Trabalho 2 - Banco de Dados I

Alunos: Erin Dante de Oliveira Vasconcelos

Rebeca Madi Oliveira

Saimon Leandro de Sousa Tavares

1. Decisões Técnicas

a. Estruturas

i. Registros

Cada registro possui no máximo 1505 bytes, ou aproximadamente 1,48 KB. Ele é estruturado como:

```
int id;                        // Código identificador do artigo. 4 bytes
char title[300];               // Título do artigo. 300 bytes
unsigned short year;           // Ano de publicação. 4 bytes
char authors[150];             // Lista dos autores. 150 bytes
unsigned int citations;        // Número de citações. 4 bytes
char lastUpdated[19];          // Data e hora da última atualização (formato YYYY-MM-DD HH:MM:SS). 19 bytes
char  snippet[1024];           // Resumo textual do artigo (+1 para o terminador nulo). 1024 bytes
```

ii. Blocos

Os registros são armazenados em um único bloco, sem espalhamento. Cada bloco armazena no mínimo 2 registros e máximo de 6 registros. Os blocos são compostos de seu ID, o registro e o tamanho do registro. Os registros possuem tamanhos flexíveis para diminuir o acesso ao disco, economizar memória e evitar colisões.

iii. Buffer

O buffer foi definido com 4MB. Inicialmente, tentamos implementar a política de substituição LRU, mas optei pelo método FIFO devido à simplicidade e melhor adequação ao projeto.

b. Organização de arquivos por hashing

i. Bucket

Um bucket é composto por uma sequência de 16 blocos, ou seja, 64 KB de dados por bucket. A escolha do tamanho do bucket foi feita para acomodar mais registros e reduzir a possibilidade de overflow.

ii. Arquivo

O arquivo de dados é organizado por hash e é composto de Buckets, Blocos e Registros. No início, consideramos o tamanho do registro como fixo, porém, para a diminuição de colisões e para diminuir acessos ao disco, foi adotado tamanho flexível (de 2 a 6 registros).

O arquivo de teste contém aproximadamente 1.022.000 registros. Cada registro pode ter até 1,48 KB. No pior caso, cada bloco suporta apenas dois registros de tamanho máximo, dessa forma, são necessários cerca de 511.000 blocos para armazenar todos os registros. Para acomodar todos os blocos necessários, o arquivo de hash foi definido com 2000 MB, o que equivale a aproximadamente 512.000 blocos de 4 KB.

Em caso de colisão foi criado um arquivo secundário de overflow. Definimos que teria um tamanho equivalente a 100 buckets. Se todos os blocos de um bucket estiverem cheios, o registro irá para o arquivo de overflow.

c. Índice primário

Utilizamos índices densos, pois o arquivo não estava fisicamente ordenado, o que não viabiliza um índice esparso. O índice primário, ao apontar para o arquivo de hash, acaba se tornando denso, já que não há possibilidade de ordenação eficiente dentro do disco. O arquivo de índice tem tamanho de 11MB. O arquivo de entrada possui cerca de 1.022.000 registros/chaves. No nível 1, há 510 chaves, armazenadas em 1 bloco. No nível 2, há 260.610 chaves, armazenadas em 511 blocos. No nível 3, teriam 132.911.100 chaves, que seria suficiente para armazenar o mínimo de registros.

i. Árvore B

Utiliza-se a árvore para implementar o índice primário que permite busca rápida pois seu formato balanceado permite as operações de inserção e pesquisa tenham custo logaritmo. Os nós de uma árvore B têm a capacidade de guardar diversas chaves, diminuindo a profundidade da árvore e, consequentemente, a quantidade de acessos necessários para localizar dados

ii. Escolha da ordem

Para definir a ordem da árvore primeiro consideramos a chave primária, que é o ID (inteiro de 4 bytes). O ponteiro de nó offset (inteiro de 4 bytes) para apontar para outros nós. Para ter a informação se um nó é folha um campo é adicionado como verificador_de_no (unsigned short 2 bytes). Para a localização do bloco utilizamos um identificador do bloco (inteiro 4 bytes). Resultando no tamanho do registro sendo de 14 bytes. Sendo o tamanho do bloco de 4096 bytes, então fizemos o cálculo para a ordem m da árvore:

$$2 * m * 4 + (2 * m + 1) * 4 + 6 <= 4096$$

$$8m + 8m <= 4092$$

$$16m = 4086$$

$$m = 255$$

iii. Inserção

Na inserção, a raiz da árvore é sempre armazenada no primeiro bloco do arquivo de índice, sendo lida e mantida no buffer. Durante o percurso na B-tree, caso um nó não esteja carregado, ele é buscado primeiro no buffer e, se não encontrado, é localizado no disco por meio do ponteiro armazenado no bloco atual. Para evitar inconsistências, os blocos envolvidos na inserção são marcados como pinned_blocks. Ao final da inserção, esses blocos são atualizados.

2. Bibliotecas e Classes

a. **B-Plus-Tree**
    i. **Classe PrimaryIndex**

    A classe PrimaryIndex implementa uma B-tree para gerenciamento de índices primários. Ela permite inserir, buscar e balancear registros, utilizando nós que armazenam chaves e ponteiros, com um limite definido pela constante M. O construtor aloca a memória necessária e o destrutor a libera. Métodos de serialização, como serializeNode e deserializeNode, convertem nós para bytes e recuperam dados do disco. Para inserções, o método insertRecord adiciona registros e divide nós quando cheios. A busca de registros é realizada por searchRecord e a árvore pode ser visualizada com printTree. A busca binária é usada em binarySearch para localizar chaves nos nós de forma eficiente.

b. **Block**
    i. **Classe Block**

    A classe Block representa um bloco de dados com capacidade de armazenar múltiplos registros. O bloco possui um identificador (id), um buffer (bytes_block) que armazena os dados, um contador de registros (record_counter), e a quantidade de espaço livre (freeSize). Na inicialização, ele aloca memória para o buffer e, opcionalmente, carrega dados existentes em blockb para reconstruir o bloco.

    A função addRecord adiciona um registro serializado ao bloco, controlando se há espaço disponível e atualizando o espaço livre e a posição atual. O método setId insere o identificador do bloco no buffer, enquanto deserializedBlock reconstrói registros a partir de dados serializados. A função findRecord busca um registro pelo idb e retorna o registro se encontrado. A classe libera toda a memória alocada ao ser destruída.

c. **Bucket**
    i. **Classe Bucket**

    A classe Bucket representa um agrupamento de blocos em um balde para armazenamento de dados, projetado para conter até

16 blocos (MAX_BLOCK_COUNTER) e com um tamanho máximo definido (BUCKET_SIZE_T). A classe armazena uma chave de hash (hash_key) que identifica o balde, um contador de blocos (block_counter) que controla quantos blocos foram inseridos, e a quantidade de espaço livre (freeSize) e a posição atual (currentOffset) para gerenciamento de armazenamento.

No construtor, Bucket(int id), o balde é inicializado com um ID. A função setBucket permite configurar o balde com dados pré-existentes, enquanto insertBlock insere um bloco no balde, atualizando o espaço livre e o contador de blocos. A função findRecord busca um registro específico por ID dentro dos blocos armazenados. A initializeBucket prepara o balde para uso, e o destrutor (~Bucket) libera a memória alocada para o balde.

d. Buffer

i. Classe Buffer

A classe Buffer implementa um sistema de gerenciamento de buffer, utilizando uma estratégia de LRU (Least Recently Used) para paginação e permitindo armazenar até 4MB de dados. Ela lida com operações de adição e remoção de blocos, busca de registros e atualização de dados tanto no buffer quanto no disco. O construtor inicializa o buffer e o destruidor libera a memória alocada. Métodos como addBlocks, removeBlock e removeNode gerenciam a adição e remoção de dados do buffer, utilizando deslocamento e controle de tamanho livre. A classe oferece funcionalidades de busca, com métodos como findRecord e findIndexNode, permitindo localizar registros ou nós dentro do buffer. Além disso, ela mantém a sincronização dos dados entre o buffer e o banco de dados ou a árvore B-Plus por meio dos métodos updateBuffer, updateNodeBuffer e updateDisk. Para manipulação de registros, oferece métodos como addRecordToBlock e findBlockInBucket, que facilitam a adição de registros a blocos ou a busca por blocos específicos. Métodos como clear permitem limpar o buffer e alterBuffer

substituem o conteúdo de um bloco existente. Em resumo, a classe gerencia dados temporários, realizando cache e leitura/gravação de blocos entre o buffer e o disco de forma eficiente.

e. Database

i. Classe Database

A classe Database gerencia arquivos de disco para simular um sistema de banco de dados em que os dados são armazenados em três arquivos distintos: um para o hash, um para overflow e um para o índice primário. Ela implementa métodos para inicializar e resetar os discos, alocando espaço e preenchendo-os com zeros. Além disso, oferece funções para ler e gravar blocos de dados nesses arquivos usando operações de leitura e escrita binária. A classe também lida com o controle de posições no disco usando lseek e realiza operações de gravação de blocos através de métodos como writeBlockToDisk, writeOverflowDisk, writePrimaryDisk, e leitura com findBlockInDisk e readPrimaryDisk. O gerenciamento de arquivos é feito por meio de descritores de arquivos, que são abertos no construtor e fechados no destruidor.

f. Record

i. Classe Record

O código define a classe Record no namespace Record, que representa um artigo acadêmico. Ele tem um construtor para inicializar os dados do artigo, incluindo ID, título, ano, autores, citações, última atualização e um snippet. Outro construtor permite criar um objeto a partir de uma string de bytes serializada. A classe também possui métodos para exibir os dados no console, converter o objeto para uma representação em bytes (toString) e restaurar o objeto a partir desses bytes (toObject). A serialização envolve o cálculo do tamanho total necessário e a alocação dinâmica de memória. O código pode ser otimizado em relação ao controle de memória e

redundância, além de considerar o uso de std::string para maior flexibilidade e segurança.

3. **Funções Principais**
    a. **findrec.cpp**
        i. int **hashFunction**(int id, int numBlocks)
        Essa função recebe um id e o número total de blocos (numBlocks). Ela calcula um índice de hash, retornando o resto da divisão do id pelo número de blocos. Esse índice será usado para localizar em qual bloco o registro pode estar.
        ii. void **findRecordById**(int id, const std::string& filename)
        Essa função busca um registro específico (id) em um arquivo binário com nome filename. Ela usa a função de hash para encontrar o bloco onde o registro provavelmente está armazenado.
        iii. int **main**(int argc, char* argv[])
        A função main é responsável por receber o id do registro como argumento de entrada e chamar findRecordById para localizar o registro. Se o usuário não fornece um ID como argumento, exibe uma mensagem de uso correto. Caso contrário, converte o argumento em um inteiro e chama findRecordById.

        Essas funções juntas buscam registros em arquivos binários por meio de uma função hash e de uma lógica de armazenamento em blocos, usando um arquivo de overflow para dados extras.
    b. **upload.cpp**
        i. void **hashingFile**(Database::Database* db, const string& filename, int sz, int f)
        Essa função organiza o arquivo de entrada (filename) em um arquivo binário, usando hashing para distribuir registros em "buckets" e blocos. Ela lê cada linha de um arquivo de entrada, cria um objeto Record para representar cada linha, e usa o hashing para determinar o bucket correto onde o registro será armazenado. Caso o bucket tenha espaço, o registro é

adicionado; se não, o registro é salvo em um arquivo de overflow para armazenar dados excedentes.

ii. int **main**(int argc, char* argv[])

A função main é responsável por configurar o programa, processando os argumentos de entrada e chamando hashingFile. Os parâmetros opcionais permitem especificar a quantidade de linhas a serem lidas e uma posição inicial. Em geral, o código otimiza a escrita e leitura de dados com o uso de um buffer intermediário e um arquivo de overflow.

c. seek1.cpp

i. void **findRecordById**(int id)

A função findRecordById busca um registro no banco de dados usando um ID fornecido. Primeiro, cria objetos para acessar o banco de dados e a árvore B-Plus. Ela lê o bloco inicial do banco de dados e desserializa a árvore B-Plus. Com a árvore carregada, a função procura o ID na árvore e, se encontrado, obtém o endereço do bloco onde o registro está armazenado. O bloco é então lido do disco e convertido em um objeto Record. Então, o registro encontrado é exibido. Se o ID não for encontrado, uma mensagem de erro é exibida.

ii. int **main**(int argc, char* argv[])

O programa espera receber um único argumento da linha de comando: o ID do registro a ser localizado. Se o número de argumentos for diferente de 2, exibe uma mensagem de erro sobre o uso correto do programa. O ID passado como argumento é convertido para um número inteiro usando std::atoi. A função findRecordById(id) é chamada para localizar e exibir o registro correspondente.

4. Ata

Erin documentou o trabalho e revisou o código, assim como teve um papel principal no desenvolvimento da função seek. Rebeca ficou responsável por fazer as classes e bibliotecas, desenvolvendo também as funções principais do trabalho, como o upload. Saimon documentou, revisou o código e atuou no desenvolvimento da função de findrec.