Segundo Trabalho Prático - Banco de Dados I

Ayumi Aoki Santana[1], **Erllison Reis**[1], **Maria Luiza Saldanha**[1]

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

1. Introdução

Este projeto é desenvolvido para manipular e organizar registros de dados usando índices primários e secundários implementados por meio de árvores B+. A aplicação visa armazenar, buscar e gerenciar registros em arquivos de dados organizados por hashing, facilitando a recuperação eficiente de informações. O projeto consiste em múltiplos arquivos que desempenham funções específicas, desde a criação de índices até a recuperação de dados com base em critérios de busca primária (por ID) e secundária (por título).

2. Estrutura do Arquivo de Dados e de Índices

As decisões de projeto para a estrutura dos arquivos de índices e o arquivo de dados foram tomadas de acordo com o tamanho de bloco do sistemas de arquivos Linux, que são 4096 *bytes*, e o tamanho em *bytes* dos campos dos registros de dados. Somando o número de bytes reservados para cada campo do registro, tem-se que registro possui 592 *bytes*. As seguintes definições a serem feitas são para os tamanho de *Bucket*, e a ordem (M_PRIM) para árvore B+ para o índice primário e secundário (M_SEC).

Para a ordem da árvore de índice primário o cálculo feito foi baseado na estrutura do nó da árvore segundo a implementação proposta pelo projeto. Um nó da árvore é composto pelos seguintes atributos:

- `int chaves[`$(2 \times M\_PRIM) + 1$`]`: Chaves de um nó
- `long enderecos[`$(2 \times M\_PRIM) + 2$`]`: apontadores de um nó
- `bool ehFolha` (1 *byte*): Indica se o nó é um nó-folha
- `int enderecoNo (4 bytes)`: Guarda a posição do nó dentro do arquivo de índices

Vale ressaltar que os atributo `chaves` e `enderecos` possuem um entrada a mais que o esperado para auxiliar na inserção ordenada, porém a regra de que um nó pode ter no máximo $2 * M$ chaves é respeitada. Dessa forma, o cálculo 1 resultou na ordem $M\_PRIM = 169$ e em um bloco de 4088 bytes. Dado que um nó deste tamanho não preenche um bloco inteiro, criamos o atributo `preenchimento` para preencher o restante do bloco com 8 bytes.

$$4 * (2 \times M\_PRIM + 1) + 8 \times (2 \times M\_PRIM + 2) + 2 + 1 <= 4096 \qquad (1)$$

Dessa forma, o cálculo da ordem da árvore é feito a partir da soma do tamanho em bytes dos campos do nó, considerando que o nó tenha que ser menor ou igual a 4096 *bytes* (tamanho de um bloco).

O mesmo se aplica para o cálculo da ordem da árvore B+ do índice secundário, com a diferença de que o formato da chave neste caso é um vetor de *char* (1 *byte*) de tamanho 301. O cálculo 2 resultou em uma ordem igual a 6 e em um nó de tamanho 4040 bytes. Nesse caso, utilizamos um atributo `preenchimento` com 56 bytes.

$$101 * (2 \times M\_SEC + 1) + 8 \times (2 \times M\_SEC + 2) + 2 + 1 <= 4096 \qquad (2)$$

3. Ordem de Dependências

```
|-- definicoes.h (define constantes usadas em todo o sistema)
|
|-- database.h
|   |-- Inclui definicoes.h (usa constantes definidas)
|
|-- primary_index.h
|   |-- Inclui database.h (usa estruturas e constantes definidas)
|
|-- secondary_index.h
|   |-- Inclui database.h (usa estruturas e constantes definidas)
|
|-- upload.cpp
|   |-- Inclui primary_index.h e secondary_index.h
|       (manipula índices primário e secundário)
|   |-- Inclui definicoes.h
|       (usa constantes para configurar buckets e blocos)
|
|-- findrec.cpp
|   |-- Inclui database.h (para operações com registros e blocos)
|   |-- Inclui primary_index.h (realiza busca pelo índice primário)
|
|-- seek1.cpp
|   |-- Inclui database.h
|       (para operações de leitura de blocos e registros)
|   |-- Inclui primary_index.h
|       (busca registro pelo índice primário com base no ID)
|
|-- seek2.cpp
|   |-- Inclui database.h (para operações com blocos e registros)
|   |-- Inclui secondary_index.h
|       (busca registro pelo índice secundário com base no título)
```

4. Descrição dos Programas

4.1. Descrição Geral das Bibliotecas

- `#include <iostream>, <fstream>, <sstream>, <string>, <vector>, <algorithm>, <cstring>`: Bibliotecas utilizadas para entrada/saída, manipulação de arquivos, e manipulação de strings.
- `database.h`: Define estruturas como `TableRow` e as constantes usadas.

4.2. Definições de Constantes: definicoes.h

As constantes definidas no projeto são utilizadas para configurar o tamanho dos blocos, registros e buckets, além de definir parâmetros para a organização dos índices primário e secundário. Essas constantes estão definidas no arquivo de cabeçalho e são usadas em todas as funções.

- `#define BLOCK_SIZE 4096`: Define o tamanho de um bloco em 4096 bytes.
- `#define REGISTER_SIZE 592`: Define o tamanho de um registro da base de dados em 592 bytes, calculado a partir da soma dos campos de um registro individual.
- `#define QTD_BLOCKS 170241`: Define a quantidade total de blocos necessários para armazenar toda a base de dados.
- `#define BLOCK_FACTOR 6`: Determina a quantidade máxima de registros que podem ser armazenados em um bloco, considerando o tamanho de cada registro e o espaço disponível no bloco.
- `#define BLOCKS_BY_BUCKETS 4`: Especifica que cada bucket contém 4 blocos.
- `#define PRIMO 120767`: Constante utilizada como número primo na função de hashing, garantindo a distribuição dos dados nos buckets.
- `#define BUCKET_SIZE (BLOCKS_BY_BUCKETS * BLOCK_SIZE)`: Define o tamanho de um bucket, que é a quantidade de blocos por bucket multiplicada pelo tamanho de cada bloco, resultando em um total de 16384 bytes (4 blocos de 4096 bytes cada).
- `#define M_PRIM 169`: Define a ordem da árvore B+ para o índice primário, definida a partir do cálculo do número máximo de chaves e ponteiros que um nó pode possuir.
- `#define M_SEC 6`: Define a ordem da árvore B+ para o índice secundário, baseada no tamanho máximo do título dos registros e na capacidade de armazenamento do nó.

4.3. Upload.cpp

O programa `upload` realiza a carga inicial de dados a partir de um arquivo CSV, organizando os registros em buckets e criando os arquivos de índices primário e secundário.

- **hash_table.bin**: Armazena os registros da tabela de dados, organizados em buckets de tamanho fixo, com um sistema de hashing que define o índice de cada bucket.
- **primary_index.bin**: Índice primário em árvore B+ com chave baseada no ID dos registros.
- **secondary_index.bin**: Índice secundário em árvore B+ para buscas, armazenando o endereço do registro no arquivo hash.

Descrição das Fontes e Funções

Fontes

O programa é composto pelos seguintes arquivos fonte:

- **main.cpp**: Implementa a função principal.

- **primary_index.h** e **primary_index.cpp**: Implementam o índice primário, incluindo as funções para inserir e buscar dados na árvore B+ pelo ID.
- **secondary_index.h** e **secondary_index.cpp**: Implementam o índice secundário, permitindo a busca por título.
- **utils.h** e **utils.cpp**: Contem funções para manipulação das strings.

Funções e Responsabilidades

- `int contarCaractere(const std::string &str, char caractere)`: Conta a ocorrência de um caractere específico em uma string.
- `std::vector<std::string> dividirString(const std::string &texto, char delimitador)`: Divide uma string com base em um delimitador, retornando um vetor com os tokens..
- `std::string removerAspas(const std::string &texto)`: Remove aspas de uma string.
- `void salvarBlocoNoArquivo(const std::vector<TableRow> &bloco, int bucket, std::ofstream &arquivo, FILE *arqPrimaryIndex, FILE *arqSecondaryIndex)`: Salva um bloco de registros no arquivo hash_table.bin, ajustando a posição do ponteiro de gravação com base no bucket. Adiciona os registros ao índice primário e secundário.
- `void analisarArquivo(const std::string &nomeArquivo)`: Analisa o arquivo CSV com base no hash dos IDs. Controla a criação de blocos e chama a função de salvamento.
- `int main(int argc, char *argv[])`: Função principal que recebe o nome do arquivo CSV de entrada e executa o fluxo principal de processamento.

4.4. Findrec.cpp

O programa `findrec` busca um registro no arquivo de dados com base no ID fornecido, utilizando o índice primário para encontrar a localização do bloco onde o registro está armazenado.

Fontes e Funções

- `lerRegistroDoBloco(const char *block, int &posBloco, TableRow &entrada)`: Lê um registro do bloco de dados, verificando se o ID do registro é válido e armazenando os campos na estrutura `TableRow`.
- `findrec(const std::string &nomeArquivo, int ID)`: Busca o registro pelo ID. Calcula o bucket, carrega o bloco e busca o registro.
- `main(int argc, char *argv[])`: Função principal para encontrar o ID do registro de entrada e chamar a função de busca com o `findrec`.

4.5. Seek1.cpp

O programa `seek1` realiza a busca no registro no arquivo de dados com base no ID. Utiliza o índice primário para encontrar o registro, retornando os campos do registro e informações sobre a quantidade de blocos lidos.

Fontes e Funções

- `int main(int argc, char *argv[])`: Verifica o argumento passado para garantir que um ID foi fornecido. Ele abre os arquivos de dados e índice primário e executa a busca por meio da `buscaRegistroIndicePrimario`:
  - **Abertura de Arquivos**: Verifica a existência dos arquivos `hash_table.dat` e `primary_index.txt`.
  - **Execução da Busca**: Chama a função `buscaRegistroIndicePrimario` passando o ID buscado, estruturas de dados, e os arquivos abertos.
  - **Cálculo dos Blocos Lidos e Exibição**: Calcula a quantidade total de blocos, a quantidade de blocos lidos e fecha os arquivos.

Funções Utilizadas

- `int buscaRegistroIndicePrimario(int id, registro_t *registro, FILE *arqIndicePrimario, FILE *arqDadosTxt)`: Realiza a busca do registro com base no ID. Ela percorre o índice para encontrar o bloco certo e retorna o registro correspondente. Contabiliza e retorna a quantidade de blocos lidos.
- `int main(int argc, char *argv[])`: Busca o ID informado pelo usuário e chama a função `buscaRegistroIndicePrimario`. Exibe o total de blocos lidos e a quantidade de blocos total do índice primário.

4.6. Seek2.cpp

O programa `seek2` realiza a busca de um registro no arquivo de dados com base no `titulo`. Utiliza o índice secundário para localizar o registro desejado, retornando os campos do registro e informações sobre a quantidade de blocos lidos durante o processo de busca.

Fontes e Funções

- `int main(int argc, char *argv[])`: Verifica o argumento passado para garantir que um título foi fornecido. Ele abre os arquivos de dados e índice secundário e executa a busca por meio da função `buscaRegistroIndiceSecundario`.
  - **Abertura de Arquivos**: Verifica a existência dos arquivos `hash_table.dat` e `secondary_index.txt`.
  - **Execução da Busca**: Chama a função `buscaRegistroIndiceSecundario` passando o título buscado, as estruturas de dados necessárias e os arquivos abertos.
  - **Cálculo dos Blocos Lidos e Exibição**: Calcula a quantidade total de blocos, a quantidade de blocos lidos e fecha os arquivos.

Funções Utilizadas

- `int buscaRegistroIndiceSecundario(const char *titulo, registro_t *registro, FILE *arqIndiceSecundario, FILE *arqDadosTxt)`: Realiza a busca do registro com base no título. A função percorre o índice secundário para encontrar o bloco correto e retorna o registro correspondente. Ela também contabiliza e retorna a quantidade de blocos lidos.
- `int main(int argc, char *argv[])`: Obtém o título informado pelo usuário e chama a função `buscaRegistroIndiceSecundario`. Exibe o total de blocos lidos e a quantidade total de blocos do índice secundário, calculada com base no tamanho do arquivo e no tamanho do bloco.

4.7. primary_index.h

O arquivo `primary_index.h` implementa as estruturas e funções necessárias para manipular o índice primário do sistema. Este índice é uma árvore B+ que organiza os registros com base em um campo de chave primária (por exemplo, o `id` do registro), permitindo buscas rápidas e eficientes.

Estruturas Definidas

- **`no_index_primario_t`**: Estrutura para representar um nó da árvore B+ no índice primário. Contém:
  - `chaves`: Vetor de chaves (`id`) para armazenar entradas de busca.
  - `enderecos`: Vetor de endereços que apontam para a localização dos registros no arquivo de dados.
  - `ehFolha`: Indica se o nó é uma folha.
  - `numChaves`: Número de chaves no nó.
  - `enderecoNo`: Endereço do nó dentro do arquivo de índice primário.
  - `preenchimento`: Área de preenchimento para garantir que o nó ocupe 4096 bytes.

Funções e Papéis

- `void inicializarIndicePrimario(FILE *arqIndex)`: Inicializa o arquivo de índice primário criando um nó raiz vazio. O nó é configurado como folha e salvo no início do arquivo de índice.
- `void criarNoPrimario(bool folha, no_index_primario_t *novo)`: Inicializa um novo nó do índice primário em memória, configurando-o como folha ou não, e preenchendo as chaves e endereços com valores padrão.
- `void escreveNoDiscoPrimario(no_index_primario_t *no, long endereco, FILE *arqIndex)`: Escreve um nó do índice primário da memória para o disco, posicionando-o no endereço especificado no arquivo de índice.
- `int lerNoDiscoPrimario(no_index_primario_t *outputNode, unsigned long endereco, FILE *arqIndex)`: Lê um nó do índice primário do disco para a memória, posicionando o cursor no endereço especificado e retornando o conteúdo no `outputNode`.

- `void inserirIndexPrimario(int chave, unsigned long enderecoDados, FILE *arqIndex)`: Realiza a inserção de uma nova chave no índice primário, chamando a função recursiva de inserção para localizar o nó onde a chave deve ser adicionada.
- `void inserirRecursivoPrimario(no_index_primario_t *no, no_index_primario_t *pai, int chave, int dataOffset, FILE *arqIndex)`: Insere uma nova chave recursivamente na árvore. Caso o nó seja uma folha, a chave é adicionada diretamente. Caso contrário, a função localiza o próximo nó até encontrar a posição correta para a chave.
- `void dividirNoPrimario(no_index_primario_t *no, no_index_primario_t *pai, int dataOffset, FILE *arqIndex)`: Realiza a divisão de um nó que excedeu a capacidade máxima, seguindo as propriedades da árvore B+. Caso o nó seja a raiz, cria um novo nó pai. A chave do meio é promovida para o pai, e o nó original é dividido em dois.
- `void deslocarChavesPrimario(no_index_primario_t *no, int pos)`: Desloca as chaves em um nó para abrir espaço para uma nova chave na posição especificada. As chaves e endereços são movidos para a direita para acomodar a nova entrada.
- `int buscarChaveMaiorPrimario(no_index_primario_t *no, int chave)`: Retorna o índice da primeira chave maior ou igual à chave passada como parâmetro. Essa função ajuda a localizar a posição de inserção ou busca de uma chave.
- `int buscaBinariaPrimario(int chave, int *vetor, int tam)`: Executa uma busca binária no vetor de chaves para encontrar a posição da chave ou a posição onde a chave deveria estar. Retorna o índice correspondente.
- `int buscaRegistroIndicePrimario(int chave, registro_t *registro, FILE *arqIndex, FILE *arqDados)`: Realiza a busca de um registro no índice primário. A função lê o nó raiz e chama a função recursiva para navegar pela árvore B+ até localizar o nó que pode conter a chave.
- `int buscaIndicePrimarioRecursivo(no_index_primario_t *raiz, int chave, registro_t *registro, FILE *arqIndex, FILE *arqDados)`: Navega recursivamente pela árvore B+ do índice primário para encontrar o nó que pode conter a chave. Se a chave for encontrada, carrega o bloco correspondente do arquivo de dados e busca o registro.

4.8. secondary_index.h

O arquivo `secondary_index.h` define e implementa as estruturas e funções necessárias para manipular o índice secundário do sistema. Esse índice secundário é organizado como uma árvore B+ e armazena os registros com base em um campo de chave secundária, como o `titulo`. O índice secundário facilita a busca eficiente de registros textuais, permitindo que os registros sejam localizados rapidamente a partir do arquivo de dados.

Estruturas Definidas

- **`chave_t`**: Define um tipo de dado para representar a chave, com tamanho fixo de 301 caracteres.

- **`no_index_secundario_t`**: Estrutura para representar um nó da árvore B+ no índice secundário. Contém:
  - `chaves`: Vetor de chaves (`titulo`) para armazenar entradas de busca.
  - `enderecos`: Vetor de endereços que apontam para a localização dos registros no arquivo de dados.
  - `ehFolha`: Indica se o nó é uma folha.
  - `numChaves`: Número de chaves no nó.
  - `enderecoNo`: Endereço do nó dentro do arquivo de índice secundário.
  - `preenchimento`: Área de preenchimento para garantir que o nó ocupe 4096 bytes.

Funções e Papéis

- `void inicializarIndiceSecundario(FILE *arqIndex)`: Inicializa o arquivo de índice secundário criando um nó raiz vazio. O nó é configurado como folha e salvo no início do arquivo de índice.
- `void criarNoSecundario(bool folha, no_index_secundario_t *novo)`: Inicializa um novo nó do índice secundário em memória, configurando-o como folha ou não, e preenchendo as chaves e endereços com valores padrão.
- `void escreveNoDiscoSecundario(no_index_secundario_t *no, long endereco, FILE *arqIndex)`: Escreve um nó do índice secundário da memória para o disco, posicionando-o no endereço especificado no arquivo de índice.
- `int lerNoDiscoSecundario(no_index_secundario_t *outputNode, unsigned long endereco, FILE *arqIndex)`: Lê um nó do índice secundário do disco para a memória, posicionando o cursor no endereço especificado e retornando o conteúdo no `outputNode`.
- `void inserirIndexSecundario(chave_t chave, unsigned long enderecoDados, FILE *arqIndex)`: Realiza a inserção de uma nova chave no índice secundário, chamando a função recursiva de inserção para localizar o nó onde a chave deve ser adicionada.
- `void inserirRecursivoSecundario(no_index_secundario_t *no, no_index_secundario_t *pai, chave_t chave, int dataOffset, FILE *arqIndex)`: Insere uma nova chave recursivamente na árvore. Caso o nó seja uma folha, a chave é adicionada diretamente. Caso contrário, a função localiza o próximo nó até encontrar a posição correta para a chave.
- `void dividirNoSecundario(no_index_secundario_t *no, no_index_secundario_t *pai, int dataOffset, FILE *arqIndex)`: Realiza a divisão de um nó que excedeu a capacidade máxima, seguindo as propriedades da árvore B+. Caso o nó seja a raiz, cria um novo nó pai. A chave do meio é promovida para o pai, e o nó original é dividido em dois.
- `void deslocarChavesSecundario(no_index_secundario_t *no, int pos)`: Desloca as chaves em um nó para abrir espaço para uma nova chave na posição especificada. As chaves e endereços são movidos para a direita para acomodar a nova entrada.

- `int buscarChaveMaiorSecundario(no_index_secundario_t *no, chave_t chave)`: Retorna o índice da primeira chave maior ou igual à chave passada como parâmetro. Essa função ajuda a localizar a posição de inserção ou busca de uma chave.
- `int buscaBinariaSecundario(chave_t chave, chave_t *vetor, int tam)`: Executa uma busca binária no vetor de chaves para encontrar a posição da chave ou a posição onde a chave deveria estar. Retorna o índice correspondente.
- `int buscaRegistroIndiceSecundario(chave_t chave, registro_t *registro, FILE *arqIndex, FILE *arqDados)`: Realiza a busca de um registro no índice secundário. A função lê o nó raiz e chama a função recursiva para navegar pela árvore B+ até localizar o nó que pode conter a chave.
- `int buscaIndiceSecundarioRecursivo(no_index_secundario_t *raiz, chave_t chave, registro_t *registro, FILE *arqIndex, FILE *arqDados)`: Navega recursivamente pela árvore B+ do índice secundário para encontrar o nó que pode conter a chave. Se a chave for encontrada, carrega o bloco correspondente do arquivo de dados e busca o registro.
- `int buscarRegistroBlocoSecundario(chave_t titulo, bloco_t *bloco, registro_t *registro)`: Realiza a busca de um registro em um bloco específico do arquivo de dados. Caso o título do registro coincida com a chave buscada, o registro é retornado; caso contrário, retorna 0.

4.9. database.h

O arquivo `database.h` define as estruturas e funções principais para a manipulação dos registros de dados no sistema. Ele fornece a base para a criação, leitura e manipulação dos registros no arquivo de dados, bem como utilitários para inicialização e manipulação de blocos. Este arquivo é essencial para definir como os dados são armazenados e acessados em disco.

Estruturas Definidas

- **`TableRow`**: Estrutura que representa um registro de dados. Cada `TableRow` contém:
  - `id`: Identificador único do registro.
  - `titulo`: Título do registro, armazenado como uma string de até 300 caracteres.
  - `ano`: Ano de publicação.
  - `autores`: Nomes dos autores, com até 150 caracteres.
  - `citacoes`: Número de citações do registro.
  - `atualizacao`: Data de atualização do registro, com até 20 caracteres.
  - `snippet`: Um trecho de texto descritivo ou resumo do registro.
- **`registro_t`**: Estrutura usada para representar cada registro em um formato otimizado para armazenamento. Contém campos semelhantes aos de `TableRow`, mas com limites de tamanho fixos para facilitar a gravação em blocos.
- **`bloco_t`**: Estrutura que representa um bloco de dados no arquivo. Cada bloco contém:

  - `registros`: Um vetor de `registro_t`, contendo vários registros de dados.
  - `preenchimento`: Área de preenchimento para garantir que o tamanho do bloco seja 4096 bytes, o que corresponde ao tamanho de bloco do sistema de arquivos.

Funções e Papéis

- `void printTableRow(const TableRow &row)`: Exibe os detalhes de um `TableRow` no console, listando cada campo de forma legível. Esta função é útil para depuração e para visualizar dados armazenados.
- `int calcularHash(int numero)`: Calcula o índice do `bucket` para um determinado número (neste caso, o `id` de um registro), usando uma função de hash. Esse índice é utilizado para distribuir registros em blocos na tabela hash, facilitando o acesso direto.
- `void inicializarArquivo(FILE *arquivoDados)`: Inicializa o arquivo de dados preenchendo-o com blocos vazios. Esta função percorre o arquivo e grava blocos vazios em cada posição, garantindo que o arquivo esteja pronto para receber novos registros.
- `int inserirRegistroEmBloco(registro_t *registro, bloco_t *bloco)`: Insere um novo registro em um bloco específico. A função verifica cada posição do bloco em busca de uma posição livre. Se encontrar, copia o registro para essa posição e retorna 1; caso contrário, retorna 0, indicando que o bloco está cheio.
- `long pegarProximoNoOffset(FILE *arqIndex)`: Retorna o próximo deslocamento (offset) disponível no arquivo de índice. Esta função posiciona o cursor no final do arquivo e retorna a posição atual, facilitando a inserção de novos nós no índice.
- `int buscarRegistroBloco(int id, bloco_t *bloco, registro_t *registro)`: Busca um registro específico dentro de um bloco, comparando o campo `id` de cada registro com o `id` desejado. Se encontrar o registro, ele é copiado para o parâmetro `registro` e a função retorna 1; caso contrário, retorna 0.
- `bool lerRegistroDoBloco(const char *block, int &posBloco, TableRow &entrada)`: Lê um registro do bloco especificado em memória, extraindo cada campo de dados para a estrutura `TableRow`. A função é usada para decodificar blocos lidos do disco e processar seus registros.