

**UNIVERSIDADE FEDERAL DE SANTA MARIA
CAMPUS FREDERICO WESTPHALEN
DEPARTAMENTO DE TECNOLOGIA DA INFORMAÇÃO
CURSO DE SISTEMAS DE INFORMAÇÃO**

Maurício Witter

**PESQUISA E ORDENAÇÃO DE DADOS APLICADO A BANCOS DE
DADOS**

Frederico Westphalen, RS
2020

Maurício Witter

PESQUISA E ORDENAÇÃO DE DADOS APLICADO A BANCOS DE DADOS

Trabalho apresentado na disciplina de
Pesquisa e Ordenação de dados,
Curso de Sistemas de Informação,
Universidade Federal de Santa Maria
(UFSM-FW, RS).

Prof.^a Dr.^a Joel da Silva

Frederico Westphalen, RS

2020

SUMÁRIO

1 INTRODUÇÃO	3
2 CLASSIFICAÇÃO INTERNA E EXTERNA.....	4
2.1 CLASSIFICAÇÃO POR DISTRIBUIÇÃO.....	4
3 ORDENAÇÃO EM BANCOS DE DADOS – EXEMPLOS	4
3.1 APLICABILIDADE	5
3.1.1 Order By	5
3.1.2 Hash.....	6
3.1.3 Árvores B	9
4 CONCLUSÃO	9

1 INTRODUÇÃO

Os algoritmos de pesquisa e ordenação de dados são essenciais desde que os computadores surgiram, isso porque, tanta a memória estática ou dinâmica era muito limitada. Nesse contexto, a ordenação de dados era essencial para diminuir os custos de pesquisa.

Muitos estudos e algoritmos foram propostos para resolver estes problemas, alguns deles são: Bubble Sort, Selection Sort, Merge Sort, Insertion Sort, Quicksort, Heap Sort, Bucket Sort, Counting Sort, Cycle Sort, Gnome Sort, Cocktail Sort, Odd-even Sort, Radix Sort (classificação por distribuição), Shell Sort entre outros. Cada algoritmo tem uma finalidade, o qual foi proposto e avaliado para um cenário específico para obter o máximo de desempenho.

O desempenho, por conseguinte, é avaliado pelo Big O (Order Of Magnitude), a qual avalia a taxa de crescimento ou complexidade linear, isto é, o limite máximo de operações realizadas por um algoritmo. Assim, o Big O descreve o pior cenário possível e, logo, pode ser usado para obter o tempo de execução necessário e o espaço de memória usado.

Na ciência da computação, ainda, há as notações formais de Little-O, Theta e Omega. O Little-O, por sua vez, conta com uma diferença do Big O — que é o limite inclusivo (\leq), O Little-O é um limite estrito, ou seja ($<$), em resumo, O Big O mantém um limite mínimo além de n , já o Little-o tem o seu limite mínimo estrito, mas ambos descrevem os limites superiores. Ademais, o Theta — $\Theta(n)$ significa limite estreito, ou seja, o Big Θ demonstra que o algoritmo em questão será executado em menos etapas do que na expressão fornecida, por exemplo: (n^2) . Já o Omega (Ω), representa a menor complexidade de tempo, isto é, o melhor cenário possível.

Assim, com esta arquitetura os algoritmos de pesquisa e ordenação podem ser avaliados para o melhor cenário possível para cada aplicação, maximizando o desempenho e processamento de dados, hoje, memória não é mais um gargalo, mas o desempenho de pesquisa é algo crucial no desenvolvimento de softwares por isso, realizar operações rápidas, otimizadas e com máxima eficiência entre a interface e o database é algo primordial.

2 CLASSIFICAÇÃO INTERNA E EXTERNA

A classificação interna diz respeito à memória em tempo de execução, ou seja, memória RAM (Random-access memory), esta memória é utilizada quando a classificação é feita durante a execução e não necessita de muita memória para ser salva em unidades externas.

A classificação externa, por sua vez, é a memória salva em discos e carregada na memória RAM em blocos. Nesse contexto, os dados não podem ser acessados de forma aleatória no disco por ter um custo muito alto de execução, necessitando de algoritmos de pesquisa e ordenação para ordenar ao salvar e buscar esses dados de forma eficiente.

2.1 CLASSIFICAÇÃO POR DISTRIBUIÇÃO

Esse tipo de ordenação faz uma classificação por chave, dessa maneira não cria uma árvore de decisão. Além disso, eles podem classificar em tempo $O(n)$, enquanto algoritmos de classificação por comparação como Quicksort, Bubble Sort e Merge Sort conseguem manter a complexidade de tempo e espaço em $O(n \log n)$. Alguns algoritmos são: Bucket Sort, Counting Sort e Radix Sort.

3 ORDENAÇÃO EM BANCOS DE DADOS — EXEMPLOS

Os bancos de dados precisam ser robustos, eficazes e simples de manter, naturalmente esses sistemas empregam a classificação de dados para muitas finalidades, principalmente, operações de ordenação e operações de consulta, essas devem ser eficientes, logo, precisa-se de algoritmos de baixa complexidade para realizar algumas operações como verificação de valores duplicados, merge, join, intersecção, união e diferença.

Pode-se implementar vários algoritmos de pesquisa e ordenação sob bancos de dados de modo a otimizar as operações. Em resumo, algumas delas são: filas prioritárias e binary heaps, quicksort, merge sort, shell sort, radix sort.

De acordo com Graefe G., 2006, as filas prioritárias e binary heaps são estruturas de dados boas para o reaproveitamento de código, isso porque a fila prioritária pode ser usada para diversas funções como, por exemplo, run generation, merging cache-sized rodando em memória, merging disk-based, prever a E/S (Entrada e Saída) de leitura antecipada mais eficaz, implementar patterns de merge sort, concatenação virtual entre outras. Além disso, para buscas baseadas em disco existem as estruturas de Árvore-B com nós semelhantes às estruturas utilizadas para cache. Graefe G., 2006, complementa que os bancos de dados, tradicionalmente, utilizam algoritmos como, Quicksort, Shell Sort, Merge Sort e Radix Sort por ter melhor custo e menor complexidade de tempo e espaço. Dessa forma, a classificação baseada em comparação se sobressai sobre a classificação por distribuição.

3.1 APLICABILIDADE

3.1.1 Order By

Algoritmos de ordenação são totalmente relevantes na utilização de bancos de dados. Os RDBMS (Relational Database Management System), utilizando a linguagem (SQL) já implementam algoritmos como Quicksort e Merge Sort para ordenar o input (entrada) de dados e realizar consultas de dados ordenados.

Figura 1: consulta utilizando order by.



Fonte: (guru99).

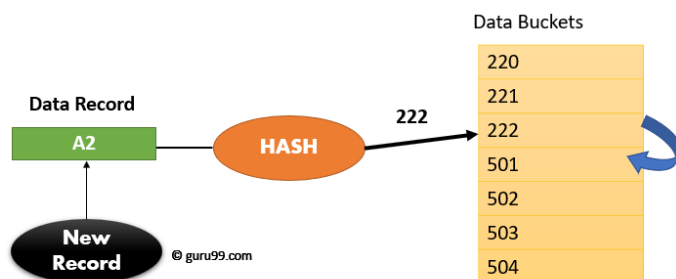
Na figura 1, a consulta utiliza um algoritmo order by e realiza uma operação de busca de forma decrescente, empregando algoritmos de comparação como o Quicksort.

3.1.2 Hash

Outra função que RDBMS utilizam são estruturas de dados do tipo tabela de hash, que são um tipo de identificador, mais precisamente, distribuem pares de chave/valor uniformemente através de um array, nos bancos de dados a função hash mapeia o conjunto de chaves de pesquisa para o endereço real dos buckets que armazenam os registros. Ao utilizar hash, a pesquisa de dados em disco é feita sem utilizar estruturas de índices, isso porque, os dados são armazenados em forma de blocos de dados e gera uma hash em memória local onde esses registros são armazenados em buckets (unidades de armazenamento). Hash são maneiras de indexar e recuperar itens em bancos de dados mais rapidamente, além disso, contam com outros benefícios.

As hash podem ser do tipo estático ou do tipo dinâmico. As funções de hashing estáticas permitem a inserção de novos registro na tabela e gerar um novo endereço para esse registro utilizando a hash key, pesquisar e recuperar dados em buckets com as funções de hash e deletar um registro, por exemplo.

Figura 2: inserção de um novo registro com hash estática



Fonte: (guru99).

Nas funções de hashes estáticas, os dados são armazenados em buckets estáticos com memória definida, caso um bucket esteja ocupado o novo registro é

colocado na próxima função de hash, como ilustra a figura 2. Já nas funções de hash dinâmica, os dados são armazenados também em buckets, porém, os buckets por serem dinâmicos podem diminuir ou aumentar de acordo com a quantidade de dados sob demanda e são armazenados de forma ordenada afim de aumentar o desempenho ao fazer operações de insert, delete e update, isso, no entanto, é aconselhável para bancos de dados que não a muitos dados armazenados, pois, torna a manutenção complexa e de alto custo de memória. (guru99.com).

Pode-se acessar rapidamente um elemento pegando o índice a partir da chave de identificação, a função de hash faz um cálculo e posteriormente tira o módulo da hash pelo tamanho do array como na figura 3, uma operação de pesquisa utilizando hash tem complexidade de $O(1)$.

Figura 3: encontrando o index a partir de uma hash

```
hash = hashfunc(key)
index = hash % array_size
```

Fonte: (hackerearth).

Supondo uma contagem de todos os caracteres de uma string $S = \text{"abcde...z"}$ iterar sobre isso teria complexidade $O(26*n)$, onde "n" é a sequência de caracteres como exemplifica a figura 4.

Figura 4: iterar uma sequência de caracteres



```

void countFre(string S)
{
    for(char c = 'a'; c <= 'z'; ++c)
    {
        int frequency = 0;
        for(int i = 0; i < S.length(); ++i)
            if(S[i] == c)
                frequency++;
        cout << c << ' ' << frequency << endl;
    }
}

```

Fonte: (hackerearth).

Aplicando hashing nesse exemplo, temos um array para os 26 caracteres que armazena chave e valor, em seguida a cada iteração, a verificação de igualdade não é mais necessária, a frequência é aumentada reduzindo a complexidade de ordenação para $O(n)$.

Figura 5: iterando sobre hash



```

int Frequency[26];

int hashFunc(char c) {
    return (c - 'a');
}

void countFre(string S) {
    for(int i = 0; i < S.length(); ++i) {
        int index = hashFunc(S[i]);
        Frequency[index]++;
    }
    for(int i = 0; i < 26; ++i) {
        cout << (char)(i + 'a') << ' ' << Frequency[i] << endl;
    }
}

```

```

let Frequency = new Array(
    'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n',
    'o', 'p', 'q', 'r', 's', 't', 'u',
    'v', 'x', 'w', 'y', 'z'
);

function hashFunc(c) {
    return (c - 'a');
}

function countFre(S) {
    for(let i = 0; i < S.length; ++i) {
        let index = hashFunc(S[i]);
        Frequency[index]++;
    }
    for (let i = 0; i < 26; ++i) {
        console.log((i + 'a'), Frequency[i]);
    }
}

```

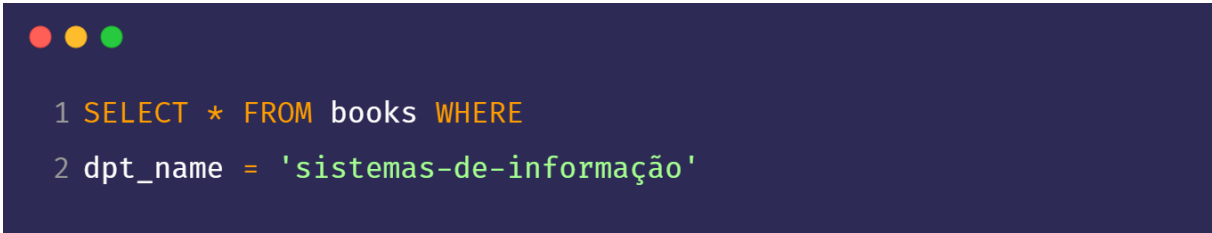
Fonte: (hackerearth).

De fato hash pode ser muito útil em diversas ocasiões como busca, cache e indexação em bancos de dados, por ser uma busca rápida. Ainda, há vários temas a serem explorados como colisões, tabelas de hashing, linear probing, quadratic probing e double hash.

3.1.3 Árvores B

Outra estrutura de dados de pesquisa são Árvores B. Elas são extremamente eficientes para realizar busca em bancos de dados bem como fazer indexação. Por exemplo, vamos supor uma biblioteca tem as seguintes tabelas: relações-públicas, agronomia, jornalismo e sistemas-de-informação. Os departamentos são separados, respectivamente, por: A-E, F-O, P-U e V-Z. Então para encontrar os livros de sistemas de informação podemos realizar a seguinte consulta — figura 6.

Figura 6: consulta de livros



```
1 SELECT * FROM books WHERE
2 dpt_name = 'sistemas-de-informação'
```

Fonte: (whatisdbms).

Primeiramente, a raiz da Árvore-B é verificada, com os intervalos de departamentos definidos abaixo da raiz, os nodes são separados por blocos que, exceto a última camada, são denominados ramos, os últimos blocos são denominados de folhas. As folhas contêm o nome da tabela, que no caso da consulta seria "sistemas-de-informação". Trazendo, dessa forma os dados da tabela em menos tempo.

As Árvores B são ótimas candidatas para trabalhar com grandes blocos de dados, são otimizadas para leitura e escrita, e são indicadas para manipular em conjunto com a memória externa por isso, Graefe, 2006, indica o seu uso para lidar com bancos de dados e sistemas de arquivos.

4 CONCLUSÃO

As pessoas sempre tentam otimizar o máximo que podem determinadas tarefas, a ideia de algoritmos surgiu com Alan Turing em 1936 com a (Máquina de Turing) e Alonzo Church. Entretanto, otimizar e tornar mais rápido as tarefas diárias são algo intrínseco de todo mundo.

O campo da Ciência da Computação e algoritmos ganharam força com os computadores modernos de transistores a partir de 1936. Como o armazenamento era curto e caro, a pesquisa e desenvolvimento de algoritmos de ordenação e pesquisa trouxe muito avanço a área, por tornar todos os processos mais rápidos.

Hoje, o armazenamento e processamento não é mais um problema gigantesco para muitos, no entanto, os algoritmos de pesquisa e ordenação ainda fazem muito sentido seja para ordenar e buscar imensos terabytes até petabytes de dados ou otimizar consultas em bancos de dados para tornar a experiência de usuário mais positiva em questões de velocidade. Ainda pode ser aplicado em transferências de dados em servidores de cloud computing (computação em nuvem) e docker, novos campos de devops (development e operações) para aprimorar serviços de busca, ou simplesmente em e-commerce ou qualquer lugar que tenha uma base de dados.

REFERÊNCIAS

GRAEFE G. **Implementing Sorting in Database Systems**. ACM Computing Surveys. 2006. Vol. 38, No. 3. Disponível em: < http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/DBSREAL/SS2005/Vorlesungsunterlagen/Implementing_Sorting.pdf>. Acesso em: 13 mar. 2020.

GURU99. **Hashing in DBMS: Static & Dynamic with Examples**. Disponível em: <<https://www.guru99.com/hashing-in-dbms.html>>. Acesso em 15 mar. 2020.

HACKEREARTH. **Basics of Hash Tables**. Disponível em: <<https://www.hackerearth.com/pt-br/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/>>. Acesso em 17 mar. 2020.

MYSQL. **8.2.1.16 ORDER BY Optimization**. Disponível em: <<https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html>>. Acesso em: 14. Mar 2020.

THAKUR. **B-TREE Indexing in DBMS: Why we use B-Tree**. Disponível em: <<https://www.hackerearth.com/pt-br/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/>>. Acesso em 17 mar. 2020.