Banco de Dados

Trabalho Prático 2 - Armazenamento e consulta em banco de dados

Davi Simite Damasceno, Luis Guilherme Ferreira Sena e Helio Endrio Cardoso Rodrigues.

1. Organização do Banco de dados

O banco de dados foi organizado por hashing de acordo com a especificação do trabalho. Ele é gerado a partir do programa *hash.cpp* o qual faz as seguintes tarefas:

A. Recebe um arquivo CSV para adquirir os registros do banco de dados

O programa busca no diretório no qual está e lê um arquivo CSV que contém os campos de cada registro. Para isso, foi utilizado expressões regulares para identificar automaticamente o início e final de cada campo.

B. Trata erros contidos no arquivo de entrada.

Foi necessário substituir, inserir e/ou excluir caracteres devido ao arquivo CSV conter vários erros de digitação, lixo e caracteres nulos. Após a correção todos os campos obedeciam a gramática estabelecida nas expressões regulares e portanto foram reconhecidas.

C. Define a estrutura do banco de dados.

Conforme especificado, cada registro é composto por os seguintes campos e no nosso banco de dados alteramos a forma de representação de alguns deles:

Id	inteiro
Titulo	char [300]
Ano	inteiro
Autores	char [150]
Citações	inteiro
Atualização	char [20]
Snippet	char [1024]

Essa alteração fez o desenvolvimento do banco de dados ser mais rápido e um pouco mais leve devido ao gasto de representação de data e hora ser mais caro que uma string contendo o mesmo conteúdo.

Com essa estrutura o custo para representar um registro é:
id(4 bytes) + titulo (300 bytes) + ano (4 bytes) + autores (150 bytes) + citacoes(4 bytes) + atualizacao(20 bytes) + snippet (1024 bytes) + custo da struct (2 bytes) = 1508 bytes

Portanto cada registro pesa 1508 bytes para ser representado.

Decidimos que a função de hashing terá inicialmente:

$$\frac{\textit{Quantidade de registros no documento de entrada}}{\textit{Quantidade de Blocos no bucket} \bullet \textit{Quantidade de Registros por bloco}}$$

O motivo foi que a quantidade de buckets será suficiente para armazenar todos os registros. Porém apenas se a função de hashing for muito eficiente, o que não é o caso com esse número de buckets. Portanto o número de buckets será o próximo número primo após o valor do resultado. Desse modo teremos buckets para todos os registros, a taxa de overflow será baixa, mas o gasto de memória será maior.

A quantidade de registros no bloco foi a maior possível. Normalmente no linux o tamanho do bloco de disco é 4096, como o registro pesa 1508 então couberam 2 registros por bloco e sobrou um espaço de 1080 bytes que não foi utilizado.

Por fim, cada bucket tem 2 blocos porque usamos outra função de hash dentro do bloco, caso o id do registro seja par então irá ser encaminhado para o primeiro registro e caso contrário o segundo registro.

D. Cria o banco de dados

Quando o programa chega este momento toda a estrutura já está estabelecida e então é criado um arquivo binário *bancoDeDados.dat* o qual será escrito quando todo o arquivo CSV estiver na memória principal e devidamente corrigido. O espaço que não foi utilizado é gravado como um espaço vazio (‘ ‘) para manter o padrão estabelecido de espaço utilizado.

2. Descrição das funções

A. artigo PegarCampos(string line,regex reg, FILE *bd)

Retorna o artigo que a expressão regular *reg* conseguir identificar no arquivo apontado pelo ponteiro *bd*.
Dentro delas existem mais casos e condições para o tratamento de erros.

B. void hash_function(artigo paper, vector<bucket> &buckets, int bucket_num)

Adiciona o *artigo* paper no vetor *buckets* utilizando a função de hashing com a chave sendo o id do paper e a quantidade de buckets que não são de overflow especificada por *bucket_num*.

C. int inserir(bucket &bucket_destino, artigo paper)

Essa função é chamada pela função B anterior e adiciona o artigo no bucket de destino fazendo a escolha sobre qual bloco escolher para ser inserido.

D. int primo(int num)

Utilizada para verificar se um número é primo ou não. Tem a finalidade de estabelecer um tamanho de buckets que seja primo para melhorar o espalhamento dos registros.

3. Consulta no banco de dados por hashing

Depois de executar o programa hash.cpp será criado o banco de dados e um arquivo chamado *MetaDados.txt* o qual contém a quantidade de buckets criada durante leitura dos campos do arquivo CSV.

Com essas informações é possível fazer um programa para fazer consultas no banco de dados levando em consideração outras informações que foram definidas antes como a quantidade de registros por bloco e etc.

Para fazer a consulta é necessário executar o programa consulta_hash.cpp e informar o número de id do artigo desejado. Caso esteja presente então retornará os seus campos e blocos lidos. Esse programa realiza as seguintes tarefas.

A. **Buscar metadados**
   Ao ser executado irá primeiramente verificar a quantidade de buckets existente no banco de dados para poder fazer os cálculos necessário para achar o registro. Portanto ele abre o arquivo de metadados e recolhe essa informação.
B. **Calcular o bucket do registro consultado**
   Com a quantidade de buckets é possível determinar a posição do registro com o custo de execução igual a O(1) através da função de hashing.
C. **Verificar se ele existe**
   Quando o bucket for achado será verificado se em um de seus registros contém o id informado ao programa.
D. **Mostrar o registro e o bucket ao qual pertence**
   Se ele estiver presente então o programa mostra todos os campos do registro e o bucket ao qual ele pertence.
   Caso contrário retorna uma mensagem informando que o artigo não consta na base de dados criada
E. **Informar a quantidade de blocos lidos**
   Durante o percurso até o registro é possível que seja visitado outros buckets devido a overflow durante a criação do banco de dados. Portanto essa informação é mostrada ao final de toda consulta

4. Funções de busca

A. **int procurar(int id, bucket bucket_consultado,FILE *arq, int bucket_num,int blocos_lidos)**
   Essa é a função mais importante do programa. Ela busca em *bucket_consultado* a qual é o bucket resultante do hashing se o registro com o *id* informado está presente no banco de dados apontado pelo ponteiro *arq* e incrementa *blocos_lidos* o qual indica a quantidade de blocos que foram lidos para chegar a uma conclusão. A variável *bucket_num* é usada para calcular em quais blocos está o registro caso seja encontrado.
B. **void mostrar_artigo(artigo registro)**
   Função que mostra na tela o registro caso encontrado;

C. **void mostrar_bloco(int consulta, int bucket_num)**

Calcula o bloco no qual está o registro com id consultado.

Com todas essas funcionalidade implementadas é possível criar um banco de dados e consultar nele em pouco tempo. Os programas *hash.cpp* e *consulta_hash.cpp* foram criados pelo aluno davi simite damasceno.

1. **Estrutura dos nós da Árvore B+:**
    a. **typedef int tipoBloco;**
    Estrutura que pode ser alterada para se adequar ao endereço do bloco(guardado em um nó folha)

    b. **typedef int tipoValor;**
    Estrutura que pode ser alterada para se adequar ao valor do registro

    c. **typedef struct{**
    tipoValor elemento;
    tipoBloco endereco;
    } tipoInfo;
    Estrutura que guarda ID e endereço do bloco do artigo:
    “elemento” guarda o valor chave do nó;
    “endereco” guarda o endereço de disco onde o valor, caso o nó seja uma folha, ou é vazio, caso contrário.

    d. **typedef struct BNo{**
    tipoInfo* dados;
    struct BNo filhos;**
    int nChaves;
    bool folha;
    } TipoBNo;
    Estrutura que gerencia as folhas e os nós internos da árvore:
    “dados” guarda endereço de um vetor de elementos;
    “filhos” guarda o endereço de um vetor que contém ponteiros para os filhos do nó atual;
    “nChaves” registra a quantidades de elementos no nó atual;
    “folha” indica com “true” caso o nó atual seja uma folha.

2. **Funções da implementação da Árvore B+:**
    a. **void criarArvore(TipoBNo** raiz);**
    Função responsável por gerar uma raiz nula, para garantir que a árvore possua apenas elementos válidos.

    b. **TipoBNo* criarFolha(tipoInfo dado);**
    Função responsável pela criação das folhas da árvore. Usada como função interna para adequar o dado a estrutura dos nós da árvore.

    c. **TipoBNo* criarBloco(tipoValor dado);**
    Função usada para criar blocos/nós internos da árvore, estes blocos devem

conter apenas o valor chave do elemento.

d. **void mostrarFolha(TipoBNo* folha);**
   Função usada internamente para ajustar o print de um nó folha;

e. **void mostrarArvore(TipoBNo* raiz);**
   Função usada para exibir elementos que estão inseridos na árvore.

f. **void inserirFolha(TipoBNo** folha, tipoInfo dado);**
   Função usada internamente para inserir um elemento em uma folha que ainda não excedeu a capacidade máxima de chaves.

g. **TipoBNo* dividirFolha(TipoBNo** folha, tipoInfo dado);**
   Função usada quando é necessário uma divisão da folha para a inserção de uma nova chave.

h. **void inserirBNo(TipoBNo** pai, TipoBNo* filho);**
   Função semelhante a “inserirFolha”, porém é aplicada apenas em nós internos/não-folhas da árvore.

i. **void dividirBNo(TipoBNo** Bloco, TipoBNo* filho);**
   Função usada para lidar com a divisão em cascata causada pela inserção de uma chave que gerou uma divisão de uma folha e consequentemente a divisão de um nó interno/não-folha.

j. **void inserirDado(TipoBNo** raiz, tipoInfo dado, TipoBNo** pai);**
   Função principal usada para inserir elementos-chaves na árvore. Ela é a função gatilho para “criarFolha”, “inserirFolha”, “criarBloco”, “inserirBNo”, “dividirFolha” e “dividirBNo”

k. **TipoBNo* buscarDado(TipoBNo* raiz, tipoValor dado);**
   Função que dado uma chave, ela retorna o endereço do nó em que a chave está.