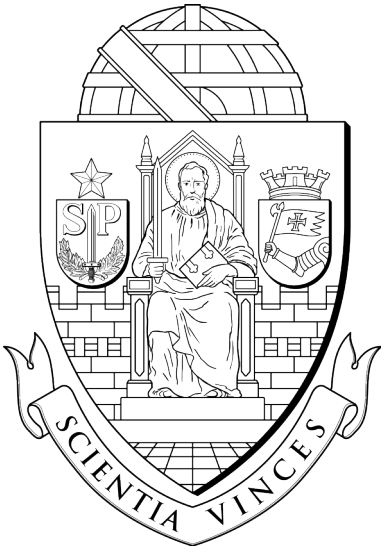

Universidade de São Paulo

Algorítmos e estrutura de dados 2

B-tree

Alunos:
Henrique Gomes Zanin
Gabriel Guimarães Vilas Boas
Marin

Sumário

Lista de Códigos

1 Resumo

O seguinte trabalho propõem a implementação de uma árvore B para indexação de registros de tamanho fixo. Optou-se por uma abordagem flexível para a definição dos campos do registro o que torna o modelo apto a gerenciar múltiplas tabelas em tempo de execução. A implementação da árvore B escolhida armazena nos nós os *relative record numbers* definidos com o tipo *long*, além de variáveis auxiliares de controle para manutenção das páginas de disco. Todos os arquivos gerados estão em formato binário, evitando que espaços desnecessário fossem usados para representar tipos primitivos da linguagem C.

2 Introdução

Originalmente concebida por R. Bayer e E. McCreight em uma competição organizada pela Boeing[1], a árvore B tem como objetivo prover uma estrutura de dados que otimize a inserção e consulta de registros por meio da paginação dos nós da árvore. A paginação do nó permite reduzir o numero de *Seeks* no disco rígido necessários para a recuperação de uma chave de registro. Este trabalho implementou em linguagem C as operações de busca e inserção em árvore B em sua forma mais encontrada na literatura, com as chaves dos registros nos nós intermediários. Outras implementações são possíveis sem considerar as variantes B* e B+ mas não serão discutidas.O trabalho está organizado da seguinte forma: introdução, aspectos técnicos da implementação, operacionalização dos menus e conclusão.

3 Implementação

A estrutura dos registros são flexíveis por permitir que o usuário defina em um arquivo texto os campos desejados. Essa implementação permite a coexistência de varias tabelas no mesmo diretório, basta definir um nome único para o arquivo de registros, um exemplo pode ser visto no diretório *bin* com o nome **alunos.table**. Nesse trabalho em questão foi pedido que os arquivos de registros e de índice primário fossem nomeados como dados.txt e index.dat respectivamente. Devido a isso não é possível que dois registros coexistam no mesmo diretório pois o arquivo de índices sempre chama index.dat, mas uma implementação genérica pode ser encontrada no repositório do projeto [1].
Sempre que o programa é executado deve-se carregar a estrutura de registros a ser usada, a exceção cabe ao menu principal, aqui modelado para seguir as exigências do trabalho proposto. Após carregar a estrutura da tabela pode-se verificar sua

[1]https://github.com/henriquezanin/B-Tree-DBMS

consistência por meio do comando *table* na interface de linha de comando. Toda a avaliação de query é feita pela função *evalQuery*, essa desempenha um papel fundamental em pois o menu principal apenas gera as consultas em conformidade com o padrão das consultas e as passa como parâmetro. Outra estrutura fundamental é a TAD **Metadata** responsável por administrar todos os campos definidos pelo usuário, também permite que os ponteiros de arquivos permaneçam abertos enquanto o usuário não troca a tabela ou o programa está em execução. Isso torna a implementação mais eficiente por evitar operações desnecessárias de *Open* e *Close* do sistema operacional.

3.1 Registros

Ao carregar a tabela o programa avalia todos os campos para atestar sua integridade, estão disponíveis 4 tipos primitivos da linguagem C: int, float, double e char. O ultimo tipo permite o usuário escolher o tamanho máximo da cadeia de caracteres. Optamos por armazenar os registros no formato binário, tal implementação otimiza o uso de espaço nos registros em detrimento do formato de texto puro. Vejamos um exemplo mais tangível, o tipo primitivo *int* em uma arquitetura de 64 bits possui 4 bytes de tamanho, quando salvo em binário esse inteiro sempre possuirá os 4 bytes. Em texto puro o tamanho varia de acordo com o tamanho do número, um exemplo para o número 10441321 pode ser visto na Tabela 1:

Tabela 1: Diferença de tamanho entre texto puro e binário

Binário	**Texto Puro**
4 bytes	8 bytes

O código fonte que trata o processamento da tabela está no **parser.c**, esse arquivo fonte também trata o formato das consultas e extração dos dados necessários para realizar a operação desejada. O comando *load* abstrai toda as operações necessárias para o carregamento da tabela em tempo de execução. Após carregada as informações ficam armazenadas na TAD Metadata e pode ser usada por todas as funções que necessitem de acesso aos campos e descrição dos mesmos. Uma observação importante é que o arquivo de tabelas não pode ser alterado após a primeira inserção de um registro, isso ocorre pois os registros tem tamanho fixo e são recuperados pelo seu *relative record number*, esse multiplicado pelo tamanho em bytes do registro[2]. Não há cabeçalho no arquivo de registros e todos os dados são escritos no formato binário, isso facilita o deslocamento no arquivo por meio da função *fseek()*

[2]O tamanho pode ser visto por meio do comando table, executado na interface de linha de comando do programa

disponível na *stdio.h*. Os campos do registro são escritos na ordem que aparecem no arquivo de tabela, facilitando a depuração do registro binário por meio de qualquer ferramenta que permita visualizar os dados em hexadecimal. Uma ferramenta muito útil é o *hexdump*, encontrado em sistemas operacionais GNU/Linux.

3.2 B-tree

Quando um registro é inserido no arquivo a função *storeData* definida em **pager.c** retorna o *RRN* do registro no arquivo de dados. Isso serve de insumo para a construção do índice primário em árvore B. Para a inserção de um novo registro o índice primário é consultado na B-tree para evitar a inserção de chaves duplicadas. Novas chaves são adicionadas em nós folhas, caso o nó supere o limite máximo de chaves as operação de divisão e promoção são chamadas para garantir o balanceamento da árvore. O aquivo da árvore está organizado da seguinte forma:

Header
[RRN da Raiz][Espaço livre]

Nós
[Chave primária][RRN do registro][Filhos][Numero de chaves][Flag isLeaf][Espaço livre]

A escolha de deixar uma página disponível para o header evita o acréscimo de espaços livres nas demais páginas que contém os nós. Caso contrário cada nova página deveria retirar 8 bytes de espaço útil para manter todos os nós com o mesmo tamanho. Uma página para o header também viabiliza implementações de variáveis de controle bem como a adição da estrutura da tabela, oque evitaria inconsistências caso o arquivo seja modificado pelo usuário[3].
Cada nó quando carregado na memória fica armazenado na TAD *btPage*. Como suas informações foram armazenadas como vetor no arquivo index.dat sua extração se dá de forma a manter um vetor de TAD's *record* que contém a chave e o RRN do registro. Os filhos se mantém como vetor pois são compartilhados entre os registros, isso evita o uso de memória desnecessária definindo um filho direito e esquerdo para cada registro.
Entende-se que armazenar *structs* diretamente no arquivo binário pode gerar falta de compatibilidade caso a linguagem de programação que opere a B-tree seja trocada. Isso se dá pois a arquitetura de Von Neumann otimiza a alocação de memória para *structs* e em certas ocasiões disponibiliza mais bytes que a soma de todas as variáveis da *struct*. Armazenar os dados brutos também otimiza o uso do espaço em disco pois evita salvar mais bytes que o necessário.

[3]Não implementado nesse trabalho

As operações de divisão(*split*) e promoção estão definidas no arquivo **btree.c**. A primeira consiste na divisão do nó quando este ultrapassa o numero máximo de registros definido no arquivo **btree.h** em MAXKEYS. Nesse caso é criado um novo nó contendo a metade superior do nó dividido e a menor das maiores chaves estabelecidas no novo nó é selecionada para a promoção. A operação de promoção é recursiva e opera no retorno das inserções, nesse passo as chaves promovida dos nós inferiores são adicionadas nos nós internos e caso superem o numero máximo são divididos e uma nova chave é promovida. O máximo de promoções e divisões que uma árvore pode experimentar é igual a sua altura. Todas as inserções nos nós são feitas de forma ordenada, dispensando algoritmos de ordenação.
A quantidade de registros em cada nó pode variar de acordo com o tamanho da pagina do sistema operacional. Para definir-la o usuário deve levar em consideração que há 2 campos que não variam com a quantidade de registros, são eles o Numero de chaves e a *flag* que indica se o nó é uma folha ou um nó intermediário, totalizando 3 bytes fixos. Os demais campos variam de acordo com a equação (1), o espaço livre é inversamente proporcional ao numero de chaves e é dado pela equação (2):

$$bytes = (MAXKEYS * 4) + (MAXKEYS * 8) + ((MAXKEYS + 1) * 8) + 3 \quad (1)$$

$$Livre = TamPag - bytes \quad (2)$$

Onde TamPag é o tamanho da página definida na formatação do disco

3.3 Query

O mecanismo de *querys* é fundamental para operar o programa. A interface de menu se comunica diretamente com o avaliador de *query*, formatando os campos inseridos pelo usuário para o formato da *query*. A inserção por lote também se torna viável, basta formatar o comando *insert* para o número de campos definido na tabela. Todo o processamento dos comandos é feito no **query.c** com auxílio das funções de *parsing* e processamento de textos definidos em **parse.c** e **utils.c**

3.4 Menu

O Menu construído nesse projeto, se ajusta ao tamanho do terminal em que o binário é executado, independente de seu tamanho ele se ajusta da melhor maneira. Isso foi feito utilizando funções e structs do kernel do sistema operacional, que informam a altura e largura do terminal em caracteres e pixeis. Por conta disso, uma depência desse projeto seria o kernel seguir o padrão POSIX. Logo, ele é compátivel com MacOS, Linux e BSDs - Windows é instável em relação ao padrão POSIX.

O código seguinte, utiliza o POSIX para printar a altura e largura em caracteres do terminal onde foi executado.

Código 1: Menu

```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main(void) {
    struct winsize terminal;
    ioctl(STDOUT_FILENO, TIOCGWINSZ, &terminal);

    printf("ALTURA: %d\n", terminal.ws_row);
    printf("LARGURA: %d\n", terminal.ws_col);

    return 0;
}
```

4 Operacionalização do banco de dados

Esse projeto foi feito pensando em duas maneiras de utilizar, a primeira, de uma maneira clássica que o usuário interage com uma interface amigável e uma segunda, na qual é visado a praticidade e velocidade do programa.

4.1 Linha de comando

Ao executar "./main –cmd"na pasta "bin", o usuário entrará no programa no modo de linha de comando. Nessa categoria, existem os comandos "insert", "search", "rrn", "load", "table", "help", "man"e por fim "exit".

4.1.1 Insert

Para utilizar o comando "insert", os dados seguem um padrão para serem inseridos. Para strings, basta colocar a string dentro de aspas, por exemplo "gabriel", para inteiros basta o numero como 4, para numeros racionais deve se colocar os decimais após um "."da seguinte maneira: 9.25. Concatenando tudo na ordem descrita na tabela, segue um breve exemplo.

insert 11218521,"gabriel","marin","bsi",8.25

4.1.2 Search

Já a instrução "search", é necessário passar apenas a chave que deseja buscar concatenada com o próprio comando. Segue o exemplo:

search 11218521

4.1.3 Rrn

Essa ordem, ao passar uma posição como numero inteiro, printa o registro guardado na posição referênciada. Segue seu exemplo:

rrn 0

4.1.4 Load

Ao entrar com essa funcionalidade, é necessário passar o arquivo da tabela desejado, sem aspas. Como o banco de dados é flexível, é necessário declarar os tipos no arquivo "*.table"e passar como parâmetro neste comando. Segue um exemplo:

load alunos.table

4.1.5 Table

Nessa opção, simplesmente é printado os campos da tabela que foi carregada. Segue o exemplo de como utilizar:

table

4.1.6 Help

Esse comando, printa os comandos disponiveis para o usuário e como visualizar o manual dos respectivos comandos. Segue o exemplo:

help

4.1.7 Man

Para o usuário entender melhor como utilizar cada comando e o que ele faz, foi embutido um simples manual no padrão dos manuais do Linux para cada comando. Para ver o manual, basta concatenar o comando que deseja aprender com o "man". Segue exemplo:

man insert

4.1.8 Exit

Simples comando para sair do programa, segue exemplo:

exit

4.2 Interface de usuário

Diferente da outra forma, se executarmos apenas "./main", o programa iniciará com as funcionalidades de interface de usuário, interagindo mais para obter as entradas de quem está utilizando sendo mais amigável. Dessa forma, os comandos disponibilizados são: "insert", "search", "help"e "exit".

4.2.1 Insert

Para entrar esta instrução, basta seguir as descrições do programa digitnado "1"e clickando em enter. Dessa forma, ele irá perguntar ao usuário as informações necessárias. No final das perguntas, o cadastro terá sido feito.

4.2.2 Search

Da mesma forma que o insert, este comando segue o mesmo padrão, ao digitar "2"e dar enter, o programa perguntará a chave e entrando com a chave que no caso é o número usp, já encontrados o cadstro e as informações são exibidas.

4.2.3 Help

Seguindo o mesmo padrão, ao digitar "3"e clickar em enter, o programa exibirá os comandos disponiveis e quais seus respectivos números.

4.2.4 Exit

Respectivamente, dando a entrada "0", o programa finaliza.

4.3 Comandos em lote

Para carregar comandos em lote, é necessário passar a opção batch no momento de executar o binário, seguido pelo arquivo com os comandos. Após isso, basta executar o programa no modo que preferir e fazer o que deseja. Segue o exemplo:

./main –batch caso2.in

4.4 Erros comuns

Existem alguns erros que o usuário pode cometer, desencadeando sinais aqui explicitados.

4.4.1 Failed to open *.table

Esse erro, indica que o arquivo da tabela deve estar no mesmo diretório onde o programa foi executado. Caso tentar abrir um arquivo com essa extensão e o programa não encontrar a referência, esse sinal será exposto para o usuário.

4.4.2 Query sem parâmetros necessários

Quando uma query é executada, são acessados os parâmetros nela existentes. Por conta disso, quando os parâmetros necessários não são referenciados, o programa apresentará um "segmentation fault". Ainda não há tratamento para querys de tamanho menor que o esperado na inserção. Por conta disso, erros internos da linguagem C podem aparecer para o usuário.

4.4.3 Finalização inesperada

Na interface de usuário, caso o usuário digite um texto no lugar no comando, o programa executa a finalização. Conteúdos em strings não são tratados nessa entrada.

4.4.4 Arquivo de tabela não definido

Caso, o usuário venha a fazer uma inserção sem carregar uma tabela primeiramente, o erro "ULL Metadata in function argument!"é apresentado. Basta carregar

o arquivo "*.table"que deseja, descrevendo os campos que serão armazenados no registro.

4.4.5 Query não existe

Na linha de comando, ao digitar uma instrução inexistente, o programa indicará isso com a mensagem "Invalid query!". Para visualizar as ordens disponiveis, entre com o comando "3"na interface de usuário e com o comando help na linha de comandos.

5 Conclusão

Diante das estruturas de indexação mais simples como índices ordenados, a árvore B tem como vantagem a otimização da recuperação de registros por operar com registros do tamanho da página de disco, diminuindo o a quantidade de *seeks* e consequentemente o tempo necessário para operar a árvore. Como forma de aproximar a implementação de um sistema gerenciador de banco de dados comercial, essa implementação levou em consideração a possibilidade de que várias tabelas definidas pelo usuário coexistam no mesmo diretório. O programa conta com um avaliador de *querys* funcionando como uma API para interfaces se comunicarem com os registros. Melhorias futuras devem ser feitas para que seja possível remover registros.

Referências

[1] Michael J Folk and Bill Zoellick. *File structures*, volume 2. Addison-Wesley Reading, 1992.