Relatório Trabalho 2
Banco de Dados I

Aline de Souza Sahdo e Silva, Carolina Falabelo Maycá,
Fernando Lucas Almeida Nascimento

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

{aline.silva,carolina.mayca,fernando.nascimento}@icomp.ufam.edu.br

Sumário

1. Decisões de Projeto

1.1. Organização Geral do Sistema

O sistema foi dividido nos seguintes componentes:

- **upload:** Responsável por ler o arquivo *artigo.csv*, armazenar os registros em um arquivo binário (artigos.dat) organizado em hashing em disco e construir dois índices auxiliares:
  - Índice primário baseado no campo **ID**, armazenado em uma Árvore B+ (arquivo prim_index.idx);
  - Índice secundário baseado no campo **Título**, armazenado em uma Árvore B+ (arquivo sec_index.idx) após normalização e hashing do texto.

  Antes de iniciar o processamento, todos os arquivos anteriores são removidos, garantindo um ambiente limpo e determinístico.
- **hashing_file:** Implementa a classe responsável pela manipulação do arquivo de dados em hashing (`artigos.dat`) e do diretório auxiliar `tabela_hash.idx`. É utilizada pelo programa `upload` durante a etapa de inserção fisicamente em disco. Realiza:
  - criação dos arquivos binários de dados e da tabela de hash quando inexistentes;
  - cálculo da posição (bucket) a partir do ID e acesso ao bloco inicial correspondente;
  - inserção em blocos com encadeamento em caso de colisões;
  - recuperação de registros por ID, percorrendo a cadeia de blocos;

  Encapsula toda a lógica de leitura, escrita e alocação de blocos, mantendo a organização física dos artigos acessível a partir de `findrec`.
- **findrec:** Localiza artigos com base no **ID**, consultando o hash.
- **seek1:** Localiza artigos com base no **ID**, consultando o índice primário armazenado em Árvore B+.
- **seek2:** Localiza artigos com base no **Título**, consultando o índice secundário em Árvore B+ a partir do valor hash da versão normalizada do texto.

Essa separação permite modularidade, reprodutibilidade das etapas e possibilidade de reindexação independente do arquivo de dados.

1.2. Gerenciamento de Caminhos e Configuração

Foi implementado um sistema de configuração centralizado no arquivo `config.h`, responsável por padronizar diretórios e nomes de arquivos utilizados em diferentes componentes da aplicação. Esse arquivo utiliza variáveis de ambiente com valores fallback para garantir execução consistente tanto em ambiente local quanto no Docker.

- **Variáveis configuráveis:** `DATA_DIR`, `BIN_DIR`, `DB_DIR`, permitindo redirecionamento de caminhos sem recompilar o código.
- **Fallback automático:** Valores padrão asseguram funcionamento mesmo sem variáveis de ambiente definidas.
- **Organização lógica:** Dados, índices e executáveis são isolados em diretórios distintos, facilitando manutenção e limpeza.
- **Portabilidade:** O mesmo código-fonte é compatível em máquinas diferentes, evitando caminhos absolutos configurados manualmente.

2. Estrutura dos Arquivos

2.1. Arquivo de Dados (CSV)

O arquivo de entrada `artigo.csv` contém os registros originais de artigos científicos. Os campos estão estruturados no formato abaixo:

- **ID** (inteiro)
- **Título** (string, até 300 caracteres)
- **Ano** (inteiro)
- **Autores** (string, separação interna por vírgulas)

- **Citações** (inteiro)
- **Atualização** (string formato YYYY-MM-DD)
- **Snippet** (trecho textual adicional, opcional)

Os campos são delimitados por ponto e vírgula. Em casos de conteúdo textual contendo ponto e vírgula, o campo está protegido por aspas duplas

2.2. Arquivo de dados em Hashing

O Arquivo de Dados em Hashing, denominado `artigos.dat`, é responsável por armazenar os registros completos dos artigos científicos. A sua organização visa permitir o acesso rápido aos dados a partir do ID do artigo, utilizando a técnica de hashing estático com tratamento de colisões por encadeamento. Sua estrutura e organização estão compostos por:

- **Conteúdo**: Armazena os registros completos de cada artigo, definidos pela estrutura *Artigo*, que inclui campos como ID, Título, Ano, Autores, Citações, Data de Atualização e Snippet.
- **Estrutura de Bloco**: O arquivo é fisicamente dividido em blocos de tamanho fixo. Cada bloco é um contentor capaz de armazenar múltiplos registros. Esta abordagem visa melhorar a utilização do espaço e reduzir o número de operações de leitura/escrita no disco.
- **Hashing**: A localização inicial de um artigo é determinada pela aplicação de uma função de hash ao ID do artigo. O resultado desta função corresponde a um índice no arquivo auxiliar `tabela_hash.idx`.
- **Tratamento de Colisões**:
  - O arquivo auxiliar `tabela_hash.idx` funciona como um diretório, onde cada entrada armazena o offset (posição em bytes) do primeiro bloco de dados (Bloco) correspondente àquele resultado de hash no arquivo `artigos.dat`.
  - Quando ocorrem colisões, os artigos são inseridos no primeiro bloco disponível na cadeia. Se um bloco fica cheio, um novo bloco (bloco de overflow) é alocado no final do arquivo `artigos.dat`, e o bloco anterior é atualizado para apontar para este novo bloco através do campo *proximo_bloco_offset*. Isto forma uma lista encadeada de blocos para cada entrada da tabela hash.
- **Registros de tamanho fixo**: A estrutura *Artigo* utiliza campos de tamanho fixo para garantir que cada registro ocupe o mesmo espaço. Consequentemente, a estrutura Bloco também tem um tamanho fixo, simplificando os cálculos de offset para leitura e escrita no arquivo.

2.3. Arquivo de Índice Primário (prim_index.idx)

O índice primário tem como objetivo acelerar a localização de artigos a partir do campo **ID**. Ele não armazena os registros completos, mas sim pares `(ID, RID)`, onde o `RID` (Record Identifier) identifica de forma lógica a posição do registro dentro do arquivo `artigos.dat`.

Sua construção ocorre após o carregamento dos dados através dos seguintes passos:

- Leitura sequencial do arquivo `artigos.dat`, bloco a bloco;
- Para cada registro ocupado, é calculado um **RID** por:
  - multiplicação do índice do bloco por 2 (número de registros por bloco),
  - somado à posição do registro dentro do bloco;
- Armazenamento dos pares `(ID, RID)` em memória;
- Ordenação dos pares por `ID`;
- Inserção ordenada em uma Árvore B+ persistida em disco.

O arquivo gerado (`db/prim_index.idx`) contém apenas chaves e ponteiros de navegação da Árvore B+, permitindo buscas logarítmicas.

Ao consultar um ID, o programa `seek1`:

1. Busca o `RID` correspondente na Árvore B+;
2. Converte o `RID` em:

    - índice físico do bloco no arquivo `artigos.dat`;
    - posição do registro dentro do bloco;
3. Carrega apenas o bloco necessário do disco;
4. Exibe o conteúdo completo do artigo.

Dessa forma, o índice primário atua como um mapa entre IDs e posições físicas, reduzindo drasticamente leituras desnecessárias no arquivo de dados.

2.4. Arquivo de Índice Secundário

O índice secundário tem como objetivo acelerar a localização de artigos a partir do campo **Título**. Como o título não é um campo de chave primária, podem existir valores repetidos na árvore, correspondendo a diferentes artigos com o mesmo nome.

Sua estrutura segue o mesmo formato do índice primário, com a diferença de que, antes de realizar operações de inserção ou busca, a chave do tipo *string* é convertida em um valor inteiro por meio de uma função de hash. Dessa forma, a árvore B+ armazena e organiza os títulos de forma eficiente, permitindo consultas rápidas mesmo em casos de duplicidade.

Além disso, ao realizar uma busca, são retornados todos os artigos cujos títulos correspondem ao valor pesquisado, garantindo que todas as ocorrências sejam recuperadas.

3. Fontes do Projeto

3.1. `include/BPlusTree.hpp`

Este arquivo define a implementação de uma Árvore B+. A implementação gerencia nós em arquivos binários com suporte a inserções, buscas pontuais e buscas por múltiplas ocorrências.

- **Macros e constantes**:
  - `BLOCK_SIZE`: Define o tamanho de bloco de disco como 4096 bytes (padrão de sistemas operacionais);
  - `M`: Define a ordem da árvore B+ (mínimo M chaves, máximo 2*M chaves por nó).
- **Classe `FileManager`**:
  - Gerencia operações de I/O em arquivos binários, incluindo leitura/escrita de cabeçalho, nós e dados;
  - Mantém metadados do arquivo (`FileHeader`) contendo offset da raiz, próximo offset livre e ordem da árvore;
  - Controla alocação de espaço em disco via `getNewOffset()`, incrementando offsets em blocos de `BLOCK_SIZE`;
  - Fornece métodos para leitura/escrita de nós (`readNode`, `writeNode`) e dados (`writeData`);
  - Rastreia estatísticas de I/O através do contador `blocksRead`.
- **Classe `BPlusTree`**:
  - **Estrutura `BPlusTreeNode`**: Define a estrutura de um nó da árvore contendo:
    - Array de chaves (`keys`) e número atual de chaves (`numKeys`);
    - Flag `isLeaf` indicando se o nó é folha;
    - Array de offsets para filhos (`childrenOffsets`) ou dados (em folhas);
    - Offset para próxima folha (`nextLeafOffset`) para varredura sequencial.
  - **Construtor e destrutor**:
    - Inicializa `FileManager`, carrega offset da raiz ou cria nova árvore se necessário;
    - Atualiza offset da raiz no destrutor para persistência.
  - **Inserção**:
    - `insert(key, data)`: Insere chave e dados, alocando offset para dados e invocando inserção recursiva;
    - `insert(key, dataOffset, nodeOffset)`: Desce recursivamente até a folha correta, inserindo ordenadamente ou dividindo nós cheios;

        * `splitLeaf`: Divide nó folha cheio, criando nova folha e promovendo primeira chave do novo nó;
        * `splitInternal`: Divide nó interno cheio durante inserção, promovendo chave do meio para o pai.
    - **Buscas**:
        * `search(k)`: Retorna offset do nó folha contendo a chave `k` (ou 0 se não encontrada);
        * `searchAll(k)`: Retorna todos os offsets de dados associados à chave `k`, percorrendo folhas consecutivas;
        * Utiliza funções auxiliares `upperBound` e `lowerBound` para buscas binárias em arrays ordenados.
    - **Utilitários**:
        * `newNode`: Aloca novo nó no arquivo, inicializando campos e retornando offset;
        * `findParent`: Encontra offset do nó pai de um nó dado (usado durante divisões).

3.2. `src/upload.cpp`

Este arquivo é responsável pela etapa de carga (upload) do projeto, realizando a preparação inicial do ambiente, leitura e normalização dos dados, e construção dos arquivos binários fundamentais para as consultas posteriores. Toda a execução exibe progresso detalhado, estatísticas de tempo e feedback incremental.

- **Função principal (`main()`)**:
    - Remove arquivos previamente gerados (`artigos.dat`, índices primário e secundário), garantindo ambiente limpo e determinístico;
    - Invoca `insereHashing()`, responsável por ler o `artigo.csv`, interpretar corretamente campos entre aspas e inserir cada registro em um arquivo de hashing em disco;
    - Executa `insereIdxPrim()`, que percorre todos os blocos do arquivo de dados, coleta pares `(ID, RID)`, ordena em memória e constrói uma Árvore B+ otimizada;
    - Chama `insereIdxSec()`, que gera um índice secundário baseado em hash FNV-1a aplicado ao título normalizado e insere os pares ordenados em uma segunda Árvore B+.
- **`insereHashing()`**:
    - Realiza leitura linha a linha utilizando `parseCSVLine()`, tratando aspas e delimitadores corretamente;
    - Constrói a estrutura `Artigo` populando todos os campos relevantes (ID, título, ano, autores, citações, data de atualização e snippet);
    - Insere cada registro no arquivo `artigos.dat` via `HashingFile`, utilizando bucket computation e encadeamento interno para colisões;
    - Executa medição de tempo através de `std::chrono` e exibe logs periódicos a cada 50.000 registros inseridos;
    - Emite mensagens de alerta para linhas inválidas ou mal formatadas, incluindo contexto da linha original.
- **`insereIdxPrim()`**:
    - Percorre sequencialmente `artigos.dat` em blocos, examinando somente registros marcados como ocupados;
    - Calcula o **RID** (Record Identifier) combinando índice de bloco e posição no bloco;
    - Armazena os pares `(ID, RID)` em um vetor temporário pré-alocado em memória;
    - Ordena os pares para reduzir splits e incrementar o fator de compactação da Árvore B+;
    - Realiza inserções em lotes de 100.000 chaves, exibindo percentual, tempo acumulado e throughput;
    - Exibe estatísticas consolidadas após o término (tempo total e total de chaves inseridas).
- **`insereIdxSec()`**:
    - Normaliza o título de cada registro (remoção de extremos e conversão para minúsculo);
    - Aplica o hash FNV-1a de 32 bits para gerar uma chave compacta e comparável;
    - Coleta pares `(hash do título, RID)` em vetor temporário;

- Descarta registros cujo título normalizado resulte vazio, mantendo contagem para relatório;
- Ordena os pares por chave para favorecer inserções contínuas na Árvore B+;
- Executa inserção em lotes de 100.000 entradas com indicação percentual e tempo decorrido;
- Finaliza exibindo tempo total e estatísticas de processamento.

Este arquivo é, portanto, responsável por todas as transformações de dados, persistência e pré-processamento para consultas futuras.

3.3. `src/hashing_file.cpp`

Este arquivo implementa a classe *HashingFile*, responsável pela gestão do arquivo de dados principal (`artigos.dat`) e do seu respetivo índice de hash `tabela_hash.idx`. Utiliza a técnica de hashing estático com encadeamento separado por blocos para organizar e aceder aos registros de artigos.

- Classe *HashingFile*: Encapsula toda a lógica de manipulação dos arquivos de hashing.
  - `Construtor(HashingFile(dataFile, tableSize))`:
    - Recebe o nome do arquivo de dados e o tamanho da tabela hash.
    - Tenta abrir o arquivo de dados para leitura e escrita. Se não conseguir, regista um aviso.
    - Nota: O nome do arquivo de índice (`tabela_hash.idx`) não é passado; caminhos fixos são usados internamente nas outras funções.
  - `Destrutor(HashingFile())`:
    - Garante que o arquivo de dados seja fechado corretamente ao destruir o objeto.
- `criarArquivos()` (privado):
  - Cria um arquivo `artigos.dat` vazio (usando o nome passado ao construtor).
  - Cria um arquivo `tabela_hash.idx` com um caminho fixo (`/app/bin/tabela_hash.idx`), preenchendo-o com offsets -1.
- `inserirArtigos(Artigo& novoArtigo)`:
  - Verifica se o arquivo de dados está aberto. Se não estiver, chama `criarArquivos()` e tenta reabrir o arquivo de dados.
  - Calcula o endereço hash para o *novoArtigo* baseado no seu id.
  - Abre o arquivo de índice (`/app/bin/tabela_hash.idx`) para encontrar o início da cadeia de blocos.
  - Percorre a cadeia de blocos em `artigos.dat`:
    - Se encontrar espaço num bloco existente, insere o artigo e atualiza o bloco no disco.
    - Se chegar ao final da cadeia sem encontrar espaço, aloca um novo bloco de overflow no final do `artigos.dat`, insere o artigo nele e atualiza o ponteiro *proximo_bloco_offset* do bloco anterior.
    - Se a cadeia estava vazia, atualiza a entrada correspondente no `tabela_hash.idx` para apontar para o novo bloco criado.
  - Retorna 0
- `buscarPorId(int id, int& blocosLidos)`
  - Calcula o endereço hash para o id procurado.
  - Abre o arquivo de índice para obter o offset do primeiro bloco da cadeia.
  - Percorre a lista encadeada de blocos em `artigos.dat`:
    - Lê cada bloco do disco (incrementando blocosLidos).
    - Procura o artigo com o id correspondente dentro dos registros do bloco atual.
    - Se encontrar, retorna o Artigo.
    - Se não encontrar no bloco, segue para *proximo_bloco_offset*.
  - Se percorrer toda a cadeia e não encontrar, retorna um *Artigo* vazio (com *ocupado* = *false*).
- `getTotalBlocos()`:
  - Calcula e retorna o número total de blocos presentes no arquivo `artigos.dat`, baseado no tamanho total do arquivo.

3.4. `src/findrec.cpp`

Este arquivo implementa o programa findrec, responsável por realizar buscas diretas no arquivo de dados (`artigos.dat`) por um registro específico, utilizando o seu ID como chave. A busca é efetuada através da estrutura de hashing implementada na classe *HashingFile*.

- **Funcionalidade Principal**:
  - Recebe um ID de artigo como argumento da linha de comando.
  - Valida se o argumento fornecido é um número inteiro válido.
  - Instancia a classe *HashingFile*, fornecendo o nome do arquivo de dados (`artigos.dat`) e o tamanho da tabela hash (que deve ser o mesmo utilizado durante a carga pelo upload). Nota: O código atual assume um caminho fixo (`/data/artigos.dat`) e não recebe o nome do arquivo de índice hash como parâmetro, o que pode requerer ajuste dependendo da versão final da classe *HashingFile*.
  - Chama o método `buscarPorId` da *HashingFile* para localizar o registro correspondente ao ID fornecido.
  - Chama o método `getTotalBlocos` da *HashingFile* para obter o tamanho total do arquivo de dados.
  - Utiliza a função `imprimirArtigoCompleto` para exibir todos os campos do registro encontrado ou uma mensagem indicando que o registro não foi localizado.
  - Exibe a quantidade de blocos lidos do disco durante a operação de busca (*blocosLidos* retornado por `buscarPorId`).
  - Exibe a quantidade total de blocos no arquivo de dados (`artigos.dat`).
- **Funções Auxiliares:**
  - `imprimirArtigoCompleto(const Artigo& art)`: Formata e imprime os detalhes de um Artigo encontrado de forma legível no terminal.
- **Tratamento de erros:**
  - Verifica se o número correto de argumentos foi passado na linha de comando.
  - Utiliza um bloco try-catch para lidar com possíveis exceções durante a conversão do ID (argumento inválido) ou erros que possam ocorrer na classe *HashingFile* (ex: arquivo de dados não encontrado)

3.5. `src/seek1.cpp`

Este arquivo implementa consultas de artigos pelo campo **ID** utilizando o índice primário armazenado em uma Árvore B+ persistida em disco.

- **Configuração de Log (variável de ambiente `LOG_LEVEL`)**:
  - O programa lê a variável de ambiente `LOG_LEVEL` (*error*, *warn*, *info*, *debug*);
  - Os níveis controlam granularidade de mensagens;
  - Requerido para depuração sem recompilação;
  - Auxilia na verificação de caminho de arquivos e offsets.
- **Funções auxiliares**:
  - `logError()`, `logWarn()`, `logInfo()` e `logDebug()`:
    - Centralizam formatação de mensagens;
    - Respeitam o nível atual de log;
    - Mantêm a saída padronizada.
  - `fixEncoding()`:
    - Sanitiza caracteres UTF-8 mal interpretados;
    - Substitui sequências inválidas por símbolos seguros.
  - `truncateSnippet()`:
    - Evita poluição visual do terminal;
    - Limita o campo snippet;
    - Concatena indicador de truncamento.
- **Estrutura de Retorno: `SearchResult`**
  - Encapsula estatísticas da execução:

        * sucesso da operação;
        * blocos da árvore B+ lidos;
        * blocos do arquivo de índice acessados;
        * blocos de dados em `artigos.dat`;
        * tempo total em milissegundos.
    - Facilita geração do relatório final ao usuário.
- **`search_primary_index()`**:
    - Temporiza a execução usando `std::chrono`;
    - Invoca `idx.search(ID)`, contabilizando páginas acessadas;
    - Realiza `searchAll()` para recuperar offset(s) associados;
    - Realiza leitura direta do arquivo `prim_index.idx`:
        * valida limites de offset,
        * detecta corrupção de índice,
        * informa erros com contexto.
    - Calcula `RID` $\Rightarrow$ bloco e posição (0 ou 1);
    - Efetua **seek** direto em `artigos.dat`;
    - Lê o bloco inteiro para acesso local;
    - Exibe:
        * ID, título, ano, autores,
        * data de atualização, citações,
        * snippet tratado e truncado.
    - Atualiza contadores internos de blocos acessados.
- **Função principal (`main()`)**:
    - Recebe o ID desejado via linha de comando;
    - Configura nível de log a partir de `LOG_LEVEL`;
    - Inicializa a Árvore B+ com o arquivo `prim_index.idx`;
    - Reseta métricas internas da estrutura;
    - Executa `search_primary_index()`;
    - Imprime relatório consolidado:
        * blocos da árvore lidos;
        * blocos do índice primário acessados;
        * blocos de dados acessados;
        * tempo total da operação em milissegundos;
        * total de blocos envolvidos na consulta.

Erros comuns (ID inexistente, offset inválido, registro não ocupado, arquivo corrompido) são reportados com mensagens descritivas, armazenando contexto para facilitar depuração.

3.6. `src/seek2.cpp`

As funções de log do `seek1` também estão presentes no `seek2`.

- **Layout dos Registros em Disco**:
    - `ArticleDisk`: espelha o formato de um artigo em `artigos.dat` (campos fixos em C `char[]` e inteiros);
    - `Bloco`: agrega `REGISTROS_POR_BLOCO = 2` artigos por página física no arquivo de dados, além de metadados:
        * `num_registros_usados`: quantos slots estão ocupados (0, 1 ou 2);
        * `proximo_bloco_offset`: encadeamento físico (caso necessário).
    - **Mapeamento RID $\Rightarrow$ posição**: dado `RID` (índice lógico do artigo), O bloco é lido integralmente e o slot é validado (`ocupado` e dentro de `num_registros_usados`).
- **Normalização e Hash de Títulos**:
    - `trim()`: remove espaços em branco das extremidades da *string*;
    - `normalize()`: aplica o `trim` (mantém o conteúdo original do título após limpeza de bordas);

  - `fnv1a32()`: implementa o **FNV-1a 32 bits** sobre a *string* normalizada, produzindo a chave inteira usada na Árvore B+.
  - **Duplicidades**: títulos idênticos geram a mesma chave. O código utiliza a função `searchAll()` para recuperar *todas* as ocorrências associadas àquela chave.
  - **Colisões**: títulos diferentes possuem uma probabilidade ínfima (mas não nula) de gerar o mesmo valor de hash. Nesses casos, há um tratamento adicional que verifica se o título encontrado é realmente igual ao título buscado, prevenindo a exibição de resultados irrelevantes.

- **Tratamento de Codificação para Exibição**:
  - `fixEncoding()`: corrige alguns casos comuns de UTF-8 mal interpretado (por exemplo, travessão e apóstrofo), mantém `\n`/`\t` e substitui bytes não imprimíveis por espaço, melhorando a legibilidade no terminal.

- **Fluxo de Busca: `search_bplus_index()`**:
  - Normaliza o título de entrada; falha rápido se vazio;
  - Converte o título em `key = fnv1a32(normalized)` (com cast para `int`);
  - `idx.search(key)`: verifica existência da chave (retorna offset de folha; `0` indica não encontrado);
  - `idx.searchAll(key)`: coleta todos os *offsets* no arquivo de índice (`sec_index.idx`) onde estão gravados os `RID`s correspondentes;
  - Para cada ocorrência:
    * Abre `sec_index.idx` em modo binário, faz `seek` até o *offset* retornado e lê um `long` (`actualRID`);
    * Converte `actualRID` em `blockIndex` e `positionInBlock` e lê o bloco alvo em `artigos.dat`;
    * Valida limites e `ocupado`; em caso afirmativo, imprime os campos do artigo com `fixEncoding()` aplicado.
  - Mensagens descritivas são emitidas para casos de: título não encontrado, erro ao abrir/lêr arquivos, offsets inválidos, slot desocupado ou fora do limite do bloco.

- **Função Principal (`main()`)**:
  - Concatena todos os argumentos da linha de comando em um título único (preserva espaços);
  - Informa o termo pesquisado no `stdout`;
  - Inicializa a BPlusTree utilizando o arquivo de indíce criado durante o upload (`sec_index.idx`) e zera estatísticas com `resetStats()`;
  - Chama `search_bplus_index(idx, titulo)`;
  - Ao final, imprime os blocos lidos da Árvore B+ via `idx.getBlocksRead()`.

- **Saída e Comportamento em Duplicidades**:
  - Para uma chave válida, `searchAll()` pode retornar uma ou várias ocorrências;
  - O programa imprime "*Encontradas N ocorrências*" e lista cada artigo correspondente (títulos iguais), garantindo que todas as ocorrências sejam recuperadas para o valor pesquisado.

4. Divisão de Trabalho

Tabela com a divisão de tarefas na equipe:

Arquivo/Função	Responsável	Descrição
`upload.cpp`	Todos	Construção e gravação do arquivo de dados organizado por hashing, indice primário e secundário organizados por B+ Tree
`BPlusTree.hpp`	Carolina	Implementação da estrutura de dados B+Tree
`hashing_file.cpp`	Fernando	Implementação da estrutura de dados Hash
`findrec.cpp`	Fernando	Busca direta no arquivo de dados (hashing) a partir do ID informado
`seek1.cpp`	Aline	Busca de registros pelo ID utilizando o índice primário (B+Tree)
`seek2.cpp`	Carolina	Busca de registros pelo Título utilizando o índice secundário (B+Tree)
Documentação	Todos	Redação, organização e formatação da documentação do projeto
Dockerização	Aline	Criação e testagem dos arquivos relacionados ao docker