrito corretamente no arquivo de saída final. Cada arquivo temporário também precisa ter seu próprio

cabeçalho para ser um CSV válido, mas esse cabeçalho precisa ser ignorado durante a fase de mesclagem.
- **Delimitadores e Citações:** Arquivos CSV podem ter peculiaridades, como vírgulas dentro de campos (que exigem aspas) ou diferentes delimitadores. Utilizar uma biblioteca CSV robusta é essencial, mas ainda assim é preciso garantir que a leitura e a escrita preservem a formatação original das linhas.

6. Lidando com Ordenação Descendente

A ordenação descendente não é tão simples quanto adicionar um reverse=True.

- **Na Ordenação Interna:** Na Fase 1, ordenar cada chunk em ordem descendente é fácil (a maioria das funções sort tem um parâmetro reverse).
- **Na Mesclagem com Heap:** A dificuldade surge na Fase 2. Bibliotecas padrão geralmente fornecem apenas uma min-heap (que retorna o menor item). Para simular uma max-heap (que retornaria o maior item) para a ordem descendente, é preciso usar um truque, como negar os valores numéricos antes de inseri-los na heap. Isso adiciona uma lógica condicional extra e funciona bem para números, mas exige uma abordagem diferente para strings (por exemplo, criando uma classe wrapper que inverte a lógica de comparação).

Conclusão

O uso de LLM para visualização geral do problema e, o auxílio do Github Copilot, nas sugestões de bibliotecas, funções, e limpeza de código via *autocomplete* tornaram o processo de desenvolvimento mais rápido, permitindo um enfoque maior no entendimento do algoritmo. Apesar do uso de ferramentas do tipo frearem o desenvolvimento e robustez da programação, temos como lado positivo a possibilidade de se explorar diversas formas de resolução de um problema já bem estabelecido, usufruindo da máxima de ***não reinventar a roda***.