Equipe:
Lucas de Souza Cerveira Pereira
Mikaelle Costa de Santana
Roberta Graziela de Oliveira Brasil

JUSTIFICATIVAS A PARTE

1. **Uso de gravação estática ao invés de dinâmica:**

Foi decidido utilizar gravação estática dos artigos ao invés de dinâmica a fim de simplificar o trabalho e trazer ganhos de velocidade no programa upload, que já é lento devido a sua natureza.

2. **Por que duas árvores B +?**

Realizar um hashing do título do artigo para int resultaria em muitas colisões. Para garantir menor uso de memória no índice primário (em que cada artigo tem um ID int) e a singularidade de cada key do índice secundário, foi tomada a decisão de fazer uma árvore B+ para o índice primário (int) e uma árvore B+ idêntica à primeira, mas com long long, a fim de mitigar quaisquer chances de dois títulos gerarem a mesma key.

3. **Uso de fstream ao invés de mmap**

O fstream oferece uma interface de alto nível e mais simples. Ele abstrai a complexidade do gerenciamento de buffers e da interação com o sistema operacional. Com fstream, é muito mais difícil cometer erros de programação que causem falhas no programa, como "bus errors".

Estrutura de cada arquivo de dados e índices

1. **Arquivo de dados (/data/data_file.dat)**
    a. **Propósito:** Armazena os registros completos dos artigos científicos. É a fonte principal de dados consultada diretamente pelo findrec e indiretamente (via ponteiros) pelos programas seek1 e seek2.
    b. **Organização Lógica:** Tabela Hash com Endereçamento Aberto (Sondagem Linear). O arquivo é pré-alocado com um tamanho fixo (block_qntd, configurado em upload.cpp com 750.000) de blocos lógicos.

c. **Estrutura Física:** Uma sequência contígua de blocos (DataBlock). Não há metadados armazenados neste arquivo; seu tamanho é fixo e conhecido pelo programa.
d. **Estrutura do Bloco (DataBlock):** Cada bloco possui um tamanho fixo de sizeof(DataBlock) bytes e contém:
    i. Artigo records[RECORDS_PER_BLOCK]; : Um array com capacidade para RECORDS_PER_BLOCK (calculado como 2) registros completos de Artigos.
    ii. int record_count;: Um inteiro indicando quantos registros estão atualmente armazenados neste bloco (de 0 a RECORDS_PER_BLOCK).
e. **Estrutura do Registro (Artigo):** Cada registro possui um tamanho fixo de sizeof(Artigo) bytes e segue a estrutura definida em record.hpp, utilizando arrays de char de tamanho fixo para garantir a consistência do tamanho:
    i. int ID; (4 bytes)
    ii. char Titulo[301]; (301 bytes)
    iii. int Ano; (4 bytes)
    iv. char Autores[151]; (151 bytes)
    v. int Citacoes; (4 bytes)
    vi. time_t Atualizacao_timestamp; (geralmente long, 8 bytes)
    vii. char Snippet[1025]; (1025 bytes)
f. **Mapeamento e Colisões:** A função de hash ID % total_blocks determina o bloco inicial para um dado ID. Em caso de inserção ou busca, se o bloco inicial estiver cheio ou não contiver o ID desejado, a sondagem linear ((bloco_atual + 1) % total_blocks) é aplicada para verificar os blocos subsequentes.

2. **Arquivo de Índice Primário (/data/primary_index.idx)**
    a. **Propósito:** Permite a busca rápida de registros no data_file.dat com base na chave primária (ID). Utilizado pelo programa seek1.
    b. **Organização Lógica:** Árvore B+ balanceada, otimizada para memória secundária.
    c. **Estrutura Física:** O arquivo consiste em:
        i. **Metadados:** Um bloco inicial de sizeof(BPlusTreeMetadata) bytes no offset 0.
        ii. **Blocos de Nós:** Uma sequência de blocos (BPlusTreeNode), cada um com sizeof(BPlusTreeNode) bytes, começando a partir do offset DATA_START_OFFSET (sizeof(BPlusTreeMetadata)).
    d. **Estrutura dos Metadados (BPlusTreeMetadata):** Armazenada no início do arquivo:
        i. f_ptr root_ptr_offset;: O offset (endereço) do bloco que contém o nó raiz atual da árvore.

    ii. long block_count;: O número total de blocos de nós (BPlusTreeNode) alocados no arquivo.
e. **Estrutura do Bloco/Nó (BPlusTreeNode):** Baseada na constante ORDER = 340, projetada para aproximar 4KB:
    i. bool is_leaf;: Indica se é um nó folha (true) ou interno (false).
    ii. int key_count;: Número de chaves válidas no nó (0 a ORDER - 1).
    iii. int keys[ORDER - 1];: Array ordenado das chaves (ID dos artigos).
    iv. f_ptr children[ORDER];: Array de ponteiros (f_ptr).
        1. Em nós internos: children[i] aponta para o offset (no mesmo arquivo .idx) do nó filho que contém chaves menores que keys[i]. children[key_count] aponta para chaves maiores ou iguais à última chave.
        2. Em nós folha: children[i] armazena o f_ptr (offset no arquivo /data/data_file.dat) do registro Artigo correspondente a keys[i]. Os ponteiros restantes são não usados (-1).
    v. f_ptr next_leaf;: Usado apenas em nós folha. Aponta para o offset (no *mesmo* arquivo .idx) do próximo nó folha na sequência ordenada, formando uma lista ligada. Vale -1 no último nó folha.

3. **Arquivo de Índice Secundário (/data/secondary_index.idx)**
    a. **Propósito:** Permite a busca rápida de registros no data_file.dat com base no Título do artigo. Utilizado pelo programa seek2.
    b. **Organização Lógica:** Árvore B+ balanceada, similar à do índice primário, mas utilizando chaves diferentes.
    c. **Chave Indexada: Importante:** Este índice **não** armazena os títulos diretamente. Ele armazena valores long long que são o resultado da aplicação de uma função de hash (definida em BPlusTree_long.hpp) sobre o campo Titulo (truncado para 300 caracteres) de cada artigo.
    d. **Estrutura Física:** Idêntica à do índice primário, consistindo em metadados seguidos por blocos de nós.
        i. **Metadados:** Um bloco inicial de sizeof(BPlusTree_long_Metadata) bytes no offset 0.
        ii. **Blocos de Nós:** Uma sequência de blocos (BPlusTreeNode_long), cada um com sizeof(BPlusTreeNode_long) bytes, começando a partir do offset DATA_START_OFFSET_LONG (sizeof(BPlusTree_long_Metadata)).
    e. **Estrutura dos Metadados (BPlusTree_long_Metadata):** Idêntica à BPlusTreeMetadata, mas é uma struct separada para melhor organização :
        i. f_ptr root_ptr_offset;

ii. long block_count;

f. **Estrutura do Bloco/Nó (BPlusTreeNode_long):** Baseada na constante ORDER_LONG = 255 (recalculada devido ao tamanho maior da chave long long), também projetada para aproximar 4KB:
   i. bool is_leaf;
   ii. int key_count; (0 a ORDER_LONG - 1)
   iii. long long keys[ORDER_LONG - 1];: Array ordenado dos **hashes** dos títulos.
   iv. f_ptr children[ORDER_LONG];: Array de ponteiros (f_ptr).
      1. *Em nós internos:* Apontam para outros nós no *mesmo* arquivo .idx, separados pelas chaves de hash.
      2. *Em nós folha:* children[i] armazena o f_ptr (offset no arquivo /data/data_file.dat) do registro Artigo cujo hash do título corresponde a keys[i].
   v. **Tratamento de Colisões de Hash:** A busca (seek2) utiliza o hash do título para encontrar um ponteiro f_ptr no índice. No entanto, como colisões de hash (hashes iguais para títulos diferentes) são teoricamente possíveis (embora raras com long long), o programa seek2 deve, após encontrar um f_ptr, ler o Artigo correspondente do data_file.dat e **comparar o Titulo real** com o título buscado para garantir a correspondência exata.

FONTES PRINCIPAIS

record.hpp

- **Objetivo principal**: referência única da struct Artigos para que não seja necessário repetir a sua definição em cada fonte.

log.hpp

- **Objetivo principal:** Fornecer um sistema de logging centralizado e configurável para todas as fontes do projeto. Permite a emissão de mensagens de diferentes níveis (ERROR, WARN, INFO, DEBUG) cuja visibilidade é controlada pela variável de ambiente LOG_LEVEL.
- **Desenvolvido por:** Lucas Pereira
- **Funções principais:**
  - **LogLevel getCurrentLogLevel():** Função (inline) que lê a variável de ambiente LOG_LEVEL na primeira vez que é chamada, determina o nível de log configurado (padronizando para INFO se a variável estiver ausente ou inválida) e retorna esse nível. Nas chamadas subsequentes, retorna o nível já determinado eficientemente.
- **Obs: #define LOG_ERROR(...), #define LOG_WARN(...), #define LOG_INFO(...), #define LOG_DEBUG(...)** são macros que chamam LOG_MSG com o nível e prefixo apropriados, simplificando o uso no código.

hashing.cpp

- **Objetivo principal:** Gerenciar um arquivo de dados utilizando uma estratégia de hashing estático com endereçamento aberto. A tabela é indexada pelo ID (inteiro) para armazenar e recuperar registros do tipo Artigo.
- **Desenvolvido por:** Mikaelle Santana
- **Funções principais:**
    - **HashingFile(const std::string& data_file_path, long num_total_blocks)** - Construtor que abre o arquivo de dados. Se o arquivo não existir, ele o cria e inicializa todos os blocos como vazios.
    - **~HashingFile()** - Destrutor que descarrega as modificações do cache para o disco (flush_cache) e fecha o arquivo de dados.
    - **insert(const Artigo& new_artigo)** – Calcula o hash a partir do ID do Artigo para encontrar um bloco e o insere. Se o bloco estiver cheio, aplica sondagem linear para encontrar o próximo espaço disponível. Retorna o endereço do registro inserido.
    - **find_by_id(int id, int& blocks_read)** – Busca um Artigo pelo seu ID. Começa pelo bloco indicado pela função de hash e, se necessário, continua a busca sequencialmente (sondagem linear) até encontrar o artigo ou um espaço vazio. Retorna o artigo encontrado ou um com ID -1 se a busca falhar.
- **Funções auxiliares:**
    - **hash_function(int key)** - Converte uma chave (ID do artigo) em um endereço de bloco no arquivo usando uma operação de módulo.
    - **read_block(long block_number)** - Lê um bloco de dados do disco. Antes, verifica se o bloco já existe no cache para otimizar o acesso.
    - **write_block(long block_number, const DataBlock& block)** - Escreve uma estrutura DataBlock que está na memória para a sua posição correspondente no arquivo em disco.
    - **flush_cache()** - Garante que todos os blocos que estão no cache de memória sejam escritos de volta no arquivo em disco, para consolidar as alterações.
- **Usa:** log.hpp e record.hpp.

BPlusTree.cpp

- **Objetivo principal:** gerenciar a **árvore B+ primária**, indexada por ID (inteiros).
- **Desenvolvido por:** Lucas Pereira
- **Funções principais:**
    - **BPlusTree(const std::string& index_file_path)** - abre/cria o arquivo de índice primário.
    - **~BPlusTree()** - descarrega modificações restantes no disco, salva metadados se forem atualizados e fecha o arquivo.
    - **insert(int key, f_ptr value)** – insere um par (ID, endereço) no índice.
    - **search(int key)** – busca pelo ID e devolve o ponteiro com a localização do Artigo no data_file.dat.
    - **get_total_blocks()** - retorna a quantidade de blocos no índice.
- **Funções auxiliares:**

    - **BPlusTreeNode read_block(f_ptr block_ptr)** - lê um bloco do arquivo e carrega seus dados para uma estrutura BPlusTreeNode na memória.
    - **void write_block(f_ptr block_ptr, const BPlusTreeNode& node)** - escreve um nó (BPlusTreeNode) no arquivo de índice na posição especificada por block_ptr.
    - **void flush_cache()** - Escreve todos os nós armazenados no cache (node_cache) de volta para o arquivo, garantindo persistência das alterações antes do fechamento.
    - **f_ptr allocate_new_block()** - Cria um novo bloco vazio no final do arquivo de índice e retorna seu deslocamento (f_ptr).
    - **void insert_into_leaf(BPlusTreeNode& leaf, int key, f_ptr data_ptr)** - insere uma nova chave e ponteiro de dado em um nó folha, mantendo a ordenação das chaves.
    - **void split_leaf(BPlusTreeNode& leaf, int key, f_ptr data_ptr, int& promoted_key_out, f_ptr& new_leaf_ptr_out)** - Divide um nó folha cheio em dois novos nós, redistribuindo as chaves e ponteiros.
    - **void insert_into_internal(BPlusTreeNode& node, int key, f_ptr child_ptr)** - Insere uma nova chave e ponteiro de filho em um nó interno, preservando a ordenação das chaves.
    - **void split_internal(BPlusTreeNode& node, int& key_in_out, f_ptr& child_in_out)** - Divide um nó interno cheio em dois, promovendo uma chave ao nível superior.
    - **insert_internal(f_ptr current_ptr, int key, f_ptr data_ptr, int& promoted_key_out, f_ptr& new_child_ptr_out)** - Função recursiva que percorre a árvore a partir do nó atual (current_ptr), inserindo a chave em uma folha apropriada, retorna True se houver uma chave que deve se tornar a nova raiz da árvore.
- **Usa:** log.hpp

BPlusTree_long.cpp

- **Objetivo principal:** gerenciar a **árvore B+ secundária**, indexada pelo hash do título (tipo long long).
- **Desenvolvido por:** Lucas Pereira
- **Funções principais:**
    - **BPlusTree_long(const std::string& index_file_path)** - abre/cria o arquivo de índice secundário.
    - **~BPlusTree_long()** - descarrega modificações restantes no disco, salva metadados se forem atualizados e e fecha o arquivo.
    - **insert(long long key, f_ptr value)** – insere um par (Hash do título, endereço) no índice.
    - **search(long long key)** – busca pelo hash do título e devolve o primeiro ponteiro com a localização do Artigo no data_file.dat.
    - **hash_string_to_long(const char* str)** - transforma o índice
    - **get_total_blocks()** - retorna a quantidade de blocos no índice secundário.
- **Funções secundárias:**
    - **BPlusTree_long_Node read_block(f_ptr block_ptr)** - lê um bloco do arquivo e carrega seus dados para uma estrutura BPlusTree_long_Node na memória.

    - **void write_block(f_ptr block_ptr, const BPlusTree_long_Node& node)** - escreve um nó (BPlusTree_long_Node) no arquivo de índice na posição especificada por block_ptr.
    - **void flush_cache()** - Escreve todos os nós armazenados no cache (node_cache) de volta para o arquivo, garantindo persistência das alterações antes do fechamento.
    - **f_ptr allocate_new_block()** - Cria um novo bloco vazio no final do arquivo de índice e retorna seu deslocamento (f_ptr).
    - **void insert_into_leaf(BPlusTreeNode& leaf, long long key, f_ptr data_ptr)** - insere uma nova chave e ponteiro de dado em um nó folha, mantendo a ordenação das chaves.
    - **void split_leaf(BPlusTreeNode& leaf, long long key, f_ptr data_ptr, long long& promoted_key_out, f_ptr& new_leaf_ptr_out)** - Divide um nó folha cheio em dois novos nós, redistribuindo as chaves e ponteiros.
    - **void insert_into_internal(BPlusTree_long_Node& node, long long key, f_ptr child_ptr)** - Insere uma nova chave e ponteiro de filho em um nó interno, preservando a ordenação das chaves.
    - **void split_internal(BPlusTree_long_Node& node, long long& key_in_out, f_ptr& child_in_out)** - Divide um nó interno cheio em dois, promovendo uma chave ao nível superior.
    - **insert_internal(f_ptr current_ptr, long long key, f_ptr data_ptr, long long& promoted_key_out, f_ptr& new_child_ptr_out)** - Função recursiva que percorre a árvore a partir do nó atual (current_ptr), inserindo a chave em uma folha apropriada, retorna True se houver uma chave que deve se tornar a nova raiz da árvore.
- Usa: log.hpp.

upload.cpp

- **Objetivo principal:** realizar a carga inicial do arquivo CSV, tratando os dados e criando o arquivo de dados (hashing) e os índices primário e secundário em disco.
- **Desenvolvido por:** Roberta Brasil
- **Funções principais:**
    - **void trim(std::string& s)** - Remove espaços em branco do início e fim da string (modifica in-place).
    - **parse_csv_line(const std::string& line, Artigo& artigo)**, converte uma linha CSV em um objeto Artigo.
    - **main(int argc, char** argv)**, lê o CSV, preenche os registros e insere cada um no arquivo de dados e nas B+ Trees.
- Usa: hashing.hpp, BPlusTree.hpp, BPlusTree_long.hpp, record.hpp e log.hpp.

findrec.cpp

- **Objetivo principal:** Fornecer um programa executável de linha de comando para buscar um Artigo específico no arquivo de dados (data_file.dat) usando seu ID. O programa exibe o registro encontrado ou uma mensagem de falha, juntamente com métricas de desempenho da busca.

- **Desenvolvido por:** Mikaelle Santana
- **Funções principais:**
    - **main(int argc, char* argv[])** - Ponto de entrada do programa. Ele processa os argumentos da linha de comando para obter o ID a ser buscado, inicializa a classe HashingFile para interagir com o arquivo de dados, chama a função de busca e, por fim, exibe os resultados (o artigo encontrado ou uma mensagem de erro) e as estatísticas da operação (blocos lidos).
    - **print_artigo(const Artigo& artigo)** - Função auxiliar que recebe um Artigo e imprime todos os seus campos (ID, Titulo, Ano, etc.) no console de forma legível e formatada.
- **Usa:** record.hpp, hashing.hpp e record.hpp.

seek1.cpp

- **Objetivo principal:** buscar um registro específico, dado o seu ID, através do índice primário (a árvore B+ baseada em ID).
- **Desenvolvido por:** Lucas Pereira
- **Funções principais:**
    - **print_artigo(const Artigo& artigo)** - função para auxiliar a impressão dos campos de um artigo
    - **main()** - valida o ID passado, busca-o na B+ Tree primária (/data/primary_index.idx) para obter o ponteiro do registro no arquivo de dados (/data/data_file.dat), lê o Artigo na posição indicada, exibe-o com print_artigo e mostra as métricas da busca.
    - (usa BPlusTree.hpp e record.hpp).
- **Usa:** record.hpp, BPlusTree.hpp e log.hpp.

seek2.cpp

- **Objetivo principal:** buscar um registro específico, dado o seu Título, através do índice primário (a árvore B+ baseada em hash de título).
- **Desenvolvido por:** Lucas Pereira
- **Funções principais:**
    - **print_artigo(const Artigo& artigo)** - função para auxiliar a impressão dos campos de um artigo
    - **main ()** - valida e reconstrói o Título, trunca para 300 caracteres, calcula o hash com hash_string_to_long, busca o hash na B+ Tree secundária (/data/secondary_index.idx), lê o registro correspondente em /data/data_file.dat, verifica colisões de hash e, se o título coincidir, exibe o artigo e as métricas da busca.
- **Usa:** record.hpp, BPlusTree_long.hpp e log.hpp.

PROGRAMAS

1. **Upload**

- **Função:** lê o arquivo CSV de entrada e cria:
    - Arquivo de dados (organizado por hashing);
    - Índice primário (B+Tree de ID);
    - Índice secundário (B+Tree de hash de título).

- **Fontes utilizadas:**

    - upload.cpp (lógica principal, desenvolvido por Roberta Brasil)
    - hashing.cpp (manipulação do arquivo de dados, desenvolvido por
    - BPlusTree.cpp (índice primário, desenvolvido por Lucas)
    - BPlusTree_long.cpp (índice secundário, desenvolvido por Lucas)

- **Headers correspondentes:**

    - upload.hpp (Roberta Brasil)
    - hashing.hpp (Mikaelle Santana)
    - BPlusTree.hpp (Lucas)
    - BPlusTree_long.hpp (Lucas)
- **Obs:** esse programa utiliza a definição de artigo presente em record.hpp e os LOG_LEVELS definidos em log.hpp.

2. Findrec

- **Função:** Busca um registro de Artigo no arquivo de dados (data_file.dat) com base em um ID fornecido como argumento de linha de comando. Exibe o registro encontrado ou uma mensagem de falha, juntamente com as métricas da busca.
- **Fontes utilizadas:**
    - **findrec.cpp (**lógica principal, desenvolvido por Mikaelle Santana)
    - **hashing.cpp** (manipulação do arquivo de dados, desenvolvido por Mikaelle Santana)
- Headers correspondentes:
    - **findrec.hpp** (desenvolvido por Mikaelle Santana)
    - **hashing.hpp** (desenvolvido por Mikaelle Santana)

Obs: Esse programa utiliza a definição de artigo presente em record.hppe os LOG_LEVELS definidos em log.hpp.

3. Seek 1

- **Função:** Busca um registro específico, dado o seu ID, utilizando o índice primário (BPlusTree de ID) para encontrar a localização do registro no arquivo de dados.
- **Fontes utilizadas:**
    - **seek1.cpp** (lógica principal, desenvolvido por Lucas)
    - **BPlusTree.cpp** (manipulação do índice primário, desenvolvido por Lucas)
- **Headers correspondentes:**
    - **seek1.hpp** (desenvolvido por Lucas)
    - **BPlusTree.hpp** (desenvolvido por Lucas)

Obs: Esse programa utiliza a definição de artigo presente em record.hpp e os LOG_LEVELS definidos em log.hpp.

3. Seek 2

- **Função:** Busca um registro específico, dado o seu Titulo, calculando o hash do título e utilizando o índice secundário (BPlusTree de hash de título) para encontrar a localização do registro no arquivo de dados. Realiza verificação final do título para tratar colisões de hash.
- **Fontes utilizadas:**
    - seek2.cpp (lógica principal, desenvolvido por Lucas)
    - BPlusTree_long.cpp (manipulação do índice secundário, desenvolvido por Lucas)
- **Headers correspondentes:**
    - seek2.cpp (desenvolvido por Lucas)
    - BPlusTree.hpp (desenvolvido por Lucas)

Obs: Esse programa utiliza a definição de artigo presente em record.hpp, a função de hash de título definida em BPlusTree_long.hpp e os LOG_LEVELS definidos em log.hpp.