Relatório do TP2

**Guilherme Louro de Salignac e Souza, Nicolas Mady Corrêa Gomes,
Victor Hugo Oliveira de Melo**

[1]Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio 6200, Coroado I, 69080-900 – Manaus – AM

1. Estrutura dos Arquivos de Dados e Índices

Nesta seção, explicaremos as decisões de projetos inerentes as estruturas de arquivos de dados e índices presentes neste trabalho (B+Tree e Hash).

1.1. Tabela Hash

A classe HashTable implementa o arquivo de dados presente neste trabalho, sendo responsável por toda a interação de baixo nível com o disco, incluindo a organização física dos registros, a serialização/deserialização de dados e a estratégia de acesso e resolução de colisões. A decisão de projeto foi implementar um Hashing Estático em Disco com Endereçamento Aberto (Sondagem Linear, ou Linear Probing), pois como não precisaremos fazer inserções ou remoções no arquivo, apenas o povoamento do arquivo de dados, um hash estático é mais coerente com o comportamento dos dados, visto que não precisaremos fazer muitos rehashings ou tratar colisões.

1.1.1. Estrutura e Construtor

```
HashTable::HashTable(const std::string &filename, int
 ↪  table_size)
    : filename(filename), table_size(table_size) {
  record_size = Artigo::getRecordSize();
  records_per_block = BLOCK_SIZE / record_size;
  if (records_per_block == 0)
    records_per_block = 1;
}

HashTable::~HashTable() {
  // Destrutor - não há recursos específicos para limpar
}
```

O arquivo (dados.hash) não é visto como um contêiner de registros individuais, mas como um grande array de blocos de tamanho fixo (BLOCK_SIZE). O construtor calcula quantos registros (Artigo) cabem em um único bloco

$$(records\_per\_block = BLOCK\_SIZE/record\_size)$$

A table_size (calculada no upload com base no fator de carga, como será mostrado na Seção 2.2) define o tamanho lógico do array de “slots”(espaços para registros). O arquivo é, portanto, um array de blocos, onde cada bloco contém records_per_block slots.

1.1.2. Formatação

Antes que qualquer registro possa ser inserido, o initialize pré-aloca todo o espaço necessário no disco. Para isso, o programa calcula o total_blocks necessário para armazenar o table_size (número de slots lógicos), usando a fórmula

$$(table\_size + records\_per\_block - 1)/record\_per\_block$$

Obs: Isto resulta em uma divisão inteira com arredondamento para cima.

Após isso, o programa escreve fisicamente total_blocks blocos vazios (preenchidos com zeros) no arquivo. Esta é a marca de um Hashing Estático: o arquivo é criado em seu tamanho máximo de uma só vez.

1.1.3. Inserção

Para a inserção, hash_value (slot lógico ideal) é calculado via Artigo::hash_id. O block_num (o bloco ideal) é então determinado por

$$hash\_value/records\_per\_block$$

O método findFreeSlot(block_num, slot) é chamado. Ele lê o bloco ideal e procura por um slot vazio (marcado por um ID == 0). Se findFreeSlot falhar (o bloco ideal está cheio), a sondagem linear é iniciada. O block_num é incrementado sequencialmente, lendo os blocos subsequentes até que um bloco com um slot livre seja encontrado.

$$((block\_num + 1)\%total_blocks)$$

Se a sondagem der a volta completa na tabela (block_num == original_block), a tabela é considerada ”cheia”e a inserção falha.

Uma vez que um block_num e slot vagos são encontrados (seja no bloco ideal ou em um bloco de sondagem), o registro Artigo é serializado (decisão de projeto: o código usa arquivos /tmp/ para a serialização). O registro serializado é escrito no buffer do bloco (char buffer[BLOCK_SIZE]) na posição correta (slot_position), e o bloco inteiro (buffer) é então escrito de volta ao disco na posição block_num usando writeBlock.

1.1.4. Busca

A busca em uma tabela hash, de certa forma, espelha a inserção. A busca começa calculando o block_num inicial (o bloco ideal) com base no hash do id. A partir daí, a operação entra em um loop do-while, que garante que pelo menos o bloco ideal seja lido. A primeira ação dentro deste loop é incrementar o contador stats.blocos_lidos++, assegurando que cada acesso ao disco seja contabilizado. Em seguida, readBlock(block_num, buffer) carrega o bloco do disco.

Uma vez carregado, um loop for interno realiza uma varredura linear em todos os slots daquele bloco, lendo o ID de cada slot. Se o stored_id corresponder ao id buscado,

o registro completo é deserializado para o objeto artigo, o status é marcado como encontrado = true, e a função retorna com sucesso. Caso o ID não seja encontrado no bloco atual, a sondagem linear continua, avançando para o próximo bloco

$$(block\_num = (block\_num + 1)\%stats.total\_blocos)$$

O loop de busca termina sob duas condições: ou o registro é encontrado, ou a sondagem dá a volta completa na tabela (block_num == original_block), indicando que o registro não existe.

1.2. B+Tree

A classe BPlusTree é o componente de indexação do projeto. Ela fornece um método de acesso aos dados alternativo e muito mais eficiente do que a varredura completa do arquivo de dados (ou mesmo do que o hashing em cenários de alta colisão). Esta classe é usada para implementar tanto o Índice Primário (mapeando ID -¿ offset) quanto o Índice Secundário (mapeando Título -¿ ID). A fim de maior reaproveitamento de código, a implementação foi feita em um formato template C++ (template <typename T>).

1.2.1. Template e Especialização

A classe BPlusTree é genérica (typename T). Ela não sabe se está armazenando inteiros ou strings. Isso significa que a mesma lógica de travessia, leitura/escrita de nós e alocação de blocos pode ser usada para ambos os índices. Para isso, no final do arquivo, funções de comparação são explicitamente especializadas:

Tabela 1. Implementação dos Métodos de Comparação dos Índices

Tipo de Índice	Método	Descrição / Implementação
PrimIdxEntry (Índice Primário)	`compare`	Compara dois IDs (subtração de inteiros).
	`matches`	Verifica igualdade exata (`entry.id == key.id`).
SecIdxEntry (Índice Secundário)	`compare`	Compara duas strings (`strcmp(a.titulo, b.titulo)`).
	`matches`	Verifica igualdade exata (`strcmp(...) == 0`).

Quando o upload instancia BPlusTree<PrimIdxEntry>, o compilador automaticamente usa as funções de comparação de ID. Quando instancia BPlusTree<SecIdxEntry>, ele usa as funções de strcmp.

Diferente do arquivo hash, o arquivo B+Tree (.btree) é uma coleção de nós (BTreeNode<T>), que é uma estrutura de tamanho fixo (node_size) que contém um array de chaves (keys[BTREE_ORDER]) e um array de ponteiros para filhos (children[BTREE_ORDER]). Esses "ponteiros" não são ponteiros de memória, mas sim offsets de byte (tipo long) que indicam a posição exata (em bytes) onde o nó filho está armazenado no arquivo. Já o initialize() cria o arquivo .btree e escreve o primeiro nó: um

nó raiz, vazio, marcado como folha (is_leaf = true). Este nó é escrito na posição 0 do arquivo.

Toda a interação com o disco é encapsulada em duas funções de baixo nível que garantem que todo o I/O seja feito em unidades de "nó", permitindo que o método de busca conte precisamente os acessos ao disco. A função bool readNode(long position, BTreeNode<T> &node) abre o arquivo de índice, usa file.seekg(position) para saltar diretamente para o offset de byte do nó, e lê exatamente node_size bytes, preenchendo a estrutura node. Inversamente, a função bool writeNode(long position, const BTreeNode<T> &node) usa file.seekp(position) para saltar para a posição correta e escrever o conteúdo da estrutura node de volta no disco.

1.2.2. Busca e Inserção na Árvore

A busca, implementada por searchInNode, é a operação principal para recuperação de dados. O método público search atua como um wrapper, iniciando a chamada recursiva a partir do nó raiz. A lógica de searchInNode é direta: a primeira ação ao entrar na função é incrementar o contador blocos_lidos++, tratando cada chamada recursiva (ou seja, cada nó visitado) como uma operação de I/O de disco. Em seguida, o nó é carregado do disco (readNode). A implementação realiza uma busca linear simples (for (int i...)) em vez de binária dentro das chaves do nó. Usando a função especializada matches(...), ela verifica se a chave foi encontrada. Se for, a busca termina com sucesso. Caso contrário, e se o nó não for uma folha (!node.is_leaf), a função desce recursivamente na árvore.

A inserção, implementada por insertIntoNode, segue uma lógica similarmente simplificada. O método lê o nó do disco e verifica se há espaço

$$(node.num\_keys < BTREE\_ORDER - 1)$$

2. Outros Códigos-Fonte

Nesta seção, apresentaremos os códigos-fonte realizados na Linguagem C++, com suas respectivas funções e respectiva importância nesta estrutura.

2.1. artigo.cpp

2.1.1. Construtor

Função padrão para o construtor da classe Artigo. O construtor pode ser inicializado tanto sem quanto com parâmetros.

2.1.2. Cópias

Estas linhas de código fazem cópia do título, dos autores e do snippet, limitando a um tamanho determinado pelo include "ìnclude.h".

2.1.3. Serialização, desserialização e impressão dos registros dos artigos

Estas linhas de código abaixo fazem jus as conversões necessárias para a serialização ou desserialização dos campos de dados do Artigo, tal qual para a impressão.

2.1.4. Função de Hash e de Tamanho do Registro

As linhas de código abaixo fazem caso ao cálculo da posição do artigo específico na tabela hash, determinada pelo resto da divisão entre ID e o tamanho da tabela; e também ao tamanho do registro do artigo em específico, dado pelo tamanho real dos atributos.

```
// Função hash para o ID
int Artigo::hash_id(int id, int table_size) { return id %
 ↪  table_size; }

// Retorna o tamanho fixo do registro
size_t Artigo::getRecordSize() {
  return sizeof(int) +                        // id
         sizeof(char) * (MAX_TITULO + 1) +    // titulo
         sizeof(int) +                        // ano
         sizeof(char) * (MAX_AUTORES + 1) +   // autores
         sizeof(int) +                        // citacoes
         sizeof(time_t) +                     // atualizacao
         sizeof(char) * (MAX_SNIPPET + 1);    // snippet
}
```

Por fim, há algumas funções utilitárias para auxiliar no parsing do arquivo .csv ao fazer o povoamento das estruturas de dados.

2.2. upload.cpp

2.2.1. Parsing da linha CSV

A função de parsing percorre cada caractere da linha, mantendo uma variável booleana de estado (inQuotes) que rastreia se o analisador está atualmente dentro de um campo entre aspas. Se o caractere atual for uma aspa dupla, o estado inQuotes é invertido. Isso significa que aspas são usadas apenas como delimitadores e não como parte do conteúdo do campo.

O ponto e vírgula é reconhecido como um separador de campo somente se o estado inQuotes for false. Todos os outros caracteres, incluindo o ponto e vírgula quando inQuotes é true, são acumulados na variável field. Quando um separador é encontrado (ou o laço termina), o conteúdo de field é submetido à função trim (para remover espaços em branco iniciais/finais) e adicionado ao vetor fields.

Após o loop principal, o código garante a adição do último campo da linha, pois não há um separador final após ele. O resultado final é um vetor que contém todos os valores da linha CSV, prontos para serem convertidos para seus respectivos tipos (inteiro, string, etc.) pelo programa principal.

```cpp
// Função para fazer parse de uma linha CSV
std::vector<std::string> parseCSVLine(const std::string
 ↪  &line) {
  std::vector<std::string> fields;
  std::string field;
  bool inQuotes = false;

  for (size_t i = 0; i < line.length(); i++) {
    char c = line[i];

    if (c == '"') {
      inQuotes = !inQuotes;
    } else if (c == ';' && !inQuotes) {
      fields.push_back(trim(field));
      field.clear();
    } else {
      field += c;
    }
  }

  // Adiciona último campo
  fields.push_back(trim(field));

  return fields;
}
```

2.2.2. Limpeza pós-parsing

A limpeza após o parsing das linhas é dada pela função removeQuotes, que verifica se a string tem pelo menos dois caracteres e se o primeiro e o último caracteres são aspas. Se todas as condições forem verdadeiras (o campo está delimitado por aspas), ela retorna a sub-string interna, efetivamente removendo a aspa inicial (posição 0) e a aspa final. Se as condições não forem atendidas (o campo não estava delimitado por aspas, como IDs ou números), a string original é retornada sem alterações.

```cpp
// Remove aspas duplas do início e fim de uma string
std::string removeQuotes(const std::string &str) {
  if (str.length() >= 2 && str[0] == '"' &&
   ↪  str[str.length() - 1] == '"') {
    return str.substr(1, str.length() - 2);
  }
  return str;
}
```

2.2.3. Povoamento usando o arquivo CSV

O processo inicia validando os argumentos de linha de comando para obter o nome do arquivo CSV (csv_filename). Em seguida, ele determina o diretório de saída dos dados, priorizando a variável de ambiente DATA_DIR e, caso não esteja definida, utilizando o caminho padrão ”data/db”. O diretório é então criado no sistema de arquivos, se ainda não existir. O programa lê o arquivo CSV uma primeira vez, linha por linha, apenas para contar o número total de registros (total_records). Com esse total, ele calcula o table_size (o número de slots lógicos na tabela hash) aplicando um fator de carga de aproximadamente 0.7, calculado como

$$(total\_records * 10)/7)$$

Esta decisão de design visa deixar cerca de 30% da tabela livre, a fim de evitar colisões para melhorar a inserção.

No loop de processamento principal, o arquivo CSV é lido pela segunda vez. Um cronômetro (std::chrono) é iniciado para medir o desempenho da ingestão. Cada linha é lida e processada dentro de um bloco try...catch — e para cada linha válida, a função parseCSVLine é chamada, seguida por removeQuotes e a conversão de tipo para cada campo. Ao final, um objeto Artigo é instanciado com esses dados. O registro é então inserido na HashTable, e o programa cria as entradas para os índices: uma entrada de índice primário (PrimIdxEntry), que mapeia o ID ao seu offset no arquivo hash, e uma entrada de índice secundário (SecIdxEntry), que mapeia o titulo ao ID do artigo.

2.3. findrec.cpp

Primeiramente, o programa valida os argumentos da linha de comando. É esperado exatamente um argumento (argc != 2): o ID do registro a ser buscado. O argumento fornecido é então convertido de string para inteiro (std::stoi). Esse bloco é envolto em um try...catch para garantir que, se o usuário fornecer uma entrada inválida (ex: ”abc”), o programa termine de forma controlada com uma mensagem de erro, em vez de travar. Após isso, o programa utiliza std::getenv(”DATA_DIR”) para determinar o local dos arquivos de banco de dados (se não houver nada definido, ele adota o diretório padrão “data/db”.

A lógica de hashing (calcular o hash do ID e a sondagem linear) depende diretamente do table_size usado durante a criação do arquivo (no programa upload). Nesta implementação, o table_size é definido com um valor estático (100.000). Esta é uma simplificação para um programa de teste autocontido. Em um sistema de banco de dados robusto, esta informação (metadados) seria lida diretamente do cabeçalho do arquivo dados.hash para garantir que findrec e upload estejam perfeitamente sincronizados, mesmo que o fator de carga ou o número de registros mudem.

O núcleo crucial da função ocorre nestas quatro linhas:

```
1 auto start_time =
   ↪  std::chrono::high_resolution_clock::now();
2 Artigo artigo;
3 SearchStats stats = hash_table.search(id, artigo);
4 auto end_time = std::chrono::high_resolution_clock::now();
```

O cronômetro (std::chrono) é iniciado imediatamente antes da operação de busca e, assim, o objeto Artigo vazio é criado na stack e passado por referência (Artigo &artigo) para o método search — cuja lógica não está contida na I/O da função, mas sim no método da hash_table. O método search executa a lógica completa de hashing: calcula o hash do ID para encontrar o bloco inicial, lê este bloco (readBlock), e (se o registro não for encontrado) inicia a sondagem linear, lendo os blocos subsequentes e incrementando o contador blocos_lidos a cada operação de readBlock.

$$((block\_num + 1)\%total\_blocos)$$

Se o ID é encontrado, ele deserializa os dados do bloco para o objeto Artigo e retorna. Se stats.encontrado for verdadeiro, os detalhes do registro são exibidos (chamando artigo.print()). Caso contrário, uma mensagem de "NÃO ENCONTRADO" é mostrada. Ao final, as estatísticas de acesso são impressas, informando exatamente quantos blocos de disco foram acessados (blocos_lidos) para satisfazer a consulta, comparado ao tamanho total do arquivo (total_blocos). O programa finaliza retornando 0 se o registro foi encontrado (sucesso) ou 1 se não foi (falha).

2.4. seek1.cpp

Diferente do findrec (que calcula a posição do registro), o seek1 utiliza uma estrutura de índice B+Tree (indice_primario.btree) para descobrir a localização do registro no arquivo de dados (dados.hash) antes de buscá-lo. O início do seek1 é similar ao findrec — verificações iniciais, definição do diretório e iniciação da cronometragem, então vamos nos aprofundar a partir do início na busca nos índices primários da B+ Tree.

2.4.1. Busca no Índice Primário

A primeira ação de I/O é a busca na B+Tree primária. A lógica é delegada ao método indice_primario.search(). Este método atravessa a árvore (lendo blocos de índice do disco) até encontrar o ID em um nó folha. O número de blocos lidos nesta travessia é capturado em index_stats.blocos_lidos. O programa então exibe imediatamente as estatísticas de I/O apenas do índice, o que isola o custo de "apontar" para o registro. Se o ID não for encontrado no índice (!index_stats.encontrado), o programa termina prematuramente, economizando o acesso desnecessário ao arquivo de dados.

Após o ID ser encontrado no índice, a B+Tree retorna a result_entry, que contém a posicao_hash. Em vez de criar um novo método na HashTable para ler um registro diretamente de um offset específico (ex: readRecordAt(result_entry.posicao_hash)), o programa reutiliza o método hash_table.search(id, artigo), decisão de projeto tomada visando simplificar a busca.

```
int table_size = 100000; // Valor padrão
HashTable hash_table(hash_file, table_size);

Artigo artigo;
SearchStats data_stats = hash_table.search(id, artigo);
```

O cronômetro (auto end_time) é parado após esta segunda busca, capturando o tempo total das duas operações. Ao final, o custo total de I/O do seek1 é a soma dos

blocos lidos na B+Tree mais os blocos lidos na tabela hash, além de retornar 0 ou 1, como no findrec.

2.5. seek2.cpp

Na Seção 2.4, a busca foi feita pelo índice primário, utilizada quando sabemos a chave primária dos dados (no caso, o ID). Agora, o seek2 fará a busca a partir do índice secundário, utilizado quando não sabemos a chave primária, e a busca é feita por um campo secundário (neste exemplo, o título do artigo). O custo de I/O foi medido em duas fases: na Fase 1, acessamos a B+Tree secundária (indice_secundario.btree) para encontrar o ID correspondente ao titulo fornecido. Após isto, utilizamos o ID obtido na Fase 1 para buscar o registro completo no arquivo de dados (dados.hash). A instanciação da validação, da configuração e do início da cronometragem é similar ao das outras funções.

2.5.1. Busca no Índice Secundário

```
// Inicializa índice secundário
SimpleIndiceSecundario
 ↪  indice_secundario(secondary_index_file);

// Mede tempo de busca no índice
auto start_time =
 ↪  std::chrono::high_resolution_clock::now();

// Busca no índice secundário
SecIdxEntry key_entry(titulo, 0);
SecIdxEntry result_entry;
SearchStats index_stats =
 ↪  indice_secundario.search(key_entry, result_entry);
```

A lógica é delegada ao método indice_secundario.search(). Este método atravessa a B+Tree, comparando strings (o titulo) em cada nó, até encontrar a entrada correspondente em um nó folha. O custo (em blocos lidos) desta travessia é capturado em index_stats.blocos_lidos. Assim como no seek1, o programa exibe as estatísticas de I/O do índice e termina prematuramente se o titulo não for encontrado, evitando a busca desnecessária no arquivo de dados.

2.5.2. Busca no Arquivo de Dados

Se a Fase 1 for bem-sucedida, a result_entry agora contém o ID do artigo (result_entry.id). Este ID é então usado para a segunda fase da busca:

```
// Agora busca o registro completo no arquivo de dados
 ↪  usando o ID encontrado
int table_size = 100000; // Valor padrão
HashTable hash_table(hash_file, table_size);

Artigo artigo;
```

```
6 SearchStats data_stats = hash_table.search(result_entry.id,
   ↪  artigo);
```

O programa adota a mesma decisão de projeto do seek1: ele reutiliza o método hash_table.search(). Em vez de buscar por um offset (que o índice secundário nem armazena), ele executa uma busca completa por hashing usando o ID que acabou de descobrir. Isso significa que o seek2 paga o custo de I/O de uma travessia de B+Tree (comparando strings) somado ao custo de I/O de uma busca por hashing (calculando hash e fazendo sondagem linear).

Após a Fase 2, o cronômetro é parado. O programa então exibe o registro completo (com artigo.print()), mas realiza uma verificação de consistência:

```
// Verifica se o título realmente corresponde
    std::string titulo_encontrado = artigo.titulo;
    if (titulo_encontrado != titulo) {
      std::cout << std::endl
                << "AVISO: Título no registro difere do
                 ↪  buscado!" << std::endl;
      std::cout << "Buscado: \"" << titulo << "\"" <<
       ↪  std::endl;
      std::cout << "Encontrado: \"" << titulo_encontrado <<
       ↪  "\"" << std::endl;
    }
  } else {
    std::cout << "REGISTRO NÃO ENCONTRADO NO ARQUIVO DE
     ↪  DADOS" << std::endl;
    std::cout << "INCONSISTÊNCIA: ID " << result_entry.id
              << " existe no índice mas não no arquivo de
               ↪  dados!" << std::endl;
  }
```

O programa compara o titulo originalmente buscado com o titulo_encontrado no registro. Um mismatch (improvável se a B+Tree estiver correta) é reportado como um AVISO. Já a INCONSISTÊNCIA é um cenário de erro grave: o índice secundário afirmou que o ID existia, mas o hash_table.search falhou em encontrá-lo, sugerindo que os arquivos de índice e de dados estão dessincronizados. Ao final, a função quantifica o custo total da busca indireta e retorna 0 ou 1.