16 de Janeiro de 2022

Universidade Federal do Rio de Janeiro

Construção de Banco de Dados

Avaliação 1

Bernardo Maiorano - 118057795
Bruno Dantas - 118048097
Gabriel Ruas - 119054685
Guilherme Bergman - 118063665
Thiago Guimarães - 118053123

Sumário

1 Introdução

1.1 Dataset

O dataset foi escolhido de forma que fosse um dataset simples, porém ainda assim trouxesse valor ao sistema desenvolvido. Deste modo, foi gerado um dataset inteiramente único, representando um banco de dados de usuários de um sistema de horóscopo.

A ideia é que cada usuário possua um id único e que este mesmo usuário possua seu signo atrelado a fim do sistema utilizar seus dados para mapear seus signos.

2 Dataset

2.1 Descrição dos campos

2.1.1 Id

O Id é o campo que representa o usuário dentro do banco de dados, desta forma, cada usuário único possuirá o seu id, diferenciando-o de outra pessoa.

2.1.2 Nome

Este é o nome cadastrado no sistema pelo usuário, podendo conter caracteres especiais.

2.1.3 Signo

O Signo é o campo que indica o signo do usuário, este foi calculado por meio da data de nascimento do usuário e armazenado no dataset.

3 Estrutura do projeto

3.1 Estrutura do projeto

Para este projeto, foi utilizada a linguagem Python para seu desenvolvimento. Mesmo tendo em mente que tal linguagem não permite um controle de memória tão simples, a escolha foi dada devido à simplicidade ao escrever as rotinas, deste modo sendo possível ganhar tempo em como otimizar cada uma delas e adotar a melhor implementação possível. Além disso, devido à adequação do grupo ao conhecimento de tal tecnologia, todo o processo foi construído adotando o paradigma orientado à objetos, devido à sua importância para abordar problemas atuais.

3.2 Opções adotadas no desenvolvimento

3.2.1 Números de objetos em um bloco

O número setado de objetos em um bloco foi de 10. Julgamos esse valor aceitável para o tamanho dos nossos dados.

3.2.2 Tamanho do bloco de memória

Para esse valor, definimos o tamanho do bloco de memória como um múltiplo do tamanho dos nossos dados multiplicado pelo número de objetos armazenáveis em um bloco. O valor definido então foi de 189, pois o tamanho máximo dos dados que armazenamos é de 63.

3.2.3 Estrutura dos arquivos

Para todas as soluções, possuímos um arquivo de metadados, que inclui informações relevantes sobre a relação em questão, além de dados auxiliares para a manipulação dos dados presentes. Para os arquivos de dados, incluímos um File Header, que possui a descrição da relação em si, possuindo uma descrição das colunas que fazem parte da relação em questão. Após o File Header, os arquivos armazenam os dados crus referentes a devida relação.

3.2.4 Campos utilizados em ordenações

Em um primeiro momento, somente o arquivo ordenado está atrelado à ideia de manter seus registros sempre ordenados. Porém, ainda que as outras

organizações não tenham esta exigência, preferimos escolher um único campo que pudesse ser aplicável às suas peculiaridades - um exemplo é selecionar um valor único para o hash e para identificação dos registros. Por conta desses fatores, selecionamos o campo 'id' como chave principal de ordenação para o arquivo ordenado e para os demais processos que envolvam unicidade e ordenação.

3.2.5 Funções de Hash adotada

Por motivos de simplicidade e encaixe de implementação, utilizamos como função hash dos endereços o resto da chave inteira(id) quando divida pelo número de buckets alocados no momento. Com isso, conseguimos identificar também o bucket de overflow, que se necessário é adicionado ao final do arquivo com os outros buckets (cada um com 10 blocos).

4 Descrição das rotinas

Cada uma das classes foi criada com o intuito de organizar melhor toda a estrutura dos códigos, deste modo é possível tornar mais fácil a leitura do projeto e deixar cada classe desempenhando somente a sua ação, tornando-a coesa.

Considerando que o código foi escrito em uma linguagem de alto nível, temos em mente que a leitura dos trechos de códigos das classes que serão abordadas possuem uma alta legibilidade e consequentemente facilitará o entendimento de cada rotina. Portanto, optamos por explicar somente as classes que encapsulam esses métodos e indicaremos, também o local no repositório onde estarão localizados o código.

4.1 Loader

A classe Loader possui as configurações de blocagem e buckets para o sistema e implementa funções de carregamento de arquivos.

4.2 Block

A classe Block implementa todo o controle interno de um bloco, desde sua atualização até o controle de seus offsets internos.

4.3 Column

A classe Column implementa a estrutura de uma coluna de um banco, expondo inclusive os metadados de uma coluna específica.

4.4 FileOrg

A classe FileOrg implementa o funcionamento básico de todos os tipos de organização de armazenamento. A partir desta classe, todas as organizações de armazenamento se formam e utilizam estas funcionalidades em comum.

4.5 Hash

A classe Hash implementa o funcionamento do sistema de armazenamento baseado em hash, inclusive o algoritmo custom de hash que utilizamos para separar os records entre os buckets.

4.6 Heap

A classe Heap implementa o funcionamento do sistema de armazenamento baseado em heap, implementando principalmente a função de inserção deste tipo de organização.

4.7 Ordered

A classe Ordered implementa a terceira organização primária demandada pelo trabalho, herdando também da classe FileOrg. Inicialmente, temos de relembrar que a inserção dessa estrutura funciona de maneira diferente: utilizamos um arquivo de extensão - basedo na Heap - que serve como uma espécie de interposto antes de inserir no arquivo "original". Igualmente diferente é o processo de deleção, cujo algoritmo está associado à marcação dos dados excluídos com uma flag de indicação. Tanto o processo de inserção como o de deleção estão diretamente ligados à função 'reorganize', tanto que, no código, são responsáveis por notificar a classe que é necessária a realização dessa tarefa ao pôr o atributo 'need_reorganize' como verdadeiro. Bom, vejamos como essa rotina funciona então. Quando a flag supracitada estiver setada, qualquer método chamado vai, por sua vez, ativar a reorganização. Nesse momento, todos os registros guardados no arquivo de extensão serão

unidos àqueles já armazenados no arquivo "original- retirando os que estiverem marcados como deletados - e o conjunto passará pelo algoritmo de merge-sort, sendo o identificador o campo de ordenação escolhido. Após o término do algoritmo, teremos o arquivo original corretamente ordenado e atualizado, e o arquivo de extensão estará novamente livre para inserções inéditas.

4.8 Record

A classe Record implementa a estrutura de um record em si no banco. Funciona como objeto para tratarmos de uma linha específica do banco, independentemente do tipo da organização.

4.9 Relation

A classe Relation faz a conexão entre o Schema e uma FileOrg. Basicamente, é a classe que declara uma relação em si, podendo declarar o schema desta e sua FileOrg.

4.10 Schema

A classe Schema possui as configurações de coluna e nome para uma dada relação. Através dela (utilizando seus metadados), montamos os file headers para cada um dos sistemas de armazenamento.

4.11 VLHeap

A classe VLHeap implementa a solução para o Heap de Tamanho variável. Esta classe permite a utilização do sistema de armazenamento do Heap de Tamanho Variável.

4.12 main

A main foi utilizada para validação de cada uma das organizações primárias sendo utilizado o dataset proposto, no qual, criamos funções referentes à cada uma das organizações para validar todas com o mesmo dataset.

5 Resultados

Após a execução da base de dados escolhida para cada um dos tipos de organizações primárias, foi desenvolvida a seguinte tabela contendo os valores de desempenhos produzidos, como é possível observar abaixo.

Esquema de Armazenamento	Número de Blocos Acessados	Tamanho do Arquivo Gerado
Hash	56 blocos	129,6 KiB
Heap	675 blocos	124,5 KiB
VLHeap	205 blocos	39,2 KiB
Ordered	1 bloco	133,3 KiB

Figura 1: Tabela contendo os resultados do benchmark

6 Conclusão

6.1 Comparação entre os resultados

Para a avaliação dos resultados acima, foram utilizados os métodos de seleção de cada um dos tipos implementados, analisados os números de blocos acessados para obter esses dados e o tamanho dos arquivos gerados.

Com isso, podemos afirmar que o melhor método definido para o tipo de dados que estamos tratando e o dataset utilizado para gerar este benchmark, o que se comportou de forma mais adequada, trazendo um menor número de operações de I/O foi o VLHeap (Heap de tamanho variável). Sendo esta também a que gerou um arquivo de dados e metadados menor.

6.2 Recomendações

Para cada uma das organizações abaixo, fizemos recomendações para quais bases de dados e aplicações devem utilizá-las.

6.3 Heap

6.3.1 Tamanho Variável

No Heap de Tamanho Variável, temos a vantagem de poder armazenar dados com maior flexilidade. Ou seja, não estamos presos a um tamanho específico de coluna. Assim, este tipo de Heap é bom para ser utilizado em dados que podem variar bastante em seu tamanho, como, por exemplo, imagens encodadas ou textos que podem crescer infinitamente.

6.3.2 Tamanho Fixo

No Heap de Tamanho Variável, temos a vantagem de poder armazenar dados com maior previsibilidade. Ou seja, a localização dos dados torna-se algo mais previsível dentro do arquivo de armazenamento. Assim, este tipo de Heap é bom para ser utilizado em dados com tamanho previsível, como, por exemplo, a data de nascimento de um usuário.

6.4 Arquivo Sequêncial Ordenado

Este método não é um tipo recomendado para dados que necessitam de buscas recorrentes, como por exemplo uma base de dados onde a empresa precisa de trazer as informações de usuários que métricas que geram valor de negócio. Estes dados possivelmente teriam um amplo número de inserções e também um amplo número de queries, trazendo uma maior necessidade de reorganização, aumentando o número de operações de I/O.

6.5 Hash Externo Estático

Este tipo é indicado para quando você possui um amplo número de inserções ou número de leituras, pois esses dados estão organizados linearmente para busca, possibilitando encontrar-los de forma mais rápida performando melhor.

7 Repositório

Todo o projeto desenvolvido, arquivos se encontram no seguinte repositório: https://github.com/DantasB/gondola.