Trabalho 2 de Banco de Dados

Thiago Reis Santana - matricula: 22153143

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200, Coroado I, Setor Norte
Campus Universitário – 69080-900 – Manaus – AM

thiago.santana@icomp.ufam.edu.br

Resumo. *Este trabalho consiste na implementação de programas para armazenamento e consulta em dados armazenados em memória secundária utilizando as estruturas de arquivo de dados e índice estudadas nas aulas. Os programas devem fornecer suporte para a inserção, assim como diferentes formas de busca, seguindo as técnicas apresentadas nas aulas de organização e indexação de arquivos.*

1. Estrutura usada para criar os registros no arquivo de dados

O registro usar o modelo de campos de tamanho fixo e variavel, com alocacão não espalhada. Seguindo o que foi apresentado nas aulas o arquivo de dados é organizado em Buckets que guardam Blocos, que por sua vez guardam os Registros. Abaixo está o codigo da estrutura de cada um desses componentes.

1.1. Registro

```
struct Registro {
    int id;
    string titulo;
    int ano;
    string autores;
    int citacoes;
    string atualizacao;
    string snippet;
    int tamanho;
};
```

Figura 1. Estrutura do registro

1.2. Cabeçalho do Bloco

```
struct BlocoCabecalho {
    int quantidade_registros;
    int tamanho_disponivel;
    int posicoes_registros[18];
};
```

Figura 2. Estrutura do Bloco onde os registros são salvos

1.3. Bloco

```
struct Bloco {
    BlocoCabecalho* cabecalho;
    unsigned char dados[TAMANHO_BLOCO];
};
```

Figura 3. Estrutura do Bloco onde os registros são salvos

1.4. Bucket

```
struct Bucket {
    int quatidade_blocos;
    int ultimo_bloco;
    Bloco* blocos[NUMERO_BLOCOS];
};
```

Figura 4. Estrutura do Bucket onde os blocos são salvos

2. As partes do programa

O meu trabalho é dividido em os três partes que são:

A pasta CodigoAuxiliar: guarda funções todas as funções e metricas usadas para contruir o ”banco de dados”.

A pasta CodigoPrincipal: guarda as funções pedidas no trabalho, alem de outras auxiliares.

E o arquivo makefile: um arquivo makefile para facilitar a execução do programa

3. Pasta CodigoAuxiliar

3.1. Funções em `ArvoreBPlus.hpp`

3.1.1. Função `gerar_inteiro`

Esta função converte um título em uma chave inteira usando uma função de hash simples.

3.1.2. Estrutura `RegArvore`

Esta estrutura representa um registro na árvore B+, contendo uma chave e um valor.

3.1.3. Classe `No`

Esta classe define um nó na árvore B+, contendo informações sobre se é uma folha, o grau, a quantidade de elementos, os itens (registros) e os filhos.

3.1.4. Classe `ArvoreBPlus`

Esta classe implementa uma árvore B+ e suas operações básicas, como busca, inserção, serialização e desserialização.

3.1.5. Função `contarNos`

Esta função conta o número total de nós na árvore B+.

3.1.6. Função `buscar_registro_bpt`

Esta função busca um registro na árvore B+ com base em um ID fornecido e retorna o registro correspondente se encontrado.

3.2. Funções em `Base_Insercao_Hash.hpp`

3.2.1. `int hashFunction(int id)`

Esta função é responsável por calcular o índice de um bucket usando uma função de distribuição hash. Ela recebe um ID como entrada e retorna o índice calculado, que é usado para determinar em qual bucket um registro será armazenado. A fórmula utilizada é uma simples operação de módulo $(37 \times \text{id}) \,\%\, \text{NUMERO\_BUCKETS}$, onde NUMERO_BUCKETS é o número total de buckets na tabela hash. Isso ajuda a distribuir os registros de forma uniforme pelos buckets.

3.2.2. `void Criar_Base(ofstream& dataFile)`

Esta função é responsável por criar a estrutura básica da hash table e escrevê-la em um arquivo de dados. Ela itera sobre todos os buckets e chama a função `criarBucket()` para cada um deles, garantindo que a estrutura inicial esteja pronta para receber registros.

3.2.3. `void inserir_registro_bloco(ifstream& leitura, ofstream& escrita, Bloco* bloco, Registro* registro, int ultimo_bloco, int index_bucket)`

Esta função insere um registro em um bloco específico dentro de um bucket. Ela recebe como entrada o bloco onde o registro será inserido, o registro em si, o último bloco usado no bucket e o índice do bucket. A função manipula a inserção dos dados no bloco, atualizando o cabeçalho do bloco e escrevendo os dados atualizados no arquivo.

3.2.4. `int inserir_registro_bucket(Registro *registro, ifstream &entrada, ofstream &saida)`

Esta função é responsável por inserir um registro em um bucket. Ela recebe como entrada o registro a ser inserido, os fluxos de entrada e saída para manipulação do arquivo de dados. A função calcula o índice do bucket usando a função `hashFunction()`, percorre os blocos do bucket procurando por espaço disponível para inserção do registro e chama a função `inserir_registro_bloco()` para inserir o registro no bloco adequado.

3.2.5. `Registro* buscar_registro(ifstream& leitura, int id_busca)`

Esta função busca um registro no arquivo de dados com base no ID fornecido. Ela utiliza a função `hashFunction()` para calcular o índice do bucket onde o registro deve estar. Em seguida, percorre os blocos do bucket, lendo cada um deles e procurando pelo registro desejado. Se o registro for encontrado, ele é desserializado e retornado; caso contrário, a função retorna `nullptr`.

3.3. Funções em `bloco.hpp`

3.3.1. Função `criarBlocoCabecalho()`

Esta função aloca dinamicamente e inicializa um novo cabeçalho de bloco. Define a quantidade de registros como 0 e inicializa todas as posições de registros como 0. Calcula o tamanho disponível do bloco subtraindo o tamanho do cabeçalho e o tamanho do vetor de posições de registros do tamanho do bloco.

3.3.2. Função `criarBloco()`

Esta função aloca dinamicamente e inicializa um novo bloco de dados. Chama a função `criarBlocoCabecalho()` para criar o cabeçalho do bloco e preenche o array de bytes de dados com zeros.

3.3.3. Função `destruirBloco(Bloco* bloco)`

Esta função é responsável por liberar a memória alocada para um bloco de dados, incluindo o seu cabeçalho. Ela recebe um ponteiro para o bloco a ser destruído como parâmetro.

3.4. Funções em `bucket.hpp`

3.5. Função `destruirBucket(Bucket* bucket)`

Esta função é responsável por desalocar a memória associada a um bucket. Ela itera sobre cada bloco no vetor de blocos do bucket e chama a função `destruirBloco` para desalocar cada bloco individualmente. Em seguida, desaloca o próprio bucket.

3.6. Função `criarBucket(ofstream &dataFile)`

Esta função é responsável por criar um novo bucket. Ela aloca dinamicamente memória para um novo bucket, inicializa os campos `quantidade_blocos` e `ultimo_bloco`, e itera sobre cada posição do vetor de blocos, criando um novo bloco usando a função `criarBloco`. Os cabeçalhos e dados de cada bloco são copiados para um buffer temporário e escritos no arquivo de dados usando `dataFile.write`. Finalmente, chama a função `destruirBucket` para desalocar o bucket e os blocos da memória principal.

3.7. Função `criarRegistro.hpp`

Esta função é responsável por criar e inicializar um novo registro com os valores fornecidos como parâmetros. Ela aloca dinamicamente memória para o registro, atribui os valores dos parâmetros aos campos correspondentes e calcula o tamanho total do registro. Em seguida, retorna o ponteiro para o registro criado.

3.7.1. Função `imprimeRegistro`

Esta função imprime na saída padrão os valores dos campos de um registro específico. É útil para visualizar os dados contidos em um registro.

3.7.2. Função `remove_unicode`

Esta função recebe uma string como entrada e remove os caracteres Unicode da mesma, substituindo-os por espaços em branco. Isso é útil para limpar os dados de entrada antes de processá-los.

3.7.3. Função `converter_linha_registro`

Esta função converte uma linha do arquivo de entrada em um registro. Primeiro, ela remove os caracteres Unicode da linha usando a função `remove_unicode`. Em seguida, divide a linha em campos delimitados por aspas duplas e ponto e vírgula. Cada campo é armazenado em um vetor. Se o número de campos for diferente de 7, a função retorna `NULL`. Caso contrário, chama a função `criarRegistro` para criar um novo registro com os valores dos campos. Se ocorrer algum erro na conversão dos valores para inteiros, a função também retorna `NULL`.

4. Pasta CodigoPrincipal

4.1. Função `findrec.hpp`

Esta função é responsável por buscar um registro com um ID específico no arquivo de dados organizado por hashing. Ela recebe como parâmetro o ID a ser buscado.

- **Abertura do arquivo de dados:** A função abre o arquivo de dados "arquivo_de_dados.bin"no modo de leitura binária.
- **Busca pelo registro:** Em seguida, chama a função `buscar_registro`, que busca o registro com o ID especificado no arquivo de dados. O resultado da busca é armazenado em `registro_busca`.
- **Verificação do registro encontrado:** A função verifica se o registro foi encontrado com sucesso. Se `registro_busca` for diferente de `NULL`, significa que o registro foi encontrado. Nesse caso, os campos do registro são impressos na saída padrão utilizando a função `imprimeRegistro`.
- **Liberação de memória:** Após imprimir os campos do registro, a memória alocada dinamicamente para `registro_busca` é liberada usando o operador `delete`, evitando vazamentos de memória.
- **Fechamento do arquivo:** Por fim, o arquivo de dados é fechado usando o método `close()`.

Esta função é útil para encontrar e exibir informações detalhadas sobre um registro específico com base no ID fornecido.

4.2. Função `seek1.hpp`

Esta função é responsável por buscar um registro com um ID específico no arquivo de dados organizado por hashing, utilizando uma estrutura de índice primário. Ela recebe como parâmetro o ID a ser buscado.

- **Abertura do arquivo de dados:** A função abre o arquivo de dados "arquivo_de_dados.bin"no modo de leitura binária.
- **Definição do índice primário:** Define o nome do arquivo que contém o índice primário como "indice_primario.bin".
- **Busca pelo registro:** Chama a função `buscar_registro_bpt`, que busca o registro com o ID especificado no arquivo de dados utilizando o índice primário. O resultado da busca é armazenado em `registro_busca`.
- **Verificação do registro encontrado:** Verifica se o registro foi encontrado com sucesso. Se `registro_busca` for diferente de `nullptr`, significa que o registro foi encontrado. Nesse caso, os campos do registro são impressos na saída padrão utilizando a função `imprimeRegistro`.
- **Liberação de memória:** Após imprimir os campos do registro, a memória alocada dinamicamente para `registro_busca` é liberada usando o operador `delete`, evitando vazamentos de memória.
- **Fechamento do arquivo:** Por fim, o arquivo de dados é fechado usando o método `close()`.

Esta função utiliza um índice primário para otimizar a busca por registros com base no ID fornecido, garantindo uma busca eficiente e rápida.

4.3. Função `seek2.hpp`

Esta função é responsável por buscar um registro com um título específico no arquivo de dados organizado por hashing, utilizando uma estrutura de índice secundário. Ela recebe como parâmetro o título a ser buscado.

- **Abertura do arquivo de dados:** A função abre o arquivo de dados "arquivo_de_dados.bin"no modo de leitura binária.
- **Definição do índice secundário:** Define o nome do arquivo que contém o índice secundário como "indice_secundario.bin".
- **Busca pelo registro:** Chama a função `buscar_registro_bpt`, que busca o registro com o título especificado no arquivo de dados utilizando o índice secundário. O título é convertido em um valor inteiro usando a função `gerar_inteiro` aplicada à versão sem caracteres unicode do título fornecido. O resultado da busca é armazenado em `registro_busca`.
- **Verificação do registro encontrado:** Verifica se o registro foi encontrado com sucesso. Se `registro_busca` for diferente de `nullptr`, significa que o registro foi encontrado. Nesse caso, os campos do registro são impressos na saída padrão utilizando a função `imprimeRegistro`.
- **Liberação de memória:** Após imprimir os campos do registro, a memória alocada dinamicamente para `registro_busca` é liberada usando o operador `delete`, evitando vazamentos de memória.
- **Fechamento do arquivo:** Por fim, o arquivo de dados é fechado usando o método `close()`.

Esta função utiliza um índice secundário para otimizar a busca por registros com base no título fornecido, garantindo uma busca eficiente e rápida.

4.4. Função `upload.cpp`

Esta função principal é responsável por criar um arquivo de dados, inserir registros nele, criar e salvar índices primário e secundário em árvores B+.

- **Criação do arquivo de dados:** A função cria o arquivo "arquivo_de_dados.bin"no modo de escrita binária.
- **Criação da base de dados:** Chama a função `Criar_Base` para criar a base de dados, que inclui a criação de todos os buckets e blocos necessários.
- **Abertura do arquivo de entrada:** Abre o arquivo de entrada "artigo.csv"no modo de leitura.
- **Abertura do arquivo de dados organizado por hashing:** Abre o arquivo de dados "arquivo_de_dados.bin"no modo de leitura binária.
- **Inserção de registros:** Lê os registros do arquivo CSV e os insere no arquivo de dados. Para cada registro lido, é chamada a função `inserir_registro_bucket`, que insere o registro no arquivo de dados, e em seguida adiciona o registro às árvores B+ primária e secundária.
- **Salvamento dos índices:** Após inserir todos os registros, as árvores B+ primária e secundária são salvas nos arquivos "indice_primario.bin"e "indice_secundario.bin", respectivamente.
- **Liberação de memória:** As árvores B+ são desalocadas da memória para evitar vazamentos de memória.

- **Fechamento dos arquivos:** Por fim, os arquivos de entrada, dados e o arquivo de dados são fechados.

Essa função constitui a operação principal do programa, que cria e popula um arquivo de dados com registros do arquivo CSV fornecido, além de construir e salvar índices para otimizar futuras operações de busca.

4.5. Função `menu.cpp`

Esta função principal é responsável por exibir um menu interativo para o usuário, permitindo a execução de diferentes operações de busca nos registros armazenados no arquivo de dados.

- **Exibição do menu:** A função exibe um menu de opções para o usuário, incluindo as operações ”findrec”, ”seek1”, ”seek2”e ”Sair”.
- **Obtenção da escolha do usuário:** Obtém a escolha do usuário em relação à operação que deseja realizar.
- **Execução da operação correspondente:** Com base na escolha do usuário, a função chama a operação correspondente:
  - **findrec:** Solicita ao usuário que insira o ID do registro a ser buscado e chama a função `findrec` para executar a busca.
  - **seek1:** Solicita ao usuário que insira o ID do registro a ser buscado e chama a função `seek1` para executar a busca.
  - **seek2:** Solicita ao usuário que insira o título do registro a ser buscado e chama a função `seek2` para executar a busca.
  - **Sair:** Encerra o programa.
  - **Opção inválida:** Se o usuário inserir uma opção inválida, uma mensagem de erro é exibida e o usuário é solicitado a escolher novamente.
- **Repetição do processo:** O menu é exibido novamente após a execução de uma operação, e o processo continua até que o usuário escolha sair (opção ’x’).

Essa função serve como a interface principal do programa, permitindo ao usuário interagir com as operações de busca nos registros do arquivo de dados.

5. Como usar o Makefile

5.1. Makefile

O Makefile é um arquivo de configuração usado em projetos de software para automatizar o processo de compilação e execução de programas. Ele define regras e comandos que o utilitário make usa para compilar e construir o software de maneira eficiente.

5.2. Definição de Variáveis

No Makefile fornecido, duas variáveis são definidas:

SRC_DIR: Esta variável define o diretório onde estão localizados os arquivos-fonte do programa. Neste caso, os arquivos-fonte estão no diretório chamado Codigo-Principal.

CSV_FILE: Esta variável contém o caminho para o arquivo CSV que será usado como entrada para o programa. O arquivo artigo.csv deve ser colocado dentro da pasta Scripts.

O Makefile contém várias regras que definem como compilar e executar os programas:

- `upload`: Esta regra especifica como compilar o programa `upload`. Utiliza o compilador `g++` para compilar o arquivo-fonte `upload.cpp`, localizado no diretório definido pela variável SRC_DIR. O programa resultante é nomeado como `upload`.

- `menu`: Similar à regra `upload`, esta regra compila o programa `menu` usando o arquivo-fonte `menu.cpp` no diretório definido por SRC_DIR.

- `Preparar`: Esta regra depende da regra `upload` e é usada para preparar o arquivo CSV para uso pelo programa. Executa o programa `upload`, passando o caminho do arquivo CSV especificado pela variável CSV_FILE.

- `Rodar`: Esta regra executa o programa `menu` após a compilação bem-sucedida. Invoca o executável gerado pela regra `menu` para rodar o programa principal.

- `clean`: Esta regra é usada para limpar o diretório de trabalho removendo os arquivos executáveis gerados. Remove os programas `upload` e `menu`.

5.3. Por que usar o Makefile?

Crie essa arquivo Makefile para facilitar o teste do meu programa, mas acho que ficou bom o suficiente para ser usado como padrão para rodar o programa.