Organização e Indexação de Arquivos em Memória Secundária
Trabalho Prático II – Documentação Detalhada

Abel Severo Rocha, Ana Carla de Araújo Fernandes, Natasha Araújo Caxias

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

{abel.severo}{ana.carla}{natasha.caxias}@icomp.ufam.edu.br

1. Introdução

Este trabalho implementa uma solução de armazenamento persistente para registros de artigos científicos em arquivos binários, com dois índices B+ (primário e secundário). Os objetivos principais são:

- garantir inserção eficiente em disco a partir de um CSV de entrada;
- implementar busca direta no arquivo de dados e buscas indexadas por B+;
- mensurar blocos lidos e tempos de operação para avaliação de desempenho;
- seguir critérios de portabilidade via Docker e instruções de build via Makefile.

A implementação foi feita em C++ e testada em ambiente Linux/Windows (a medição do bloco foi feita em Windows).

2. Especificação do Registro

Cada registro representa um artigo científico e foi desenhado para ter tamanho fixo — isto simplifica cálculo de offsets e acesso direto por posição.

2.1. Campos e tamanhos

A estrutura em C++ (resumida) é:

```
struct Registro {
   int id; // 4 bytes identificador
   array<char,300> titulo; // 300 bytes ttulo do artigo
   char ano[8]; // 8 bytes ano de publicao
   char autores[150]; // 150 bytes lista de autores
   char citacoes[16]; // 16 bytes n de citaes
   char data_atualizacao[32]; // 32 bytes data/hora da ltima atualizao
   char snippet[1024]; // 1024 bytes resumo do artigo
   ptr prox; // 8 bytes ponteiro p/ encadeamento no hash
};
```

Tamanho total do registro Somando os campos (valores usados na implementação):

$$T_{reg} = 4 + 300 + 8 + 150 + 16 + 32 + 1024 + 8 = 1542\ bytes$$

Para alinhamento em sistemas distintos, o sizeof(Registro) pode ser arredondado pelo compilador; na implementação gravamos em binário usando

`write(reinterpret_cast<char>(registro), sizeof(Registro))`
e confirmamos que cada registro ocupa o valor retornado por `sizeof(Registro)`
Para a documentação, usaremos o valor real obtido com `sizeof` (que aqui é 1542 bytes conforme soma direta — a implementação zera o padding manualmente para reprodutibilidade).

3. Organização do Arquivo de Dados (HashFile)

3.1. Parâmetros globais

- **NUM_BUCKETS** = 100003 (primo grande, reduz colisões)
- **BUCKET_SIZE** = 10 (nº de registros na área primária por bucket)

3.2. Layout lógico do arquivo

O arquivo é dividido logicamente em *buckets* consecutivos. Cada bucket ocupa exatamente:

$$bytes\_por\_bucket = BUCKET\_SIZE \times T_{reg}$$

Offset (em bytes) do início do bucket b (0-indexado):

$$offset\_bucket(b) = b \times bytes\_por\_bucket$$

Dentro do bucket, a i-ésima posição ($0 \leq i <$ BUCKET_SIZE) tem offset:

$$pos\_registro(b, i) = offset\_bucket(b) + i \times T_{reg}$$

Se o bucket primário está cheio, novos registros daquele bucket são escritos no fim do arquivo (área de overflow). A lista de overflow é encadeada através do campo `prox` (offset absoluto no arquivo em bytes).

3.3. Função de hash

Usamos função simples:

$$h(key) = key \bmod NUM\_BUCKETS$$

Nota: O uso de um número primo para `NUM_BUCKETS` ajuda a dispersar melhor a chave inteira.

3.4. Tamanho estimado do arquivo

Supondo N registros efetivos, número de registros na área primária é no máximo:

$$N_{prim} = NUM\_BUCKETS \times BUCKET\_SIZE$$

Se $N \leq N_{prim}$, não haverá overflow. Caso contrário, haverá $N - N_{prim}$ registros em overflow.

Tamanho do arquivo se inicializamos o arquivo vazio com todos os buckets pré-alocados:

$$T_{arquivo} = N_{prim} \times T_{reg}$$

Com nossos parâmetros:

$$N_{prim} = 100003 \times 10 = 1\,000\,030 registros$$

Estimativa:

$$T_{arquivo} \approx 1\,000\,030 \times 1538 \approx 1.538 \times 10^9\ bytes \approx 1.43\ GB$$

Observação: na prática o arquivo é inicializado gravando registros vazios (id=0, prox=-1) para todas as posições primárias.

4. Algoritmo de Inserção (Hash + Overflow)

Em média, cada inserção em bucket com espaço causa 1 operação de escrita do registro no bucket (escrita posicional): custo amortizado $O(1)$ em acesso randômico, medido em blocos lidos/gravações. Quando há overflow, há uma escrita no final do arquivo e uma leitura/escrita para atualizar o ponteiro do último nó da cadeia. Em lote, atualizamos menos vezes ao agrupar por bucket. Ao agrupar inserções por bucket reduzimos o número de seeks (custo de movimentação do cabeçote lógico) porque acessamos sequencialmente as posições do mesmo bucket.

5. Índice: B+Tree (implementação em disco)

A implementação das operações na estrutura adota uma abordagem iterativa. Cada nó armazena suas chaves e ponteiros em vetores. Para localizar a posição correta de uma chave dentro de um nó, utiliza-se o algoritmo de busca binária.

A utilização de vetores nos nós otimiza significativamente a performance e a eficiência de memória. A estrutura de dados contígua permite a aplicação de busca binária ($O(\log n)$) para localizar chaves, superando a busca linear $O(n)$. Adicionalmente, esta abordagem explora a localidade espacial, o que resulta em melhor aproveitamento da cache da CPU e reduz o overhead de alocações dinâmicas de memória.

Implementamos duas B+Trees armazenadas em arquivos binários diferentes:

- `/data/bptreeId.idx` — índice primário: chaves `int` (ID), ordem $M_{ID} = 341$.
- `/data/bptreeTitulo.idx` — índice secundário: chaves fixas `char[300]`, ordem $M_{TITULO} = 14$.

5.1. Motivação e organização

A B+Tree armazena chaves e ponteiros para os registros (offsets em `data.db`). Todas as chaves estão nas folhas; nós internos armazenam separadores (chaves guia) e ponteiros para filhos.

Escolhas de projeto:

- **Ordem dinâmica:** usamos o utilitário `calcularM` para estimar a ordem M a partir do tamanho de bloco detectado (4 KB) e dos tamanhos de chave e ponteiro;
- **Nós com tamanho de bloco:** cada nó deve caber aproximadamente em um bloco do sistema de arquivos (4096 bytes), reduzindo leituras de múltiplos blocos para um mesmo nó;
- **Armazenamento em disco:** cada nó tem identificador equivalente ao seu offset no arquivo do índice. Um cache em memória foi implementado (flush e salvarMetadados) para reduzir I/O.

5.2. Layout de nó (modelo)

A estrutura de dados empregada utiliza uma representação unificada tanto para nós folha quanto para nós internos. A distinção entre esses dois tipos de nós é realizada através de um campo booleano denominado `folha`, onde o valor 1 (*verdadeiro*) designa um nó folha e o valor 0 (*falso*) indica um nó interno.

Nó interno

- bool folha (1 byte)
- int qtdKeys (4 bytes)
- array ponteiros filhos (M+1) de 8 bytes cada (offsets)
- array keys (M) (tipo depende da chave: int ou char[300])

Nó folha

- bool folha (1 byte)
- int qtdKeys (4 bytes)
- array keys (até M)
- array ponteiros para registros (offsets 8 bytes cada)
- ponteiro `nextLeaf` (8 bytes) para iteração em sequência

A implementação reserva espaço adicional para operações temporárias (insersão), conforme mostrado no utilitário `calcularM` (ex.: overhead = sizeof(int) + sizeof(bool) + tamTipo + 2 * tamPoint).

5.3. Cálculo da ordem M – demonstração

O utilitário `calcularM` usa a equação aproximada:

$$M = \left\lfloor \frac{B + tamTipo - overhead}{tamTipo + tamPoint} \right\rfloor$$

onde:

- B = tamanho do bloco do filesystem (4096 bytes);
- $tamTipo$ = tamanho da chave (int: 4 bytes; título: 300 bytes);
- $tamPoint$ = tamanho de um ponteiro/offset no arquivo (int64_t: 8 bytes);
- $overhead$ = bytes reservados para metadados no nó (ex.: qtdKeys, flag folha, buffers temporários).

Substituindo para ID:

$$M_{ID} \approx \left\lfloor \frac{4096 + 4 - o}{4 + 8} \right\rfloor$$

Com o overhead observado no código (variável o calculada com sizeof(int)+sizeof(bool)+tamTipo+2*tamPoint), o utilitário retornou empiricamente:

$$M_{ID} = 341$$

Para Título:

$$M_{TITULO} \approx \left\lfloor \frac{4096 + 300 - o}{300 + 8} \right\rfloor = 14$$

Estes valores refletem a capacidade de um nó de caber num bloco com o layout escolhido.

5.4. Operações principais

Busca

1. Ler nó raiz (1 bloco lido).
2. Percorrer comparando chaves guia nos nós internos (1 bloco por nível).
3. Ler folha e localizar chave; se encontrada, obter offset e acessar `data.db`.

A função `bptree.buscar(key, registro, &db)` retorna `pair<bool,long>` onde `first` indica sucesso e `second` contém blocos lidos no índice.

Inserção

1. Percorrer até a folha destino;
2. Inserir chave/ponteiro; em caso de overflow (>= M), dividir a folha;
3. Propagar split para cima, criando nós internos quando necessário.

6. Implementação dos Programas

6.1. upload (./bin/upload /data/input.csv)

Fluxo:

1. Validação do argumento (caminho do CSV).
2. Leitura do CSV linha a linha via `parseCSV` (remove BOM, trata quebras).
3. Conversão de linha para `Registro` fixo (função `toRegistro`).
4. Inserção em buffer até `BATCH_SIZE` e chamada para `HashFile::inserirEmLote`.
5. Para cada inserção bem-sucedida obtém-se posição (offset) e atualiza-se:
   - `bptreeId.inserir(id, pos);`
   - `bptreeTitulo.inserir(titulo, pos);`
6. Ao final flush dos caches e fechamento dos arquivos.

Saídas impressas no terminal:

- caminhos dos arquivos usados;
- progresso (registros processados, tempo e velocidade);
- estatísticas finais: registros válidos/inválidos, tempo total, blocos no arquivo, local dos índices.

6.2. findrec (./src/findrec <ID>)

Busca direta no arquivo hash:

1. calcular bucket = h(ID);
2. ler bucket primário sequencialmente (até BUCKET_SIZE) procurando ID;
3. se não encontrado, seguir ponteiros de overflow encadeados (campo `prox`);
4. contar blocos lidos e imprimir registro (se encontrado) e estatísticas.

6.3. seek1 (./src/seek1 <ID>)

Busca via índice primário:

1. abrir `bptreeId.idx` e `data.db`;
2. carregar estrutura B+ (metadados) e chamar `bptree.buscar(id, registro, &db)`;
3. impressão detalhada do registro, blocos lidos no índice e tempo.

6.4. seek2 (./src/seek2 «Título>")

Busca via índice secundário (título):

1. converter título de entrada para array char[300] (zfill/truncate);
2. carregar `bptreeTitulo.idx` e realizar busca;
3. caso encontrado, ler registro em `data.db` e imprimir.

7. Logging e Medição

Cada programa tem saídas obrigatórias:

- **Arquivos usados:** caminho do banco e do índice.
- **Blocos lidos:** contagem de blocos no índice (retornada pela busca B+) e bloco/-leitura no hash (por follow chain).
- **Tempo de execução:** medido internamente com `chrono::high_resolution_clock`, em ms.
- **Nível de log:** controlado por ENV `LOG_LEVEL` (error, warn, info, debug). Mensagens de debug mostram offsets e passos internos.

8. Estrutura do Repositório

```
/app
  /src
  /include
  /bin
  /data # vazio no repo; montado em runtime
  /utils # scripts Python e calcularM
  Dockerfile
  docker-compose.yml
  Makefile
  TP2_documentao.pdf
  README.md
```

9. Testes e Validação

9.1. Testes de performance

- Medir tempo e blocos lidos para busca direta (findrec) vs indexada (seek1) em 100 amostras aleatórias.
- Medir throughput de inserção em lote com diferentes `BATCH_SIZE`.
- Forçar overflow: gerar $N > NUM\_BUCKETS \times BUCKET\_SIZE$ e medir crescimento do arquivo e custo médio de inserção.

9.2. Casos de borda

- IDs repetidos: a estratégia atual não admite duplicatas (ID é chave única). Se ID já existente, o comportamento deve ser definido (ex: ignorar/atualizar).
- Campos ausentes/no formato errado: `parseCSV` aplica valores default (ex.: "Sem Titulo").

10. Decisões de Projeto e Justificativa

- **Hash estático com overflow encadeado:** simples e determinístico; facilita acesso direto e implementação em C++ sem estruturas complexas.
- **B+Tree em disco:** excelente para buscas por intervalo e permitindo operações em tempo logarítmico por número de chaves.
- **Eficiência de memória e comparações:**A utilização de vetores promove a localidade espacial, armazenando dados de forma contígua. Esta organização otimiza o aproveitamento da cache da CPU.
- **Ordem M calculada por bloco:** manter nós com tamanho de bloco reduz número de I/Os por operação de nó.
- **Registros em formato fixo:** simplifica offsets e evita parsing caro durante seeks.
- **Agrupamento por bucket (inserirEmLote):** reduz seeks e melhora throughput de inserção.

11. Conclusão

A solução implementada combina uma organização por hashing (para inserção e busca direta) com índices B+ (para busca eficiente por chave primária e secundária). A escolha de parâmetros (NUM_BUCKETS = 100003, BUCKET_SIZE = 10, M_ID = 341, M_TITULO = 14) é justificada por considerações de espaço de bloco e tamanho de chaves. O sistema entrega:

- um pipeline completo de ingestão (upload);
- buscas diretas e indexadas com medição de blocos e tempos;
- ferramentas auxiliares (corrigir CSV, calcular M, Makefile e Docker) que garantem reprodutibilidade.

12. Divisão de Trabalho

- **Abel Severo Rocha** – (TP2_documentação.tex).
- **Ana Carla de Araújo Fernandes** – Implementação das B+Trees (bptreefile.*).
- **Natasha Araújo Caxias** – Limpeza e normalização do CSV, Implementação do HashFile (hashfile.*).

13. Referências

- Visualização e conceitos de B+Tree: Galles, University of San Francisco. https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- Apostila/Slides sobre organização de arquivos: ICMC/USP. http://wiki.icmc.usp.br/images/8/8e/SCC578920131-B.pdf
- Knuth, D. E., *The Art of Computer Programming* — vol. 3 (searching and hashing) — para princípios teóricos.