Estruturas de Dados

aula 29: Espertezas de dispersão

1 Introdução

Na aula de hoje, nós vamos ver algumas espertezas que podem ser feitas com funções de dispersão.

A primeira esperteza é usada na situação onde a grande maioria das consultas é feita por elementos que não estão lá.

Por exemplo, imagine que nós temos um banco de dados de criminosos muito perigosos foragidos.

E imagine que esse banco de dados é consultado sempre que alguém é parado na rua por um guarda de trânsito.

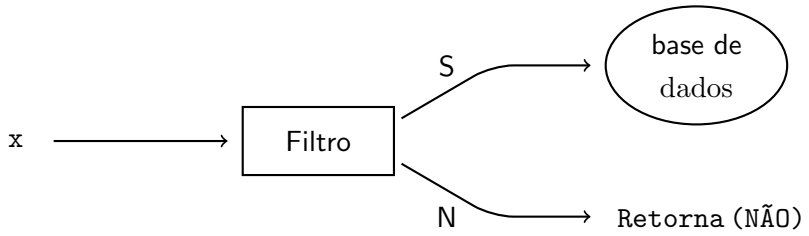
Daí, como a grande maioria das pessoas não é um criminoso muito perigoso foragido, a grande maioria das consultas tem resultado negativo.

Então, a ideia é filtrar as consultas negativas para evitar a realização de um acesso ao banco de dados nesses casos.

Mas, como é que a gente sabe que a consulta é negativa antes de acessar os dados?

Bom, é aqui que vem a esperteza.

Quer dizer, a ideia é criar uma outra estrutura de dados para separar os casos negativos dos casos positivos



2 Filtros de Bloom

Certo.

Para essa ideia fazer sentido, é preciso que o acesso ao filtro seja muito rápido.

E a melhor maneira de conseguir isso é usar um filtro imperfeito.

Quer dizer, vão haver casos em que o filtro vai dizer SIM mas o elemento não está na base de dados.

Nesses casos, a base de dados terá que ser consultada para verificar que o elemento realmente não está lá.

Por outro lado, quando o filtro responde NÃO é porque o elemento não está lá mesmo.

Legal.

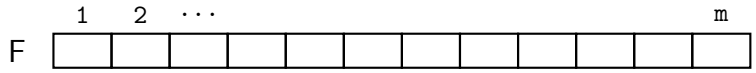
Mas como é que a coisa funciona?

Bom, a ideia é simples.

Quer dizer, nós vamos usar k funções de dispersão

$$\mathtt{h}_1(.), \quad \mathtt{h}_2(.), \quad \ldots, \quad \mathtt{h}_k(.)$$

E um vetor de bits com tamnanho m, que vai fazer o papel de filtro

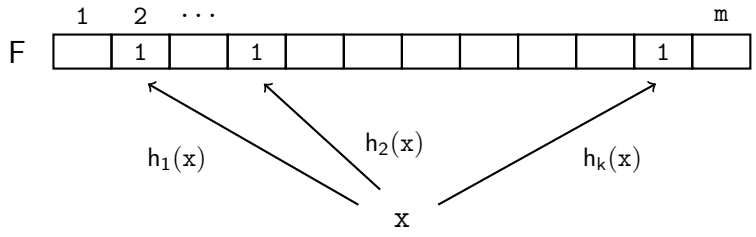


Agora imagine que nós fazemos a inserção do elemento `x` na base de dados.

Daí, para registrar esse fato no filtro, nós calculamos as posições,

$$\mathtt{i}_1 \leftarrow \mathtt{h}_1(\mathtt{x}), \qquad \mathtt{i}_2 \leftarrow \mathtt{h}_2(\mathtt{x}), \qquad \ldots \qquad \mathtt{i}_k \leftarrow \mathtt{h}_k(\mathtt{x}),$$

e ajustamos os bits nas posições $\mathtt{i}_1, \ldots, \mathtt{i}_k$ para o valor 1 — (*no início, todos os bits tem o valor 0*)



E a mesma coisa acontece com todos os elementos que são inseridos na base de dados.

Agora, imagine que ocorre uma consulta pelo elemento `z`.

Daí, nós calculamos as posições

$$\mathtt{j}_1 \leftarrow \mathtt{h}_1(\mathtt{z}), \qquad \mathtt{j}_2 \leftarrow \mathtt{h}_2(\mathtt{z}), \qquad \ldots \qquad \mathtt{j}_k \leftarrow \mathtt{h}_k(\mathtt{z}),$$

e examinamos os bits nessas posições do filtro.

Caso ao menos um desses bits seja igual a zero, nós temos a certeza de que o elemento `z` não está na base de dados — (*porque?*)

Por outro lado, se todos os bits são 1, então existem dois casos

1. pode ser que o elemento `z` tenha sido inserido na base de dados
   — (*o que fez com que os bits recebessem o valor 1*)
2. ou pode ser que `z` não esteja lá, e que os bits tenham recebido o valor 1
   na inserção de outros elementos

E é assim que surge a possibilidade de um falso positivo.

2.1 Análise

Certo.

Mas essa ideia só tem utilidade se a probabilidade de falso positivo for bem pequena.

E é isso o que nós vamos verificar agora.

Como usual, nós vamos assumir que as funções de dispersão tem comportamento aleatório.

Daí que, a probabilidade de que um certo `i` seja selecionado pela função de dispersão $\mathsf{h_j}(.)$ no momento de uma inserção é de $\frac{1}{m}$.

A probabilidade de que ele não seja selecionado é de $1 - \frac{1}{m}$.

E a probabilidade de que nenhuma função selecione o bit `i` é de

$$\Big(1 \ - \ \frac{1}{m}\Big)^k$$

A esperteza nesse ponto é reescrever a expressão assim

$$\Big(1 \ - \ \frac{1}{m}\Big)^k \quad = \quad \Big[(1 \ - \ \frac{1}{m})^m\Big]^{k/m} \quad \simeq \quad e^{-k/m}$$

Legal.

Agora imagine que existem n elementos na base de dados.

Então, a probabilidade de que o bit `i` ainda mantenha o valor 0 após as inserções é de

$$\Big(1 \ - \ \frac{1}{m}\Big)^{kn} \quad \simeq \quad e^{-kn/m}$$

E a probabilidade de que ele tenha o valor 1 é de

$$1 \ - \ e^{-kn/m}$$

Certo.

Agora nós já estamos prontos para calcular a probabilidade de um falso positivo.

Quer dizer, imagine que `z` é um elemento que não está na base de dados.

Aplicando as k funções de dispersão a esse elemento, nós obtemos as posições

$$\mathtt{i_1} \ \leftarrow \ \mathsf{h_1}(\mathtt{z}), \qquad \mathtt{i_2} \ \leftarrow \ \mathsf{h_2}(\mathtt{z}), \qquad \ldots \qquad \mathtt{i_k} \ \leftarrow \ \mathsf{h_k}(\mathtt{z}),$$

O falso positivo acontece quando todas essas posições tem o valor 1.

E a probabilidade de que isso ocorra é de

$$\varepsilon \quad = \quad \Big(1 \ - \ e^{-kn/m}\Big)^k$$

Bom, aqui nós podemos fazer algumas observações simples

1. A probabilidade de falso positivo ε diminui quando m aumenta
   — (*o que faz sentido*)

2. A probabilidade de falso positivo ε aumenta quando n aumenta
   — (*o que também faz sentido*)

Mas, o comportamento da probabilidade ε com relação a k não é tão evidente.

E a sua análise não é tão fácil assim ...

O fato é que, a medida que k aumenta

- a probabilidade ε primeiro diminui
- e depois ela volta a aumentar ...

E o valor mínimo da probabilidade ε é atingido quando

$$k \quad = \quad \frac{m}{n} \ln 2$$

Quer dizer, esse é o número de funções de dispersão que nós devemos utilizar para um certo número de elementos n e um certo tamanho do filtro m.

Para essa escolha de k, a probabilidade de falso positivo é dada por

$$\varepsilon \quad = \quad \left(1 \quad - \quad e^{-\left(\frac{m}{n} \ln 2\right)\frac{n}{m}}\right)^{\frac{m}{n} \ln 2}$$

o que simplifica para

$$\ln \varepsilon \quad = \quad - \frac{m}{n} \big(\ln 2\big)^2$$

o que resulta em

$$m \quad = \quad - \frac{n \ln \varepsilon}{\big(\ln 2\big)^2}$$

Portanto, o número ótimo de bits por elementos (para uma certa probabilidade de falso positivo ε) é

$$\frac{m}{n} \quad = \quad \frac{\ln \varepsilon}{\big(\ln 2\big)^2} \quad \simeq \quad -1,44 \ln \varepsilon$$

2.2 A operação de remoção

Da maneira como foi definido acima, o filtro de Bloom não suporta a operação de remoção.

Porque ao remover um elemento da base de dados, nós não podemos simplesmente ajustar os bits correspondentes no filtro para 0 — (*porque?*)

Mas uma pequena esperteza resolve esse problema.

Quer dizer, a ideia é trocar os bits por contadores

F	0	2	0	0	1	1	3	0	1	2	0	0

onde cada contador indica quantos elementos na base de dados selecionaram essa posição.

Daí, na operação de inserção os contadores são incrementados.

Na operação de remoção os contadores são decrementados.

E o filtro responde NÃO para uma consulta quando ao menos um dos contadores tem valor 0.

O problema com essa solução é que os contadores não podem ocupar muito espaço
— (*para que o filtro não ocupe muito espaço ...*)

Quer dizer, a ideia é que os contadores tenham apenas 2 ou 3 bits.

Mas um contador com 3 bits só pode contar até 7.

E quando ele chega em 7, novos incrementos mantém o seu valor em 7.

Mas daí, nós já não podemos mais decrementar o valor do contador.

Porque nós não sabemos se existem 7 ou mais que 7 elementos na base de dados que selecionam aquela posição.

Quer dizer, os contadores que alcançam o valor máximo não podem mais ser decrementados.

E nós vemos que essa não é uma solução perfeita para o problema da remoção.

3 Funções de dispersão perfeitas

Legal.

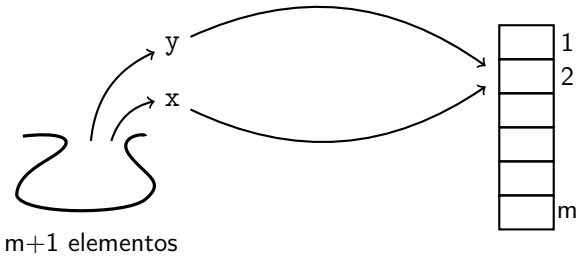
Nós acabamos de ver como é possível utilizar funções de dispersão (imperfeitas) para construir um filtro imperfeito.

Agora, nós vamos ver como é possível utilizar funções de dispersão imperfeitas para construir uma função de dispersão perfeita.

Quer dizer, uma função de dispersão perfeita é aquela onde nunca ocorre uma colisão.

Ora, mas isso é impossível.

Porque se a tabela tem tamanho m e nós temos um saco com $m + 1$ elementos, então certamente existem dois elementos no saco que são mapeados para a mesma posição da tabela



Certo.

Mas então a gente muda o problema.

Quer dizer, imagin que nós temos um saco com m elementos.

E imagine que a tabela tem tamanho m.

O problema é

→ *Construir uma função de dispersão perfeita* $\mathsf{H}()$*, que mapeia os* m *elementos do saco para* m *posições diferentes da tabela — (i.e., sem colisão)*

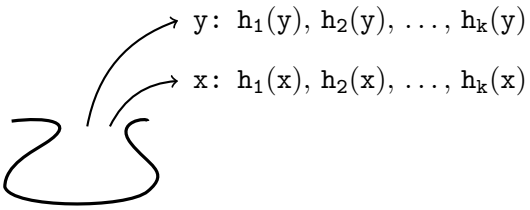
Bom, esse problema a gente consegue resolver.

E a ideia é resolvê-lo utilizando uma coleção de funções de dispersão imperfeitas.

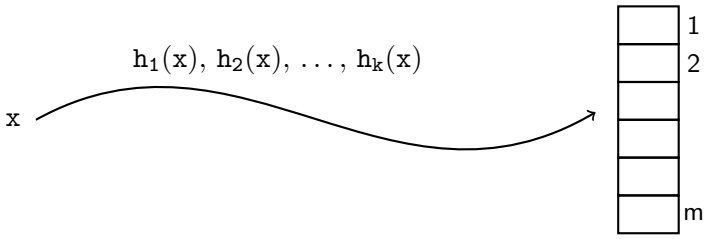
Quer dizer, as funções imperfeitas vão produzir colisões.

Mas elas vão produzir colisões de maneiras diferentes.

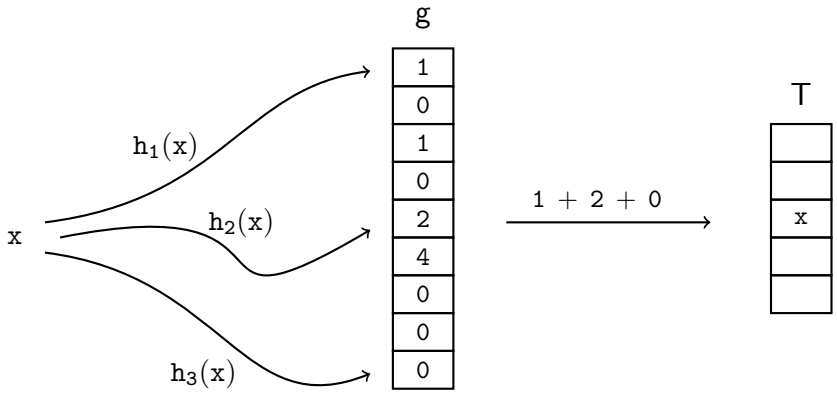
Daí que, ao aplicar as funções a todos os elementos do saco, cada um deles vai receber um conjunto de valores diferentes



E daí, a ideia é obter a posição de cada elemento na tabela T a partir da coleção de valores atribuídos pelas funções de dispersão.



Uma maneira simples de implementar essa ideia consiste em utilizar uma tabela intermediária g



Quer dizer, as funções h_1, h_2, h_3 indicam posiçõs na tabela intermediária.

E a posição de x na tabela T é dada pela soma dos elementos de g que foram selecionados pelas funções h_1, h_2, h_3.

Abaixo nós temos a definição das funções $\mathsf{h_1}$, $\mathsf{h_2}$, $\mathsf{h_3}$ para 5 elementos $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w}, \mathtt{u}$, que são mapeados todos para posições diferentes de T

	$\mathsf{h_1}$	$\mathsf{h_2}$	$\mathsf{h_3}$	
x	1	5	9	$\rightarrow \quad \mathtt{1+2+0} \;=\; \mathtt{3}$
y	3	6	8	$\rightarrow \quad \mathtt{1+4+0} \;=\; \mathtt{5}$
w	1	4	8	$\rightarrow \quad \mathtt{1+0+0} \;=\; \mathtt{1}$
z	2	5	7	$\rightarrow \quad \mathtt{0+2+0} \;=\; \mathtt{2}$
u	2	6	8	$\rightarrow \quad \mathtt{0+4+0} \;=\; \mathtt{4}$

Quer dizer, as funções $\mathsf{h_1}$, $\mathsf{h_2}$, $\mathsf{h_3}$ são imperfeitas e produzem colisões.

Mas a sua combinação por meio da tabela intermediária g resulta em um tabela de dispersão perfeita.

Legal, não é?

Bom, o truque é a escolha adequada dos valores que ficam armazenados na tabela intermediária g.

Quer dizer, tudo o que a gente espera das funções de dispersão $\mathsf{h_1}$, ..., $\mathsf{h_k}$ é que elas tenha comportamento (aproximadamente) aleatório.

Daí, elementos diferentes vão selecionar coleções de posições diferentes na tabela g.

E daí, se não houverem coincidências numéricas, os elementos vão ser todos mapeados em posições diferentes da tabela T.

Sim.

Mas cmo é que a gente garante que não ocorrem coincidências numéricas?

Bom, é isso o que nós vamos ver a seguir.

Quer dizer, suponha que nós temos m elementos para serem mapeados nas m posições da tabela T.

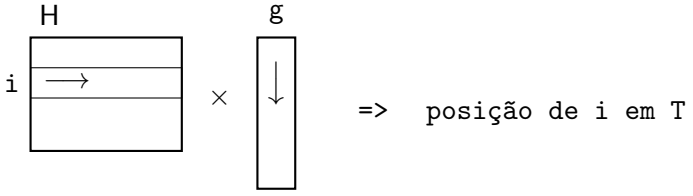
Suponha que nós vamos utilizar uma tabela intermediária g com tamanho N.

E suponha que as funções de dispersão $\mathsf{h_1}$, $\mathsf{h_2}$, ..., $\mathsf{h_k}$ já foram dadas.

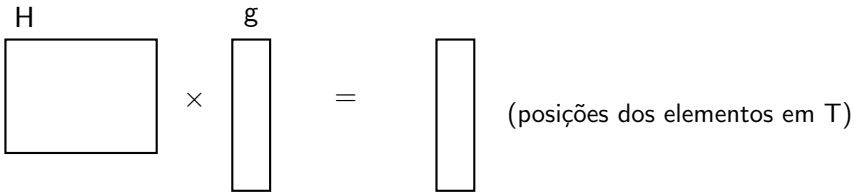
Daí, não é difícil construir a matriz

$$\mathsf{H(i,j)} \quad = \quad \begin{cases} 1 \quad , & \text{se alguma função de dispersão seleciona a posição } \mathtt{j} \text{ para o elemento } \mathtt{i} \\ 0 \quad , & \text{caso contrário} \end{cases}$$

A ideia aqui é que a multiplicação da linha $\mathtt{i}$ da matriz H pelo vetor g vai produzir a posição do elemento $\mathtt{i}$ na tabela T

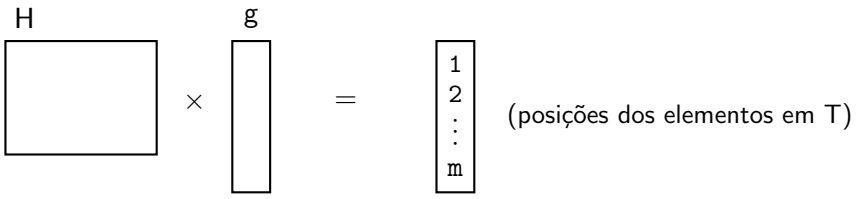


E a multiplicação da matriz inteira H pelo vetor g produz as posições de todos os elementos em T



A esperteza agora é escolher as posições em que a gente quer colocar os elementos na tabela T.

Por exemplo,



E daí, basta resolver esse sistema de equações lineares para determinar os valores de g que produzem essa distribuição dos elementos.

Se o vetor g é grande o suficiente, e as funções $h_1, \ldots h_k$ tem comportamento aleatório, o sistema tem solução com alta probabilidade

Não é legal?

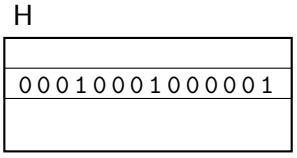
4 Mais uma esperteza

Sim, é legal.

Mas a tarefa de resolver um sistema muito grande de equações lineares é bem custosa computacionalmente.

Daí que, as pessoas pensaram em ainda mais uma esperteza para evitar ter que fazer isso.

Quer dizer, a observação chave é que as linhas da matriz H possuem muitos 0's, e apenas alguns poucos 1's



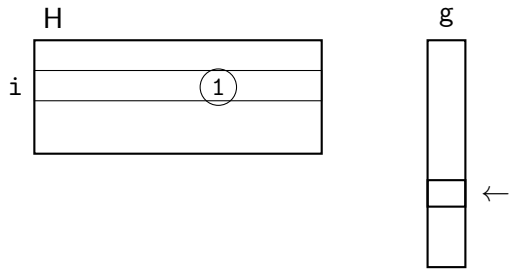
— (*porque o número k de funções de dispersão é bem menor do que o tamanho da tabela* g)

Além disso, como as funções $h_1, \ldots, h_k$ tem comportamento aleatório, linhas diferentes vão ter os seus 1's em posições diferentes.

E é aqui que aparece a esperteza.

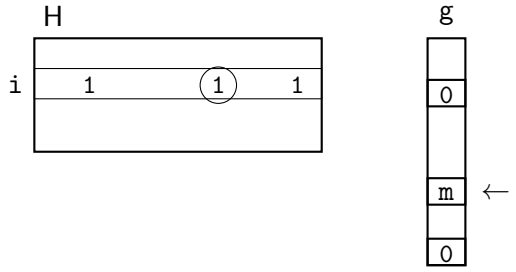
Quer dizer, escolha uma linha qualquer da matriz, e escolha um valor `1` qualquer nessa linha.

A posição desse `1` corresponde a uma posição no vetor `g`



E a ideia é que nós vamos usar essa posição para escolher o lugar do elemento `i` na tabela `T`.

Por exemplo, se a gente quiser colocar o elemento `i` na posição `m`, basta fazer o seguinte



— (*note que os outros* `1`*'sda linha* `i` *colocaram* `0`*'s em outras posições de* `g`)

Certo.

Daí, nós escolhemos uma outra linha da matriz `H`

A ideia é que essa linha possui ao menos um valor `1` em uma posição que ainda não foi preenchida na tabela intermediária `g`.

E daí, escolhendo o valor dessa posição de maneira adequada (e zerando as demais posições não preenchidas), nós conseguimos colocar o elemento `j` em qualquer posição da tabela `T`.

Pronto, é só isso!

Quer dizer, a ideia é continuar fazendo as coisas dessa maneira.

Se o vetor `g` é grande o suficiente, e as funções de dispersão $\mathtt{h}_1, \ldots, \mathtt{h}_k$ tem comportamento aleatório, nós sempre conseguimos encontrar valores `1` em posições de `g` que ainda não foram preenchidas
— (*com alta probabilidade*)

E assim, todos os elementos são colocados em posições diferentes na tabela `T`.

Na prática, a gente pode ser um pouco mais esperto na hora de definir a ordem em que a gente escolhe as linhas da matriz `H`.

E isso permite controlar o tamanho do vetor `g`.

Mas isso é uma outra esperteza, que fica para um outro dia ...