

# Documentação do TP2 de Banco de Dados

Vinicius L. N. Fonseca - 22050031,  
Matheus Silva dos Santos - 22052573

<sup>1</sup>Instituto de Computação – Universidade Federal do Amazonas (UFAM)  
Caixa Postal 69077-470 – Manaus – AM – Brazil

{vinicius.fonseca, matheus.silva@icomp.ufam.edu.br}}

## 1. Estrutura do Registro

Decidimos implementar o registro utilizando campos de tamanho fixo e variável, com alocação não espalhada. Cada campo de tamanho variável (string) será delimitado pelo caractere especial “\0” para indicar o seu final. A seguir, apresentamos uma representação e o cálculo do tamanho de um registro:

nome	id	title	year	authors	citations	update	snippet
tipo	int	string	int	string	int	string	string
tamanho (bytes)	4	w+1	4	x+1	4	y+1	z+1

$TamR = (4 * 3) + (w + x + y + z + 4)$ , onde  $w$ ,  $x$ ,  $y$  e  $z$  é o tamanho de cada string.

Dessa forma, supondo que um registro possa ter todos os seus campos variáveis vazios, o tamanho mínimo para um registro seria de 16 bytes. No entanto, optamos por armazenar a string que representa o valor “NULL”, resultando em um tamanho mínimo de 36 bytes por registro.

## 2. Estrutura do arquivo de dados e de índices

### 2.1. Arquivo de dados organizado por hash

Durante a criação do arquivo de dados, foi escolhida uma função hash simples para determinar em qual bucket um registro seria inserido. Essa função hash é calculada da seguinte forma:

$f(id) = (37 * id) \% nb$ , onde  $nb$  representa o número de buckets.

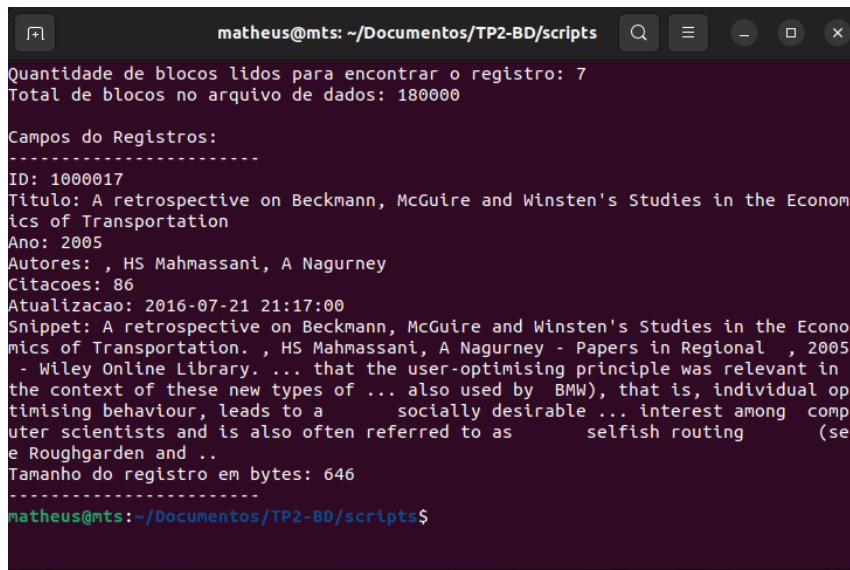
Após uma análise, foi estabelecido o uso de 15.000 buckets, cada um com 12 blocos. Cada bloco possui um tamanho de 4096 bytes. Essa decisão foi tomada com o objetivo de simplificar o código, evitando a necessidade de criar um arquivo de overflow. O cálculo do espaço total que será utilizado pelo arquivo de dados é realizado da seguinte forma:

$$TamAD = 15.000 \times 12 \times 4096 = 737.280MB$$

Também foi definido que cada bloco possuiria um cabeçalho, contendo as seguintes informações:

nome	qtd_registros	tam_disponivel	pos_registros[18]
tipo	int	int	int[18]
tamanho (bytes)	4	4	72
total (bytes)	80		

Isso resulta em 4016 bytes úteis por bloco. O vetor “pos\_registros” corresponde a um vetor com 18 posições, contendo o offset de cada registro no bloco. Estimamos que, devido à variação de tamanho dos registros, seria possível alocar aproximadamente 18 registros em cada bloco. Na figura 1 é mostrado um exemplo de saída do programa `./findrec <ID>`, no qual busca diretamente no arquivo de dados por um registro com o ID informado e, se existir, retorna os campos do registro, a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos do arquivo de dados.



```
matheus@mts: ~/Documentos/TP2-BD/scripts
Quantidade de blocos lidos para encontrar o registro: 7
Total de blocos no arquivo de dados: 180000

Campos do Registros:
-----
ID: 1000017
Titulo: A retrospective on Beckmann, McGuire and Winsten's Studies in the Economics of Transportation
Ano: 2005
Autores: , HS Mahmassani, A Nagurney
Citacoes: 86
Atualizacao: 2016-07-21 21:17:00
Snippet: A retrospective on Beckmann, McGuire and Winsten's Studies in the Economics of Transportation. , HS Mahmassani, A Nagurney - Papers in Regional , 2005 - Wiley Online Library. ... that the user-optimising principle was relevant in the context of these new types of ... also used by BMW), that is, individual optimising behaviour, leads to a socially desirable ... interest among computer scientists and is also often referred to as selfish routing (see Roughgarden and ..
Tamanho do registro em bytes: 646
-----
matheus@mts:~/Documentos/TP2-BD/scripts$
```

**Figura 1. Exemplo de saída do programa `./findrec <ID>` com o registro de ID 1000017.**

## 2.2. Arquivos de índice primário e secundário

Ambos os arquivos foram implementados utilizando árvores B+. Cada nó da árvore possui 510 chaves (IDs) e 2 endereços de bloco de 8 bytes. Para o índice primário utilizamos dois inteiros: o ID de um registro e seu endereço em um bloco, resultando em 8 bytes. Para o índice secundário, que utiliza o título, optamos por remover todos os caracteres Unicode e convertê-lo em valores inteiros, a fim de simplificar a implementação, ficando com a mesma forma que o índice primário. Segue um cálculo para justificar a seleção desses valores:

$$b = 510 \times 8 + 8 \times 2 = 4096 \text{ bytes}, \text{ que resulta em um tamanho de um bloco.}$$

Após a conclusão da implementação dos índices primário e secundário, constatou-se que as árvores B+ correspondentes alcançaram uma altura de 3. No caso do índice primário, a árvore totalizou 4021 blocos, enquanto o índice secundário apresentou uma quantidade menor, com 2930 blocos. Essa disparidade surgiu devido à leitura dos registros do arquivo “artigo.csv”, onde foram identificados registros sem títulos. Para resolver essa questão, optou-se por atribuir a esses registros o título “NULL” como uma string. Uma vez que a árvore B+ implementada não permite chaves duplicadas, somente o primeiro registro sem título foi adicionado à árvore após a conversão da string “NULL” para um valor inteiro. Os demais registros sem título foram ignorados.

Portanto, ao final da implementação do índice secundário, a árvore B+ conterá menos chaves (representando os títulos convertidos em inteiros), resultando em um número menor de nós. As figuras 2 e 3 exemplificam a saída dos programas que utilizam os arquivos de índice primário e secundário. O arquivo de índice primário é empregado pela função `./seek1 <ID>`, onde o ID de um registro é passado como parâmetro, enquanto o índice secundário é utilizado pela função `./seek2 <Titulo>`, que recebe um título como parâmetro.

```
matheus@mts: ~/Documentos/TP2-BD/scripts
Quantidade de blocos lidos para encontrar o registro no arquivo de índice: 4
Quantidade total de blocos do arquivo de índice primário: 4021

Campos do Registros:
-----
ID: 178911
Titulo: Real-Time Lane Estimation Using Deep Features and Extra Trees Regression
Ano: 2015
Autores: Vijay John|Zheng Liu|Chunzhao Guo|Seiichi Mita|Kiyosumi Kidono
Citacoes: 0
Atualizacao: 2016-10-19 21:56:30
Snippet: Real-Time Lane Estimation Using Deep Features and Extra Trees Regression. V John, Z Liu, C Guo, S Mita, K Kidono - Pacific-Rim Symposium on Image , 2015 - Springer. Abstract In this paper, we present a robust real-time lane estimation algorithm by adopting a learning framework using the convolutional neural network and extra trees. By utilising the learning framework, the proposed algorithm predicts the ego-lane location in the given ..
Tamanho do registro em bytes: 613
-----
matheus@mts:~/Documentos/TP2-BD/scripts$
```

Figura 2. Exemplo de saída do programa `./seek1 <ID>` com o registro de ID 178911.

```
matheus@mts: ~/Documentos/TP2-BD/scripts
Quantidade de blocos lidos para encontrar o registro no arquivo de índice: 4
Quantidade total de blocos do arquivo de índice primário: 2930

Campos do Registros:
-----
ID: 273192
Titulo: Adaptive control system for electrohydraulic camless engine gas valve actuator
Ano: 2005
Autores: TC Hanks
Citacoes: 4
Atualizacao: 2016-11-08 11:55:53
Snippet: Adaptive control system for electrohydraulic camless engine gas valve actuator. TC Hanks, JH Lumkes - Proceedings of the 2005, American , 2005 - ieeexplore.ieee.org. Abstract If realized, the camless internal combustion engine offers significant advantages over the engines on the market today in the areas of efficiency, fuel economy, and emissions reduction [1-5]. In previous work completed, a mathematical model of an electrohydraulic ..
Tamanho do registro em bytes: 574
-----
matheus@mts:~/Documentos/TP2-BD/scripts$
```

Figura 3. Exemplo de saída do programa `./seek2 <Titulo>` passando o título do registro de ID 273192.

Após realizar testes nos programas principais implementados, observamos que o uso dos índices primário e secundário oferece vantagens na busca direta de registros no arquivo de dados quando um registro específico está armazenado além do quarto bloco alocado em um bucket. Isso ocorre porque as árvores B+ dos índices possuem uma altura de 3, o que implica que o máximo de blocos a serem lidos para obter o endereço do bloco

de um registro é de 3 (altura da árvore) + 1 (bloco onde o registro está), totalizando 4 blocos.

Por exemplo, ao buscar o registro com ID 1000017 no arquivo de dados, foram necessários ler 7 blocos para localizar o registro correspondente. No entanto, ao utilizar os índices, a busca desse mesmo registro exigirá apenas a leitura de 4 blocos. É importante ressaltar que, se um registro estiver no primeiro bloco de um bucket, a busca direta no arquivo de dados será mais rápida e menos dispendiosa. No entanto, não é possível determinar todos os registros contidos nos 3 primeiros blocos, e o número de blocos lidos ao utilizar a busca direta no arquivo de dados pode variar consideravelmente. Por outro lado, ao utilizar os índices, a leitura será sempre fixa, com um valor de 4 blocos, até encontrar o registro desejado.

### 3. Fontes dos programas

Todos os programas criados compartilham duas fontes principais: fontes primárias e fontes secundárias. As fontes primárias são os arquivos ou recursos que contêm o código-fonte principal do programa. As fontes secundárias são os arquivos ou recursos que são referenciados ou importados pelas fontes primárias. Para mais informações sobre cada fonte abaixo, recomenda-se visualizar o seu respectivo código fonte, no qual possui comentários e descrições mais detalhadas sobre cada parte das fontes.

#### 3.1. Fontes secundárias

- **hash.hpp:**
  - registro.hpp;
  - constantes.hpp;
  - bloco.hpp;
  - bucket.hpp.
- **bplustree.hpp:**
  - registro.hpp;
  - constantes.hpp.

#### 3.2. Fontes primárias

- **upload.cpp:**
  - hash.hpp;
  - bplustree.hpp.
- **findrec.cpp:**
  - hash.hpp
- **seek1.cpp:**
  - hash.hpp
  - bplustree.hpp
- **seek2.cpp:**
  - hash.hpp
  - bplustree.hpp

## 4. Descrição das fontes

### 4.1. registro.hpp

- **Registro\* criarRegistro(int id, string title, int year, string authors, int citations, string update, string snippet):** Função construtora de um registro.
- **void imprimeRegistro(Registro registro):** Dado um registro imprime seus campos no terminal.
- **string remove\_unicode(string str):** Dado uma string retorna a mesma sem os caracteres unicode, os substituindo por .
- **Registro\* lineToRegister(string entry\_line):** Dado uma string, nesse caso uma linha do arquivo .csv, extrai os campos do registro e o retorna.

### 4.2. hash.hpp

- **int hashFunction(int id):** Função de distribuição hash, dado um id retorna o índice de um bucket.
- **void escreveHashTable(ofstream& dataFile):** Escreve a tabela hash vazia em memória secundária, dado um arquivo de entrada;
- **void inserir\_registro\_bucket(Registro \*registro, ifstream &entrada, ofstream &saida):** Insere um registro em um bucket, calcula utilizando a hashFunction o índice do bucket e procura um bloco com espaço disponível dentro do bucket para realizar a inserção, quando encontra chama: `inserir_registro_bloco`. Recebe como entrada: registro a ser inserido, arquivo de leitura e arquivo de escrita. Caso não haja espaço retorna uma mensagem de erro.
- **void inserir\_registro\_bloco(ifstream& leitura, ofstream& escrita, Bloco\* bloco, Registro\* registro, int ultimo\_bloco, int index\_bucket):** Escreve o registro no bloco e atualiza o cabeçalho do bloco. Recebe: arquivo de leitura e escrita, o bloco no qual vai ser inserido o registro, o registro e o índice do ultimo bloco, que representa o índice de um bloco com espaço livre para o registro.
- **Registro\* buscar\_registro(ifstream& leitura, int id\_busca):** Realiza a busca de um registro no arquivo de dados. Percorre os blocos de um bucket, lendo o cabeçalho e os dados de cada bloco. A função verifica se há registros no bloco e, em caso afirmativo, itera sobre cada registro. Para cada registro, é verificado se o ID corresponde ao ID buscado. Se houver correspondência, as informações do registro são extraídas dos dados do bloco e retornadas como um objeto **Registro**. A função exibe informações relacionadas à busca, como a quantidade de blocos lidos e o total de blocos no arquivo. Caso o registro não seja encontrado, a função retorna **nullptr**.

### 4.3. bucket.hpp

- **void destruirBucket(Bucket\* bucket):** Função para desalocar a memória de um bucket.
- **Bucket\* criarBucket(ofstream &dataFile):** Escreve um bucket na memória secundária.

#### 4.4. bloco.hpp

- **BlocoCabecalho\* criarBlocoCabecalho()**: Função construtora do cabeçalho do bloco.
- **Bloco\* criarBloco()**: Função construtora de um bloco.
- **void destruirBloco(Bloco\* bloco)**: Função destrutora de um bloco.

#### 4.5. bplustree.hpp

- **int gerar\_inteiro(string titulo)**: Dado uma string a transforma em um inteiro e retorna o valor obtido.
- **Node(size\_t \_degree)**: Função construtora de um nó degree equivale ao número de itens que um nó pode ter.
- **BPlusTree(size\_t \_degree)**: Função construtora de um árvore B+, degree equivale ao número de itens que um nó pode ter.
- **void destroyTree(Node<RegArvore>\* node)**: Função destrutora da árvore B+ dado um nó (raiz).
- **Node<RegArvore>\* getroot()**: Retorna a raiz da árvore B+.
- **Node<RegArvore>\* BPlusTreeSearch(Node<RegArvore>\* node, RegArvore key)**: Busca um registro na árvore B+ e o retorna se existir.
- **Node<RegArvore>\* search(int chave)**: Busca o Id de um registro na árvore B+ e o retorna se existir utilizando BPlusTreeSearch.
- **RegArvore\* item\_insert(RegArvore\* arr, RegArvore data, int len)**: Responsável por inserir um item em um array ordenado de nós. Ela encontra a posição correta para inserir o novo item com base em sua chave e realoca os elementos subsequentes para abrir espaço para o novo item. Retorna o array atualizado.
- **Node<RegArvore>\*\* child\_insert(Node<RegArvore>\*\* child\_arr, Node<RegArvore>\*child,int len,int index)**: insere um filho em um array de ponteiros para nós. Ela encontra a posição correta para inserir o novo filho e realoca os ponteiros subsequentes para abrir espaço para o novo filho. Retorna o array atualizado de ponteiros para nós.
- **Node<RegArvore>\* child\_item\_insert(Node<RegArvore>\* node, RegArvore data, Node<RegArvore>\* child)**: Insere um item e um filho em um nó. Ela encontra as posições corretas para inserir o novo item e filho com base na chave do item e realoca os elementos subsequentes para abrir espaço para o novo item e filho. Retorna o nó atualizado.
- **void InsertPar(<RegArvore>\* par, Node<RegArvore>\* child, RegArvore data)**: Inserir um item em um nó folha. Ela verifica se há espaço disponível no nó para inserir o novo item. Se houver espaço, o item é inserido na posição correta e o tamanho do nó é atualizado. Caso contrário, ocorre um overflow e realiza o balanceamento dos nós da árvore.
- **void insert(RegArvore\* data)**: Função principal para inserir um novo item na árvore B+. Ela recebe um ponteiro para o item a ser inserido. Se a árvore estiver vazia, ela cria um novo nó raiz e insere o item nele. Caso contrário, ela encontra o

nó folha adequado onde o item deve ser inserido usando a função **BPlusTreeRangeSearch**. Se o nó folha não estiver cheio, o item é inserido diretamente nesse nó. Caso contrário, ocorre um overflow e realiza o balanceamento dos nós da árvore.

- **void serializeBPlusTree(const BPlusTree& tree, const string& filename):** Serializa uma árvore B+ em um arquivo binário. Ela recebe a árvore e o nome do arquivo como parâmetros. Primeiro, ela abre o arquivo para escrita em modo binário. Em seguida, escreve o grau da árvore no arquivo. Em seguida, chama a função **serializeNode** para serializar recursivamente o nó raiz e seus filhos. Por fim, fecha o arquivo.
- **void serializeNode(ofstream& file, const Node<RegArvore>\* node):** Função auxiliar recursiva usada por **serializeBPlusTree**. Ela recebe um arquivo e um nó como parâmetros. Primeiro, escreve as informações do nó no arquivo, como se é uma folha e o tamanho do nó. Em seguida, escreve os itens do nó no arquivo. Se o nó não for uma folha, a função chama a si mesma recursivamente para serializar os nós filhos.
- **BPlusTree deserializeBPlusTree(const string& filename):** Desserializa uma árvore B+ de um arquivo binário. Ela recebe o nome do arquivo como parâmetro. Primeiro, abre o arquivo para leitura em modo binário. Em seguida, lê o grau da árvore do arquivo. A função cria uma nova árvore B+ com o grau fornecido. Em seguida, chama a função **deserializeNode** para desserializar recursivamente o nó raiz e seus filhos. Por fim, fecha o arquivo e retorna a árvore desserializada.
- **Node<RegArvore>\* deserializeNode(ifstream& file, Node<RegArvore>\* parent, size\_t degree):** Função auxiliar recursiva usada por **deserializeBPlusTree**. Ela recebe um arquivo, um nó pai e o grau da árvore como parâmetros. A função lê as informações do nó do arquivo, como se é uma folha e o tamanho do nó. Em seguida, cria um novo nó e preenche suas informações. A função lê os itens do nó do arquivo. Se o nó não for uma folha, a função cria um array de ponteiros para os nós filhos e chama a si mesma recursivamente para desserializar os nós filhos. Retorna o nó desserializado.
- **void destroyNode(Node<RegArvore>\* node):** Destrói um nó e seus filhos recursivamente. Ela recebe um nó como parâmetro. A função verifica se o nó existe e, em seguida, verifica se o nó é uma folha ou não. Se não for uma folha, a função chama a si mesma recursivamente para destruir os nós filhos. Em seguida, libera a memória alocada para os itens do nó e o próprio nó.
- **int countNodes(Node<RegArvore>\* node):** A função é responsável por contar o número de nós em uma árvore B+. Ela recebe como parâmetro um ponteiro para um nó da árvore e retorna o número total de nós presentes na árvore.
- **Registro\* buscar\_registro\_bpt(string index\_filename, ifstream& dataFile, int id\_busca) :** Busca um registro em uma árvore B+ e retornar o registro encontrado (ou nullptr se não encontrado). Ela recebe como parâmetros o nome do arquivo de índice da árvore B+ (index\_filename), um objeto de leitura de arquivo (ifstream& dataFile), e o ID do registro a ser buscado (int id\_busca). A função carrega a árvore B+ do arquivo de índice, realiza a busca do ID na árvore, busca o registro correspondente no nó encontrado, e se o registro for encontrado, lê os demais campos do registro no arquivo de dados. Em seguida, a função calcula o tamanho do registro

e conta o número total de blocos da árvore B+ usando a função `countNodes`. Por fim, a função libera a memória alocada para a árvore B+ e seus nós, e retorna o registro encontrado ou `nullptr` se não encontrado.

## 5. Desenvolvimento das fontes/funções

Todas as fontes e funções foram desenvolvidas colaborativamente pela dupla, durante sessões de chamada pelo aplicativo Discord. Para realizar essa tarefa, foram alocadas algumas horas de tempo livre para programação e desenvolvimento de acordo com as exigências estabelecidas no enunciado do trabalho.

## 6. Valgrind

O programa principal do trabalho é denominado `.upload <file>`, pois é através dele que os três arquivos necessários para a implementação dos demais programas serão gerados (arquivo de dados, índice primário e índice secundário). Dedicamos uma atenção especial a esse programa, uma vez que o objetivo é criar esses arquivos utilizando acessos em blocos na memória secundária e minimizando o uso da memória principal. Além disso, é essencial analisar se o programa apresenta vazamento de memória.

O vazamento de memória ocorre quando um programa aloca dinamicamente memória durante sua execução, mas não libera essa memória quando não é mais necessária. Isso resulta na perda gradual de recursos de memória à medida que o programa é executado repetidamente, podendo ocasionar problemas de desempenho e instabilidade. Em C++, a alocação dinâmica de memória é realizada através dos operadores **new** e **new[]**, e a liberação de memória é feita com os operadores **delete** e **delete[]**. Quando a memória alocada dinamicamente não é liberada adequadamente utilizando os operadores correspondentes, ocorre um vazamento de memória.

Para verificar se o programa `.upload <file>` apresentava vazamento de memória, utilizamos a ferramenta Valgrind. O Valgrind é uma ferramenta poderosa de depuração e análise de memória em sistemas Linux. Seu principal propósito é detectar e diagnosticar problemas relacionados à memória, como vazamentos de memória, uso incorreto de ponteiros, acesso a áreas inválidas de memória e condições de corrida. Além da detecção de erros de memória, o Valgrind também oferece suporte a outras ferramentas, como o Memcheck (para detecção de erros de memória), o Cachegrind (para análise de desempenho do cache) e o Helgrind (para detecção de condições de corrida em threads).

Inicialmente, ao realizar os testes, identificamos que o programa apresentava vazamento de memória, o que poderia ser a causa do alto consumo de memória RAM (em média, 1,7 GB de RAM). Realizamos várias iterações de testes utilizando a ferramenta Valgrind para corrigir todos os vazamentos de memória encontrados. Após a correção, observamos que o programa tornou-se consideravelmente mais rápido e passou a consumir uma quantidade significativamente menor de memória RAM (no máximo 60 MB, independentemente do computador utilizado).

Na figura 4 é mostrado a saída do programa `.upload <file>`, no qual também é o tempo de execução até a finalização do programa e a criação dos três arquivos e o uso máximo de memória RAM em quilobytes (KB). Por outro lado, a figura 5 mostra a saída do programa `.upload <file>` com o uso da ferramenta Valgrind.



```
matheus@mts: ~/Documentos/TP2-BD/scripts
Tabela hash criada com sucesso!
Inserindo registros no arquivo de dados...
Arquivo de dados criado com sucesso!
Índice primário e secundário criado com sucesso!
Tempo de CPU: 20.2809 segundos
Uso máximo de memória: 59520 kilobytes
matheus@mts:~/Documentos/TP2-BD/scripts$
```

Figura 4. Exemplo de saída do programa `./upload <file>`.

```
matheus@mts: ~/Documentos/TP2-BD/scripts
Tabela hash criada com sucesso!
Inserindo registros no arquivo de dados...
Arquivo de dados criado com sucesso!
Índice primário e secundário criado com sucesso!
Tempo de CPU: 490.593 segundos
Uso máximo de memória: 157688 kilobytes
==10653==
==10653== HEAP SUMMARY:
==10653==    in use at exit: 0 bytes in 0 blocks
==10653==   total heap usage: 37,073,565 allocs, 37,073,565 frees, 26,174,336,973 bytes allocated
==10653==
==10653== All heap blocks were freed -- no leaks are possible
==10653==
==10653== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
matheus@mts:~/Documentos/TP2-BD/scripts$
```

Figura 5. Exemplo de saída do programa `./upload <file>` com o uso da ferramenta Valgrind.

Portanto, o uso do Valgrind foi efetivo para identificar e corrigir os vazamentos de memória presentes no programa, garantindo um uso adequado dos recursos de memória. A saída do Valgrind indica que nenhum vazamento de memória foi detectado durante a execução do programa. O relatório mostra que nenhum byte está em uso no momento da saída e que todos os blocos de memória alocados foram liberados corretamente. Além disso, é destacado que não foram encontrados erros em nenhum contexto durante a análise. Isso significa que o programa foi executado sem problemas relacionados à alocação e liberação de memória, garantindo assim um uso adequado dos recursos de memória disponíveis.