Teste prático para vaga de desenvolvedor na Zukkin

Áquila Oliveira Souza -
www.linkedin.com/in/áquila-oliveira-souza-47a965234
Belo Horizonte - MG – Brasil

1. Introdução

Este script foi criado para resolver o teste prático da **Zukkin**. Ele aborda o desafio da ordenação externa de arquivos, fornecendo uma solução para ordenar grandes volumes de arquivos de entrada de qualquer tamanho. O objetivo é garantir a máxima eficiência tanto em termos de utilização de memória quanto de tempo de execução.

2. Método

O programa foi desenvolvido na linguagem PHP
• Sistema Operacional: Linux
• Processador: 11th Gen Intel(R) Core(TM) i3
• RAM: 8,00 GB (utilizável: 7,80 GB)

2.1 Estrutura e implementação

Para resolver este problema, optei pelo algoritmo de ordenação **merge sort**. Inicialmente, separo os 100 mil números em 100 blocos de mil números cada. Em seguida, aplico o algoritmo **merge sort** a cada um desses blocos para ordená-los individualmente. Após a ordenação dos blocos, realizo a mesclagem para gerar a solução final completamente ordenada.

2.2 Gerenciamento de memória

O código apresentado implementa um algoritmo de ordenação eficiente para arquivos grandes, utilizando técnicas para gerenciar a memória de forma eficaz.

- Divide o arquivo em blocos de tamanho fixo.
- Ordena cada bloco individualmente.
- Mescla os blocos ordenados em um único arquivo ordenado.

As técnicas de gerenciamento de memória incluem:

- Usar arrays como objetos temporários.
- Usar array_slice para sub-arrays.
- Processamento em blocos com arquivos temporários.
- Fechar arquivos abertos.

2.3 Método de ordenação

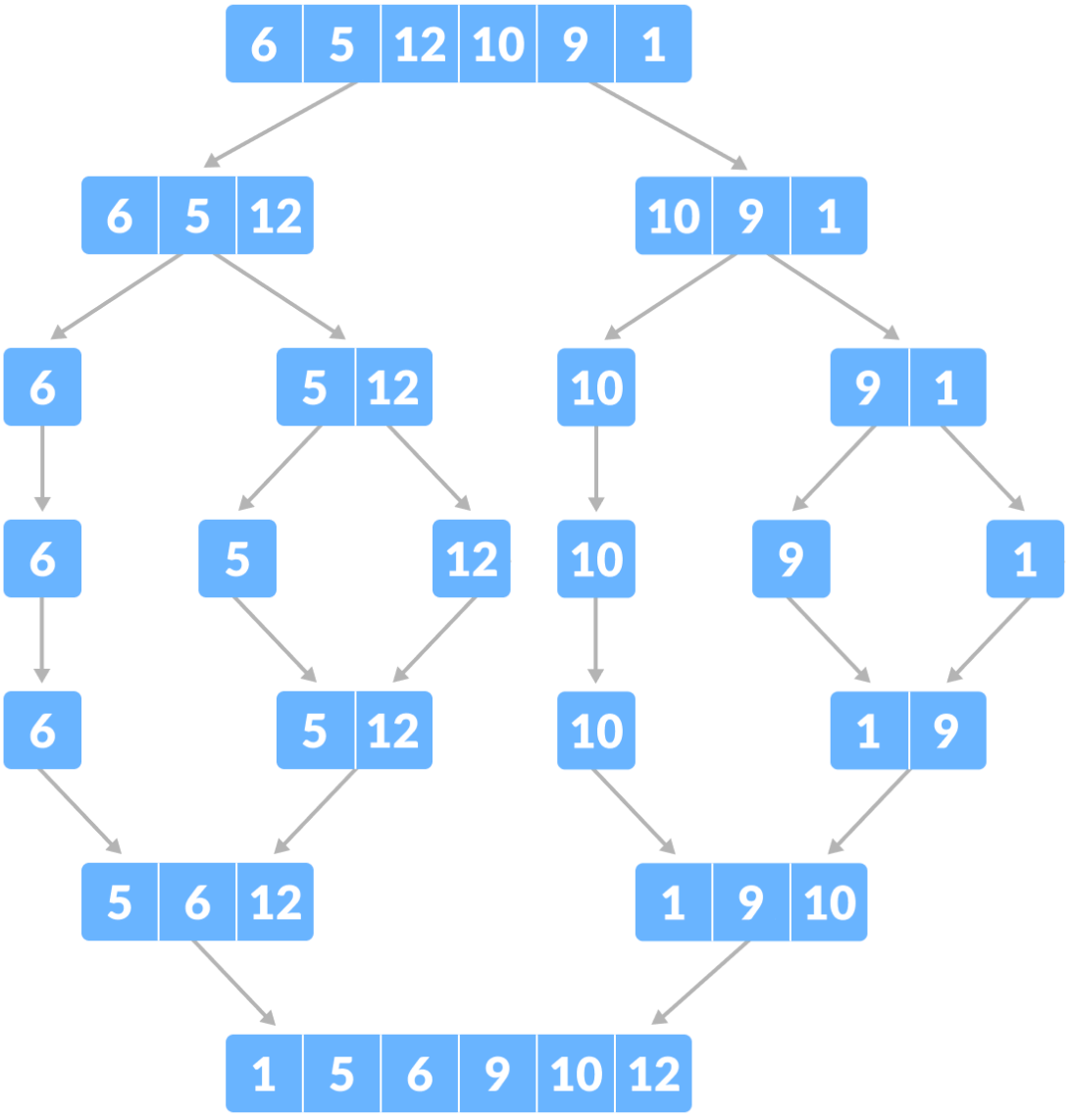
O método de ordenação escolhido por mim foi o **merge sort** para solucionar esse problema. Ele é um algoritmo eficiente que usa o conceito de divisão e conquista e se nossa missão for ordenar um array comparando seus elementos, do ponto de

vista assintótico, **n∗logn** é o nosso limite inferior, o que torna dentre os algoritmos de ordenação por comparação o mais eficiente. A vantagem desse método é que para o melhor caso, caso médio e pior caso a complexidade é **n∗log n.**

2.1 Características

- O algoritmo possui estabilidade, o que significa que nas trocas se os elementos forem iguais eles não são trocados de ordem.
- Não é adaptável, o que implica que independente da entrada ele executa o mesmo número de comparações.

2.1 Representação visual do método de ordenação



2.3 Funções

merge($left, $right, $option): Recebe dois arrays ordenados $left e $rigth e um parâmetro de opção que indica se a ordenação vai ser crescente ou decrescente e ele mescla esses dois arrays e em array ordenado de acordo com a opção especificada.

mergeSort($array, $option): Recebe um array não ordenado ($array) é um parâmetro de opção ($option) que indica a ordem de classificação. Divide recursivamente o array em metades, ordena cada metade e, em seguida, mescla as metades ordenadas usando o método merge(). Por fim, retorna o array ordenado.

sortBlocks($inputFile, $blockSize, $option): Este método lê um arquivo de entrada ($inputFile) em blocos, ordena cada bloco e grava os blocos ordenados em arquivos temporários. Utiliza o mergeSort() para ordenar cada bloco. Retorna um array contendo os nomes dos arquivos temporários que contêm os blocos ordenados.

mergeBlocks($blocks, $outputFile, $option): Utiliza uma abordagem de mesclagem de vários arquivos para mesclar os blocos ordenados. Abre os arquivos de bloco, lê uma linha de cada vez e mescla os dados em ordem no arquivo de saída final.

showMenu(): Este método exibe um menu para o usuário escolher a ordem de classificação (crescente ou decrescente) e solicita uma entrada ao usuário.

3. Análise de Complexidade

merge($left, $right, $option):

- Tempo de Execução: O tempo de execução depende do tamanho total das duas listas combinadas. Como a função itera sobre ambas as listas uma única vez, o tempo de execução é **O(n)**, onde **n** é o tamanho total das duas listas.
- Espaço: A função cria uma lista adicional para armazenar os elementos ordenados. Portanto, o espaço adicional necessário é **O(n)**, onde **n** é o tamanho total das duas listas combinadas.

mergeSort($array, $option):

- Tempo de Execução: O tempo de execução do algoritmo de merge sort é **O(n log n)**, onde **n** é o número de elementos no array. Isso ocorre porque o array é dividido ao meio recursivamente até que cada subarray contenha apenas um elemento, e depois é mesclado em ordem crescente.
- Espaço: O merge sort é um algoritmo de classificação "dividir para conquistar", que requer espaço adicional para armazenar as cópias dos

subarrays. Portanto, o espaço adicional necessário é **O(n)**, onde **n** é o número de elementos no array.

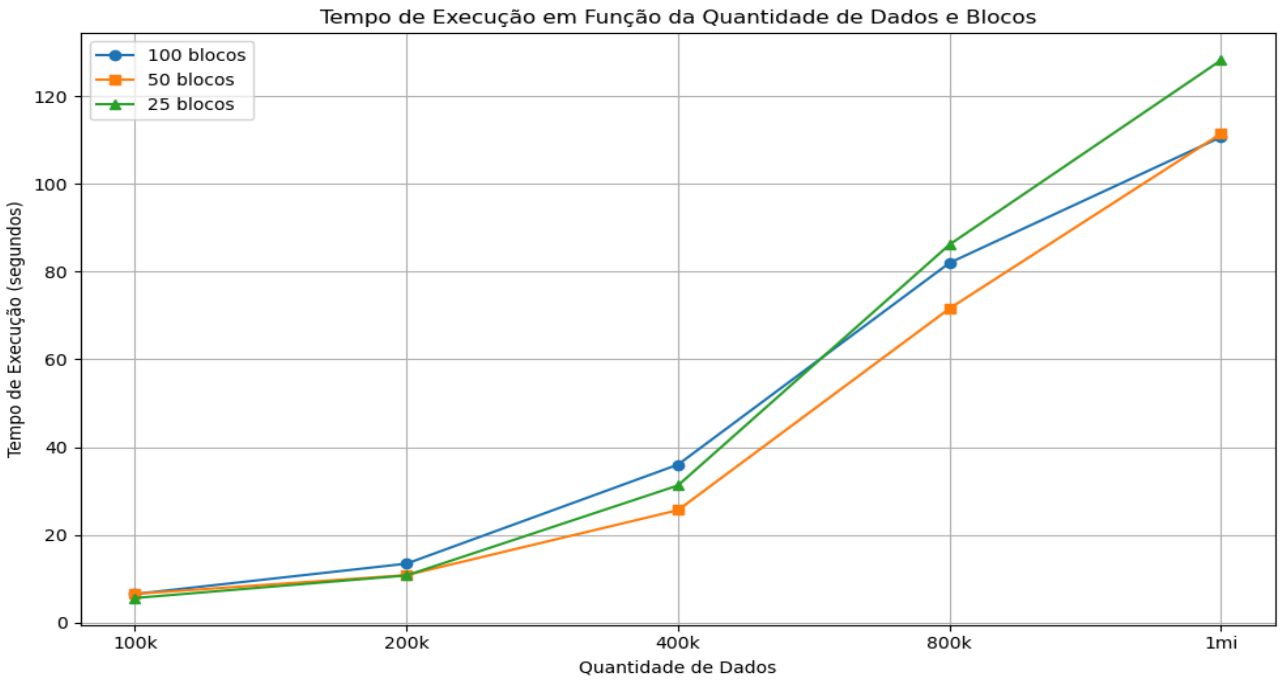
sortBlocks($inputFile, $blockSize, $option):

- Tempo de Execução: O tempo de execução depende do tamanho do arquivo de entrada e do tamanho do bloco. Se **n** for o número total de elementos no arquivo de entrada e **k** for o número de blocos a serem criados, o tempo de execução seria **O(n log n + k · log k)**. Isso ocorre porque, para cada bloco, é realizado um mergesort com complexidade **O(tamanho do bloco · log(tamanho do bloco))**, e então esses blocos ordenados são mesclados em **O(n log k)**.
- Espaço: O espaço adicional necessário para o **sortBlocks** depende do tamanho do bloco e do número de blocos a serem criados. Se **k** for o número total de blocos e **m** for o tamanho do bloco, o espaço adicional necessário seria **O(k · m)**.

mergeBlocks($blocks, $outputFile, $option):

- Tempo de Execução: O tempo de execução depende do número de elementos nos blocos. Se **n** for o número total de elementos nos blocos, o tempo de execução seria **O(n log n),** pois a mesclagem dos blocos é semelhante ao merge sort.
- Espaço: O espaço adicional necessário é **O(n)**, onde n é o número total de elementos nos blocos.

4. Análise Experimental



Com base nos resultados da ordenação para diferentes volumes de dados, pode-se tirar algumas conclusões:

Eficiência em relação ao número de blocos:
À medida que o número de blocos aumenta, o tempo de execução tende a diminuir. Isso sugere que dividir o conjunto de dados em um maior número de blocos pode melhorar o desempenho geral do algoritmo de ordenação externa.

Eficiência em relação ao tamanho dos blocos:
Quando o tamanho dos blocos é aumentado, o tempo de execução geralmente diminui. Isso indica que trabalhar com blocos maiores pode ser mais eficiente em termos de tempo de execução. No entanto, existem algumas exceções, como o caso em que 100 blocos de 8000 têm um tempo de execução maior do que 50 blocos de 16.000 para 800k de dados. Isso ocorre porque, para blocos muito grandes de números, a eficiência é menor, o que gera esse comportamento.

Escalabilidade:
Os resultados indicam que o algoritmo de ordenação externa é escalável e pode lidar com volumes crescentes de dados. O aumento do número de blocos e o tamanho dos blocos geralmente resultam em tempos de execução razoáveis, mesmo para conjuntos de dados maiores, como 1 milhão de registros.

5. Conclusões

Em resumo, os resultados mostram que o algoritmo de ordenação externa é capaz de lidar eficientemente com diferentes volumes de dados, mas a eficiência pode ser influenciada por fatores como o número e o tamanho dos blocos, bem como as características específicas do conjunto de dados e do ambiente de execução.

6. Instruções para execução

- Definir o tamanho que cada bloco vai possuir (linha 130 do script)
- Ter php instalado na sua máquina
- definir o caminho do input (linha 128 do script)
- rodar no terminal: php script.ph e escolher a opção de ordenação