Filtro de Bloom - Estrutura de Dados

Emannuel Levi de Assis Bezerra

Fevereiro de 2025

1 O que é o filtro de Bloom?

O filtro de Bloom é uma técnica eficiente e econômica para determinar se uma lista contém ou não um determinado elemento. Seu funcionamento baseia-se no uso de n funções hash: para cada elemento adicionado, cada hash mapeia o elemento para uma posição específica em um vetor de bits inicialmente zerado. As posições correspondentes ao elemento são então configuradas como 1. Para verificar se um dado elemento está presente na lista, aplicam-se as n funções hash e verifica-se se todas as posições resultantes estão marcadas com 1; se estiverem, o elemento é considerado presente. Contudo, esse método apresenta a desvantagem dos falsos positivos, isto é, existe a chance de todas as posições estarem marcadas como 1 devido a colisões de hash, mesmo quando o elemento não foi adicionado. Por outro lado, não há falsos negativos: se o filtro de Bloom indicar que um elemento não está na lista, ele realmente não está.

2 Implementação

2.1 Bit Array

A seguir, apresenta-se a implementação em Python do Bit Array:

```
class BitArray:
    def __init__(self, size):
        self.barray = [0] * (size%32)
    def set(self, idx):
        i = (idx//32)%(len(self.barray))
        bi = idx%32
        self.barray[i] |= 1 << bi
    def get(self, idx):
        i = (idx//32)%(len(self.barray))
        bi = idx%32
        return (self.barray[i] >> bi) & 1
```

Nesta implementação, inicializamos uma lista de inteiros, onde cada inteiro representa 32 bits. Por exemplo, ao inicializar o Bit Array com tamanho 70, teremos uma lista contendo 3 inteiros (ou seja, 96 bits). Para definir (setar) um bit, primeiramente é necessário identificar a posição do inteiro na lista, representada por `i`. Seguindo o exemplo do tamanho 70, se quisermos acessar o bit 59, ele estará localizado no segundo inteiro da lista (ou seja, `i = 1`). Em seguida, para determinar o índice específico do bit dentro do inteiro, representado por `bi`, utilizamos a operação de módulo: para o bit 59, temos `59 mod 32 = 27`. Dessa forma, configuramos o vigésimo sétimo bit do segundo inteiro da lista como 1, utilizando uma operação com `or`. Para isso, realizamos 27 deslocamentos para a esquerda em um inteiro com valor 1 e aplicamos a operação bitwise `OR`, garantindo que o vigésimo sétimo bit será configurado como 1, independentemente do valor previamente armazenado. Para acessar (pegar) o valor do bit, a lógica é semelhante, mas agora utilizamos uma operação bitwise `AND` com 1, verificando diretamente o valor armazenado naquele bit.

2.2 Filtro de Bloom

Segue implementação em Python:

```
class Lista:
    def __init__(self, barray_size, hash_quantity):
        self.list = []
        self.size = barray_size
        self.bit_array = BitArray(barray_size)
        self.seeds = sample(range(1, 141115), hash_quantity)
    def append(self, item):
        self.list.append(item)
        for seed in self.seeds:
            self.bit_array.set(mmh3.hash(str(item), signed=False, seed=seed)%self.size)
    def __contains__(self, key):
        for seed in self.seeds:
            if self.bit_array.get(mmh3.hash(str(key), signed=False, seed=seed)%self.size) !=
                return False
        return True
```

Nesta implementação, a classe `Lista` representa um filtro de Bloom. Durante a inicialização, definimos o tamanho do bit array (`barray_size`) e a quantidade de funções hash (`hash_quantity`). O bit array é criado a partir da classe `BitArray` e, em seguida, é gerada uma lista de sementes aleatórias que serão utilizadas para configurar as funções hash.

No método `append`, o item é adicionado à lista e, para cada semente, calculamos o hash do item utilizando a função `mmh3.hash`. O valor resultante, após a aplicação do operador módulo com o tamanho do bit array, indica a posição que deverá ser configurada como 1, marcando a presença do item naquele índice.

Para verificar se um item pertence ao conjunto, o método `__contains__` recalcula os hashes do item utilizando as mesmas sementes. Se, para alguma das sementes, o bit correspondente no bit array não estiver configurado como 1, o método retorna `False`. Caso todos os bits estejam configurados como 1, o item é considerado presente (lembrando que esse método pode ocasionar falsos positivos).

3 Testes

3.1 Implementação

Segue implementação em python dos testes para medirmos em que casos o filtro de bloom pode nos ser util, usaremos uma lista de emails criados aleatóriamente (com ajuda da biblioteca Faker) e depois criaremos outra lista de emails aleatórios de tamanho maior e veremos quantos colidem na nossa lista (Quanto menos, melhor)

```
quantities = [50, 250, 500]
hashes = [3, 6, 9]
spans = [50, 180, 350]
emails = [200, 500, 1000]
for quantity, hashy, span, email in product(quantities, hashes, spans, emails):
    lst = Lista(quantity, hashy)
    for i in range(span):
        lst.append(fake.email())
    randemails = [fake.email() for i in range(email)]
    print(f"{quantity}, {hashy}, {span}, {emails}",end=": ")
    print(len(list(filter(lambda x: x in lst, randemails))))
```

3.2 Resultados

Segue os resultados obtidos da execução do algoritmo OBS: 3,6 e 9 se refere a quantidade de Hashes, tabela confusa eu sei, mas é dificil fazer uam tabela de quatro dimensões.

Table 1: Resultados dos Testes para Quantidade = 50

	200 Emails			**500 Emails**			**1000 Emails**		
Span	3	6	9	3	6	9	3	6	9
50	157	200	200	399	436	500	1000	1000	1000
180	200	200	200	500	500	500	1000	1000	1000
350	200	200	200	500	500	500	1000	1000	1000

Table 2: Resultados dos Testes para Quantidade = 250

	200 Emails			**500 Emails**			**1000 Emails**		
Span	3	6	9	3	6	9	3	6	9
50	17	30	37	39	44	85	77	93	155
180	139	179	200	349	462	485	599	915	1000
350	190	200	200	491	500	500	971	1000	1000

Table 3: Resultados dos Testes para Quantidade = 500

	200 Emails			**500 Emails**			**1000 Emails**		
Span	3	6	9	3	6	9	3	6	9
50	4	0	4	10	5	9	18	13	6
180	59	100	117	124	256	349	286	414	699
350	137	195	195	339	414	476	662	960	972

Obtivemos um cenário ideal ao utilizar quantidade = 500, span = 50, 6 funções hash e 200 emails, que resultou em 0 (o menor valor possível, indicando o melhor desempenho do filtro). Além disso, em dois outros cenários notáveis com os mesmos parâmetros de span (50), emails (200) e quantidade (500), os testes com 3 e 9 funções hash apresentaram um resultado de 4.