Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Estudo do efeito de variações de *Bloom filters* no desempenho de algoritmos de hifenização de palavras

Matheus Barbosa Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Guilherme Oliveira Mota
Cossupervisor: Prof. Dr. Yoshiharu Kohayakawa

São Paulo
2022

Esta seção é opcional e fica numa página separada;
ela pode ser usada para uma dedicatória ou epígrafe.

Agradecimentos

Do. Or do not. There is no try.
— Mestre Yoda

Texto texto. Texto opcional.

Resumo

Matheus Barbosa Silva. **Estudo do efeito de variações de *Bloom filters* no desempenho de algoritmos de hifenização de palavras**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

Matheus Barbosa Silva. **Title of the document**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de Símbolos

ϵ Razão de resultados falsos positivos

Lista de Figuras

2.1 Esquema do funcionamento de um dicionário implementado com *hash table* 5
2.2 Esquema do funcionamento de um dicionário implementado com *hash compaction* 6

3.1 Esquema do funcionamento de um *Bloom filter* com parâmetros $m = 16$, $k = 3$, $n = 2$, $S = \{x, y\}$ com teste de membresia do elemento z que não pertence à estrutura 8
3.2 Esquema do funcionamento de um *Bloom filter* com parâmetros $m = 16$, $k = 3$, $n = 2$, $S = \{x, y\}$ com resultado falso positivo para o teste de membresia do elemento w que não pertence à estrutura 9

4.1 Esquema do funcionamento da inserção de um elemento em uma *cuckoo hash table* 16
4.2 Esquema do funcionamento da inserção de um elemento em um *Cuckoo filter* com a técnica de *partial-key cuckoo hashing* 18
4.3 Esquema do funcionamento da remoção de um elemento em um *Cuckoo filter* 20

5.1 Tempo médio de processamento de uma mesma lista de palavras em um algoritmo de hifenização implementado com um *Bloom filter* (com intervalos de confiança de 95%) 22
5.2 Tempo médio de construção de um *Bloom filter* para uma mesma lista de exceções em um algoritmo hifenizador de palavras (com intervalos de confiança de 95%) 23
5.3 Tempo médio de processamento de uma mesma lista de palavras em um algoritmo de hifenização implementado com um *cuckoo filter* (com intervalos de confiança de 95%) 24
5.4 Tempo médio de construção de um *cuckoo filter* para uma mesma lista de exceções em um algoritmo hifenizador de palavras (com intervalos de confiança de 95%) 25

5.5 Comparação entre os tempos médios de processamento de uma mesma lista de palavras em um algoritmo hifenizador de palavras implementado com *Bloom filter* e com *cuckoo filter* 25
5.6 Comparação entre os tempos médios de construção do filtro para uma mesma lista de exceções em um algoritmo hifenizador de palavras implementado com *Bloom filter* e com *cuckoo filter* 26

Sumário

1 Introdução **1**

2 *Approximate set membership* **3**

2.1 Implementação de Dicionário Completo 4

2.1.1 Interface 4

2.1.2 *Hash table* 4

2.1.3 *Hash Compaction* 5

3 *Bloom filters* **7**

3.1 Definição 8

3.2 Razão de falsos positivos 9

3.3 Comprimento mínimo do filtro 10

3.4 Algoritmos 10

3.4.1 Inserção 10

3.4.2 Consulta 11

3.5 Aplicações 11

3.5.1 Hifenização de palavras 11

3.5.2 Bancos de dados distribuídos 12

3.5.3 Estimação da diferença entre conjuntos 12

3.5.4 Estimação da intersecção de conjuntos 12

3.5.5 Protocolos de roteamento 13

4 *Cuckoo filters* **15**

4.1 *Cuckoo hash tables* 15

4.1.1 Inserção 16

4.1.2 Consulta 16

4.1.3 Remoção 17

4.2 Definição 17

4.3 Algoritmos 18

4.3.1 Inserção 18
4.3.2 Consulta 18
4.3.3 Remoção 18

5 Experimentos **21**
5.1 Análise do desempenho de *Bloom filters* 21
5.2 Análise do desempenho de *Cuckoo filters* 21
5.3 Análise comparativa do desempenho de *Bloom filters* e *Cuckoo filters* . . 21

6 Conclusão **27**

Capítulo 1

Introdução

Texto

Capítulo 2

Approximate set membership

A representação de **conjuntos** de elementos por meio de estruturas de dados requer que essas estruturas sejam capazes de indicar não só as informações armazenadas, mas também responder quais elementos estão no conjunto. Nesse contexto, a verificação de que um dado elemento x é membro de um conjunto S (isto é, $x \in S$) é chamada de *membership testing* (teste de membresia).

A construção de soluções para o problema de verificar se um elemento pertence a um conjunto pode ser baseada em uma das seguintes variantes do problema: **estática** ou **dinâmica**. Essa classificação é feita de acordo com as informações dadas no momento de execução de buscas de elementos no conjunto. Na variante estática do problema, assume-se que o conjunto S tem um tamanho fixo. Logo, todos os elementos do conjunto são expressos antes das consultas, enquanto na variante dinâmica, inserções e consultas podem estar intercaladas.

Seja $n = |S|$, uma solução intuitiva e determinística para determinar se $x \in S$ requer a comparação do elemento x com cada um dos n elementos de S. No entanto, a aplicação dessa estratégia pressupõe uma representação de S que consome espaço $\Theta(n)$, isto é, exige consumo linear de espaço. Logo, essa não é uma alternativa vantajosa para os cenários em que o **espaço é escasso** ou não é desejável ou necessário representar cada elemento do conjunto.

Para os cenários em que o espaço é escasso, propõe-se o *approximate set membership* (teste de membresia aproximado) como uma **solução probabilística** que permite responder consultas sobre um conjunto de forma aproximada.

À medida que essa estrutura para responder testes de membresia é comprimida, passa a existir o custo de resultados falsos positivos – a estrutura responde que um elemento está no conjunto, quando na verdade não está. A probabilidade de ocorrência desses resultados depende do nível da compressão e da estratégia adotada pela estrutura que responde aos testes de membresia.

2.1 Implementação de Dicionário Completo

Dicionário completo (ou simplesmente dicionário) é um tipo de estrutura de dados que apresenta uma entrada para cada elemento de um conjunto (isto é, há enumeração completa dos elementos). Logo, um dicionário provê respostas determinísticas para testes de membresia ao custo de consumo de espaço $\Theta(n)$.

Dicionários podem ser implementados a partir de *hash tables*, árvores rubro-negras ou outras estruturas de dados que permitam, usualmente, o mapeamento de pares chave-valor.

2.1.1 Interface

Um dicionário deve implementar os seguintes comandos:

- `INSERT(x)`: insere um dado elemento x no conjunto S;
- `QUERY(x)`: verifica se um dado elemento x é membro do conjunto S (*membership testing*);

2.1.2 *Hash table*

Hash table é uma implementação possível para a estrutura de dicionário completo. Essa estrutura mantém um array de *buckets* que comportam zero ou mais elementos do conjunto S.

Seja $\mathcal{U}$ o conjunto universo, também escolhe-se uma função de *hashing* h, $h : \mathcal{U} \longrightarrow \{0, 1, ..., m\}$ que mapeia cada elemento do conjunto a um *bucket* indexado por um número no intervalo $[0, m]$, em que $m < |\mathcal{U}|$. Logo, no caso de colisão de *hashing* (onde vários elementos do conjunto são mapeados para um *bucket* de mesmo índice) deve existir um **esquema de resolução de colisão**.

Um esquema de resolução de colisão deve permitir que vários elementos do conjunto mapeados para o mesmo *bucket* possam ser univocamente identificados. Um possível esquema de resolução de colisão é a construção de uma lista ligada que contenha todos os elementos mapeados para um mesmo *bucket*.

Uma implementação de dicionário completo com *hash table* deve aplicar os procedimentos seguintes para cada comando:

- `INSERT(x)`: usa-se a função de hashing h para obter um índice $i = h(x)$, $0 \leq i < m$. O elemento é, então, inserido na estrutura no *bucket* de índice i, que emprega o esquema de resolução de colisões, se necessário;
- `QUERY(x)`: usa-se a função de hashing h para obter um índice $i = h(x)$, $0 \leq i < m$. Então, busca-se o elemento no *bucket* de índice i. Se o elemento é encontrado, retorna que $x \in S$, senão retorna que $x \notin S$.

Essas dinâmicas são ilustradas na Figura 2.1.

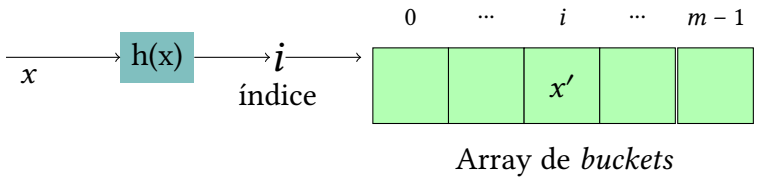

Figura 2.1: *Esquema do funcionamento de um dicionário implementado com hash table*

Seja $u = |\mathcal{U}|$ o tamanho do conjunto universo e $n = |S|$ o tamanho do conjunto contido em U, logo, há $\binom{u}{n}$ conjuntos S distintos que podem ser representados pela estrutura. Portanto, a memória necessária, em bits, para representar computacionalmente todos os possíveis conjuntos S é limitada superiormente por $\log_2 \binom{u}{n} = n \log_2 \frac{u}{n} + \Theta(n)$.

Essa implementação permite que consultas sejam realizadas com tempo esperado $O(1)$ ou, no pior caso, em tempo $\Theta(n)$. Já inserções são sempre realizadas em tempo $O(1)$ desde que se considere um esquema de resolução de conflitos com inserção em tempo $O(1)$ (como a implementação de uma lista ligada).

No entanto, o armazenamento e o acesso a elementos com comprimentos grandes (em bits) podem requerer vários acessos ao disco e causar perda de desempenho. Logo, é desejável minimizar a quantidade de acessos ao disco, de modo que o consumo real de tempo aproxime-se do esperado.

2.1.3 *Hash Compaction*

A implementação de dicionários completos com *hash compaction*, proposta por Wolper e Leroy, viabiliza uma **relação linear** entre o número de elementos do conjunto e o espaço consumido pela estrutura. Essa estrutura armazena os elementos do conjunto em **blocos de tamanho fixo**.

Assim, todo elemento inserido na estrutura consome o mesmo espaço (em bits). Para isso, usa-se uma nova função de hashing H, $H : \mathcal{U} \mapsto \{0,1\}^l$, onde l é o comprimento (em bits) dos blocos na estrutura. Desta forma, todo elemento é comprimido em l bits pela função de hashing H antes de ser inserido na estrutura.

Implementações com *hash compaction* devem aplicar os seguintes procedimentos para cada comando:

- `INSERT(x)`: aplica-se o procedimento de inserção usado na implementação com *hash table* para incluir uma representação comprimida de x na estrutura. Seja $x' = H(x)$ a representação comprimida de x e $i = h(x')$, $0 \leq i < m$, o índice do *bucket* do elemento na estrutura. A representação comprimida x' é inserida na estrutura no bucket de índice i, que emprega o esquema de resolução de colisões, se necessário;

- `QUERY(x)`: aplica-se a função de hashing H para obter $x' = H(x)$, uma representação comprimida de x. Então, usa-se a função de hashing h para obter um índice $i = h(x')$, $0 \leq i < m$. Se x' é encontrado no *bucket* de índice i, retorna que $x \in S$, senão retorna que $x \notin S$.

Essas dinâmicas são ilustradas na Figura 2.2.

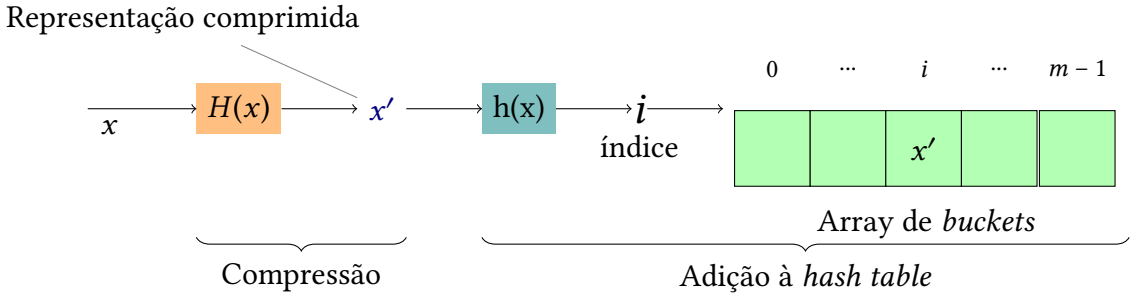

Figura 2.2: *Esquema do funcionamento de um dicionário implementado com hash compaction*

Seja k o comprimento em bits da representação comprimida de x (isto é, x') e n o número de elementos distintos que podem ser representados pela estrutura. Para $k = \log_2 n$, note que com k bits é possível representar $2^k = n$ elementos distintos. Portanto, para $k \geq \log_2 n$ não há colisão de representações comprimidas.

Já para os casos em que ocorre colisão, essa estrutura pode retornar resultados falsos positivos. Isto é, um elemento que não faz parte do conjunto representado pode retornar positivo para o teste de membresia por conta da colisão de hashing com outro elemento que pertence à estrutura. O parâmetro $\epsilon \in (0, 1)$ representa a probabilidade de ocorrência de resultados desse tipo. Assumindo que cada bit (0 ou 1) seja escolhido com probabilidade $\frac{1}{2}$ pela função de hashing $H(x)$, então é evidente que $\epsilon \leq \frac{1}{2^k}$ (pois há 2^k distintas representações comprimidas representáveis).

Capítulo 3

Bloom filters

O *Bloom filter*, estrutura de dados aleatorizada concebida por Bloom (1970), tem o objetivo de responder a testes de membresia com **eficiência no consumo de espaço** (assim como os dicionários completos implementados com *hash conmpaction*). Essa estrutura permite representar os elementos de um conjunto de forma compacta com o custo de resultados **falsos positivos** – quando a estrutura responde que um elemento é membro do conjunto, quando de fato não é – que variam de acordo com o tamanho escolhido para o filtro.

Dada a possibilidade de ocorrência de resultados falsos positivos, o *Bloom filter* é uma estrutura de dados especialmente adequada para cenários em que os métodos convencionais (não probabilísticos, livres de erros) para responder a testes de membresia demandem uma quantidade impraticável de espaço. Ademais, Bloom (1970) sugere o uso dessa estrutura para aplicações em que a grande maioria dos elementos testados não fazem parte do conjunto representado pela estrutura (isto é, resultados negativos).

Define-se **tempo de rejeição** como o tempo médio dispendido por uma estrutura de dados para decidir que um dado elemento não pertence ao conjunto representado. De fato, o *Bloom filter* permite diminuir o consumo de espaço sem acrescer o tempo de rejeição, o que ocorre com o custo de resultados falsos positivos e impossibilidade de remoção de elementos da estrutura (ainda que existam estruturas derivadas do *Bloom filter*, como o *counting Bloom filter*, que permitem remoções de elementos). A aplicação de um *Bloom filter* é vantajosa quando o **princípio do *Bloom filter*** é atendido:

> Onde quer que seja usada uma lista ou conjunto e o espaço seja valioso, considere usar um *Bloom filter* se o efeito de falsos positivos puder ser mitigado. (Broder e Mitzenmacher, 2003, tradução nossa)

Logo, essa é uma estrutura adequada para cenários em que são permitidos erros nas respostas dos testes de membresia. O processo de **hifenização** de palavras é um desses cenários e, especialmente, um clássico exemplo de aplicação de *Bloom filters*. Para cada idioma são definidas regras fixas de hifenização que podem ser aplicadas a qualquer palavra. No entanto, podem existir exceções às regras, o que demanda a busca pontual da hifenização de algumas palavras no dicionário.

Nesse cenário, um *Bloom filter* pode ser utilizado para representar o conjunto de palavras que demandam buscas no dicionário. Assim, um resultado falso positivo corresponde a uma palavra cuja hifenização pode ser obtida pelas regras, mas foi erroneamente classificada como uma exceção. Nesse caso, a palavra não seria encontrada no dicionário (já que não é uma exceção) e, em seguida, seria hifenizada pelas regras. De todo modo são evitados acessos desnecessários ao dicionário e, dado o número pequeno de exceções, usualmente a maior parte das palavras não faz parte do filtro.

3.1 Definição

Um *Bloom filter* é descrito pelos parâmetros seguintes:

- Seja $S = \{x_1, x_2, \ldots, x_n\}$ o conjunto de n elementos representado pela estrutura;
- Seja m o comprimento em bits do vetor utilizado pelo filtro para representar o conjunto (inicialmente com todos os m bits são zerados);
- Seja k a quantidade de funções de *hashing* **universais** independentes $(h_1, \ldots, h_k)$ utilizadas pelo filtro, $h_i : \mathcal{U} \longrightarrow [0, m-1], \forall i \in [1, k]$. Uma função de *hashing* universal deve mapear as entradas uniformemente no conjunto de saída;

Para a **inserção** de um dado elemento x na estrutura, todos os bits de índices $h_i(x), \forall i \in [1, k]$ recebem o valor 1. Assim, a resposta para um teste de membresia de um dado elemento z é obtida a partir da verificação dos bits de índices $h_i(x), \forall i \in [1, k]$. Caso algum dos k bits examinados contenha o valor 0, então a estrutura responde que o elemento mão está no conjunto representado (resposta negativa). Isto é, algum dos bits examinados contém o valor 0 **se e somente se** o elemento consultado não pertence ao filtro. O esquema do funcionamento da inserção de elementos e dos testes de membresia são apresentados na Figura 3.1.

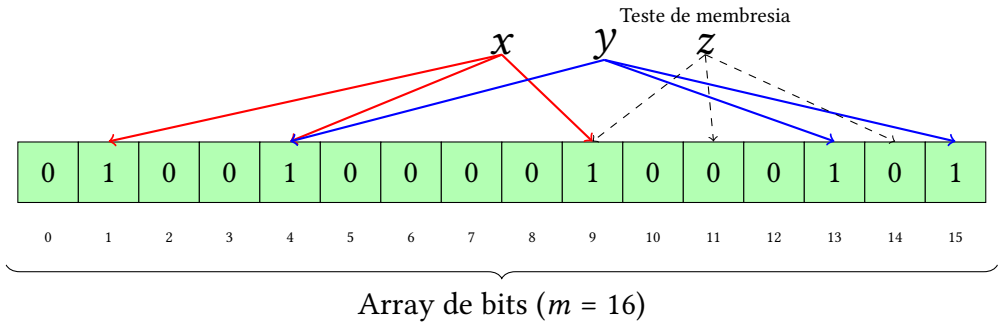

Figura 3.1: *Esquema do funcionamento de um Bloom filter com parâmetros* $m = 16, k = 3, n = 2, S = \{x, y\}$ *com teste de membresia do elemento z que não pertence à estrutura*

Já quando todos os k bits examinados para um dado elemento são iguais a 1, a estrutura assume que o elemento pertence ao conjunto representado (resposta positiva). Ainda assim, essa classificação é feita com uma probabilidade de erro, dado que essa estrutura admite resultados falsos positivos. Um exemplo de resultado falso positivo é dado na Figura 3.2.

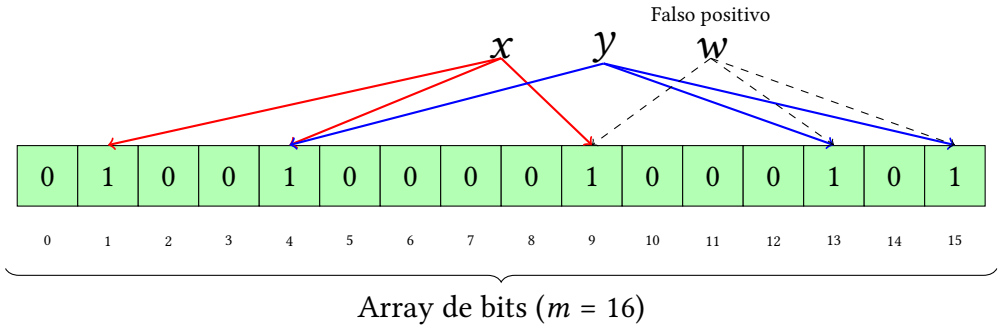

Figura 3.2: *Esquema do funcionamento de um Bloom filter com parâmetros* $m = 16, k = 3, n = 2, S = \{x, y\}$ *com resultado falso positivo para o teste de membresia do elemento* w *que não pertence à estrutura*

3.2 Razão de falsos positivos

O cálculo da razão de falsos positivos em um *Bloom filter* é realizado a partir da supoosição de que $kn < m$, de modo que sejam evitados os casos triviais em que cada um dos n elementos do conjunto S possam ser mapeados para k bits distintos do vetor de bits. Ademais, sem perda de generalização, assuma que todos os elementos de S estão inseridos na estrutura.

Assim, nesse cenário, seja p' a probabilidade de que um dado bit do vetor esteja zerado. Dada a suposição de que as funções de *hashing* escolhidas são universais, então para cada função, um bit recebe o valor 1 com probabilidade $1/m$. Com a inserção de n elementos, cada um mapeado com k funções de *hashing*, segue que

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

Tome $p = e^{-kn/m}$ uma aproximação para p'. A ocorrência de um resultado falso positivo demanda que os k bits examinados sejam iguais a 1, logo requer k repetições do evento com probabilidade complementar a p. Portanto, um resultado falso positivo ocorre aproximadamente com probabilidade

$$\left(1 - e^{-kn/m}\right)^k = (1 - p)^k.$$

Suponha que m e n são dados, então deseja-se obter um valor ótimo para k de modo que a razão de resultados falsos positivos $(1 - p)^k$ seja minimizada. Seja

$$f(k) = \exp\left(k \ln(1 - e^{-kn/m})\right)$$

uma função de k para a razão de falsos positivos (isto é, $(1-p)^k$ escrita de forma conveniente). Obtém-se que um mínimo global para essa função ocorre com $k = \ln 2 \cdot (m/n)$ de modo que, nesse caso,

$$f(k) = (1/2)^k \approx (0.6185)^{m/n}.$$

Dado que $k \in \mathbb{N}$, escolhe-se $k = \lfloor \ln 2 \cdot (m/n) \rfloor$ de modo a minimizar o número de funções de *hashing* utilizadas. No entanto, note que a determinação de um valor ótimo para o parâmetro k só é possível quando se conhece ou se projeta de antemão a quantidade de elementos que devem ser representados pela estrutura.

3.3 Comprimento mínimo do filtro

Dada uma razão de resultados falsos positivos ϵ, deseja-se obter m mínimo (o comprimento mínimo do vetor de bits do filtro) de modo a permitir resultados falsos positivos para, no máximo, uma fração ϵ de $\mathcal{U}$. Seja $u = |\mathcal{U}|$ e $n = |S|$, então o filtro deve ser capaz de identificar cada um dos $\binom{u}{n}$ conjuntos S distintos.

Seja $s = F(S)$ a sequência de bits para o qual o conjunto S é mapeado por um dado *Bloom filter*. Diz-se que s **aceita** um dado elemento $x \in \mathcal{U}$ se s é a sequência de bits associada a um conjunto S tal que $x \in S$. É evidente que toda sequência s aceita todos os elementos de S, mas também pode aceitar outros $\epsilon(u - n)$ elementos que fazem parte de $\mathcal{U}$, mas não estão no conjunto S de modo que a razão de falsos positivos seja, no máximo, igual a ϵ.

Portanto, cada sequência s deve representar, no máximo, $n + \epsilon(u - n)$ elementos de $\mathcal{U}$. Em especial, uma sequência s pode representar qualquer um dos $\binom{n+\epsilon(u-n)}{n}$ subconjuntos de S. Dado que m é o tamanho (fixo) da sequência de bits, então há 2^m sequências distintas de m bits que devem representar $\binom{u}{n}$ conjuntos distintos. Em especial, deve valer que

$$2^m \binom{n + \epsilon(u - n)}{n} \geq \binom{u}{n}.$$

A partir dessa desigualdade, obtém-se um limite inferior para o o comprimento do vetor de bits de modo que a razão de falsos positivos não ultrapasse ϵ. É necessário que m atenda a desigualdade

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 e \cdot \log_2 (1/\epsilon).$$

3.4 Algoritmos

Um *Bloom filter* permite realizar operações de inserção e consulta de elementos.

3.4.1 Inserção

O procedimento para a inserção de um elemento em um *Bloom filter* é descrito no Algoritmo 1, que itera sobre as k funções de *hashing* e pode alterar o valor de uma posição do vetor de bits a cada iteração (logo consome tempo $\Theta(k)$).

Algoritmo 1: Bloom Filter: Inserção (`Insert`)

Entrada: x - elemento a ser inserido na estrutura

```
1 para j ← 1 até k faça
2 |   i ← HASH_j(x)                ▷ Itera sobre todas as k funções de hashing
3 |   se bits[i] = 0 então
4 |   |   bits[i] ← 1
5 |   fim
6 fim
```

3.4.2 Consulta

O Algoritmo 2 descreve o processo necessário para responder a testes de membresia em um *Bloom filter*. Para isso, basta verificar se os k bits indexados pelos resultados das funções de *hashing* contêm o valor 1 (o algoritmo itera sobre as k funções de *hashing*, logo consome tempo $\Theta(k)$).

Algoritmo 2: Bloom Filter: Consulta (`Query`)

Entrada: x - elemento a ser consultado na estrutura (teste de membresia)

Saída: `Verdadeiro` ou `Falso` - indica se o elemento está na estrutura

```
1 para j ← 1 até k faça
2 |   i ← HASH_j(x)
3 |   se bits[i] = 0 então
4 |   |   retorna Falso
5 |   fim
6 fim
7 retorna Verdadeiro
```

3.5 Aplicações

De acordo com Bloom (1970), a aplicação de *Bloom filters* é vantajosa em cenários onde a grande maioria dos elementos consultados (nos testes de membresia) não fazem parte do conjunto representado pela estrutura. A seguir são dados alguns cenários convenientes para a aplicação dessa estrutura.

3.5.1 Hifenização de palavras

A aplicação de *Bloom filters* em algoritmos de hifenização de palavras foi originalmente citada e analisada por Bloom (1970). O processo de hifenização funciona a partir da análise de regras e exceções definidas para cada linguagem. O grupo de exceções, para cada idioma, compreende um grupo de palavras cujas hifenizações não podem ser obtidas de acordo com as regras usuais. Portanto, as hifenizações de palavras desse grupo são armazenadas em algum tipo de armazenamento persistente.

Evidentemente, o grupo de exceções representa apenas uma pequena porção do total de palavras um uma dada linguagem. Nesse cenário, Bloom (1970) sugere que apenas as

exceções sejam inseridas em um *Bloom filter*, de modo que apenas as palavras encontradas no filtro sejam buscadas no armazenamento. Assim, um resultado falso positivo (palavra cuja hifenização pode ser obtida pelas regras, mas é classificada como uma exceção) resultaria na busca de uma palavra no armazenamento, o que não retornaria resultados e, então, obtém-se a hifenização a partir das regras usuais.

3.5.2 Bancos de dados distribuídos

Em bancos de dados distribuídos, os dados não são centralizados e armazenados em um único local, mas sim repartidos em diversos nós que se intercomunicam. Essa estratégia é especialmente conveniente para o gerenciamento de grandes volumes de dados, já que é possível acrescer nós quando necessário e, consequentemente, aumentar a capacidade de armazenamento de dados do sistema, assim como mitigar falhas em um nó.

O uso de *Bloom filters* em bancos de dados distribuídos permite reduzir o tráfego de dados na rede. De fato, já que os dados não são centralizados, a resposta para uma consulta pode demandar informações de vários nós. Se um nó A precisa de informações de um nó B para responder a uma consulta, então o nó A pode receber um *Bloom filter* com os dados do nó B inseridos (o que diminui o tráfego de dados na rede, já que essa é uma representação comprimida dos dados de B) no lugar de uma cópia completa dos dados. Ao fim do processamento, o nó A pode enviar o resultado da consulta para o nó B de modo que falsos positivos possam ser removidos.

3.5.3 Estimação da diferença entre conjuntos

Bloom filters também podem ser utilizados na transmissão de dados em aplicações *peer-to-peer*. Dados dois pares A e B que armazenam os conjuntos S_A e S_B, respectivamente, suponha que B requer os dados de S_A que ainda não estão em S_B (isto é, $S_A - S_B$).

É possível realizar essa troca de informações a partir do envio de um *Bloom filter* com os elementos de S_B para o par A. Assim, A pode verificar quais de seus elementos não estão em B (de acordo com os testes de membresia do filtro) e transmiti-los para B. Note que, dada a possibilidade de ocorrência de resultados falsos positivos, alguns elementos que pertencem a $S_A - S_B$ podem não ser enviados para B.

3.5.4 Estimação da intersecção de conjuntos

O uso de *Bloom filters* para determinar intersecções entre conjuntos é proposto por Reynolds e Vahdat (2003) e assemelha-se à técnica utilizada para estimar a diferença entre conjuntos. Dados dois pares A e B que armazenam os conjuntos S_A e S_B, respectivamente, suponha que B requer os dados de S_A que também estão em S_B (isto é, $S_A \cap S_B$).

Uma aproximação para os elementos da interseção de S_A e S_B pode ser obtida a partir do envio de um *Bloom filter* com os elementos de S_B para A, o que diminui o uso da rede. Assim, A realiza testes de membresia com o filtro de S_B para determinar quais elementos de S_A também estão em S_B de acordo com o filtro e envia-os de volta a B. Dada a possibilidade de ocorrência de resultados falsos positivos, alguns elementos que não pertencem a S_B podem ser enviados para B como resultado de $S_A \cap S_B$. Ainda assim, B pode testar cada

um dos elementos do resultado de modo a eliminar aqueles que não pertencem a S_B, se necessário.

3.5.5 Protocolos de roteamento

Dada uma rede de nós em formato de **árvore radicada** onde cada nó detém uma lista de recursos, suponha que uma requisição por um recurso pode partir de qualquer nó dessa árvore. Cada nó deve ser capaz de responder se detém um recurso ou se é possível obtê-lo por algum de seus descendentes (assim como deve informar a lista de recursos de cada um de seus filhos). Assim, é possível decidir se uma requisição que navega por um dado nó encontrou o recurso ou se deve acessar um filho do nó ou subir um nível na árvore.

Nesse cenário, *Bloom filters* podem ser utilizados para representar as listas de recursos de cada nó. De fato, essa estrutura é especialmente vantajosa para aplicações desse tipo pois as listas de recursos dos filhos de um nó podem ser combinadas por meio de disjunções entre os binários que representam as listas de recursos de cada filho. Isto é, se A é o nó pai de B e C, cujas representações binárias por meio de *Bloom filters* são dadas por b e c, respectivamente, então a lista de recursos dos filhos de A é $b \vee c$ (ainda ocorrem apenas resultados falsos positivos, e não falsos negativos).

O **roteamento geográfico** é uma possível aplicação desse tipo de protocolo. Suponha que se deseja estabelecer um protocolo de trocas de mensagens para dispositivos móveis distribuídos em uma dada uma região quadrada de um espaço geográfico qualquer. Esse espaço é subdividido recursivamente em quatro quadrados de mesma área, o que resulta em vários níveis de hierarquia, como em uma árvore radicada.

Cada quadrado armazena um *Bloom filter* que deve representar o conjunto de dispositivos acessíveis por algum de seus filhos (subdivisões) ou irmãos (demais três quadrados). Assim, uma mensagem que parte de uma origem qualquer pode navegar pela árvore por meio de consultas aos filtros até que chegue à região de destino.

Capítulo 4

Cuckoo filters

O *cuckoo filter*, descrito por Fan *et al.* (2014), é proposto como uma alternativa ao *Bloom filter* tradicional para os cenários em que a remoção de elementos da estrutura é necessária. Para isso, utiliza-se o ***cuckoo hashing*** como um modo de construção do filtro, o que permite que elementos sejam removidos sem que seja necessário usar espaço adicional, como ocorre nos *counting Bloom filters*.

Há outras alternativas para os *Bloom filters* que permitem remoções de elementos (como os *counting Bloom filters* e *quotient filters*), mas que em contrapartida podem demandar espaço adicional para que se obtenha a mesma razão de falsos positivos de um *Bloom filter* tradicional. Ademais, as execuções de testes de membresia nessas estruturas têm, usualmente, desempenho inferior ao apresentado nos *Bloom filters*.

Destacam-se algumas vantagens do *cuckoo filter* sobre o *Bloom filter* tradicional e outros filtros que permitem remoções de elementos:

- Permite adicionar e remover itens dinamicamente;
- Fornece melhor desempenho de consultas do que o *Bloom filter* tradicional, mesmo quando está quase cheio (com 95% do espaço utilizado, por exemplo);
- É mais fácil de implementar do que alternativas como o *quotient filter*;
- Usa menos espaço que os *Bloom filters* em muitas aplicações práticas se a razão de falsos positivos ϵ alvo é menor que 3%. (Fan *et al.*, 2014, tradução nossa)

4.1 *Cuckoo hash tables*

Uma *cuckoo hash table* é um tipo especial de *hash table* que insere e remove elementos da estrutura usando a técnica de *cuckoo hashing*. Essa estrutura é composta por um vetor de *buckets* e duas funções de *hashing* $h_1(x)$ e $h_2(x)$ que mapeiam um dado elemento x para dois *buckets* da estrutura. Um *bucket* pode ser uma estrutura capaz de armazenar

um ou mais elementos (como um vetor), de modo que o funcionamento da estrutura é, essencialmente, o mesmo em qualquer caso.

4.1.1 Inserção

Caso algum dos dois *buckets* de índices dados pelas funções de *hashing* tenha espaço para um novo elemento, x é inserido no *bucket* disponível. Caso contrário, seleciona-se um dos dois *buckets* e x toma o lugar de um elemento anteriormente inserido. Nesse cenário, o elemento removido é **realocado**: o processo de inserção é executado novamente com o elemento removido. Realizam-se realocações sucessivas até que seja encontrado um *bucket* disponível (onde nenhum elemento precisa ser realocado) ou até que seja atingido um limite de realocações, onde a estrutura é considerada muito cheia para comportar o novo elemento.

Esse procedimento corresponde ao típico comando `INSERT(x)` das *hash tables* convencionais. A Figura 4.1 demonstra o processo de inserção de um elemento x em uma *cuckoo hash table*. Note que os dois *buckets* de x estão ocupados, então x é inserido no *bucket* de índice 1 e o elemento b é realocado para o *bucket* de índice 5. Como o *bucket* de índice 5 está ocupado, c é realocado para a posição 4, que está vazia – o que encerra o processo de inserção de x.

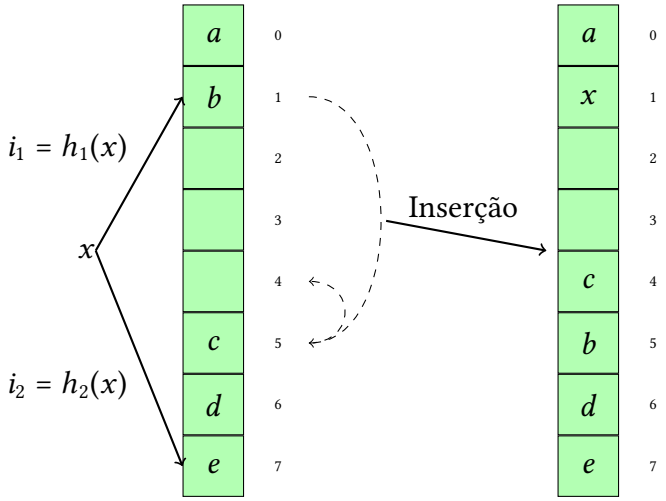

Figura 4.1: *Esquema do funcionamento da inserção de um elemento em uma cuckoo hash table*

Deste modo, todo elemento inserido na estrutura sempre está em algum dos dois *buckets* de índices dados pelas funções de *hashing*. Apesar do custo das realocações de elementos, **o tempo amortizado das inserções é** $O(1)$, de acordo com Fan *et al.* (2014).

4.1.2 Consulta

O comando `QUERY(x)` é aplicado a partir da verificação dos dois *buckets* de índices $i_1 = h_1(x)$ e $i_2 = h_2(x)$. O resultado da consulta é positivo caso o elemento seja encontrado em algum dos dois *buckets*, ou negativo caso contrário.

4.1.3 Remoção

Dado que um elemento sempre deve estar em algum dos dois *buckets* indicados pelas funções de *hashing*, uma remoção `REMOVE(x)` demanda apenas a verificação e exclusão de x de um dos dois *buckets*.

4.2 Definição

Um *cuckoo filter* é uma versão compacta de uma *cuckoo hash table* que deve armazenar ***fingerprints*** no lugar de elementos. *Fingerprints* são sequências de bits associadas a um elemento a partir de uma função de *hashing*. O comprimento dessa sequência de bits é determinado de acordo com a razão de falsos positivos ϵ, dado que podem ocorrer colisões nesse processo de compressão. Cada entrada dos *buckets* da estrutura devem comportar *fingerprints* do comprimento escolhido.

Um fator relevante na construção de *cuckoo filters* é a escolha de uma posição alternativa para cada elemento dado (isto é, a definição de $h_2(x)$). O *partial-key cuckoo hashing* é proposto por Fan *et al.* (2014) como uma forma de obter um alto nível ocupação da tabela, o que evita falhas na execução de inserções. Seja $f = \texttt{fingerprint}(x)$ e dada uma função de *hashing* $h(x)$, então a técnica de *partial-key cuckoo hashing* propõe que as funções de hashing escolhidas sejam:

$$h_1(x) = h(x) = i,$$

$$h_2(x) = h_1(x) \oplus h(f) = j.$$

A operação $\oplus$ é especialmente importante para assegurar que a partir de uma *fingerprint* f é possível obter o valor de $h_1(x)$ mesmo sem que o valor de x seja armazenado separadamente. Dado que um *fingerprint* f está armazenado em uma posição i, sua posição alternativa j pode ser facilmente obtida: $j = i \oplus h(f)$. De fato, se f está armazenado em $h_2(x)$, então $h_2(x) \oplus h(f) = h_1(x) \oplus h(f) \oplus h(f) = h_1(x)$ e se f está armazenado em $h_1(x)$, segue que $h_1(x) \oplus h(f) = h_2(x)$ da definição.

O esquema do funcionamento da inserção de um elemento em um *cuckoo filter* com a técnica de *partial-key cuckoo hashing* é apresentada na Figura 4.2.

O cálculo de $h_2(x)$ é ajustado de modo a evitar colisões e aprimorar a utilização da tabela. Usa-se $h(f)$ e não somente f para que mesmo elementos com *fingerprints* próximas tenham posições alternativas distribuídas pela tabela.

Ainda assim, é possível que ocorram colisões de *fingerprints*, o que é um problema caso as colisões ultrapassem o dobro do comprimento do *bucket*. Assim, todas as posições primárias e alternativas para esses elementos tornam-se ocupadas e não é possível realocá-los. Essas colisões também podem ocasionar resultados falsos positivos, dado que um elemento que não foi inserido na estrutura pode ser mapeado para a *fingerprint* de um item que está no filtro.

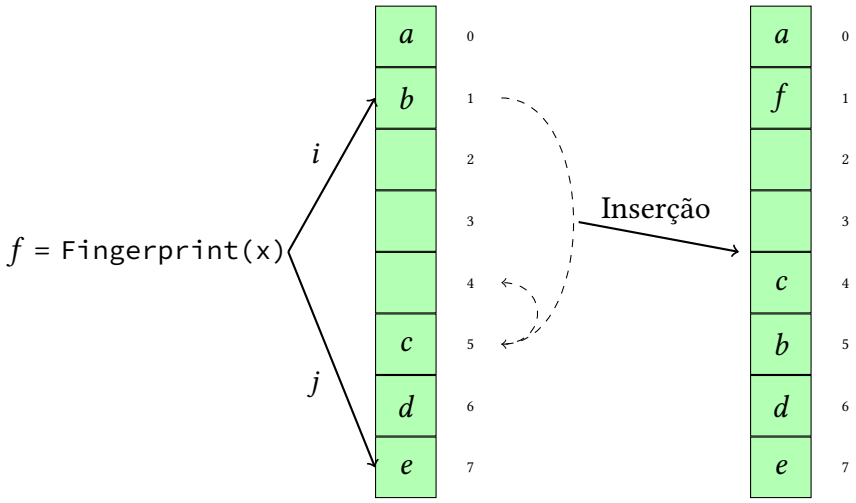

Figura 4.2: *Esquema do funcionamento da inserção de um elemento em um Cuckoo filter com a técnica de partial-key cuckoo hashing*

4.3 Algoritmos

Um *Cuckoo filter* permite realizar operações de inserção, consulta e **remoção** de elementos.

4.3.1 Inserção

O Algoritmo 3 descreve o procedimento para a inserção de um elemento em um *Cuckoo filter*, onde deve ser dado um número máximo de realocações até que a estrutura seja considerada cheia. O algoritmo emprega a técnica de *partial-key cuckoo hashing* para a determinação dos índices primário e alternativo e armazena apenas o valor calculado para a *fingerprint* na tabela.

Ademais, note que em caso de necessidade de realocação, escolhe-se aleatoriamente o *bucket* que deve ter um elemento realocado e, também, escolhe-se o elemento dentro do *bucket* aleatoriamente.

4.3.2 Consulta

O Algoritmo 4 descreve o procedimento para a realização do teste de membresia de um elemento em um *Cuckoo filter* com a técnica de *partial-key cuckoo hashing*. Dado que todo elemento x inserido na estrutura está sempre armazenado em algum dos dois *buckets* de índices $h_1(x)$ e $h_2(x)$, basta verificar se a *fingerprint* está armazenada em alguma entrada desses *buckets*. Dessa forma, resultados falsos negativos não ocorrem desde que não se exceda o tamanho do bucket (o que é garantido pelo funcionamento do algoritmo).

4.3.3 Remoção

O Algoritmo 5 descreve o procedimento para a remoção de um elemento em um *Cuckoo filter* com a técnica de *partial-key cuckoo hashing*, o que não é permitido por um *Bloom*

Algoritmo 3: Cuckoo Filter: Inserção (`Insert`)

Entrada: x - elemento a ser inserido na estrutura
Saída: `Verdadeiro` ou `Falso` - indica se o elemento foi inserido na estrutura

```
1  f ← FINGERPRINT(x)
2  i_1 ← HASH(x)
3  i_2 ← i_1 ⊕ HASH(f)
4  se bucket[i_1] ou bucket[i_2] tem uma entrada vazia então
5  |  insere f em um bucket com entrada vazia
6  |  retorna Verdadeiro
7  fim
8  i ← escolha aleatória entre i_1 e i_2
9  para n ← 0 até MaxNumRealocações faça
10 |  e ← escolha aleatória de uma entrada de bucket[i]
11 |  troca os conteúdos de f e da entrada e de bucket[i]   ▷ Realoca um elemento
12 |  i ← i ⊕ HASH(f)
13 |  se bucket[i] está vazio então
14 |  |  insere f em bucket[i]
15 |  |  retorna Verdadeiro
16 |  fim
17 fim
18 retorna Falso                    ▷ Considera-se que a hash table está cheia
```

Algoritmo 4: Cuckoo Filter: Consulta (`Query`)

Entrada: x - elemento a ser consultado na estrutura (teste de membresia)
Saída: `Verdadeiro` ou `Falso` - indica se o elemento está na estrutura

```
1  f ← FINGERPRINT(x)
2  i_1 ← HASH(x)
3  i_2 ← i_1 ⊕ HASH(f)
4  se bucket[i_1] ou bucket[i_2] tem uma entrada com o conteúdo f então
5  |  retorna Verdadeiro
6  fim
7  retorna Falso
```

filter tradicional. O processo de remoção de um elemento x é ilustrado na Figura 4.3.

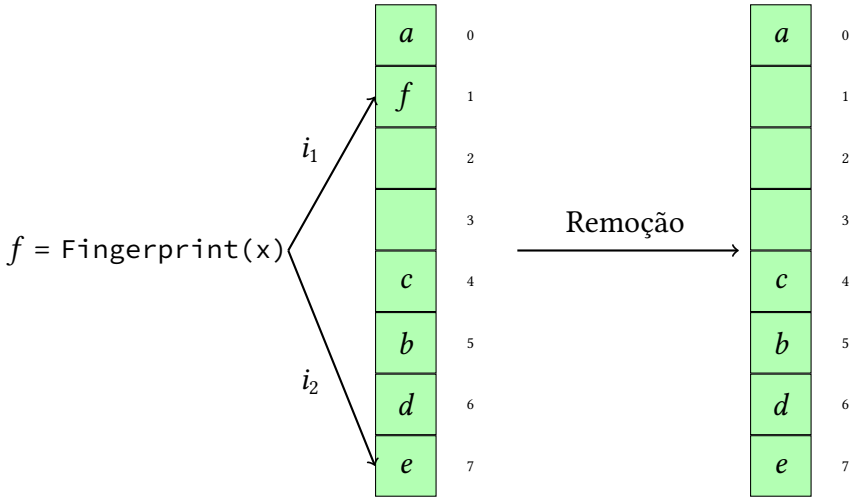

Figura 4.3: *Esquema do funcionamento da remoção de um elemento em um Cuckoo filter*

Assim como na consulta, para a remoção de um elemento x, basta verificar se a *fingerprint* está armazenada em alguma entrada dos *buckets* de índices $h_1(x)$ ou $h_2(x)$ e, então, **remover uma entrada** do elemento. Note que apenas uma entrada do item deve ser removida do *bucket*, dado que podem ocorrer colisões de *fingerprints* – portanto, nem sempre uma *fingerprint* está associada a um único elemento. Por conta disso, idealmente, a remoção só pode ser executada para elementos que foram inseridos na estrutura, caso contrário há risco de remoção de outro item cujo *fingerprint* colide com o elemento dado.

Algoritmo 5: Cuckoo Filter: Remoção (`Delete`)

Entrada: x - elemento a ser removido da estrutura
Saída: `Verdadeiro` ou `Falso` - indica se o elemento foi removido da estrutura

```
1 f ← FINGERPRINT(x)
2 i1 ← HASH(x)
3 i2 ← i1 ⊕ HASH(f)
4 se bucket[i1] ou bucket[i2] tem uma entrada com o conteúdo f então
5 |   remove o conteúdo de uma entrada que contém f
6 |   retorna Verdadeiro
7 fim
8 retorna Falso
```

Capítulo 5

Experimentos

De acordo com Bloom (1970), a aplicação de *Bloom filters* é especialmente vantajosa em algoritmos hifenizadores de palavras. Essa estrutura pode ser aplicada de modo a representar o conjunto de exceções às regras usuais de hifenização de cada idioma. Para reproduzir a utilização do `hypher`[1] implementação de *Bloom filters* e *cuckoo filters* em Javascript[2]

Todos os experimentos foram realizados em uma máquina com processador AMD Ryzen 7 3700U (2.30 GHz) com 20GB de memória RAM e sistema operacional Windows 10 de 64 bits.

5.1 Análise do desempenho de *Bloom filters*

Texto

5.2 Análise do desempenho de *Cuckoo filters*

Texto

5.3 Análise comparativa do desempenho de *Bloom filters* e *Cuckoo filters*

Texto

[1] Disponível em: https://github.com/bramstein/hypher

[2] Disponível em: https://github.com/Callidon/bloom-filters

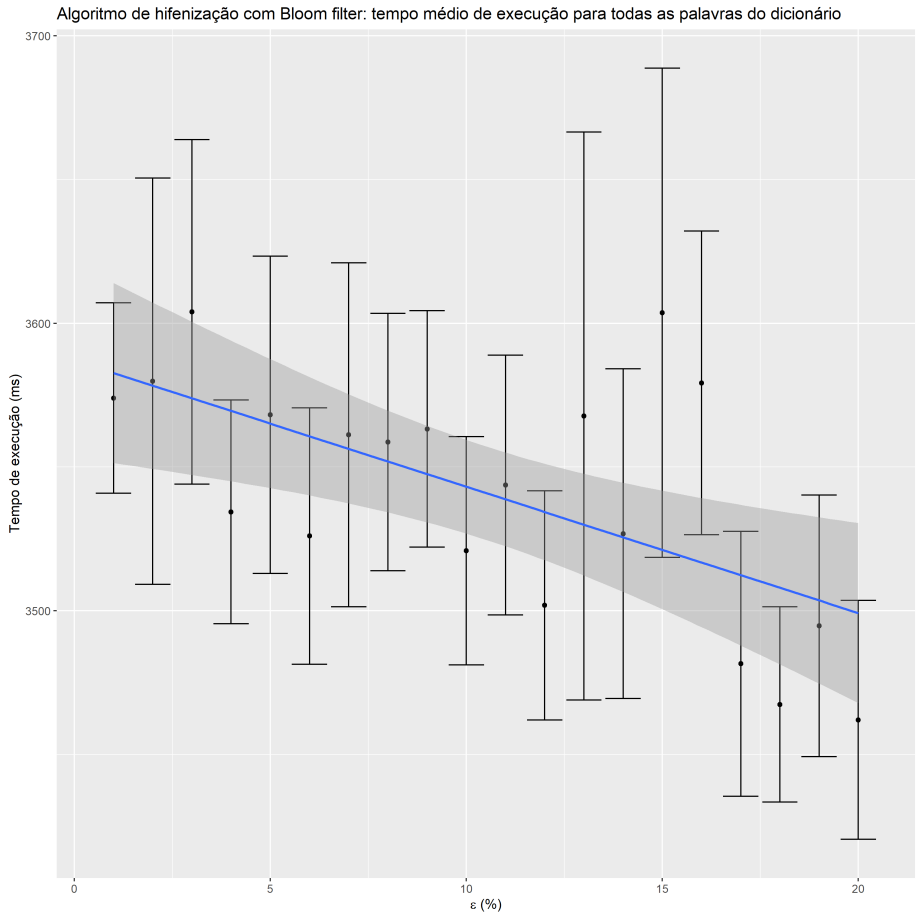

Figura 5.1: *Tempo médio de processamento de uma mesma lista de palavras em um algoritmo de hifenização implementado com um Bloom filter (com intervalos de confiança de 95%)*

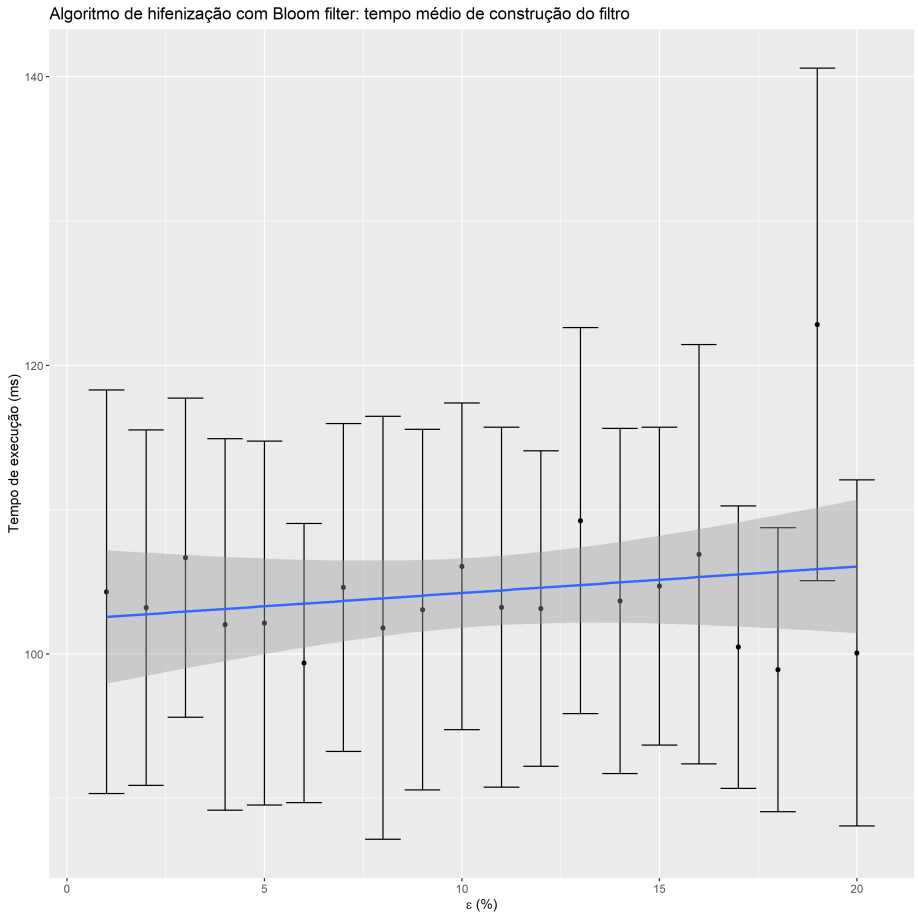

Figura 5.2: *Tempo médio de construção de um Bloom filter para uma mesma lista de exceções em um algoritmo hifenizador de palavras (com intervalos de confiança de 95%)*

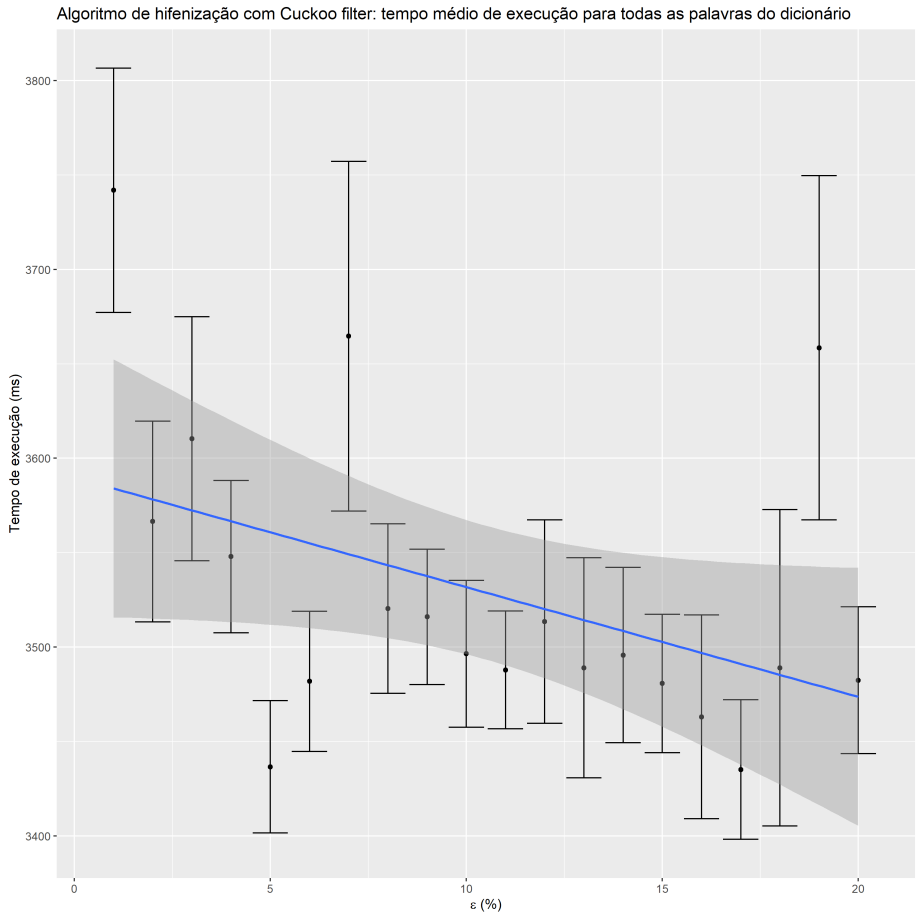

Figura 5.3: *Tempo médio de processamento de uma mesma lista de palavras em um algoritmo de hifenização implementado com um cuckoo filter (com intervalos de confiança de 95%)*

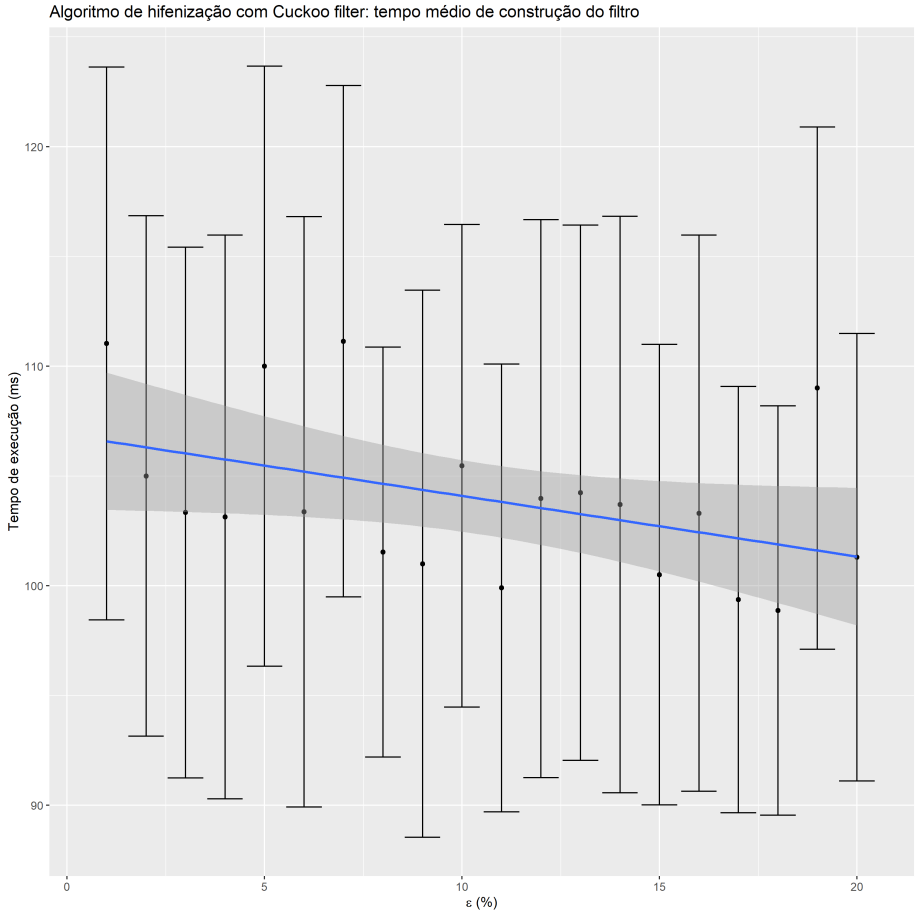

Figura 5.4: *Tempo médio de construção de um cuckoo filter para uma mesma lista de exceções em um algoritmo hifenizador de palavras (com intervalos de confiança de 95%)*

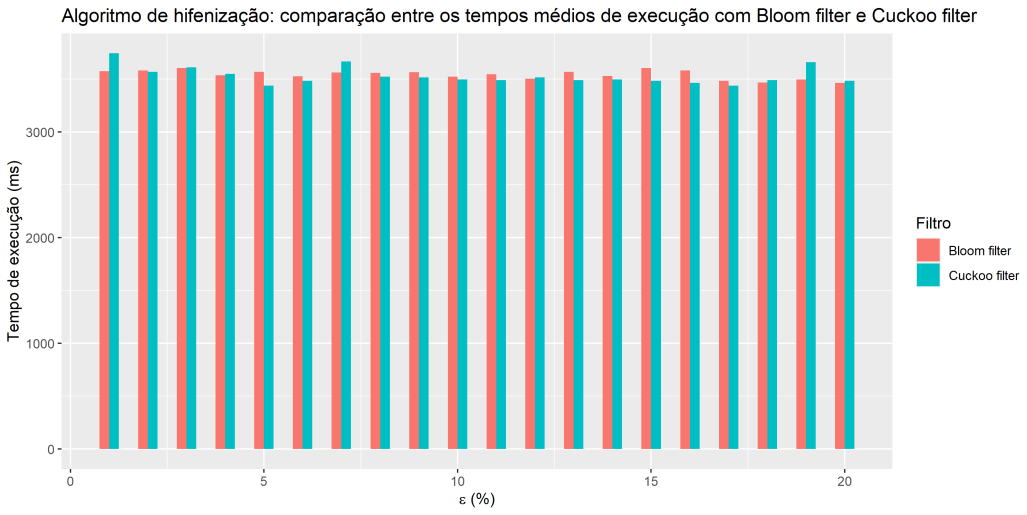

Figura 5.5: *Comparação entre os tempos médios de processamento de uma mesma lista de palavras em um algoritmo hifenizador de palavras implementado com Bloom filter e com cuckoo filter*

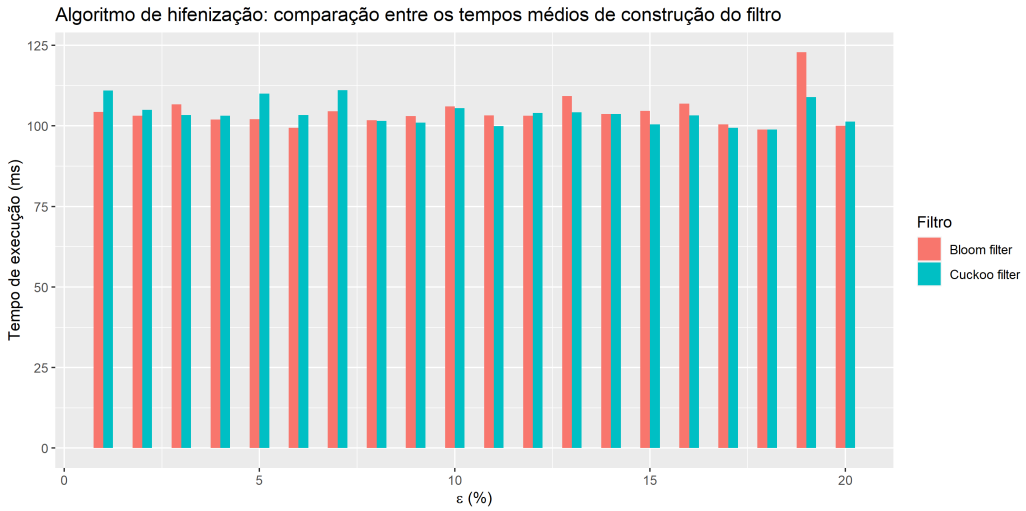

Figura 5.6: *Comparação entre os tempos médios de construção do filtro para uma mesma lista de exceções em um algoritmo hifenizador de palavras implementado com Bloom filter e com cuckoo filter*

Capítulo 6

Conclusão

Texto

Referências

[Bloom 1970] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". Em: *Commun. ACM* 13.7 (jul. de 1970), pgs. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692 (citado nas pgs. 7, 11, 21).

[Broder e Mitzenmacher 2003] Andrei Broder e Michael Mitzenmacher. "Survey: network applications of bloom filters: a survey." Em: *Internet Mathematics* 1 (nov. de 2003). DOI: 10.1080/15427951.2004.10129096 (citado na pg. 7).

[Fan *et al.* 2014] Bin Fan, Dave G. Andersen, Michael Kaminsky e Michael D. Mitzenmacher. "Cuckoo filter: practically better than bloom". Em: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, pgs. 75–88. ISBN: 9781450332798. DOI: 10.1145/2674005.2674994. URL: https://doi.org/10.1145/2674005.2674994 (citado nas pgs. 15–17).

[Mosharraf e Adnan 2022] Sharafat Ibn Mollah Mosharraf e Muhammad Abdullah Adnan. "Improving lookup and query execution performance in distributed big data systems using cuckoo filter". Em: *Journal of Big Data* 9.1 (jan. de 2022), pg. 12. ISSN: 2196-1115. DOI: 10.1186/s40537-022-00563-w. URL: https://doi.org/10.1186/s40537-022-00563-w.

[Reynolds e Vahdat 2003] Patrick Reynolds e Amin Vahdat. "Efficient peer-to-peer keyword searching". Em: *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Middleware '03. Rio de Janeiro, Brazil: Springer-Verlag, 2003, pgs. 21–40. ISBN: 3540403175 (citado na pg. 12).

[szabowexler 2014] Elias Szabo-Wexler szabowexler. "Approximate membership of sets : a survey". Em: 2014.

[Tarkoma *et al.* 2012] Sasu Tarkoma, Christian Esteve Rothenberg e Eemil Lagerspetz. "Theory and practice of bloom filters for distributed systems". Em: *IEEE Communications Surveys & Tutorials* 14.1 (2012), pgs. 131–155. DOI: 10.1109/SURV.2011.031611.00024.