

Banco de Dados 2

012 – Vacuum, Organização de Arquivos e Table Spaces.

PostgreSQL VACUUM: Limpando o Banco de dados

- É muito importante, principalmente para um **DBA (Administrador de Banco de Dados)**, conhecer e analisar as **ferramentas que possam aumentar a performance do banco de dados**.
- Isso porque geralmente **empresas que possuem DBA trabalham com uma grande quantidade de dados e a velocidade de resposta da aplicação torna-se crítica para seu negócio**.
- Imagine, por **exemplo**, uma **Operadora de Cartão de Crédito**, como **Visa** ou **Mastercard**: **quantos milhões de transações por segundo são realizadas em todo mundo?**
- É fato que um **index errado** em um **banco de dados** dessa **operadora** pode ser fatal para **perda de milhões de reais em poucos segundos**.

Vacuum

- É importante **conhecer o funcionamento interno do PostgreSQL.**
- Você já percebeu que ao **deletar registros do seu banco de dados ele não diminui ?**
- O **PostgreSQL**, assim como **outros bancos**, na verdade **não deletam os registros e sim os marca como inúteis**.
- Ou seja, se você fizer um “**DELETE FROM funcionario WHERE id > 100**” e este comando **apagar 10 mil linhas**, você na verdade **estará marcando as 10 mil linhas como inúteis e não deletando fisicamente**, o que demandaria muito mais tempo e recurso.
- A lógica é a seguinte:**é muito melhor ter uma operação rápida do que espaço em disco, sendo assim o banco ficará enorme mas sua performance compensará tal perda de espaço.**

Vacuum

- Esse mecanismo é chamado de **MVCC (Multiversion Concurrency Control)** que garante **uma performance melhor ao banco de dados**, afinal performance é o ponto chave em **aplicações críticas**.
- Quando você realiza um **UPDATE**, o seu velho registro não é atualizado. Na verdade é feita uma inserção de outra tupla na sua tabela, com os mesmos dados da tupla original, apenas alterando o que você solicitou no **UPDATE**, e a tupla anterior (não atualizada) é marcada como inútil.
- A utilização do **VACUUM**: Já que temos muitos **registros que estão marcados como inúteis**, precisamos em algum momento **limpar** estes, **para garantir ainda mais performance** em nosso banco e retirar toda sujeira de dados.
- No momento em que o comando **VACUUM** é executado, **é feita uma varredura em todo o banco a procura de registros inúteis, onde estes são fisicamente removidos, agora sim diminuindo o tamanho físico do banco**.
- Mas **além de apenas remover os registros**, o comando **VACUUM** encarrega-se de **organizar os registros que não foram deletados**, garantindo que **não fiquem espaços/lacunas em branco após a remoção** dos registros inúteis.

Vacuum

- Temos na imagem ao lado (**coluna 1**) a lista de todos os registros, incluindo os úteis e inúteis (marcados em vermelho).
- O **vacuum** apaga todos os registros inúteis (em vermelho), mas após essas deleções serem realizadas, você pode perceber que **ficam espaços em branco**, exatamente o espaço onde estavam os registros inúteis (**Coluna 2**).
- Por fim, o **vacuum encarrega-se de remover esses espaços**, garantindo que os mesmos fiquem organizados e em uma disposição correta.
- Não está descrito na figura ao lado, mas o **vacuum** também **atualiza as estatísticas que são utilizadas pelo otimizador do PostgreSQL** para determinar qual a melhor forma de realizar uma **busca no banco de dados**.
- Porém, a atualização dessas estatísticas vai depender da forma em que o **vacuum** for executado.



Vacuum

```
1 | VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ tabela ]
2 | VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ]
3 | ANALYZE [ tabela [ (coluna [, ...] ) ]]
```

Listagem Sintaxe do Vacuum

- **FULL :** Quando o vacuum é utilizado em conjunto com este parâmetro, então **é feita uma limpeza completa de todo o banco, em todas as tabelas e colunas.** Este **processo geralmente é demorado e evita que qualquer outra operação no banco seja realizada**, ou seja, ao realizar um **VACUUM FULL** você terá que esperar todo processo terminar até realizar um comando DLL ou DML.
- **VERBOSE:** produzirá um **relatório detalhado de tudo que está sendo feito** no comando **VACUUM**.
- **ANALYSE:** Este parâmetro é responsável por habilitar ou desabilitar a atualização das estatísticas que são utilizadas pelo otimizador do PostgreSQL para determinar o melhor método de consulta. Ao usar o **ANALYSE** junto ao seu comando **VACUUM** ele **irá atualizar as estatísticas do banco de dados** a fim de melhorar a performance das pesquisas.

Vaccum

- **tabela:** Caso você queira realizar o **VACUUM apenas em uma tabela**, então você deve especificar explicitamente qual tabela será, caso contrário, apenas deixe este parâmetro em branco e todas as tabelas serão consideradas.
- **coluna:** Seguindo o mesmo raciocínio da tabela, caso você deseje realizar o **VACUUM em apenas algumas colunas**, basta especificar quais são, caso contrário, deixe este parâmetro em branco e todas as colunas serão consideradas.

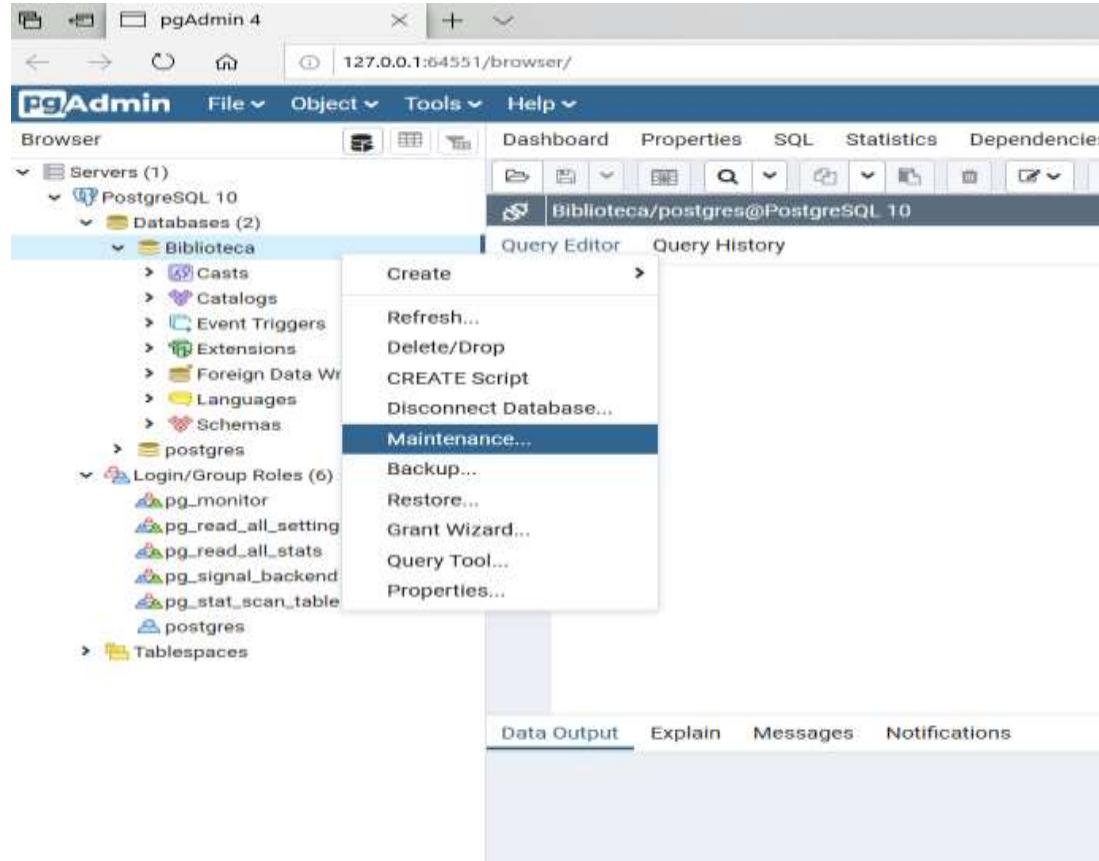
Vacuum

- A diferença entre o **VACUUM sem parâmetros** (simples) e o **VACUUM FULL** (que exige o bloqueio exclusivo das tabelas, ou seja, nenhuma operação pode ser realizada enquanto este comando estiver em processamento).
- O **VACUUM simples** apenas remove as tuplas marcadas como **inúteis/removidas em processos de UPDATE ou DELETE**. Sendo assim, não há necessidade de bloquear as operações no banco.
- O **VACUUM FULL** além de remover essas tuplas inúteis, ainda **reorganiza as tabelas, retirando os espaços em branco que ficaram após a remoção dessas colunas**, e para tal processo é necessário que o banco não esteja realizando nenhuma operação, por isso o bloqueio do mesmo é necessário.

Vacuum

- Caso você utilize o **pgAdmin** como uma **ferramenta** para realizar o **gerenciamento do seu PostgreSQL**, então nele mesmo (**sem linha de comando**) você poderá realizar o **VACUUM**, assim como outros processos **otimizadores de performance**.
- O processo é simples: basta você **clicar com o botão direito em cima da sua base de dados** e depois escolher a opção “**Maintenance...**” (**Manutenção**).

Vacuum



Vaccum

Biblioteca/postgres@PostgreSQL 10

Query Editor Query History

Maintenance....

1

Options

Maintenance operation **VACUUM** ANALYZE REINDEX CLUSTER

Vacuum

FULL No FREEZE No ANALYZE No

Verbose Messages Yes

i **?**

x Cancel **✓ OK**

Data Output Explain Messages Notifications

Vacuum

Maintenance...

Options

Maintenance operation

VACUUM ANALYZE REINDEX CLUSTER

Vacuum

FULL Yes FREEZE No ANALYZE Yes

Verbose Messages Yes

i ?

✖ Cancel ✓ OK

Vacuum

Maintenance

Maintenance (Vacuum)

Thu Jun 23 2022 16:32:39 GMT-0300 (Hora oficial do Brasil)

⌚ 7.86 seconds

[More details...](#) [Stop Process](#)

i Running...

^K ☰ 16:32
POR PTB2 23/06/2022

Maintenance

Maintenance (Vacuum)

Thu Jun 23 2022 16:32:39 GMT-0300 (Hora oficial do Brasil)

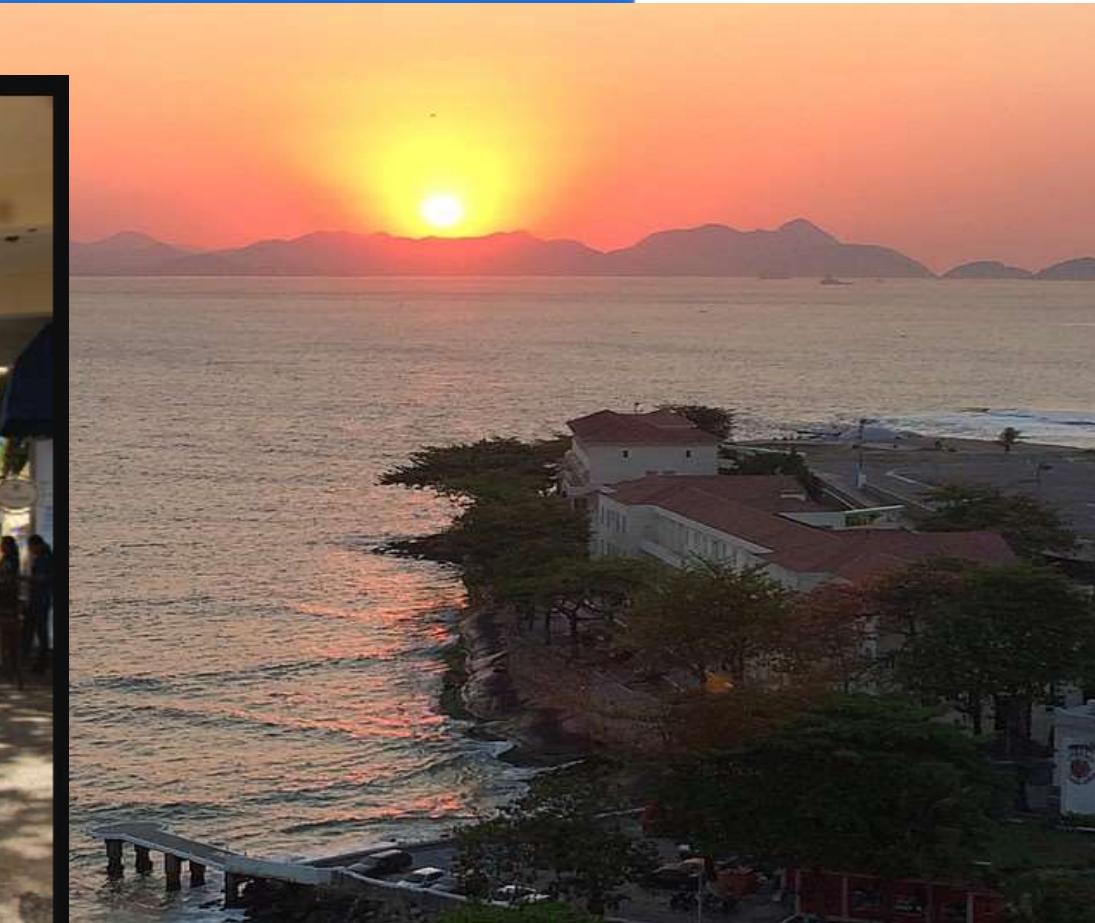
⌚ 36.7 seconds

[More details...](#) [Stop Process](#)

✓ Successfully completed.

^K ☰ 16:34
POR PTB2 23/06/2022

Organização de Arquivos



Organização de Arquivos

- A **organização de arquivos em bancos de dados** é um tema central de **SGBDs** porque afeta diretamente o **desempenho** de consultas, inserções e atualizações.

Organização de Arquivos

- **Constituição física: colunas, registros e blocos**
- Um banco de dados relacional é armazenado em arquivos no disco que seguem uma hierarquia de organização:
- **Colunas (campos / atributos):**
 - São as menores unidades lógicas, armazenam valores individuais (ex: nome, idade, salário).
 - Cada coluna tem um tipo de dado (inteiro, texto, data, etc.), que determina o espaço ocupado.
- **Registros (tuplas / linhas):**
 - Um registro é a linha da tabela: uma coleção de colunas que formam uma instância.
 - Exemplo: **(123, "Maria", 25, "Colatina", 3500.00)**.

Organização de Arquivos

- **Blocos ou Páginas de disco (pages)**
 - O sistema de banco de dados não lê dados célula por célula, mas em páginas fixas de disco (tipicamente 4 KB, 8 KB ou 16 KB dependendo do SGBD).
 - Dentro de uma página, ficam vários registros.

Organização de Arquivos

- Assim, a **hierarquia** é:
- **Disco → Arquivos → Blocos (páginas) → Registros (linhas) → Colunas (campos)**

Organização de Arquivos

Formas de Organização de Arquivos:

A forma como os **registros são dispostos dentro dos arquivos/páginas** impacta a **eficiência** de acesso:

a) Heap File (**organização por heap**):

Os registros são armazenados **na ordem em que chegam**, sem critério específico.

Vantagens:

- Inserções rápidas (só coloca no próximo espaço livre).

- Boa para tabelas de staging ou log.

Desvantagens:

- Pesquisas exigem **varredura completa** (full scan).

- Ordenação e busca por chave são lentas.

Usado quando não há chave primária ou quando a ordem não importa.

Organização de Arquivos

- Uma **varredura completa** (em inglês **full table scan** ou **sequential scan**) acontece quando o **SGBD** precisa **percorrer todos os blocos/páginas de uma tabela para encontrar os registros que atendem a uma consulta.**
- Ou seja, o banco lê linha por linha (**registro por registro**), verificando se cada uma satisfaçõa a condição do **WHERE**.

Organização de Arquivos

Suponha a tabela `CLIENTE`:

ID	Nome	Idade
1	Ana	25
2	Bruno	40
3	Carla	30
...

```
SELECT *  
FROM CLIENTE  
WHERE Idade = 30;
```

O **SGBD** faz um **full scan**: lê todos os registros (Ana, Bruno, Carla, ...), compara um por um até achar quem tem **Idade = 30**.

Organização de Arquivos

- **b) Arquivo Sequencial (Sorted File Organization):**
- Os registros são armazenados em ordem crescente de uma chave de busca (ex: ID, CPF).
- **Vantagens:**
- Busca por chave e intervalos são eficientes (pode usar busca binária).
- Ótimo para consultas por faixa (idade BETWEEN 20 AND 30).
- **Desvantagens:**
- Inserções e exclusões são custosas, porque podem exigir reordenação ou reorganização do arquivo.
- Muito usado em SGBDs analíticos (OLAP), ou quando os dados mudam pouco.

Organização de Arquivos

Salary	Ptr
10000	1
12000	2
12500	3
13250	4
17000	5
25000	6

Salary
15200

Salary	Ptr
10000	1
12000	2
12500	3
13250	4
17000	6
25000	7
15200	5

(a) Initial status of table

(b) Record to be inserted

(c) After insertion of new record

Sequential (sorted) file organization - Record insertion

Como inserimos um registro?

Insira o registro no final do arquivo.

Localize o registro anterior no valor da chave de classificação e redefina o ponteiro desse registro para apontar para o novo registro e insira o ponteiro do registro anterior no campo de ponteiro do novo registro.

Todos os ponteiros dos outros registros também devem ser alterados.

Reorganize todos os registros periodicamente para organizá-los na ordem de classificação do atributo de ordenação.

Organização de Arquivos

Como excluir um registro?

Arquivo Sequencial (Sorted File Organization):

- Localize o registro que deve ser excluído.
- Substitua o valor do ponteiro do registro anterior pelo ponteiro do registro a ser excluído.
- Atualize todos os ponteiros dos outros registros (se necessário).

Organização de Arquivos

Arquivo Sequencial (Sorted File Organization):

Vantagens:

- A recuperação de registros se torna eficiente se a consulta usar o atributo de classificação como chave de pesquisa.
- A classificação de registros no campo de ordenação é rápida. [Não é necessária classificação externa]

Desvantagens:

- A inserção e exclusão de registros são caras.
- Atualizar os valores dos atributos de classificação dos registros também é caro.
- A recuperação de registros em atributos não ordenados não é fácil.

Organização de Arquivos

- **c) Arquivo com Hashing (Hashed File Organization):**
- Os registros são armazenados em páginas determinadas por uma função de hash aplicada à chave de busca.
- Exemplo: **página = hash(ID) mod N.**
- **Vantagens:**
- Recuperação de registros por igualdade é muito rápida (WHERE CPF = '123').
- **Desvantagens:**
- Intervalos não são eficientes (WHERE idade BETWEEN 20 AND 30).
- Colisões de hash precisam ser resolvidas (encadeamento, overflow).
- **Usado em tabelas de alta taxa de consulta por igualdade exata.**

Organização de Arquivos

Organização de arquivos hash

- **Bucket de dados** - Buckets de dados são os locais de memória onde os registros são armazenados. Esses buckets também são considerados Unidades de Armazenamento.
- **Função Hash** - A função hash é uma função de mapeamento que mapeia todos os conjuntos de chaves de busca para o endereço real do registro. Geralmente, a função hash usa a chave primária para gerar o índice hash – o endereço do bloco de dados. A função hash pode ser uma função matemática simples ou qualquer função matemática complexa.
- **Índice de Hash** - O prefixo de um valor de hash inteiro é considerado um índice de hash. Cada índice de hash tem um valor de profundidade para indicar quantos bits são usados para calcular uma função de hash. Esses bits podem endereçar 2^n buckets. Quando todos esses bits são consumidos, o valor de profundidade é aumentado linearmente e o dobro dos buckets é alocado.

Organização de Arquivos

Hash estático

No hash estático, quando um valor de chave de busca é fornecido, a função de hash sempre calcula o mesmo endereço. Por exemplo, se quisermos gerar um endereço para STUDENT_ID = 104 usando uma função de hash mod(5), o resultado sempre será o mesmo endereço de bucket 4. Não haverá nenhuma alteração no endereço do bucket aqui. Portanto, o número de buckets de dados na memória para este hash estático permanece constante.

Operações:

- **Inserção** - Quando um novo registro é inserido na tabela, a função hash h gera um endereço de bucket para o novo registro com base em sua chave hash K . Endereço de bucket = $h(K)$
- **Pesquisa** - Quando um registro precisa ser pesquisado, a mesma função de hash é usada para recuperar o endereço do bucket para o registro. Por exemplo, se quisermos recuperar o registro completo para o ID 104, e se a função de hash for mod(5) nesse ID, o endereço do bucket gerado será 4. Então, iremos diretamente ao endereço 4 e recuperaremos o registro completo para o ID 104. Aqui, o ID atua como uma chave de hash.
- **Exclusão** - Se quisermos excluir um registro, usando a função hash, primeiro buscaremos o registro que deve ser excluído. Em seguida, removeremos os registros daquele endereço na memória.
- **Atualização** - O registro de dados que precisa ser atualizado é primeiro pesquisado usando a função hash e, em seguida, o registro de dados é atualizado.

Organização de Arquivos

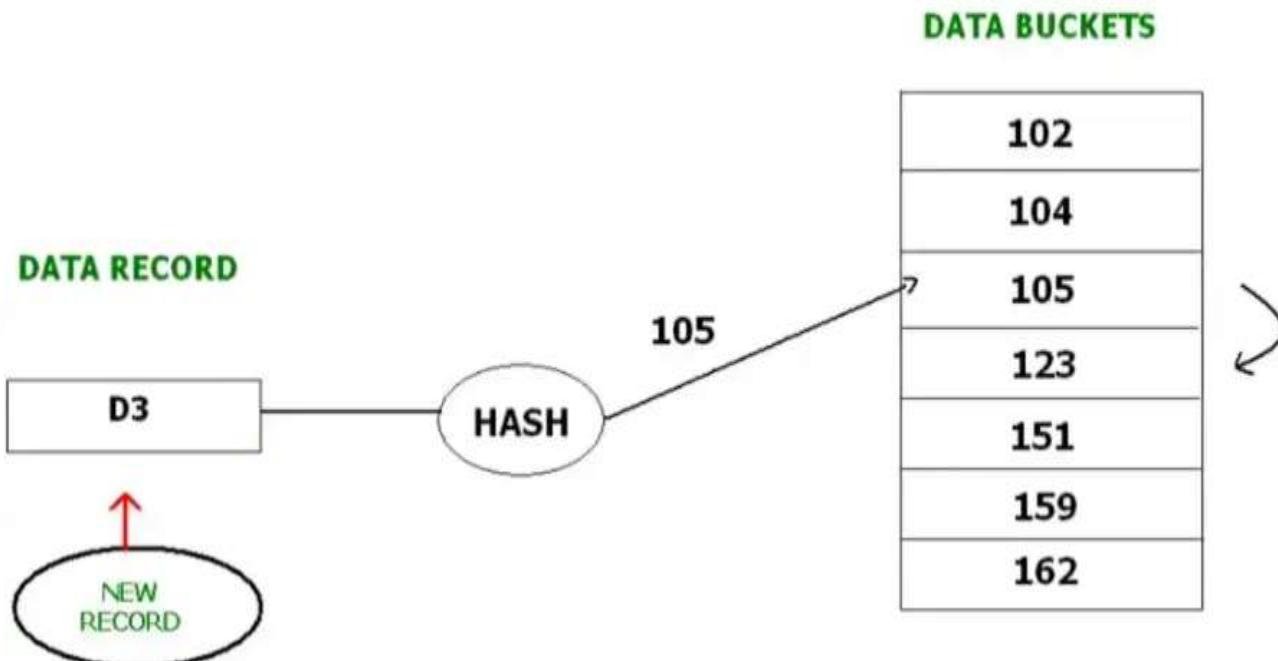
Agora, se quisermos inserir novos registros no arquivo, mas o **endereço do bucket de dados** gerado pela **função hash** não estiver vazio ou os dados já existirem nesse endereço, isso se torna uma situação crítica para lidar. Essa situação, no caso do **hash estático**, é chamada de **estouro de bucket**. Como inseriremos os dados nesse caso?

Existem vários **métodos** para superar essa situação.

Alguns métodos comumente usados são discutidos abaixo:

Hashing Aberto - O próximo bloco de dados disponível é usado para inserir o novo registro, em vez de sobrescrever o antigo. Este método também é chamado de **sondagem linear**. Por exemplo, D3 é um **novo registro** que precisa ser **inserido**; a função de hash gera o **endereço 105**. **Mas ele já está cheio**. Assim, o sistema busca o **próximo bloco de dados disponível**, 123, e atribui D3 a ele.

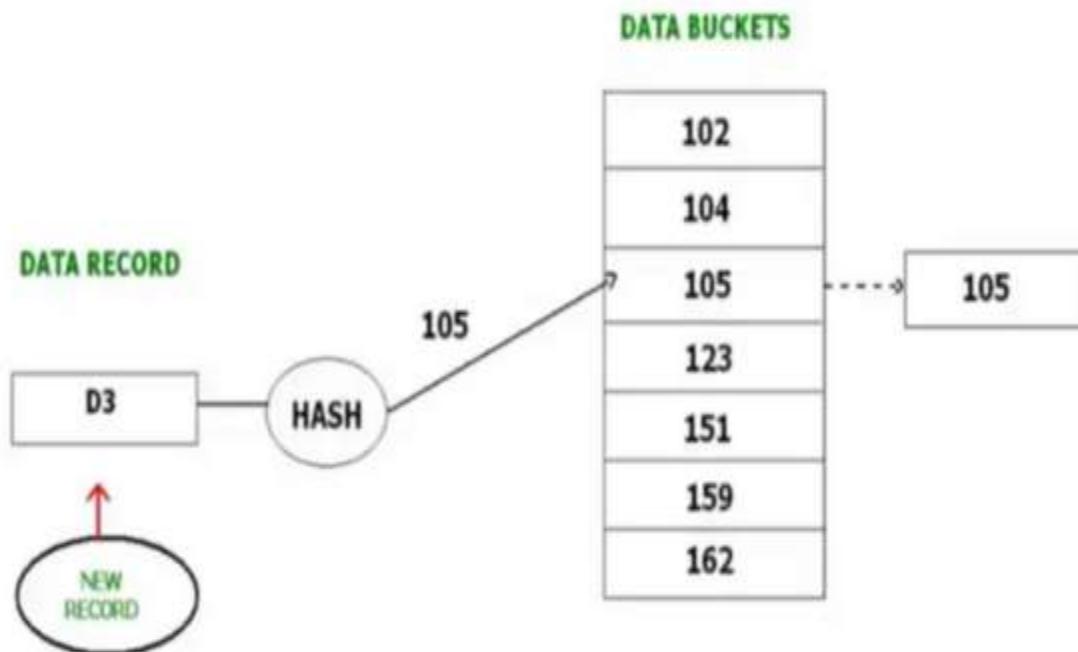
Organização de Arquivos



Organização de Arquivos

- Hash fechado - Um novo bucket de dados é alocado com o mesmo endereço e vinculado a ele após o bucket de dados estar cheio. Esse método também é conhecido como **encadeamento de estouro**. Por exemplo, precisamos inserir um **novo registro D3** na tabela. A **função de hash estática** gera o endereço do **bucket de dados como 105**. Mas esse **bucket** está **cheio** para armazenar os novos dados. Nesse caso, **um novo bucket de dados é adicionado ao final do bucket de dados 105 e vinculado a ele**. O novo registro **D3** é inserido no novo bucket.

Organização de Arquivos



Organização de Arquivos

- **(1) Hash Estático:** quando um valor de chave de busca é fornecido, a função de hash sempre calcula o mesmo endereço.
 - **(1.1) - Hashing Aberto:** a colisão de dados leva o sistema busca o próximo bloco de dados disponível.
 - **(1.2) – Hashing Fechado:** a colisão de dados gera um novo bucket de dados que é adicionado ao final do bucket de dados ocupado e vinculado a ele.

Organização de Arquivos

Hashing dinâmico

- A desvantagem do hash estático é que ele não expande ou encolhe dinamicamente conforme o tamanho do banco de dados cresce ou diminui. No hash dinâmico, os buckets de dados crescem ou encolhem (adicionados ou removidos dinamicamente) conforme os registros aumentam ou diminuem.

Organização de Arquivos

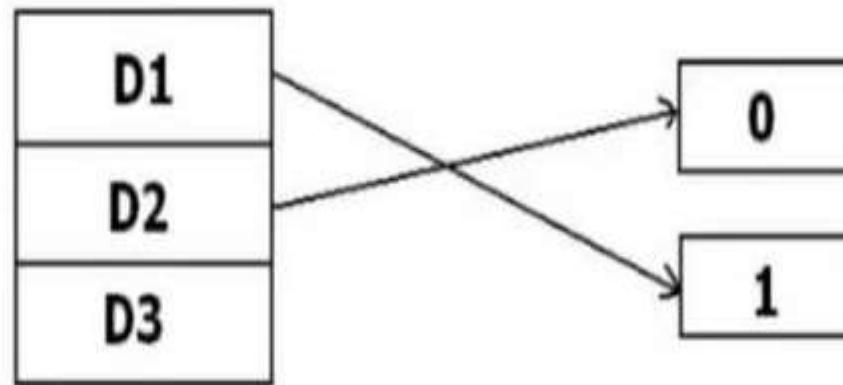
- O **hash dinâmico** também é conhecido como **hash estendido**.
- No **hash dinâmico**, a função hash é feita para produzir um grande número de valores. Por exemplo, há três registros de dados D1, D2 e D3. A função hash gera três endereços 1001, 0101 e 1010, respectivamente.
- Este método de armazenamento considera apenas parte deste endereço – especialmente apenas o primeiro bit para armazenar os dados. Então, ele tenta carregar três deles nos endereços 0 e 1.

Organização de Arquivos

$h(D1) \rightarrow 1001$

$h(D2) \rightarrow 0101$

$h(D3) \rightarrow 1010$

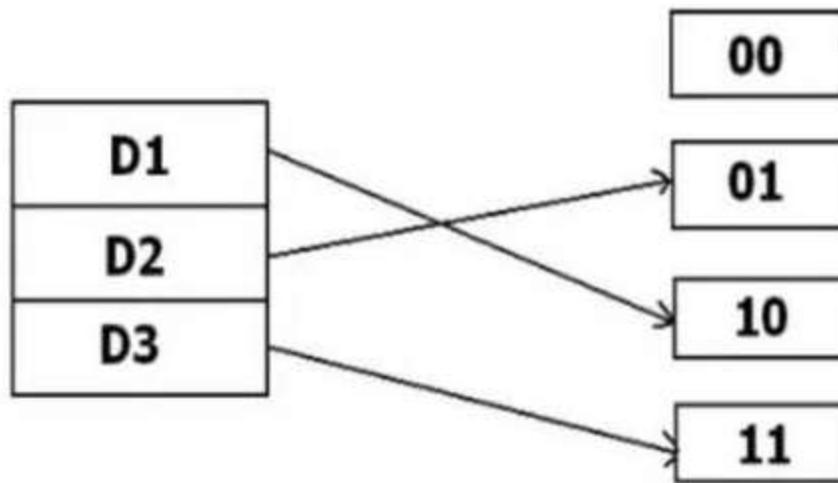


Organização de Arquivos

- **Mas o problema é que não há endereço de bucket restante para D3. O bucket precisa crescer dinamicamente para acomodar D3.**
- **Então, ele altera o endereço para 2 bits em vez de 1 bit e, em seguida, atualiza os dados existentes para um endereço de 2 bits.**
- **Em seguida, ele tenta acomodar D3.**

Organização de Arquivos

$h(D1) \rightarrow 1001$
 $h(D2) \rightarrow 0101$
 $h(D3) \rightarrow 1010$



hash dinâmico

TableSpaces

- No **PostgreSQL**, um **tablespace** é um local no sistema de arquivos onde o banco de dados pode armazenar objetos como tabelas, índices e bancos de dados inteiros.
- Por padrão, o PostgreSQL já cria dois tablespaces principais:
 - **pg_default** → usado para armazenar a maioria dos objetos, caso nada seja especificado.
 - **pg_global** → usado para objetos globais do cluster (por exemplo, catálogos de sistema compartilhados entre bancos).

TableSpaces

- **Por que usar tablespaces?**
- Eles permitem controlar onde os dados ficam fisicamente gravados no disco. Isso pode ser útil em várias situações:
- **(1) Distribuir dados em diferentes discos** (por exemplo, SSD para tabelas críticas e HDD para dados históricos).
- **(2) Melhorar desempenho** colocando índices e tabelas em dispositivos de armazenamento separados.
- **(3) Gerenciar espaço** de forma mais flexível, aproveitando diferentes partições/discos.

TableSpaces

- **Como criar um tablespace?**
- Um tablespace precisa estar associado a um diretório do sistema de arquivos já existente, ao qual o PostgreSQL tenha acesso e permissão.
- **Exemplo:**

```
1  
2 -- Criar um diretório no sistema de arquivos (feito fora do PostgreSQL)  
3 -- mkdir /mnt/disco_rapido/tablespace_app  
4  
5 -- Dentro do PostgreSQL, criar o tablespace:  
6 CREATE TABLESPACE ts_app LOCATION '/mnt/disco_rapido/tablespace_app';  
7
```

TableSpaces

- **Outro exemplo (no Windows):** vamos criar um diretório em **D:\pg_tablespaces\ts_app**.
- **Esse diretório precisa estar vazio.**
- **O usuário do Windows que executa o serviço do PostgreSQL (geralmente chamado postgres) precisa ter permissão de leitura e escrita nessa pasta.**

```
7  
8 -- Dentro do PostgreSQL, criar o tablespace:  
9 CREATE TABLESPACE ts_app LOCATION 'D:\\pg_tablespaces\\ts_app';  
10
```

TableSpaces

Atenção: No Windows, use duas barras invertidas (\\) no caminho, pois \\ é caractere de escape.

O diretório precisa estar vazio, senão dará erro.

7

8 -- Dentro do PostgreSQL, criar o tablespace:

9 CREATE TABLESPACE ts_app LOCATION 'D:\\pg_tablespaces\\ts_app';

10

11 CREATE DATABASE loja TABLESPACE ts_app;

12

TableSpaces

```
10  
11 -- Criando um banco de dados dentro de uma TableSpace específica:  
12 CREATE DATABASE loja TABLESPACE ts_app;  
13 -- Criando uma tabela dentro de uma TableSpace:  
14 CREATE TABLE produtos (  
15     id SERIAL PRIMARY KEY,  
16     nome TEXT,  
17     preco NUMERIC  
18 ) TABLESPACE ts_app;  
19 -- Criando um índice dentro de uma TableSpace:  
20 CREATE INDEX idx_produtos_nome  
21     ON produtos(nome)  
22     TABLESPACE ts_app;  
23
```

TableSpaces

```
24  
25 -- Você pode verificar os tablespaces registrados com:  
26 \db    -- no psql (lista os tablespaces)  
27  
28 -- ou via consulta SQL:  
29 SELECT spcname, pg_tablespace_location(oid)  
30 FROM pg_tablespace;  
31
```

TableSpaces

```
27  
28 -- ou via consulta SQL:  
29 SELECT spcname, pg_tablespace_location(oid)  
30 FROM pg_tablespace;  
31
```

Data output Messages



	spcname name	pg_tablespace_location text
1	pg_default	
2	pg_global	
3	meu_tablespace	D:\POSTGRESQL