Ministério da Educação
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Campus Campo Mourão
Curso de Bacharelado em Ciência da Computação

Aluno: Artur Marchi Pacagnan **RA:** 2149273
Aluno: Matheus Patriarca Santana **RA:** 2171481

Prof. Dr. Juliano Henrique Foleiss

Trabalho Prático 1: Ordenação de Arquivos Grandes de Registros

Introdução:

Devido ao tamanho do arquivo, pode acontecer de não conseguirmos carregá-lo totalmente em memória primária (RAM). Desse modo, como um meio para a ordenação de arquivos grandes podemos utilizar a Ordenação externa. Neste projeto, temos um arquivo de entrada composto por uma quantidade de N registros de 1024 bytes. Para a sua ordenação utilizaremos a Ordenação de Intercalação em K-vias, ao início deste tipo de ordenação, é necessário a requisição dos dados do arquivo inicial, o qual serão divididos através de buffers de forma ordenada em arquivos temporários. Neste caso, todo tipo de ordenação será feito comparando o "ID" correspondente a cada registro "ITEM_VENDA".

Com os arquivos temporários ordenados, utiliza-se os buffers de entrada para realizar o consumo dos respectivos menores registros para salvar no buffer de saída. Esse processo é denominado de merge sort, o qual necessita de arrays ordenados e o tamanho dos buffers de entrada são definidos na função principal no início do processo.

Após a ordenação através do k-way merge, esses elementos serão introduzidos no buffer de saída. Quando esse buffer fica cheio, tem todo o seu conteúdo despejado no arquivo de saída. Esse processo se repete até todos os elementos dos arquivos temporários serem consumidos pelos buffers de entrada e comparados na função que busca o próximo menor registro. Ao fim desse processo, o arquivo de saída estará com o conteúdo do arquivo de entrada de forma ordenada.

Funções Utilizadas:

"Ordenacao.c"

Compare
A função "compare" é uma função auxiliar para a função "qsort", própria da biblioteca "<stdlib.h>", que verifica qual elemento possui um "ID" maior entre dois elementos "ITEM_VENDA", retornando o resultado da subtração do id do ITEM_VENDA A com o id ITEM_VENDA de B.
Menor Elemento
A função "Menor Elemento" é uma função auxiliar para a função de "Intercalacao" e seu objetivo é retornar a posição do menor ID em um vetor de buffers de entrada. Nessa função, no seu primeiro loop é encontrado o primeiro próximo elemento de um buffer ativo (estado = 1) que é atribuído a uma variável menor. O segundo loop busca o próximo elemento de cada buffer ativo, comparando com o

menor elemento. Caso algum dos buffers tenha o elemento próximo menor do que o contido na variável menor elemento, realizamos a troca. Ao fim, é retornado a posição do menor elemento id, que corresponde ao próximo de um determinado buffer.

Intercalação

A função “Intercalacao” possui rotinas que representam o processo de consumação dos subsequentes menores elementos. Então, existe um loop que tem como critério de parada o número de elementos do arquivo de entrada para que todos os registros do arquivo de entrada sejam inseridos no arquivo de saída ordenadamente. Nesse processo, há a busca pelo menor elemento entre os N buffers de entrada, a inserção desse elemento no buffer de saída, a consumação do menor elemento e o despejamento caso o buffer de saída esteja cheio.

“ManutencaoArquivo.c”

Destruir_Arquivo

A função Destruir_Arquivo recebe o nome do arquivo e retorna 0 quando o arquivo foi removido com sucesso e -1 quando não foi possível removê-lo.

Conta_digitos

A função “Conta_digitos” retorna a quantidade de dígitos de um número inteiro e é utilizada para definir o tamanho da variável do nome do arquivo.

Ler_Registros

A função “Ler_Registros” faz a leitura de um arquivo com N registros com o objetivo de salvar o vetor de itens dentro de um buffer de entrada.

Dividir_Arquivo

A função “Dividir_Arquivo” tem como objetivo a criação das partições dos arquivos que serão consumidos pelos buffers de entrada. Então, enquanto o arquivo de entrada não chegar no fim, há uma leitura da quantidade de registros por partição, seguido da ordenação desses registros coletados e a inserção em uma arquivo partição, o qual servirá para os buffers de entrada consumir.

Calcular_Tamanho_Arquivo

A função “Calcular_Tamanho_Arquivo” é responsável por calcular o tamanho do arquivo. Na sua chamada é passado o nome do arquivo e ela tem como retorno o tamanho deste arquivo.

Ordenacao_Externa

A função “Ordenacao_Externa” é responsável por ordenar um arquivo de entrada composto de N registros. Essa ordenação é feita através do campo id.

“TAD_bufferEntrada.c”

Criar_BufferEntrada

A função “Criar_BufferEntrada” é responsável por realizar a criação dos buffers de entrada. Para isso, ela aloca dinamicamente as variáveis necessárias. Inicia o estado_buffer em 1, destacando que o buffer está ativo e pode realizar operações futuras.

Proximo_BufferEntrada

A função “Proximo_bufferEntrada” é responsável por retornar o próximo menor elemento contido no buffer de entrada.

Consumir_BufferEntrada

A função Consumir_BufferEntrada verifica se o buffer de entrada está vazio. Caso esteja vazio, e também não tenha consumido todos os registros do arquivo temporário, ele realiza a requisição de novos dados. Caso o buffer tenha consumido todo o arquivo temporário, o seu estado recebe o valor 0

Vazio_BufferEntrada
A função “Vazio_BufferEntrada” retorna 1 se o buffer de entrada não tem mais conteúdos a serem consumidos. Caso ele tenha conteúdos a serem consumidos, ele retorna 0.

“TAD_bufferSaida.c”

Criar_BufferSaida
A função “Criar_BufferSaida” é responsável por realizar a criação dos buffers de saída. Para isso, ela aloca dinamicamente as variáveis necessárias. O seu estado não tem relevância para o processo.

Inserir_BufferSaida
A função “Inserir_BufferSaida” é responsável por realizar a inserção do menor elemento no buffer de saída.

Despejar_BufferSaida
A função “Despejar_BufferSaida” é responsável por realizar o despejo dos dados do buffer de saída, no arquivo de saída. Isso se dá apenas após o buffer de saída estiver cheio.

Destruir_BufferSaida
A função “Destruir_BufferSaida” é responsável por desalocar a memória das estruturas alocadas, o vetor de itens e o próprio buffer.

Descrição da atividade:

Para a realização do projeto, utilizamos o Visual Studio Code para realizar a codificação e para a contribuição conjunta dos membros ao projeto, utilizamos uma ferramenta chamada live share, a qual, permite aos membros do projeto realizar colaborações no código em tempo real. Deste modo, tentamos realizar todo o processo em conjunto, debatendo ideias de como poderíamos executar e entrando em um consenso final para codificar.

Dividimos os arquivos do projeto a fim de organizar funções específicas para podermos utilizar em outros contextos quando necessário, os arquivos criados são: “ManutencaoArquivo.c”, “Ordenacao.c”, “TAD_bufferEntrada.c” e “TAD_bufferSaida.c”.

Tentamos deixar a modularização simples e intuitiva, abstraindo funções e não as deixando sobrecarregadas. Para os buffers de entrada e saída e as suas respectivas funções, criamos a “TAD_bufferEntrada.c” e uma “TAD_bufferSaida.c”. Basicamente, essas TADS realizam todo tipo de processo referente aos buffers de entrada e saída. Para os arquivos de big file, que realiza a criação do arquivo de entrada com N registros, decidimos não realizar mudanças ou sobrecarregá-lo com novas funções.

Também foi criado um arquivo de ordenação, neste arquivo está contido todas as funções referentes ao processo de Ordenação Externa. Como por exemplo, a função “compare” para a comparação realizada no qsort da biblioteca stdlib e também a função de Intercalação e Menor Elemento, responsáveis pela execução do k-way merge.

Além disso, um arquivo de manutenção do arquivo também foi criado. Como havia muitas operações com arquivo, como por exemplo a leitura de registros, criação de arquivos temporários, divisão dos arquivos, cálculo do tamanho do arquivo e até mesmo a ordenação externa, decidimos criar um arquivo responsável apenas para essas operações. Dessa forma, nenhum dos outros arquivos do projeto ficariam sobrecarregados com funções que se referem em sua maioria a operação com arquivos. Como já tínhamos o arquivo entrada.dat gerado pelas funções do arquivo big file, precisávamos realizar apenas o processo de Ordenação Externa, realizando as respectivas ordenações através do atributo id do registro ITEM_VENDA.

Diante da complexidade do projeto, decidimos iniciar pela especificação 3. Isso se deu devido ao fato dessa função ser uma das principais funções para o processamento de todo o projeto, recebendo os dados iniciais por parâmetros na main, realizando as operações necessárias encontradas na especificação 4, a Intercalação, e posteriormente finalizando todo o processo, destruindo o que for necessário. Foi importante iniciar pela função da Ordenação Externa para termos uma visão geral de todo o processo. Dessa forma, foi mais fácil entender as operações necessárias para a execução de todo o processo, principalmente a TAD_bufferEntrada e TAD_bufferSaida.

Como precisávamos testar, voltamos às especificações 1 e 2 e colocamos em prática as funções da TAD_bufferEntrada e TAD_bufferSaida. Neste momento, percebemos que alterações eram importantes para o controle de algumas operações. Para isso, criamos uma estrutura denominada BUFFER.

```
typedef struct BUFFER{
 int estado_buffer;
 int quantidade_registros;
 int quantidade_registros_total;
 int quantidade_consumidos;
 int quantidade_consumidos_total;
 ITEM_VENDA* vetor_itens;
} BUFFER;
```

O estado_buffer é responsável pelo desativamento da estrutura. Quando criado, ele é setado em 1, destacando que o buffer pode ser utilizado para a realização de operações. Quando setado em 0, está desativado. Esse atributo acaba sendo muito importante na comparação dos mínimos, já que nos buffer setados em zero não precisamos realizar a comparação.

A quantidade_registros faz a contabilização de todos os registros que o buffer de entrada contém. A quantidade_registros_total é a contabilização de todos todos os registros que o buffer de entrada pode conter ao longo do processo. Já a quantidade_consumidos e quantidade_consumidos_total é utilizada para fazermos o controle do fluxo de dados no buffer. Quando a quantidade_registros é igual a quantidade_consumidos, devemos fazer novas requisições de dados. Diferentemente de quando a quantidade_consumidos_total for igual a quantidade_registros_total, em que não temos mais elementos que podem serem inseridos no buffer de entrada, dessa forma ele pode parar de fazer requisições e ser desativado.. Após a execução do

TAD_bufferEntrada e do controle dos buffers de entrada foi possível realizar a maioria das operações e testá-las.

Assim como para os buffer de entrada, nos buffers de saída esses atributos da estrutura BUFFER também são importantes. É importante destacar que o estado_buffer não tem uma atuação relevante para os buffers de saída.

Na função de Ordenação Externa foi realizado todo o processo solicitado. Nela estava contida a maioria dos cálculos solicitados na especificação 4, calculando o "k" (Quantidade de buffers), o tamanho do arquivo de entrada, a quantidade de registros contida em cada um dos buffers de entrada, a quantidade de registros contida no arquivo de entrada, assim como também a quantidade de registros contida em cada uma das partições.

Após a realização dos cálculos na função Ordenação Externa, é realizado a divisão do arquivo de entrada em N buffers, que realizam a ordenação dos registros do arquivo utilizando o campo id. Para isso, foi utilizado a função qsort da biblioteca stdlib.h. E para a comparação dos elementos através do campo id, implementamos uma função que comparava esses elementos. Após essa ordenação, os elementos passavam por um buffer e eram inseridos em arquivos temporários. Para o nome desses arquivos eram passados números em sequência.

```
int compare (const void * a, const void * b){
 ITEM_VENDA *A = (ITEM_VENDA *)a;
 ITEM_VENDA *B = (ITEM_VENDA *)b;
 return ( A->id - B->id );
}
```

Para esta função "Dividir_Arquivo", que realiza a divisão do arquivo de entrada. Enquanto o arquivo de entrada não for lido completamente, há a leitura com a quantidade de registros do arquivo particionado, o uso do QuickSort para ordenar essa partição e a criação do arquivo particionado, respectivamente.

Depois da divisão do arquivo de entrada em N arquivos temporários e da criação dos buffers de entrada e saída, realizamos o processo de Intercalação(k-way merge). Para essa função de intercalação há uma função auxiliar denominada menor_elemento, que realiza o processo da argMin. Nessa função de menor elemento é encontrado o menor elemento do próximo buffer disponível, ou seja, setado em 1. Após isso, esse buffer tem o seu id comparado com outros próximos elementos de buffers também ativos. Encontrando o menor elemento, ele pode ser inserido no buffer de saída e consumido no buffer de entrada.

Além disso, foi criada uma função ler registros que executa a leitura desses arquivos temporários para os buffers de entrada, que armazenam apenas parte do conteúdo dos arquivos

temporários. Isso se dá devido ao fato de não podermos sobrecarregar a memória primária (RAM), que com buffer de entrada não tão grandes, acabam exigindo menos. Quando o buffer de saída é cheio, ele tem o seu conteúdo despejado no arquivo de saída.

Resultados:

Foi encontrado um certo problema na execução dos maiores casos. No caso de b = 256 mb e s = b/2 da tabela de 921600 e na tabela de 1572864 os processos acabavam não executando. Para a execução correta tivemos que desalocar a memória do arquivo de bigfile, em que alocava memória para a criação do array de registros a ser inserido no arquivo de entrada.

		S		
		B/8	B/4	B/2
B	8388608(8mb)	**9.242**	**5.816000**	**10.298000**
	16777216(16mb)	**8.545000**	**7.020000**	**10.061000**
	33554432(32mb)	**5.972000**	**7.064000**	**6.357000**

Quantidade de registros = 256000

		S		
		B/8	B/4	B/2
B	16777216(16mb)	**19.592000**	**30.215000**	**18.865000**
	33554432(32mb)	**17.328000**	**16.939000**	**19.324000**
	67108864(64mb)	**19.728000**	**16.797000**	**21.157000**

Quantidade de registros = 512000

		S		
		B/8	B/4	B/2
B	67108864(64mb)	**51.841000**	**36.814000**	**44.334000**
	134217728(128mb)	**35.884000**	**42.469000**	**36.096000**
	268435456(256mb)	**33.982000**	**38.470000**	**36.690000**

Quantidade de registros = 921600

		S		
		B/8	B/4	B/2
B	67108864(64mb)	**68.399000**	**81.938000**	**94.424000**
	134217728(128mb)	**77.944000**	**72.323000**	**79.396000**
	268435456(256mb)	**72.865000**	**78.329000**	**82.996000**

Quantidade de registros = 1572864

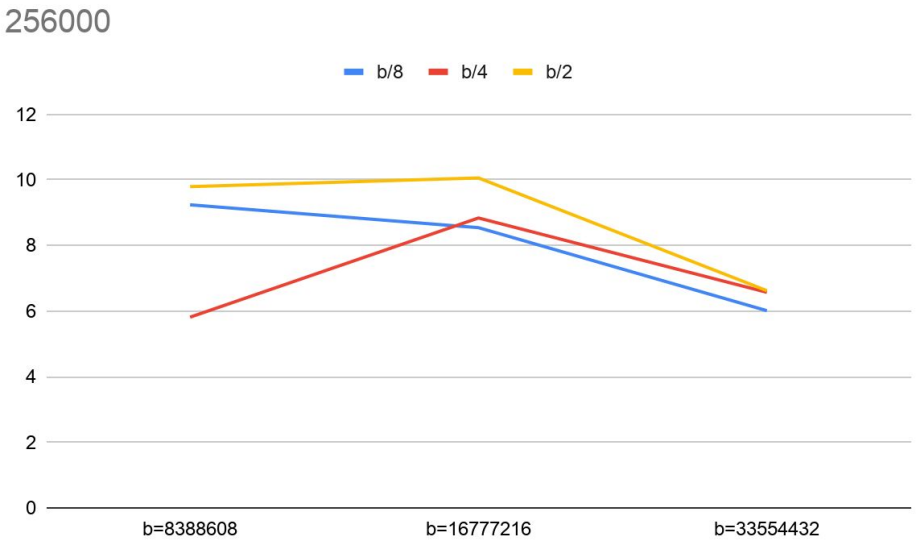

Figure 1: Tempo De Execução para 256000

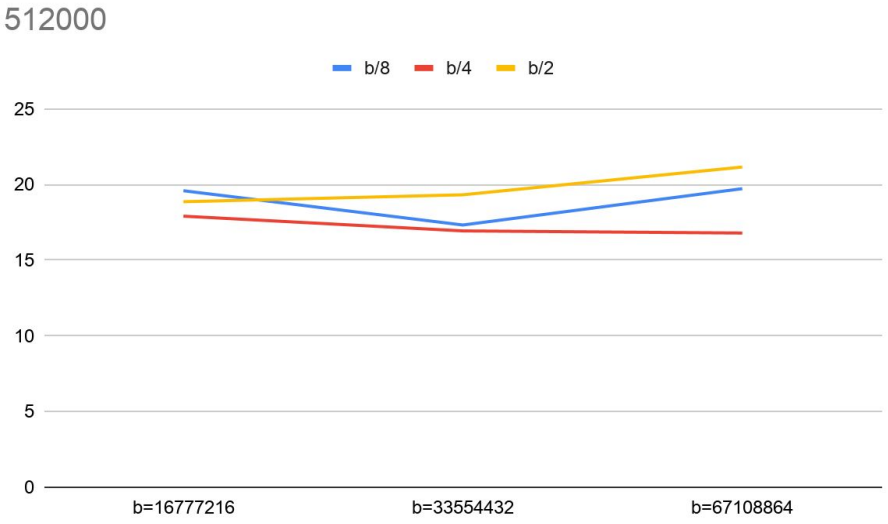

Figure 2: Tempo De Execução para 512000

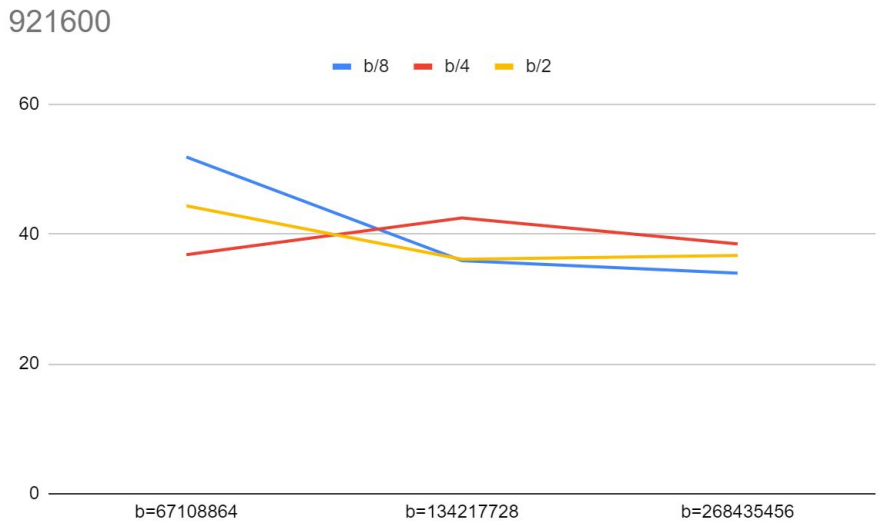

Figure 3: Tempo De Execução para 921600

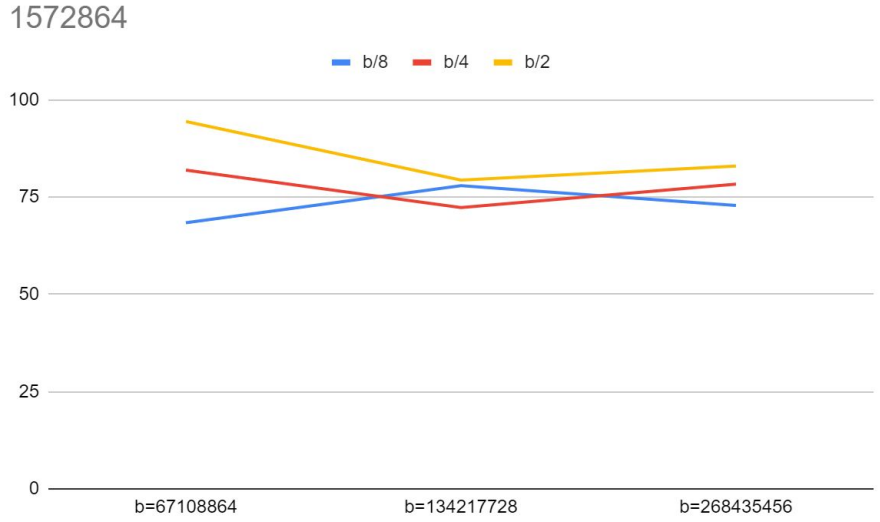

Figure 4: Tempo De Execução para 1572864

Conclusão:

Portanto, com a descrição do projeto e com os dados analisados, podemos entender mais claramente como funciona a ordenação externa e como ela foi colocada em prática neste projeto. Além disso, conseguimos entender o funcionamento específico de cada uma das funções que compõem esse projeto e da sua devida necessidade para o resultado final.

Com base nos dados da tabela apresentados, realizados no HDD, percebe-se que há uma tendência do tempo de execução em se manter constante. Com uma análise mais minuciosa, no caso do tamanho do buffer de saída “b/2” possuímos o pior resultado comparado aos outros casos.