

## Filtros de Bloom

os filtros de Bloom são usados para filtrar as transações e os blocos que as contém que um nodo VSP (nodo VSP=Verificação simplificada de pagamento, são nodos que operam sem ter a cadeia blockchain completa. Em inglês, SPV) recebe de seus vizinhos. Os nodos VSP criam um filtro que corresponde aos endereços contidos na carteira do nodo VPS. O nodo VSP vai mandar uma mensagem *filterload* para o vizinho contendo o filtro para ser usado na conexão. Após isso, o vizinho vai testar cada transação contra o filtro bloom e somente as transações que passarem no filtro é que serão enviadas.

Um filtro Bloom é um filtro probabilístico que descreve um padrão desejado sem especificá-lo exatamente. É uma maneira eficiente de expressar um padrão de busca enquanto protege a privacidade.

Fazendo uma analogia, se você for um turista e perguntar para as pessoas sobre um restaurante na rua Emiliano Pernetá 122, você estará dizendo claramente qual o seu interesse. Um filtro bloom é como se você perguntasse: existem ruas nessa vizinhança cujo nome termina por "neta"? Esta pergunta informa menos sobre o destino do que "Emiliano Pernetá 122". Usando esta técnica pode-se perguntar sobre endereços terminados em 'ta' (muitos endereços, mantida privacidade alta) ou endereços terminados em Pernetá (poucos endereços, perda de privacidade). ao variar a sua busca, o turista revela mais ou menos informações, à custa de resultados mais ou menos específicos.

Os filtros bloom servem a essa função, o permitir que um nodo especifique a busca. Um filtro bloom mais específico irá produzir resultados precisos mas às custas de revelar quais endereços são usados na carteira do usuário. Um filtro menos específico irá produzir mais dados sobre mais transações, muitas irrelevantes para o nodo, mas permitirá que o nodo mantenha uma privacidade maior.

Uma família importante de algoritmos surge aqui: os algoritmos probabilísticos: eles usam atalhos sem precisar olhar todos os candidatos de forma exaustiva (força bruta). A contrapartida ("não existe almoço grátis") é que eventualmente o resultado gerado pode ser incorreto.

A maneira de lidar com este problema é chamar o algoritmo várias vezes, usualmente, tantas quantas se queira: em geral a cada chamada a probabilidade do erro diminui.

## Erros

Há que se definir aqui dois tipos de erros: o falso positivo e o falso negativo. Antes de estudar estes erros no âmbito da ciência da computação veja-se um exemplo trivial: ao pesquisar uma possível doença em uma pessoa, o falso positivo seria acusar a doença estando a pessoa sã. O falso negativo seria afirmar que a pessoa não tem a doença e ela está doente. Na ciência da computação, o falso positivo é quando ao estudar a pertinência de um elemento a um conjunto o algoritmo erra e afirma: o elemento PERTENCE ao conjunto e como ele não pertence este é um erro.

O falso negativo para o mesmo problema é afirmar que o elemento não pertence ao conjunto e ele - pelo contrário - pertence. Daí o erro.

Conforme o problema, um dos dois erros pode ser mais provável ou mesmo mais importante que o outro. Cada caso é um caso.

## Filtro de Bloom

Uma estrutura de dados probabilística, inventada por Burton Bloom em 1970, que é usada para verificar se um elemento é membro de conjunto de maneira muito rápida e eficaz. O falso positivo (achar que o elemento é membro, mas ele não é) é possível - embora com probabilidade tão baixa quanto se queira, mas o falso negativo (achar que o elemento não é membro e ele é) nunca ocorre. Sua construção usa os seguinte artefatos:

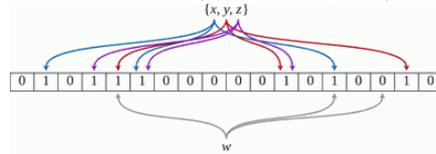
- Um conjunto  $x_1, x_2, \dots, x_n$  de elementos de um universo  $X$ .

- Um vetor  $S$  de  $m$  bits originalmente todos valendo 0.

- Um conjunto de  $k$  funções hash, todas diferentes  $h_1, h_2, \dots, h_k$  para as quais dado um valor  $X$  devolvem um valor no domínio  $(1, \dots, m)$  ou seja uma posição no vetor  $S$ .

Para agregar um elemento novo de  $X$  aplica-se a ele todas as funções  $k$  e o resultado dessas funções é tornado 1 em  $S$ . Para buscar se um novo elemento pertence ou não a  $X$ , a busca retorna "SIM" se todas as funções consultadas possuem um 1 em  $S$ .

Da maneira como o filtro é construído, é impossível eliminar um elemento. A solução pode ser um segundo filtro de Bloom contendo os eliminados. Veja no exemplo a seguir o funcionamento de um filtro de Bloom (tirado da wikipedia)



Aqui está o conjunto  $\{x, y, z\}$ . As flechas mostram as posições no vetor de bits dos resultados após a aplicação das funções de hash a cada um dos elementos. O candidato  $w$  não está no conjunto  $\{x, y, z\}$  porque ele tem um ou mais valores de hash valendo 0.

Nesta folha, vamos estudar a relação de pertinência de candidatos em um universo (denominado  $B$ ) de elementos. Cada um terá 150 bytes aleatórios e esperam-se 2.000 ocorrências em  $B$ . O algoritmo de força bruta para estudar pertinência exigiria, no pior caso, examinar  $150 \times 2000 = 300.000$  bytes. Vai-se fazer o mesmo gastando muito menos recursos. Imagine estes números elevados a expoentes grandes...

Vão-se definir 5 funções hash (denominadas  $h_1, h_2, h_3, h_4$  e  $h_5$ ) e todas elas vão gerar um resultado numérico inteiro entre os intervalos 0..19.999. Consequentemente o nosso filtro terá um comprimento de 20.000 bits. Claro que este número é inteiramente arbitrário e estimado pelo arquiteto da estrutura.

Vai-se definir um filtro (de nome  $F$ ) que será um vetor binário de 20.000 bits. Após aplicar as 5 funções hash  $h_i$  a cada elemento de  $B$  vai-se marcar com um 1 a pertinência de  $B_j$  no filtro  $F$ .

Finalmente, para estudar o funcionamento do algoritmo, vão-se gerar 10 candidatos  $C$  com 50% de probabilidade de pertencerem ao conjunto original  $B$ . Depois, far-se-á uma pesquisa de pertinência probabilística de  $C_k$  sobre  $B$  usando  $F$ .

## Funções hash propostas

Dado um elemento  $B$ , cada função hash devolverá um inteiro ( $i$ ) entre 0 e 19999 e este inteiro determinará a levantada do bit  $i$  em  $F$ .

- 1.

$$i = \sum_{j=0}^{len(X)} ord(B_j)$$

Durante o somatório, deve-se examinar o overflow para evitar dificuldades com o tamanho dos inteiros envolvidos. Como este limite de overflow é inteiramente arbitrário vai-se usar o limite 1.000.000.000 (1 bilhão). Então, se durante a soma o valor ultrapassar 1 bilhão ele será substituído por valor mod 1.000.000.000 e segue o baile. Termina a função o cálculo do resto do valor final por 20.000 mais um.

2. A segunda função só vai examinar os bytes  $j$  do elemento, onde  $j$  é a sequência de números primos: 2, 3, 5, 7, 11, 13... Também vai ser feita a soma da função *ord* de cada byte assim escolhido. O mesmo cuidado de overflow a 1.000.000.000 deve ser tomado. A função termina obtendo-se o modulo de 20.000 do resultado final mais um.

3. O elemento vai ser quebrado em pedaços de 64 bits (havendo necessidade de stuffing bits ao final colocar até 7 bytes valendo zeros binários). Esses blocos de 64 bits vão ser quebrados em 2 inteiros de 32 bits cada. Deve-se calcular o mod 20.000 de cada um

desses 2 valores e deve-se reter o maior valor alcançado. Este valor retido é o resultado final da função hash.

4. Criar uma máscara binária de 64 bits e com o valor  $X'02CAFEBABACA2288'$ . Depois o elemento deve ter blocos de 64 bits O primeiro bloco deve ser operado usando XOR com a máscara acima, gerando uma nova máscara. Essa deve ser usada na segundo bloco gerando a terceira máscara e assim por diante. Havendo necessidade de stuffing bytes ao final usar até 7 bytes valendo  $X'FF'$ . O último resultado é convertido em 2 inteiros de 32 bits. Estes valores são subtraídos, o maior menos o menor (ou o primeiro menos o segundo em módulo, tanto faz). O valor obtido é operado com mod 20000 mais um para obter o valor da função hash.
5. Os bytes do elemento das posições 1,2,4,8,16,32,... são convertidos a inteiro e somados, devidamente ponderados por 1,2,4,8,16,... Verificar overflow para 1.000.000.000 como acima descrito. O resultado final da função hash é o mod 20000 mais um desse resultado.

**Exemplo** Consulte no AVA o arquivo Fn65EXE1.myd e verifique que para este arquivo, as respostas esperadas são:

0 0 1 1 1 1 1 1 1 0

## Para você fazer

Você receberá o arquivo que foi colocado no AVA de nome

Fn65D001.myd

Este arquivo contém 2010 chaves, compostas cada uma de caracteres ASCII. Deve construir um filtro de BLOOM para as primeiras 2000 ocorrências desse arquivo. Depois disto, deve consultar as últimas 10 ocorrências para ver se elas pertencem ou não ao conjunto inicial. É óbvio que a pertinência deve ser verificada via o filtro de Bloom e não por força bruta, pois este é o objetivo de aprendizagem desta folha.

Informe 1 (pertence) ou 0 (não pertence) para cada uma das 10 últimas ocorrências do arquivo.

1	2	3	4	5	6	7	8	9	10



- 1 - /