Ordenação de Arquivos: Perguntas e Respostas

Cauã Borges Faria (834437)

6. Quais as alternativas gerais temos para ordenar um arquivo? Quais fatores devem ser levados em consideração para a escolha da melhor alternativa?

Existem duas abordagens principais para ordenar arquivos:

- **Ordenação Interna:** Isso acontece quando o arquivo inteiro cabe na **memória RAM**. Nesse caso, você pode usar qualquer algoritmo de ordenação que já conhece (como Quicksort, Mergesort, Heapsort, etc.) diretamente nos dados na memória.
- **Ordenação Externa:** Esta é a opção quando o arquivo é **grande demais** para caber por completo na memória RAM. Aqui, a ordenação precisa ser feita em etapas, lendo partes do arquivo para a memória, ordenando-as e gravando-as de volta em disco, para depois intercalar essas partes ordenadas.

Para escolher a melhor alternativa, você deve considerar os seguintes fatores:

- **Tamanho do Arquivo:** Esse é o fator mais crítico. Se o arquivo é pequeno o suficiente para a memória, a ordenação interna é a mais eficiente. Se não, a externa é a única opção.
- **Memória Disponível:** A quantidade de RAM que você tem à disposição impacta diretamente se a ordenação interna é viável e, na externa, o tamanho dos blocos que podem ser processados.
- **Tempo de Processamento:** Ordenações internas são geralmente muito mais rápidas devido ao acesso direto à memória. Ordenações externas são mais lentas por envolverem muitas operações de I/O (leitura/escrita em disco).

- **Tipo de Acesso (Sequencial vs. Aleatório):** A forma como você precisa acessar os dados pode influenciar a escolha do algoritmo de ordenação interna.

---

7. Analise o uso de algoritmos de ordenação interna para ordenar um arquivo. Por que essa alternativa não é adequada, se aplicados nos dados armazenados no próprio arquivo?

Algoritmos de ordenação interna são projetados para trabalhar com dados que estão completamente na **memória principal**. Aplicá-los diretamente aos dados armazenados no próprio arquivo, sem antes carregar para a memória, não é adequado por várias razões:

- **Acesso Lento ao Disco:** Algoritmos internos assumem acesso rápido e aleatório aos elementos. O acesso a dados em disco é ordens de magnitude mais lento do que o acesso à memória RAM. Cada troca ou comparação exigiria uma operação de I/O, tornando o processo extremamente ineficiente e demorado.
- **Custo de I/O:** Cada leitura ou escrita no disco tem um custo alto. Um algoritmo interno que faz muitas trocas resultaria em uma quantidade gigantesca de operações de disco, o que "mataria" o desempenho.
- **Complexidade de Implementação:** Adaptar um algoritmo de ordenação interna para trabalhar diretamente com acesso a disco seria complexo e ineficiente. Você teria que gerenciar blocos de leitura, escritas parciais, e o desempenho ainda seria terrível.

---

8. Caso todos os registros de um arquivo couberem na memória, como você ordenaria esse arquivo? Por que?

Se todos os registros de um arquivo couberem na memória, a forma mais eficiente de ordená-lo seria:

1. **Ler o arquivo inteiro para a memória:** Carregue todos os registros do arquivo para uma estrutura de dados na memória RAM (por exemplo, um array ou uma lista de objetos).
2. **Ordenar os dados na memória:** Utilize um **algoritmo de ordenação interna eficiente**, como **Quicksort** ou **Mergesort**, para ordenar os registros dentro da memória.
3. **Gravar os dados ordenados de volta no arquivo:** Após a ordenação, sobrescreva o arquivo original ou crie um novo arquivo com os registros já ordenados.

Por que? Porque a memória RAM oferece acesso muito mais rápido aos dados do que o disco. Ao trazer tudo para a memória, você pode aproveitar a alta velocidade dos algoritmos de ordenação interna, minimizando as operações de I/O, que são as mais lentas. Quicksort e Mergesort são excelentes escolhas por sua eficiência (O(NlogN) no caso médio e pior, respectivamente), sendo amplamente utilizados em cenários de ordenação interna.

---

9. Faça um algoritmo geral para a ordenação de um arquivo, carregando ele inteiro na memória (supondo que caiba).

```
ALGORITMO OrdenarArquivoEmMemoria:
    ENTRADA:
        caminho_arquivo_entrada: String (caminho do arquivo a ser
ordenado)
        caminho_arquivo_saida: String (caminho para o arquivo
ordenado, pode ser o mesmo do entrada)
        funcao_comparacao: Função (define como comparar dois
registros)

    VARIAVEIS:
        registros: Lista de Registros (estrutura para armazenar os
registros do arquivo)

    INICIO:
        // 1. Abrir o arquivo de entrada para leitura
        ABRIR_ARQUIVO(caminho_arquivo_entrada, MODO_LEITURA)
```

```
        // 2. Ler todos os registros do arquivo para a memória
        ENQUANTO NÃO_FIM_DE_ARQUIVO:
            LER_REGISTRO_DO_ARQUIVO(registro_atual)
            ADICIONAR_NA_LISTA(registros, registro_atual)
        FIM_ENQUANTO

        FECHAR_ARQUIVO(caminho_arquivo_entrada)

        // 3. Ordenar a lista de registros na memória usando um
algoritmo de ordenação interna
        // Exemplo: Usando um Quicksort
        ORDENAR_LISTA(registros, funcao_comparacao,
ALGORITMO_QUICKSORT)

        // 4. Abrir o arquivo de saída para escrita (sobrescreve se
já existir)
        ABRIR_ARQUIVO(caminho_arquivo_saida, MODO_ESCRITA)

        // 5. Escrever os registros ordenados de volta no arquivo
de saída
        PARA CADA registro EM registros:
            ESCREVER_REGISTRO_NO_ARQUIVO(registro,
caminho_arquivo_saida)
        FIM_PARA

        FECHAR_ARQUIVO(caminho_arquivo_saida)

        RETORNAR SUCESSO
    FIM.
```

---

10. Caso os registros de um arquivo não couberem todos na memória, mas caso for possível construir um índice sobre esse arquivo, que caiba inteiro na memória, seria possível ordenar o índice e criar um arquivo ordenado seguindo o índice. Quais os problemas desse método?

Sim, essa é uma estratégia interessante e comumente usada! Se o índice (chave do registro + Record Reference Number - RRN, que é a posição do registro no arquivo) couber na memória, você pode ordená-lo e usar essa ordem para acessar os registros. No entanto, há problemas importantes:

- **Muitas Operações de I/O Aleatórias:** O principal problema é o **acesso aleatório intensivo ao disco**. Para cada registro a ser escrito no arquivo de saída ordenado, você precisaria:
    a. Ler a próxima entrada do índice ordenado (chave, RRN).
    b. Usar o RRN para buscar o registro correspondente no arquivo original (uma leitura aleatória).
    c. Escrever esse registro no novo arquivo ordenado.

    Isso resulta em um grande número de operações de leitura/escrita em posições não sequenciais do disco, o que é extremamente lento e ineficiente devido ao tempo de busca (seek time) e à latência rotacional do disco.
- **Fragmentação do Arquivo de Saída:** O novo arquivo gerado pode não ser contíguo no disco, o que pode impactar futuras leituras sequenciais.
- **Espaço em Disco Adicional:** Você precisa de espaço para o arquivo original e para o novo arquivo ordenado, pelo menos temporariamente.

Embora conceitualmente simples, a penalidade de desempenho de I/O aleatório geralmente torna essa abordagem impraticável para arquivos muito grandes, onde a ordenação por intercalação se mostra superior.

---

11. Faça um algoritmo geral para a ordenação de um arquivo, criando um índice (chave, RRN) em memória (supondo que caiba), e usando o índice para a geração de um arquivo ordenado.

```
ALGORITMO OrdenarArquivoPorIndice:
    ENTRADA:
        caminho_arquivo_entrada: String
        caminho_arquivo_saida: String
        funcao_extrair_chave: Função (extrai a chave de ordenação
de um registro)
        funcao_comparacao_indice: Função (compara duas entradas de
índice)

    VARIAVEIS:
        indice: Lista de Tuplas (chave, RRN) (estrutura para
armazenar o índice na memória)
        RRN_atual: Inteiro = 0

    INICIO:
        // 1. Abrir o arquivo de entrada para leitura
        ABRIR_ARQUIVO(caminho_arquivo_entrada, MODO_LEITURA)

        // 2. Construir o índice na memória
        ENQUANTO NÃO_FIM_DE_ARQUIVO:
            LER_REGISTRO_DO_ARQUIVO(registro_atual)
            chave = funcao_extrair_chave(registro_atual)
            ADICIONAR_NA_LISTA(indice, TUPLA(chave, RRN_atual))
            INCREMENTAR(RRN_atual) // Incrementa o RRN para o
próximo registro
        FIM_ENQUANTO

        FECHAR_ARQUIVO(caminho_arquivo_entrada)

        // 3. Ordenar o índice na memória
        ORDENAR_LISTA(indice, funcao_comparacao_indice,
ALGORITMO_QUICKSORT) // Ou outro algoritmo de ordenação interna
```

```
        // 4. Abrir o arquivo de entrada novamente (ou manter
aberto se a linguagem permitir)
        // e o arquivo de saída para escrita
        ABRIR_ARQUIVO(caminho_arquivo_entrada, MODO_LEITURA) //
Reabre para ler registros aleatoriamente
        ABRIR_ARQUIVO(caminho_arquivo_saida, MODO_ESCRITA)

        // 5. Gerar o arquivo ordenado usando o índice
        PARA CADA entrada_indice EM indice:
            (chave_ordenada, RRN_original) = entrada_indice

            // Buscar o registro correspondente no arquivo original
usando o RRN
            POSICIONAR_PONTEIRO_ARQUIVO(caminho_arquivo_entrada,
RRN_original * TAMANHO_REGISTRO) // Assumindo registros de tamanho
fixo
            LER_REGISTRO_DO_ARQUIVO(registro_ordenado,
caminho_arquivo_entrada)

            // Escrever o registro lido no arquivo de saída
            ESCREVER_REGISTRO_NO_ARQUIVO(registro_ordenado,
caminho_arquivo_saida)
        FIM_PARA

        FECHAR_ARQUIVO(caminho_arquivo_entrada)
        FECHAR_ARQUIVO(caminho_arquivo_saida)

        RETORNAR SUCESSO
    FIM.
```

---

12. Como funciona a ordenação de arquivos por intercalação? Quais as principais fases? Faça um diagrama e explique o funcionamento. Em quais circunstâncias essa estratégia deve ser utilizada?

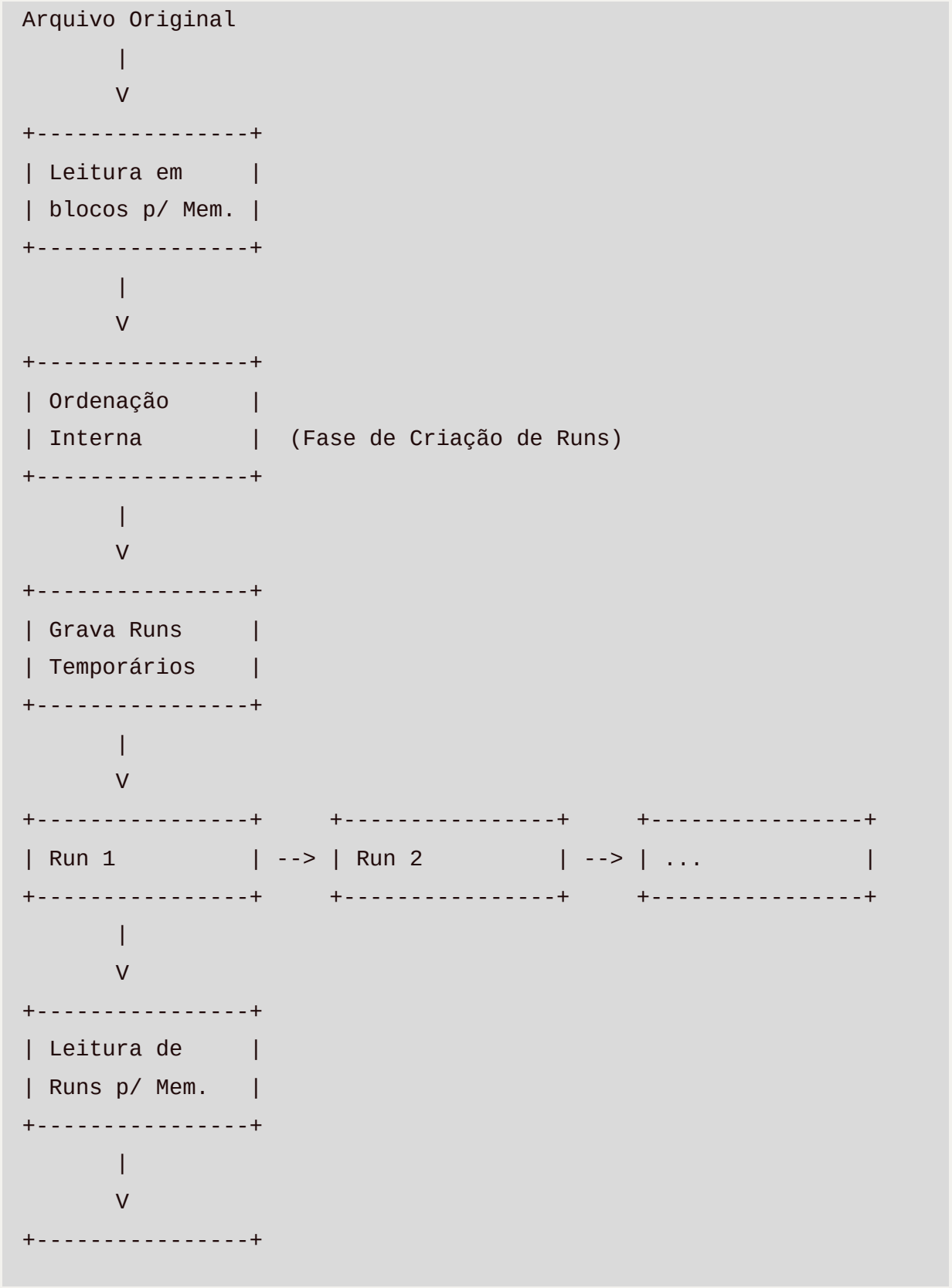
A **ordenação de arquivos por intercalação (Merge Sort Externo)** é a estratégia mais comum e eficiente para ordenar arquivos que não cabem completamente na memória principal. Ela é baseada no princípio de dividir para conquistar.

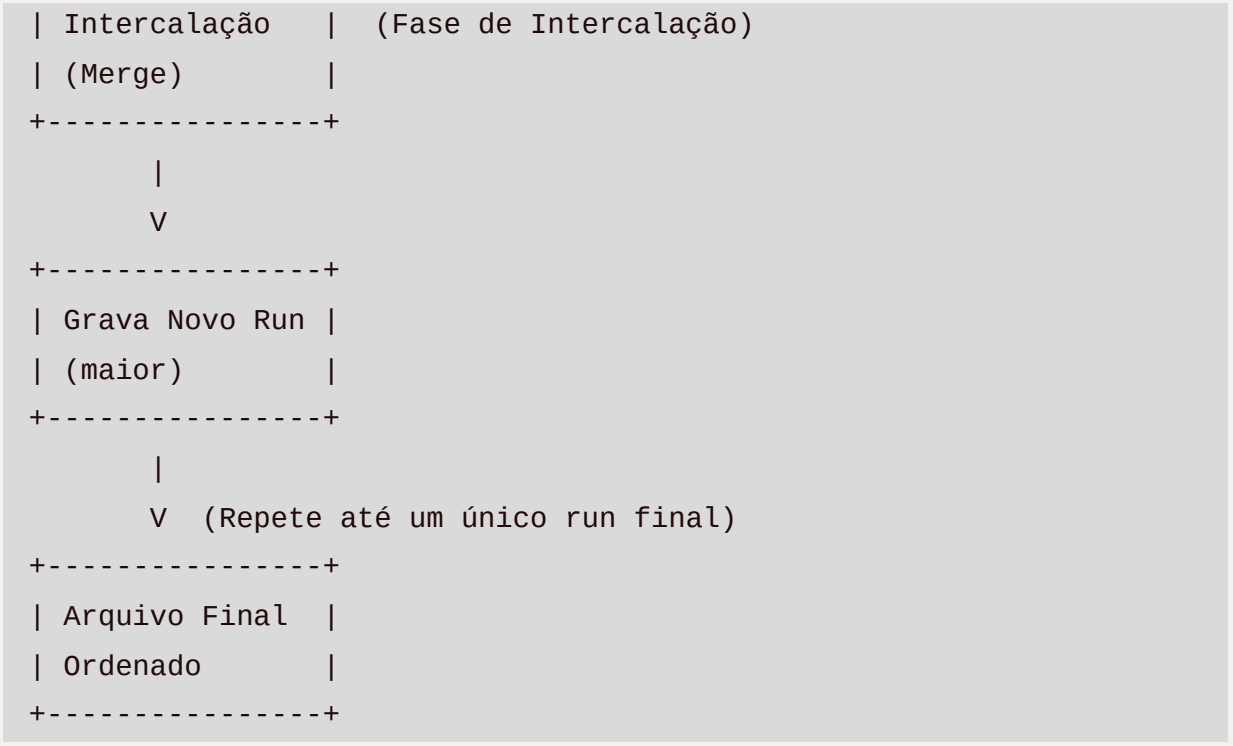
Como funciona e suas principais fases:

A ordenação por intercalação geralmente envolve duas fases principais:

1. **Fase de Criação de Runs (Ordenação Inicial):**
    - O arquivo original é lido em **blocos** que cabem na memória disponível.
    - Cada bloco é **ordenado internamente** usando um algoritmo eficiente (como Quicksort ou Heapsort).
    - Esses blocos ordenados são gravados em arquivos temporários separados no disco. Cada arquivo temporário é chamado de "run" ou "sub-arquivo ordenado".
2. **Fase de Intercalação (Merge):**
    - Os "runs" (sub-arquivos ordenados) são combinados de forma ordenada em um processo iterativo.
    - Em cada iteração, um número `K` de runs é lido parcialmente para a memória (geralmente um bloco de cada run).
    - Esses `K` blocos são intercalados (mesclados) para produzir um novo run maior e ordenado.
    - Esse processo continua até que todos os runs sejam intercalados em um único arquivo final ordenado. A intercalação pode ser binária (de 2 em 2 runs) ou multi-via (de `K` em `K` runs, onde `K` é limitado pela memória disponível).

Diagrama:


Arquivo Original
Leitura em blocos p/ Mem.
Ordenação Interna
(Fase de Criação de Runs)
Grava Runs Temporários
Run 1
Run 2
...
Leitura de Runs p/ Mem.



Explicação do Funcionamento:

A eficiência da intercalação reside no fato de que, em ambas as fases, as operações de I/O são predominantemente **sequenciais**.

- Na fase de criação de runs, você lê um grande bloco sequencialmente, ordena-o, e o escreve sequencialmente.
- Na fase de intercalação, você lê pequenos blocos sequencialmente de múltiplos runs e escreve sequencialmente no novo run maior.

Isso minimiza o número de buscas (seek times) no disco, que são as operações mais caras. O número de runs gerados e o número de passes de intercalação dependem do tamanho do arquivo e da memória disponível. Uma maior memória permite runs maiores e menos passes de intercalação.

Em quais circunstâncias essa estratégia deve ser utilizada?

A ordenação de arquivos por intercalação deve ser utilizada em três circunstâncias principais:

1. **Arquivos Muito Grandes:** Quando o arquivo a ser ordenado é **maior do que a memória RAM disponível**. Esta é a aplicação mais comum e crucial.
2. **Eficiência de I/O:** Quando a prioridade é **minimizar as operações de acesso aleatório a disco** e maximizar o desempenho através de acessos sequenciais.
3. **Ambientes de Banco de Dados/Sistemas de Arquivos:** É a técnica fundamental utilizada por sistemas de gerenciamento de banco de dados (SGBDs) e sistemas de arquivos para ordenar grandes volumes de dados que não cabem na memória.