Bancos de Dados 2

# ÍNDICES

## A) O que é um Índice em bancos de dados relacionais;

É uma otimização que os SGBDs oferecem, sendo, basicamente, uma estrutura de dados (normalmente b-trees);

Índices tornam as consultas, via de regra, extremamente mais rápidas por conta da forma que ordena/categoriza os dados.

Tem como contrapeso o aumento do uso de espaço de disco, já que os índices, em verdade, são as próprias colunas (indexadas) armazenadas extrinsecamente (isto é, são cópias). Isso se agrava ainda mais caso o programador seja displicente e indexe colunas de forma indevida (há casos inclusive que os índices pesam mais que a própria tabela) ou desnecessária, já que as b-trees se auto-manutem de forma a garantir a própria estrutura, o que implica que qualquer alteração (insert/update/delete) na tabela tem um custo a mais (O(logN)).

## B) Para que servem os Índices em bancos de dados relacionais;

Como supracitado, melhoram drasticamente consultas, tanto específicas

a exemplo de:

```
select * from pessoa
where 'nome' = 'joao da silva';
```

como, também, consultas em largura:

```
select * from pessoa
where 'idade' > 18 and 'idade' < 30;
```

Essencialmente, qualquer query que envolva 'where' terá um tempo de resposta muito menor no caso de colunas (bem) indexadas.

## C) Qual ou quais estrutura de dados (já estudadas por vocês na disciplina de EDA) que os índices mais utiliza para tornar o banco de dados eficiente (ágil)? Terminando explique como está estrutura funcionando, representando-a em uma ou figuras, mas tem a sua explicação dissertativa de como ela funcionaria.

Como já fora aludido, a principal estrutura de dados utilizada é a b+ tree (um tipo de b-tree). Se assemelha bastante com BSTs (binary search tree, árvore de busca binária), com a diferença de que cada nó pode ter mais que dois filhos, e com Tries (árvores de prefixo).

As b-trees são árvores auto-balanceadas que mantêm os dados ordenados e garante acesso, inserções e deleções em O(logN).

Cada nó pode ter T chaves, e cada chave possui 2 endereços, sendo o endereço à esquerda o que indica nós que possuem chaves com valor inferior ao atual, e à direita nós que possuem chaves com valor superior ou igual ao atual. O inteiro T, chamado de ordem/grau, há de ser maior ou igual a 2.

Para garantir a estrutura, há algumas regras/propriedades*:

1. Cada nó contém no máximo T nós filhos
2. Cada nó, exceto a raiz e as folhas, têm pelo menos ⌈T/2⌉ nós filhos
3. O nó raiz tem ao menos dois nós filhos (a menos que ela seja uma folha)
4. Toda folha possui a mesma profundidade, cuja qual é equivalente à altura da árvore
5. Um nó interno com k nós filhos contem k-1 chaves
6. Uma folha contém pelo menos ⌈T/2⌉-1 chaves e no máximo T-1 chaves

**(retiradas da Wikipédia)*

As b+ trees distinguem-se das b-trees de forma a facilitar o acesso sequencial. Para tanto, todas as chaves são mantidas nas folhas; além disso, todas as folhas estão conectadas (através de uma lista duplamente encadeada), de forma que é possível uma folha acessar a folha prima vizinha. Doravante, fica claro o porquê das consultas em largura serem tão eficientes nessa estrutura de dados.

Vale notar que embora a estrutura dos nós internos seja similar aos da b-tree, os nós internos da b+tree possuem simplesmente as chaves, mas não as tuplas -- diferentemente da b-tree, na qual cada chave é também a tupla do dado.

# Aforismos:

## *Por que não BST?*

Como os dados estão salvos no disco, cada vez que uma tupla/nó é acessada/consultada, ela há de ser primeiro carregada na memória -- isto é, então cada vez que você segue um ponteiro (indo para esquerda ou direita), a máquina têm de carregar um novo bloco de memória na memória cache (o que é extremamente mais lento que simplesmente acessar o que já está na cache); como a BST é, verticalmente, maior que a b+tree (e portanto têm muito mais ponteiros), ela é muito mais lenta em tempo de execução (já que a b+tree aloca, de uma vez, várias chaves na memória para consultar.)

## *Quando é recomendado:*

- se a coluna é muito consultada, muito utilizada em joins/group by etc
- se os dados são muito heterogêneos

## *Quando não é recomendado:*

- se a tabela tem muitas inserções/deleções***** e/ou se a coluna é constantemente atualizada
- se espaço em disco for escasso

## *Caso complexo (?):*

- se os dados são homogêneos (muitos dados repetidos)
  No caso de um index referente apenas a uma única coluna com, por exemplo, apenas 2 valores (como sexo [m/f]) resultaria em: aproximadamente metade dos dados à esquerda e o resto à direita; já que o único identificador de cada nó e folha é apenas uma letra, a b+tree não pode otimizar/segmentar mais do que isso, o que, no caso de uma consulta com mais de uma restrição (por exemplo select * from pessoas where 'sexo' = 'm' and 'idade' > 18) resultaria em: o(log n) para achar uma folha com sexo = 'm' e o(n) percorrendo cada uma delas checando se idade > 18, somando, no total, uma complexidade assintótica de o(n logn), que é pior do que se a coluna não fosse indexada e a consulta apenas percorresse todos os dados da tabela checando as restrições (que seria o(n))

Porém, talvez, isso é ser muito teórico e não se ater a realidade. Pragmaticamente, não "demoraria" O(log N) de tempo para achar uma folha 'm', seria basicamente O(1) já que N nesse caso não seria o total de registros e, sim, o total de possibilidades (que é 2)
A partir disso, no pior caso (só há registros de homens ['m'] e todos os registrados têm menos de 18 anos) demoraria-se O(n) (sendo, agora sim, n = total de registros) para achar os registros que atendam as restrições;

Entretanto, num cenário mais aproximado da realidade (metade homem, metade mulher e idades esparsas [1...100]), n dividir-se-ia por 2, demoraria-se O(n/2) (tudo bem que as constantes devem ser ignoradas, mas a análise aqui tem o intuito de ser pragmática, como já fora ressaltado)

Num cenário mais otimista, à depender da ordem dos atributos da tabela, considerando-se que, felizmente, idade seja o segundo critério de ordenação, demoraria-se O(n/2 - m) (sendo m o número de registros com idade inferior à 18 anos)

***** *deleções 'únicas', e não por range, como por exemplo:*

```
delete from pessoas

where 'idade' < 18
```

## **REFERÊNCIAS**:


b+ trees por tutorials point:

https://www.youtube.com/watch?v=9AkRxwT8T1E

b+ trees:

https://en.wikipedia.org/wiki/B%2B_tree

b-trees:

https://pt.wikipedia.org/wiki/%C3%81rvore_B

b+ trees por kerttu pollari-malmi:

https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf

slide washington:

https://courses.cs.washington.edu/courses/cse326/08sp/lectures/11-b-trees.pdf

vantagem de b-trees sobre arvores binárias (bst):

https://stackoverflow.com/questions/15485220/advantage-of-b-trees-over-bsts

livro incentivando/ensinando o uso de indices:

https://use-the-index-luke.com/

respostas explicando b-trees:

i. https://www.quora.com/What-data-structure-is-behind-the-SQL-indexes

ii. https://www.quora.com/How-is-a-B-Tree-exactly-used-by-MySQL-for-indexing

iii. https://www.quora.com/What-is-the-Order-of-a-B-Tree

cheat sheet de complexidades big-O:

http://www.bigocheatsheet.com/

caso das tabelas de index ocuparem mais que as próprias tabelas originais:

https://stackoverflow.com/questions/3727848/sql-server-2005-index-bigger-than-data-stored

indices:

https://en.wikipedia.org/wiki/Database_index

slide utexas:

i. http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture16.html

ii. http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture17.html