Estudo do efeito de variações de Bloom filters no desempenho de consultas no banco de dados Apache Cassandra

Matheus Barbosa Silva*

Resumo

Uma variante do arquivo `tese.tex` usando a classe article.

1 Introdução

Se você precisa criar um texto relativamente curto, como um artigo ou um trabalho de disciplina, este modelo pode servir como base. Observe, no entanto, que periódicos em geral nas áreas de matemática e computação costumam ter seus próprios modelos LaTeX (como é o caso da SBC); nesses casos, é melhor utilizá-los e apenas consultar este modelo para verificar como usar algum recurso específico. Fique atento: alguns modelos antigos ou de periódicos internacionais podem usar latin1 ao invés de utf8 ou mesmo não ter configuração pré-definida para caracteres acentuados. Além disso, eles muito frequentemente utilizam bibtex ao invés de biblatex para a geração automática da bibliografia.

* Instituto de Matemática e Estatística da Universidade de São Paulo

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.

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*);

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 1.

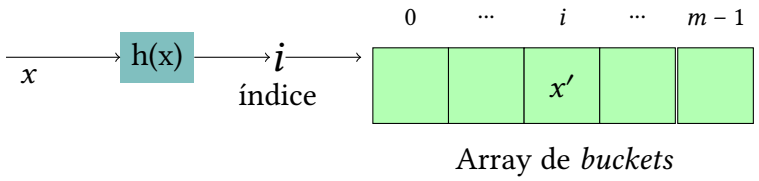

Figura 1: *Esquema do funcionamento de um dicionário implementado com hash table*

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.

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.

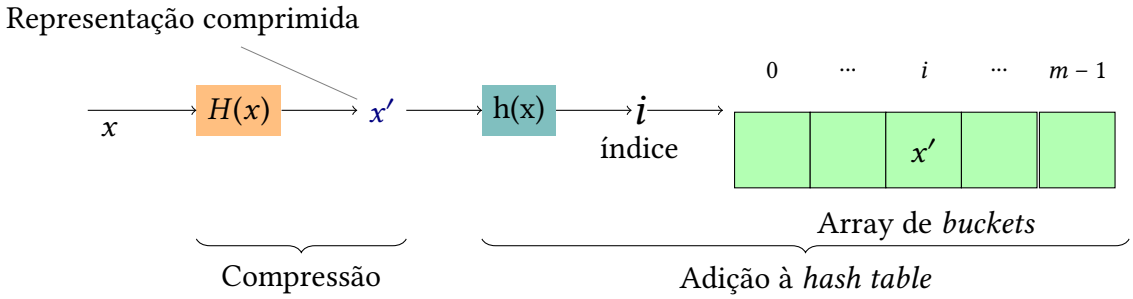

Figura 2: *Esquema do funcionamento de um dicionário implementado com hash compaction*

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).

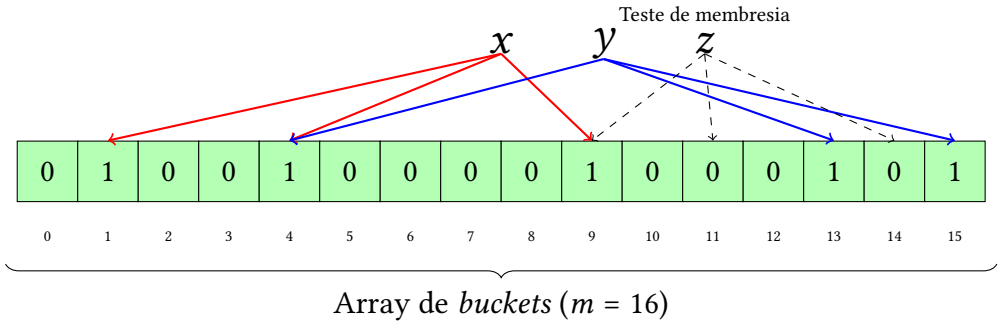

Figura 3: *Esquema do funcionamento de um Bloom filter com parâmetros $m = 16$, $k = 3$, $n = 2$*

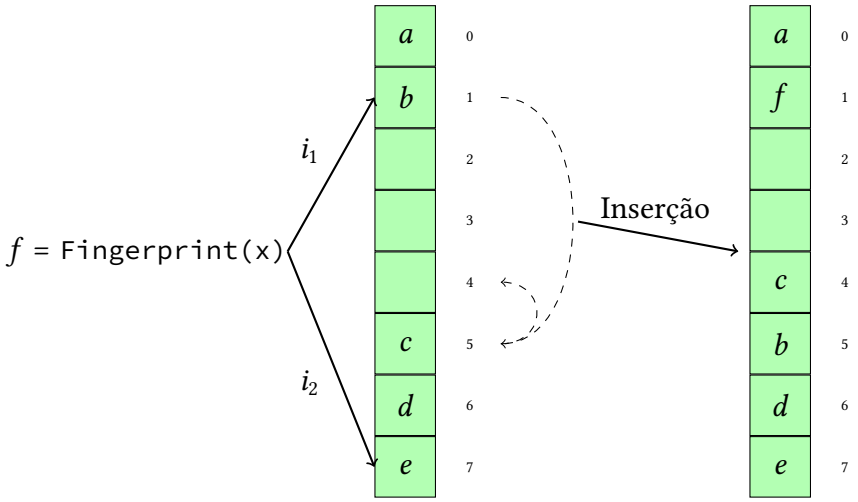

Figura 4: *Esquema do funcionamento da inserção de um elemento em um Cuckoo filter*

3 *Bloom filters*

1. **Introdução**: criador, objetivos e aplicações vantajosas dessa estrutura de dados (de forma geral);
2. Princípio do *Bloom filter*;
3. Dinâmica de funcionamento (construir figuras), evidenciar a possiblidade de falsos positivos e expor as susposições básicas de movo a evitar alguns casos triviais. **Limitações:** não permite remoção de elementos;
4. Uso de Bloom filter para estimar intersecções (aplicações em BD) com n elementos
5. Cálculo da razão de falsos positivos (*false positive rate*)

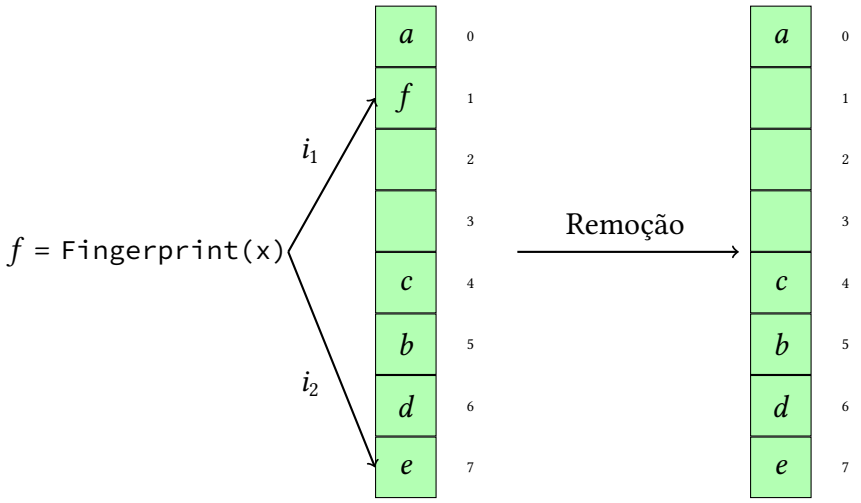

Figura 5: *Esquema do funcionamento da remoção de um elemento em um Cuckoo filter*

6. Cálculo de quantos bits são necessários para representar um conjunto
7. Algoritmos

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 bucket[i] = 0 então
4 |   |   bucket[i] ← 1
5 |   fim
6 fim
```

(a) Inserção

(b) Consulta

(c) Remoção (não há suporte nos *Bloom filters* padrões, usar como motivação para a exposição dos *Cuckoo Filters*)

4 *Cuckoo filters*

1. **Introdução**: objetivos e vantagens sobre Bloom filters;
2. *Cuckoo Hash Tables*: resultados das análises matemáticas;
3. hashing de chave parcial
   (a) Inserção
   (b) Consulta
   (c) Remoção

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

para $j \leftarrow 1$ até k **faça**
    $i \leftarrow \text{HASH}_j(x)$
    se $bucket[i] = 0$ **então**
        retorna *Falso*
    fim
fim
retorna *Verdadeiro*

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

$f \leftarrow \text{FINGERPRINT}(x)$
$i_1 \leftarrow \text{HASH}(x)$
$i_2 \leftarrow i_1 \oplus \text{HASH}(f)$
se $bucket[i_1]$ ou $bucket[i_2]$ tem uma entrada vazia **então**
    insere f em um bucket com entrada vazia
    retorna *Verdadeiro*
fim
$i \leftarrow$ escolha aleatória entre i_1 e i_2
para $n \leftarrow 0$ até *MaxNumRealocações* **faça**
    $e \leftarrow$ escolha aleatória de uma entrada de `bucket`$[i]$
    troca os conteúdos de f e da entrada e de `bucket`$[i]$ ▷ Realoca um elemento
    $i \leftarrow i \oplus \text{HASH}(f)$
    se $bucket[i]$ está vazio **então**
        insere f em `bucket`$[i]$
        retorna *Verdadeiro*
    fim
fim
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

$f \leftarrow \text{FINGERPRINT}(x)$
$i_1 \leftarrow \text{HASH}(x)$
$i_2 \leftarrow i_1 \oplus \text{HASH}(f)$
se $bucket[i_1]$ ou $bucket[i_2]$ tem uma entrada com o conteúdo f **então**
    retorna *Verdadeiro*
fim
retorna *Falso*

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 $f \longleftarrow \text{FINGERPRINT}(x)$
2 $i_1 \longleftarrow \text{HASH}(x)$
3 $i_2 \longleftarrow i_1 \oplus \text{HASH}(f)$
4 **se** $bucket[i_1]$ ou $bucket[i_2]$ 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*

Referências

[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.

[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.

[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.