Trabalho 2 de Banco de Dados

Estefany Licinha Mendes - estefany.silva@icomp.ufam.edu.br
Luiz Henrique Ribeiro - luis.henrique@icomp.ufam.edu.br
Samuel Davi Chagas - samuel.chagas@icomp.ufam.edu.br

Sumário

1. Estrutura dos Arquivos de Dados e Índices

1.1 Arquivo de dados (Hashing)

O arquivo de dados principal (`data/data.db`, nome recebido por parâmetro) armazena os registros físicos dos artigos científicos. Ele é manipulado pelos programas `upload` e `findrec`, sendo acessado via descritor de arquivo (fd) em modo binário. A organização direta dos registros é suportada por um índice hash estático armazenado no arquivo auxiliar `data/hash_index.db`.

Cada bucket do índice contém um vetor de offsets que referenciam diretamente posições no arquivo de dados, com possibilidade de encadeamento para tratamento de colisões.

1.1.1 Estrutura Física

- Arquivo principal (`data.db`): contém os registros serializados sequencialmente, gravados no final do arquivo, com retorno de offset.
- Arquivo de índice hash (`hash_index.db`): contém uma tabela de buckets de tamanho fixo. Cada bucket armazena:
    - count: número de offsets válidos;
    - offsets[]: lista de posições físicas no arquivo de dados;
    - next_overflow: ponteiro para um bucket de overflow, caso ocorra colisão.

1.1.2 Funções Relacionadas

- init_hash_file()`: inicializa o arquivo de buckets com valores nulos.
- hash_id(id)`: aplica função de espalhamento para mapear o ID a um número de bucket.
- insert_offset_into_bucket()`: insere o offset retornado por `write_record()` no bucket correspondente.
- write_record(fd, r)`: grava o registro no final do arquivo de dados e retorna seu offset.
- read_record_by_offset(fd, offset)`: acessa diretamente um registro físico usando seu deslocamento.

1.1.3 Características

- Chave de busca: ID do artigo (inteiro).
- Complexidade média: O(1) para inserção e busca.
- Tratamento de colisões: encadeamento por buckets de overflow.
- Aplicação prática: usado pelo comando `findrec <id>`, que consulta o índice hash e recupera rapidamente o registro pelo offset.

1.2 Índice Primário (B+Tree sobre ID)

O índice primário é responsável por organizar as chaves ID dos artigos em uma B+Tree persistente, armazenada no arquivo binário data/index_primary.idx. Esse índice é manipulado pelos programas upload (durante inserções) e seek1 (para consultas por ID).

1.2.1 Estrutura Física

Cada nó da B+Tree (BPTreeNode) é gravado como um bloco de tamanho fixo (BLOCK_SIZE = 4096 bytes) e contém:

- folha: indica se o nó é folha (1) ou interno (0);
- nChaves: número de chaves válidas;
- chaves[]: vetor de inteiros ordenados;
- valores[]: offsets dos registros correspondentes;
- filhos[]: ponteiros (offsets) para nós filhos;
- proxFolha: ponteiro para a próxima folha (encadeamento de folhas).

O primeiro bloco do arquivo contém o cabeçalho, que armazena:

- rootOffset: posição do nó raiz;
- totalBlocos: número total de blocos válidos.

1.2.2 Funções Principais

- criarArvore(): cria o arquivo da B+Tree, grava o cabeçalho e inicializa a raiz.
- fecharArq(): fecha o arquivo de índice.
- inserirChave(): insere uma nova chave ID e o offset correspondente.
- buscarChave(): percorre a árvore do nó raiz até a folha para localizar o offset.
- escreverBloco() / lerBloco(): gravam e leem nós em disco.

1.2.3 Uso no Sistema

Durante o upload, cada registro é inserido na B+Tree primária: inserirChave(t, r.id, offset. O programa seek1 utiliza a função buscarChave() para encontrar o offset do registro e depois lê o dado diretamente do arquivo de dados.

1.2.4 Características

- Chave primária: ID (inteiro).
- Acesso: sequencial ou direto via árvore.
- Complexidade média: O(log n) para busca/inserção.
- Bloco de armazenamento: 4 KiB (compatível com páginas de disco).
- Encadeamento de folhas: permite varredura ordenada.

1.3 Índice Secundário (B+Tree sobre título)

O índice secundário organiza os registros de artigos com base no campo Título, permitindo busca textual exata. É armazenado em data/index_secondary.idx e utilizado pelo programa seek2.

1.3.1 Estrutura Física

Cada nó é representado pela estrutura BTreeNode:

- is_leaf: indica se o nó é folha;
- num_keys: número de chaves válidas;
- keys[][]: títulos armazenados (cadeias de caracteres);
- data_offsets[]: offsets dos registros no arquivo de dados;
- children[]: ponteiros para filhos;
- next_leaf: ponteiro para a próxima folha (usado em percursos sequenciais).

O arquivo possui um cabeçalho (BTreeHeader) contendo:

- root_offset: posição da raiz da árvore;
- num_nodes: número de nós;
- height: altura atual da árvore.

1.3.2 Funções Principais

- init_btree_file(): cria e inicializa o arquivo de índice.
- insert_into_btree(): insere uma nova chave (título) e o offset.
- search_btree(): busca exata de um título e retorna o offset correspondente.
- close_btree(): fecha o arquivo da B+Tree.

1.3.3 Uso no Sistema

O programa seek2 realiza buscas com base no título: off_t data_offset = search_btree(btree_file, search_titulo.c_str()). Ao encontrar o offset, o programa lê o registro correspondente no arquivo de dados e exibe todos os campos do artigo.

1.3.4 Características

- Chave secundária: Título do artigo (texto).
- Tipo de comparação: igualdade exata (strcmp).
- Suporte a duplicatas: múltiplos títulos podem apontar para offsets diferentes.
- Complexidade média: O(log n) para busca e inserção.
- Bloco padrão: 4096 bytes (mesmo tamanho do índice primário).

2. Estrutura dos Programas

2.1 Upload

Função do módulo: Implementar o programa Upload, responsável por realizar a carga inicial dos dados a partir de um arquivo CSV, populando tanto o arquivo de dados (hashing) quanto os índices primário e secundário. O programa integra todos os módulos do sistema, garantindo a criação correta da base de dados, a inserção de registros no arquivo principal e a atualização sincronizada dos índices em disco.
Fontes: upload.cpp, bptree.cpp, btree_sec.cpp e data_engine.cpp

Funções usadas em upload.cpp:

Função	Papel
parser_csv()	Lê o arquivo CSV e converte as linhas em um Registro record
init_hash_file()	Inicializa o arquivo de índice hash, criando os buckets vazios para inserções futuras.
criarArvore(BPTree &t)	Cria e inicializa a B+Tree primária que será usada na indexação por ID.
hash_id(r.id)	Calcula o bucket correspondente ao ID do registro para uso no índice hash.
insert_offset_into_bucket(hash_path,bucket,offset)	Insere o par `(bucket → offset)` no índice hash (tratando colisões via encadeamento).
InserirChave(t, r.id, offset)	Insere a chave primária (ID) e seu respectivo offset na B+Tree primária, mantendo a ordenação.
close(fd)	Fecha o arquivo de dados após o término das gravações.
fecharArq(t)	Fecha a B+Tree primária, salvando seu estado final.

Responsável : Luis Ribeiro

Funções implementadas em bptree.cpp:

Função	Papel
criarArvore()	Inicializa o arquivo de índice e cria o nó raiz da árvore B+.
fecharArq()	Fecha o arquivo associado à estrutura da árvore, garantindo integridade dos dados.
lerBloco()	Lê um bloco específico do arquivo de índice e carrega o conteúdo do nó em memória.

escreverBloco()	Persiste um nó da árvore B+ em uma posição específica do arquivo em disco.
lerCabecalho()	Lê o cabeçalho do arquivo e atualiza o campo rootOffset na estrutura BPTree.
escreverCabecalho()	Atualiza o cabeçalho do arquivo de índice com o offset atual da raiz da B+Tree.
alocarNo()	Cria um novo nó (folha ou interno) no final do arquivo de índice, inicializando campos padrão.
buscarChave()	Busca um registro na B+Tree pelo campo ID, retornando o offset no arquivo de dados.
inserirChave()	Insere uma nova chave na árvore, mantendo a estrutura balanceada.
inserirRecursivo()	Implementa o algoritmo recursivo de inserção em nós internos e folhas.
inserirNaFolha()	Insere um novo par (chave, valor) dentro de um nó folha, realiza split quando necessário.
limiteSuperior()	Função auxiliar que realiza busca binária para determinar a posição correta de inserção.
tamanhoArquivo()	Retorna o tamanho total do arquivo de índice em bytes, usado para cálculo de blocos.
divArredondada()	Calcula a divisão inteira de dois valores com arredondamento para cima.

Responsável : Estefany Licinha

Funções implementadas em btree_sec.cpp:

Função	Papel
criarArvoreSecundaria()	Inicializa o arquivo do índice secundário (index_secondary.idx) e cria o nó raiz da B+Tree que utilizará o campo Título como chave de indexação.
buscarTitulo()	Fecha o arquivo de índice secundário, garantindo que todas as modificações sejam persistidas.

Responsável: Samuel Davi

Funções implementadas em data_engine.cpp:

Função	Papel
init_hash_file(const char *filename)	Cria e inicializa o arquivo de dados organizado por hashing estático, com buckets vazios
serialize_record(const Record &r)	Converte a estrutura Record (em memória) em uma sequência binária de bytes para gravação em disco, garantindo alinhamento e tamanho fixo dos campos.
deserialize_record(const uint8_t *buf, size_t len)	Executa o processo inverso: lê os bytes de um bloco e reconstrói a estrutura Record em memória
write_record(int fd, const Record &r)	Recebe um registro e escreve no arquivo de dados após aplicar o hashing. Retorna o offset físico onde foi armazenado, usado pelos índices
read_record_by_offset(int fd, off_t offset)	Lê um registro diretamente a partir do offset informado, reconstruindo-o em memória para exibição ou validação
insert_offset_into_bucket(const char *hash_file, uint32_t bucket_id, off_t record_offset)	Gerencia colisões no bucket: caso o bucket esteja cheio, cria uma área de overflow e insere o novo offset nessa estrutura
hash_id(int32_t id)	Calcula o valor de hash para o campo ID, determinando o bucket correto em que o registro será armazenado

Responsável: Luís Ribeiro

2.2 frindrec

Função do módulo: Implementar o programa Findrec, responsável por realizar a busca direta de registros no arquivo de dados, utilizando apenas o endereço físico (offset) gerado pelo hashing. Esse programa não utiliza nenhum índice (nem B+Tree primário, nem secundário); sua função é demonstrar o acesso direto ao arquivo de dados e comparar o custo de I/O em relação às buscas indexadas. Ele também é usado para verificar a integridade dos registros armazenados após o processo de upload.
Fontes: findrec.cpp, data_engine.cpp

Funções usadas em findrec.cpp:

Função	Papel

hash_id()	Calcula o hash do ID para determinar o bucket em que o registro deve estar armazenado
read_record_by_offset()	Lê o registro diretamente a partir do offset calculado ou obtido pela busca no bucket. Retorna a estrutura Record completa
open(hash_path, O_RDONLY)	Abre o arquivo de índice hash para leitura
open(data_path, O_RDONLY)	Abre o arquivo de dados binários
pread(hash_fd,&b, sizeof(Bucket), pos)	Lê diretamente o bucket correspondente ao índice
read_record_by_offset(fd, offset)	Lê o registro físico associado ao offset

Responsável: Luís Ribeiro

Tabela das funções implementadas em data_engine.cpp é apresentada na **seção 2.1**.

2.3 seek1

Função do módulo: Implementar o programa Seek1, responsável por realizar buscas indexadas pelo campo ID, utilizando o índice primário baseado em árvore B+. Esse programa demonstra a integração entre o módulo de índice (bptree.cpp) e o módulo de dados (data_engine.cpp), permitindo recuperar registros de forma eficiente, com custo de acesso proporcional a O(log n). Além de recuperar os dados, o Seek1 contabiliza o número de blocos lidos e o total de blocos do índice, permitindo comparar seu desempenho com a busca direta (findrec).
Fontes: seek1.cpp, bptree.cpp, data_engine.cpp

Funções usadas em seek1.cpp:

Função	Papel
criarArvore()	Inicializa ou abre o arquivo de índice primário (index_primary.idx), garantindo acesso à estrutura persistida da B+Tree
buscarChave()	Realiza a busca hierárquica do ID na árvore B+, retornando o offset do registro no arquivo de dados (hash_index.db). Essa função é a base da operação indexada
lerCabecalho()	Carrega o offset da raiz da B+Tree a partir do cabeçalho do arquivo de índice. É chamada automaticamente ao abrir a estrutura

read_record_by_offset()	Lê o registro localizado no arquivo de dados com base no offset retornado pelo índice
deserialize_record()	Reconstrói o registro em memória, permitindo a exibição legível dos campos (ID, título, autores, etc.)
fecharArq()	Fecha o arquivo de índice após a operação, garantindo integridade e atualização do cabeçalho

Responsável: Estefany Licinha

Tabela das funções implementadas em bptree.cpp e em data_engine.cpp é apresentada na **seção 2.1**.

2.4 seek2

Função do módulo: Implementar o programa Seek2, responsável por realizar buscas indexadas pelo campo Título, utilizando o índice secundário baseado em árvore B+ textual. Esse programa exemplifica a aplicação de um índice não único, permitindo recuperar um ou mais registros que compartilham o mesmo título. O Seek2 integra os módulos de índice secundário (btree_sec.cpp) e arquivo de dados (data_engine.cpp), operando de forma semelhante ao Seek1, porém com chaves textuais e comparação lexicográfica.
Fontes: seek2.cpp, btree_sec.cpp, data_engine.cpp

Funções usadas em seek2.cpp

Função	Papel
hash_id()	Calcula o hash do ID para determinar o bucket em que o registro deve estar armazenado
read_record_by_offset()	Lê o registro diretamente a partir do offset calculado ou obtido pela busca no bucket. Retorna a estrutura Record completa

Responsável: Samuel Davi

Tabela das funções implementadas em btree_sec.cpp e em data_engine.cpp é apresentada na **seção 2.1**.

3. Decisões de Projeto

Optamos pelo hashing estático como método de organização do arquivo de dados, por oferecer acesso direto rápido (O(1)), simplicidade de implementação e controle explícito de colisões por listas de overflow. Cada registro é serializado em formato binário fixo, o que permite leitura

direta via offset e integração com os índices. O hashing dinâmico foi descartado por complexidade e custo de manutenção acima do escopo do trabalho.

O índice primário utiliza uma B+Tree persistente organizada por ID, garantindo busca e inserção em O(log n) e armazenamento ordenado em disco. Essa estrutura foi escolhida por ser balanceada, suportar varreduras sequenciais e reduzir o número de acessos a disco. Cada nó é gravado em blocos de 4 KiB, com cabeçalho e encadeamento de folhas, facilitando varreduras ordenadas e reabertura segura do arquivo.

Por fim, o índice secundário também é uma B+Tree, mas adaptada para chaves textuais e não únicas, permitindo buscas por título. Foi escolhida por garantir ordenação lexicográfica e suporte a registros duplicados, mantendo a mesma estrutura física e lógica do índice primário. Índices invertidos foram descartados, pois exigiriam tokenização e buscas parciais, fora do escopo do projeto.

4. Divisão de Trabalho

4.1 Dados e upload

Responsável: Luis Ribeiro

Principais responsabilidades:

- Desenvolvimento do módulo de dados e do processo de carga inicial (upload).
- Implementação das funções de manipulação direta do arquivo de dados organizado por hashing estático com encadeamento de overflow.

- Implementação da política de tratamento de colisões por meio de buckets de overflow.
- Testes unitários de escrita, leitura e serialização de registros.

Implementações:

- Arquivos: data_engine.cpp, data_engine.h, parte de upload.cpp.
- Funções desenvolvidas:
    - init_hash_file() - Inicializa o arquivo de índice hash com todos os buckets vazios.
    - write_record() - Grava um registro no final do arquivo de dados e retorna seu offset.
    - read_record_by_offset() - Recupera um registro diretamente pelo seu deslocamento físico.
    - hash_id() - Implementa a função de dispersão para cálculo do número do bucket.
    - insert_offset_into_bucket() - Realiza o encadeamento em caso de colisões e armazena o offset.

Testes e validação:

- Validou a serialização/deserialização correta dos registros.
- Verificou o funcionamento da função de hash e o encadeamento de overflow.
- Garantiu a recuperação dos registros por ID utilizando hashing.

Documentação:

O integrante também ficou responsável pela documentação técnica associada à organização física do arquivo de dados, pelas estruturas registradas em disco (Record e Bucket), pela explicação da função de hashing e pelo fluxo completo de armazenamento e leitura de registros via offset.

4.2 Índice primário

Responsável: Estefany Licinha

Principais responsabilidades:

- Implementação completa da B+Tree primária, armazenada em disco, com nós representados por blocos binários.
- Desenvolvimento das funções de inserção, busca e persistência da árvore.
- Integração com o arquivo de dados e implementação do programa seek1 <ID>.

Implementações:

- Arquivos: bptree.cpp, bptree.h, seek1.cpp.
- Funções desenvolvidas:
    - criarArvore() – inicializa a árvore e cria o arquivo de índice primário.
    - fecharArq() - Fecha o arquivo da árvore.
    - lerBloco() / escreverBloco() – funções de I/O direto dos nós da árvore.
    - lerCabecalho() / escreverCabecalho() - Controlam o offset da raiz no arquivo.
    - alocarNo() - Cria novos nós no final do arquivo.
    - buscarChave() – Localiza o offset de um registro pelo ID.
    - inserirChave() / inserirRecursivo() / inserirNaFolha() – Insere e divide nós conforme necessário.

    - limiteSuperior() - Realiza busca binária interna.
    - tamanhoArquivo() / divArredondada() - Calcula estatísticas e medidas de arquivo.
    - Implementação principal de seek1.cpp (consulta pelo índice primário).

Testes e validação:

- Realizou testes de inserção e busca com diferentes quantidades de IDs.
- Validou o controle de splits, o balanceamento da árvore e a contagem de blocos lidos.

Documentação:

Elaborou a documentação do módulo de índice primário, descrevendo o funcionamento interno da B+Tree e sua estrutura em disco.

4.3 Índice secundário, CLI e DevOps

Responsável: Samuel Davi

Principais responsabilidades:

- Implementação do índice secundário baseado em B+Tree textual, utilizando o campo Título como chave.
- Desenvolvimento do programa seek2 <Titulo> e suporte a múltiplos registros com mesmo título.
- Implementação do programa findrec <ID>, integração do sistema e automação via Docker e Makefile.

Implementações:

- Arquivos: btree_sec.cpp, btree_sec.h, seek2.cpp, findrec.cpp, Dockerfile, Makefile, README.md.
- Funções desenvolvidas:
    - criarArvoreSecundaria(BTreeSec&) – inicializa o índice secundário.
    - buscarTitulo(BTreeSec&, std::string titulo) – busca registros pelo título textual.
    - inserirTitulo(BTreeSec&, std::string titulo, ptr_t posDado) – insere novas chaves de título no índice.
    - findrec.cpp – busca direta no arquivo de dados por ID.

DevOps e integração:

- Criou scripts e targets (make build, docker-build, docker-run-*) para execução dos binários no contêiner.
- Montou o README.md e consolidou o relatório final TP2_documentação.pdf.

Testes e validação:

- Validou a busca de títulos com diferentes tamanhos e caracteres.
- Testou integração completa dos programas via Docker (upload, findrec, seek1, seek2).

Documentação:

Responsável por integrar e revisar as partes fornecidas pelos colegas, incluindo o relatório final e a seção de instruções Docker.