Relatório do TP2 de Banco de Dados I

Guilherme Dias Corrêa, Luiza da Costa Caxeixa e Sofia de Castro Sato

[1]Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)

Av. Rodrigo Otávio, nº 6200, Coroado I, Manaus – AM, 69080-900

{guilherme.correa, luiza.caxeixa, sofia.sato}@icomp.ufam.edu.br

1. Introdução

Este documento descreve o projeto e implementação do **Trabalho Prático 2** da disciplina de Bancos de Dados I (UFAM, 2025/02). O objetivo do trabalho é desenvolver quatro programas em C++ capazes de manipular arquivos de dados e índices armazenados em memória secundária, utilizando técnicas de *hashing* e árvores **B+ Tree**.

Os programas implementados são:

- **upload** – cria o arquivo de dados com hash e os índices com a B+Tree a partir de um CSV de entrada tratado;
- **findrec** – realiza busca direta por *ID* no arquivo de dados;
- **seek1** – busca por *ID* utilizando o índice primário (B+ Tree);
- **seek2** – busca por *Título* utilizando o índice secundário (B+ Tree).

2. Estrutura do Projeto

A organização dos diretórios e arquivos do projeto segue o padrão definido no enunciado:

```
app/
 src/                # Código-fonte (.cpp)
 include/            # Cabeçalhos (.h)
 bin/                # Executáveis gerados
 data/               # Arquivos de dados e índices
 tests/              # Scripts e dados de teste
 Dockerfile          # Configuração do contêiner
 docker-compose.yml  # Orquestração Docker
 Makefile            # Automação de build
 README.md           # Instruções de uso
 TP2-BD1-2025-02.docx.pdf
```

3. Estrutura dos Arquivos de Dados e Índices

3.1. Arquivo de Dados (Hashing)

Estrutura do Registro:

- **ID** – inteiro (4 bytes, chave primária);
- **Título** – string (301 bytes fixos);
- **Ano** – inteiro (4 bytes);
- **Autores** – string (151 bytes fixos);

- **Citações** – inteiro (4 bytes);
- **Atualização** – string (21 bytes, formato: YYYY-MM-DD HH:MM:SS);
- **Snippet** – string (1025 bytes fixos).

Organização Física:

- Tamanho do bloco: 4096 bytes
- Registros por bloco: 2 (considerando overhead)
- Função hash: baseada em módulo do ID
- Utiliza 2000 buckets
- Arquivo principal de dados: `data.bin`
- Arquivo do mapeamento dos buckets: `hash.bin`
- Arquivo dos dados distribuídos em blocos: `data_hash.dat`
- Arquivo de offsets: `index.bin`

3.2. Índice Primário (B+ Tree)

A árvore B+ primária indexa o campo **ID** e está armazenada no arquivo `indice_id.bin`.

- Ordem da árvore: configurável
- Estrutura em memória secundária
- Nós folha contêm ponteiros diretos para registros
- Cada entrada: `<ID, ponteiro_para_registro>`

3.3. Índice Secundário (B+ Tree)

A árvore B+ secundária indexa o campo **Título** e está armazenada no arquivo `indice_titulo.bin`.

- Ordem da árvore: configurável
- Cada entrada: `<Título, ponteiro_para_registro>`
- Busca exata por string

4. Estrutura dos Programas e Fontes

Arquivo Fonte	Descrição	Responsável
`Documentação`	Registro técnico do desenvolvimento e guia para manutenção do sistema.	Equipe
`upload.cpp`	Lê o CSV e cria os arquivos de dados e índices.	Guilherme Dias
`findrec.cpp`	Realiza busca direta por ID no arquivo de dados.	Sofia Sato
`seek1.cpp`	Busca via índice primário (B+ Tree por ID).	Luiza Caxeixa
`seek2.cpp`	Busca via índice secundário (B+ Tree por Título).	Luiza Caxeixa
`hash.cpp`	Implementa estrutura de hashing.	Sofia Sato

`BTplus_mem.cpp`	Implementa a estrutura de B+ Tree genérica.	Guilherme Dias

4.1. Cabeçalhos (include/)

Arquivo Header	**Descrição**
`BTplus_mem.h`	Definições da classe B+ Tree e estruturas relacionadas.
`hash.h`	Definições do hash e estruturas relacionadas.

5. Funções Principais e Implementação

5.1. Sistema de Hash (hash.cpp)

Implementação da tabela hash para organização do arquivo de dados:

- Função hash baseada no ID do registro
- Função que gera o hash com tratamento de colisões por encadeamento no bucket
- Persistência em arquivos binários

No hash, foi utilizada uma estrutura de buckets, em que cada bucket armazenou uma sequência de blocos. A organização foi feita por encadeamento, de forma que os blocos de um mesmo bucket se ligassem entre si por meio de overflow quando um bloco ficasse cheio.

5.2. Árvore B+ (BTplus_mem.cpp)

Implementação da estrutura de índices:

- Inserção mantendo propriedades da B+ Tree
- Busca eficiente por chaves
- Divisão e fusão automática de nós
- Persistência em memória secundária

5.3. Programa de Carga de Dados

5.3.1. Upload

O programa `upload` realiza a carga inicial dos dados a partir do CSV (`artigo.csv`) e cria os arquivos de dados e índices necessários para os demais programas.

As etapas principais são:

- **Preparação do ambiente**: remove arquivos antigos de dados e índices;
- **Processamento do CSV**: converte cada linha em registros binários, tratando delimitadores e aspas;
- **Inserção no arquivo de dados**: armazena registros em blocos com hashing e encadeamento para colisões;
- **Criação dos índices**: gera o índice primário (`ID`) e secundário (`Título`) usando B+ Trees;
- **Métricas**: exibe quantidade de registros, blocos utilizados e tempo de execução.

Essa etapa garante que os arquivos de dados e índices estejam consistentes e prontos para buscas eficientes pelos programas `findrec`, `seek1` e `seek2`.

5.4. Programas de Busca

5.4.1. Findrec

O programa `findrec` realiza a busca sequencial no arquivo de dados através do identificador único (`ID`). Após encontrar o valor do bucket pela função de hashing, a busca deve percorrer o encadeamento de blocos daquele bucket até achar o registro que está procurando. Optamos por permanecer com uma busca sequencial, para que ela não dependesse de blocos com registros ordenados.

5.4.2. Seek1

O programa `seek1` realiza a busca de registros a partir do índice primário, que é construído com base no identificador único (`ID`) de cada artigo. A função percorre a Árvore B+ associada ao índice primário, navegando pelos nós internos até encontrar o nó folha que contém o ponteiro para o bloco de dados correspondente. A partir desse ponteiro, o programa acessa diretamente o registro no arquivo de dados, minimizando a quantidade de blocos lidos em comparação com a busca sequencial.

Essa abordagem é significativamente mais eficiente do que a busca direta no arquivo de dados, pois a estrutura de índice permite localizar o registro em tempo logarítmico em relação ao número total de chaves. Além disso, o uso da Árvore B+ garante que os registros possam ser percorridos em ordem crescente de ID, caso seja necessário realizar buscas ou varreduras ordenadas.

5.4.3. Seek2

O programa `seek2` tem como objetivo realizar buscas baseadas no campo `título` do artigo. Para isso, é utilizado um índice secundário implementado com uma Árvore B+, semelhante à estrutura do índice primário, mas onde as chaves são os títulos e os valores armazenados são ponteiros para os registros no arquivo de dados.

Durante a execução, o `seek2` recebe como entrada um título de artigo e procura por ele no índice secundário. Quando uma correspondência é encontrada, o programa utiliza os ponteiros armazenados para acessar e exibir os registros correspondentes no arquivo de dados. Além das informações do artigo, o programa também apresenta estatísticas de desempenho, como o número de blocos lidos e o tempo de execução total da busca.

É importante destacar que, na implementação atual, os títulos são armazenados e indexados de forma **literal**, ou seja, sem normalização de caracteres, remoção de espaços extras ou padronização de capitalização. Como consequência, pequenas variações na grafia do título — como diferenças em maiúsculas/minúsculas, acentuação ou espaços — podem fazer com que determinados registros não sejam encontrados durante a busca. Assim, pode haver casos em que o número de artigos retornados pelo `seek2` seja, muitas vezes, menor do que o total esperado, mesmo que existam títulos visualmente idênticos no arquivo de dados.

Essa escolha foi feita pela equipe para manter a consistência entre os dados brutos e o índice gerado durante o processo de upload, simplificando a implementação e evitando divergências entre as chaves indexadas e os registros armazenados.

6. Decisões de Projeto

- Linguagem escolhida: **C++17** pela eficiência e suporte nativo a manipulação de arquivos binários.
- Estruturas de dados: **B+ Tree** e **Hashing**.
- Armazenamento: registros binários com tamanho fixo e controle de blocos.
- Persistência via Docker, montando o diretório `/data`.
- Build automatizado via Makefile.

7. Infraestrutura e Ferramentas

7.1. Sistema de Build

O `Makefile` implementa:

- `make build` – compila todos os executáveis localmente;
- `make clean` – remove binários gerados;
- `make docker-build` – constrói a imagem Docker;
- `make docker-run-*` – executa programas no contêiner.

7.2. Containerização Docker

Dockerfile:

- Base: `gcc:latest` para compilação C++;
- Volume montado: `/data` para persistência;
- Build automático durante construção da imagem.

8. Testes e Validação

8.1. Dados de Teste

Arquivo CSV de exemplo no diretório `data/artigo.csv` com a estrutura definida no enunciado.

8.2. Métricas Coletadas

Durante a execução das buscas, cada programa coleta e exibe métricas quantitativas de desempenho:

- Total de registros encontrados;
- Total de blocos no arquivo de dados;
- Quantidade de blocos lidos na operação;
- Tempo total de execução em milissegundos.

Essas informações permitem avaliar a eficiência das estruturas de indexação e a diferença entre busca direta, sequencial e indexada.

9. Conclusão

O projeto implementa os conceitos de organização de arquivos e indexação vistos em sala de aula, oferecendo um sistema funcional para inserção e consulta eficiente de registros. A estrutura modular e containerizada garante reprodutibilidade, portabilidade e clareza no processo de desenvolvimento.