Fabio Fogarin Destro (10284667), Paulo A. de Oliveira Carneiro (10295304),
Renata Vinhaga dos Anjos (10295263), Vítor H. Gratiere Torres (102849552)

Segundo Trabalho Prático de Organização de Arquivos - árvore B Virtual

Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Bacharelado em Ciências da Computação

Departamento de Ciências de Computação

SCC0215 - Organização de Arquivos

Professora Dra. Cristina Dutra de Aguiar Ciferri

24 de Junho de 2018, São Carlos, SP

Sumário

1 Introdução

Este trabalho consiste na extensão da primeira parte, em que foi feita a manipulação do arquivo de dados, com a inclusão de três funções: `[10]`, `[11]` e `[12]`, que envolvem a utilização do arquivo de índice de árvore B + *buffer-pool*. Dentre essas são: inserção de todos os registros acoplada a função `[1]`, inserção de apenas um registro acoplada a função `[6]` e busca.

Nota: Neste trabalho não foram implementadas as funções `[13]` e `[14]` de remoção e atualização.

O programa foi feito utilizando a linguagem `C`, sendo compilado e testado no sistema operacional Linux (`Ubuntu 16.04 lts`).

2 Macros

Mais algumas substituições de sintaxe foram incluídas para facilitar o entendimento do código, com relação direta às equações para cálculo do *byte offset* no arquivo de índice e auxílio nas funções da árvore B:

- `ORDEM` - ordem da árvore B, que equivale ao número máximo de filhos de cada nó.
- `MAX_CHAVES` - número máximo de chaves que um nó pode ter (`ORDEM - 1`).
- `TAM_PAG` - tamanho de um nó (página) na árvore B virtual (116 *bytes* = 4 *bytes* para o número de chaves atuais da página + 72 *bytes* (`MAX_CHAVES * (4 + 4)`) para o vetor de chaves e RRNs do arquivo de dados + 40 *bytes* (`ORDEM * 4`) para o vetor de "ponteiros" na árvore B virtual).
- `TAM_CABECALHO_B` - tamanho do cabeçalho do arquivo de índice (13 *bytes* = 1 *byte* para *status* do arquivo + 4 *bytes* para o RRN do nó raiz + 4 *bytes* para a altura da árvore + 4 *bytes* para o RRN da última página inserida no índice).
- `TAM_BUFFER` - número de páginas máximo que o *buffer-pool* comporta.
- `NIL` - referência a vazio. Valor = `-1`
- `T` - taxa de ocupação da árvore B (`ORDEM / 2`)

3 Estruturas de Dados

Utilizou-se de mais 4 `structs` como auxílio e armazenamento temporário de dados para manipulação:

3.1 De árvore B

- `Cabecalho_B`: contém o status do arquivo de índice, o RRN do nó raiz, a altura da árvore e o RRN do último nó inserido na árvore.
- `C_PR`: contém a chave (codEscola) e seu respectivo RRN para o arquivo de dados.
- `Pagina`: contém o número de chaves atual do nó (página), um vetor (tamanho fixo) de `structs C_PR` e outro vetor (tamanho fixo) de ponteiros para os nós filhos.

3.2 De *buffer-pool*

- `BufferPool`: contém o número de *hits* (acertos) e *misses* (não encontrados), o vetor (tamanho fixo) de RRNs das páginas do arquivo de índice, um vetor (tamanho fixo) que indica se as páginas estão atualizadas ou não e um vetor (tamanho fixo) de `structs Pagina`.

4 Sobre *buffer-pool*

Buffer-pool mantém armazenado em *RAM* um certo número de páginas do arquivo de índice (`.bin`) afim de reduzir o número de acessos ao disco. Utilizou-se a política *Least Recently Used* (*LRU*) para substituições.

O algoritmo do *buffer-pool* foi inserido nas funções implementadas para diminuir o acesso ao disco. Essa ferramenta consiste na implementação de 3 funções:

4.1 *Get*

1. Percorre o *buffer-pool*, procurando a página desejada, caso encontre incremente em um o valor de *buffer hits* e pule para o passo 3.
2. Se não encontrou a página desejada, faça um acesso a disco carregando a página desejada em uma página auxiliar e chame a função *put*, que irá inseri-la no *buffer-pool*.
3. Retorne o conteúdo desta página.

4.2 *Put*

1. Percorre o *buffer-pool*, procurando a página desejada, caso encontre mude o valor da *flag* para `1`.
2. Se o valor da variável auxiliar *flag* for igual a zero, significa a página não foi encontrada, logo é necessário adicionar um *buffer miss*. Nesse caso é verificado se o

valor do RRN de `'i'` é igual a `-1`, se sim significa que a o *buffer-pool* ainda possui espaços vazios, logo a nova página é inserida nessa posição, que é a primeira posição vazia, caso o contrário o *buffer-pool* está cheio e é preciso remover uma página seguindo a política de substituição *LRU* para inserir a página desejada.

3. Se o valor da variável auxiliar *flag* for igual a `1` significa que a página já está inserida no *buffer-pool*, então *buffer hits* é incrementado e a página é atualizada e marcada como modificada.

4.3 *Flush*

1. Percorre o *buffer-pool* e para cada página marcada como modificada, a página é escrita no arquivo de índice, utilizando o seu RRN.

2. Escreve em disco, no final do arquivo `buffer-info.text`, as informações de *buffer misses* e *buffer hits*.

Tanto na função *get* quanto na função *put*, a função reorganiza é chamada, essa função é responsável por manter o *buffer-pool* organizado, isto é, a raiz sempre na posição zero, e nas seguintes posições as páginas seguindo a ordem da menos recentemente utilizada até a mais recentemente utilizada, tornando assim possível aplicar a política de substituição *LRU*, retirando do *buffer* sempre o elemento menos recentemente utilizado.

5 Criação e preenchimento inicial da árvore B com *buffer-pool*

`Função [10]` - Função acoplada dentro da função `[1]` da primeira parte do trabalho. Inserção de uma chave (codEscola) no arquivo de índice árvore B com o RRN do seu registro no arquivo de dados.

Na função `[1]` são feitos(as):

1. Criação do arquivo de índice (`.bin`), criação do seu cabeçalho definindo seu *status* para inconsistente e criação do *buffer-pool*.

2. Após a inserção de um registro no arquivo de dados, é chamada a inserção no arquivo de índice de árvore B virtual. Essa funcionalidade possui 3 funções principais de inserção: *Btree_Insert*, *Insert_Non_Full* e *Split*.

3. Atualiza o *status* do arquivo de índice para consistente e o fecha.

5.1 *Btree_Insert*

1. Lê os dados do cabeçalho do arquivo de índice e armazena na `struct` auxiliar `Cabecalho_B`.

2. Caso a árvore esteja vazia, ou seja, não tenha nó raiz, cria-se uma nova árvore inserindo a página raiz no *buffer-pool*.

3. Se a árvore não está vazia, carrega-se a página raiz do *buffer-pool*[1]. Verifica-se se ela está cheia. Se estiver, realiza-se o *split* preventivo nessa e é chamada a função para busca do filho em que deve se inserir a chave (*Insert_Non_Full*). Se não, apenas chama esta mesma função.

4. Atualiza o Cabeçalho.

5.2 *Insert_Non_Full* (Recursiva)

1. Caso base: Verifica se a página (nó) atual é folha. Se for, faz uma busca sequencial para encontrar a posição correta de inserção da chave e seu RRN. Insere a página atualizada no *buffer-pool* e finaliza.

2. Se a página atual não é folha, faz uma busca sequencial nas chaves para encontrar o filho correto para inserir a chave. Ao achá-lo, através do vetor de "ponteiros" (RRN do arquivo de índice) carrega a página filha correta do *buffer-pool*.

3. Se a página carregada estiver cheia, é feito o *split* preventivo no nó e a função *Insert_Non_Full* é chamada para o novo filho correto (após o *split*) para inserção da chave, esse que também é carregado do *buffer-pool*. Se não, apenas chama a mesma função.

5.3 *Split*

1. Recebe o nó pai antigo (R) (nó que está cheio), o novo pai (S) e cria um novo filho (Z).

2. Z irá receber metade das chaves de R (taxa de ocupação - 1) com seus respectivos RRNs e se R não for folha, receberá metade de seus filhos (taxa de ocupação). R agora terá metade das suas chaves antigas (taxa de ocupação - 1).

[1] Ao se referir do carregamento de páginas do *buffer-pool*, subentende-se que as funções já descritas na seção 4 tratam do caso da página não estar presente nesse e ser preciso fazer o carregamento do arquivo. Não sendo necessário portanto, repetir o mesmo ponto nas funções seguintes.

3. S desloca seus filhos (“ponteiros”) e recebe Z como um deles (Z é uma nova página, portando seu RRN será o último RRN inserido na árvore B + 1), tal como desloca suas chaves e recebe a chave central de R que foi promovida.

4. As páginas R e S atualizadas e a nova página Z são carregadas para o *buffer-pool*.

Após a finalização dessa funcionalidade, obtivemos os seguinte dados finais com a inserção dos 3.000 registros contidos no arquivo de `dados.csv`:

- RRN raiz: 312;
- Altura árvore B: 5;
- Ultimo RRN inserido: 746;
- *Page hit*: 17.785;
- *Page Fault*: 1.620.

6 Inserção em árvore B com *buffer-pool*

`Função [11]` - Funcionalidade acoplada dentro da função `[6]`. Inserção de uma chave (codEscola) no arquivo de índice árvore B com o RRN do seu registro no arquivo de dados.

Na função `[6]` são feitos(as):

1. Abertura do arquivo de índice (`.bin`) definindo seu *status* para inconsistente.

2. Após a inserção do registro no arquivo de dados, é chamada a inserção no arquivo de índice de árvore B virtual. Funcionalidade essa que já foi especificada na seção 5.

3. É dado *flush* no *buffer-pool*.

4. Atualiza o *status* do arquivo de índice para consistente e o fecha.

7 Busca

`Função [12]` - Funcionalidade recebe uma chave (codEscola) e realiza uma busca no índice de árvore B

7.1 busca

1. Chama a função buscaArvoreB passando como argumento o codEscola a ser buscado.
2. Atribui o valor de retorno da função buscaArvoreB à variável RRN.
3. Se o valor RRN for igual a `-1`, o codEscola buscado não foi encontrado.
4. Caso o RRN não seja igual a `-1`, abre o arquivo `saida.bin`.
5. Realiza um `fseek` nesse arquivo para ir até a posição do *byte offset* (`(RRN*TAM_REG)+TAM_CABECALHO`) no arquivo.
6. Verifica se o registro nesta posição é válido, isto é, ainda não foi removido.
7. Se for um registro válido, imprime seu conteúdo, caso contrário imprime registro inexistente.

7.2 buscaArvoreB

1. Abre o arquivo de `indice.bin` somente para operações de leitura.
2. Carrega o cabeçalho do arquivo.
3. Inicia o *buffer-pool* e já insere a raiz da árvore no *buffer*.
4. Enquanto a página auxiliar não for folha, procura a chave na página atual.
5. Caso a chave seja encontrada, chama a função *flush* para limpar o *buffer*, fecha o arquivo `indice.bin` retornando o RRN desejado.
6. Se a chave da página de `'i'` for maior que a chave a buscada, deve-se descer para o filho `'i'` da página, chamando a função *get* para atualizar página atual para a página de RRN igual ao `'i'` filho, voltando então ao passo 4.
7. Se percorreu `MAX_CHAVES` e não satisfez as condições 5 e 6, deve-se descer para o último filho da página, chamando a função *get* para atualizar a página atual para a página de RRN igual ao `'i'` filho, voltando então ao passo 4.
8. Se chegou a esse passo, então o nó atual é folha, logo deve-se realizar uma busca sequencial pela chave. Se encontrar, chama a função *flush* para limpar o *buffer*, fecha o arquivo de índice (`.bin`) e retorna o RRN desejado, caso contrário chama a função *flush* para limpar o *buffer*, fecha o arquivo de índice (`.bin`) e retorna `-1` para demonstrar que a árvore não contém essa chave.