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. Já na variante dinâmica, inserções e consultas podem estar intercaladas.

Para ambas as variantes do problema, a construção de uma estratégia eficiente que permita obter uma **resposta determinística** para os *membership tests* pode não ser imediata. Uma solução intuitiva, mas que demanda o consumo linear de espaço é a comparação do elemento a ser verificado com cada um dos elementos do conjunto. No entanto, em alguns cenários, a escalabilidade é um requisito fundamental, portanto é desejável que se evite o consumo linear de espaço.

Assim, propõe-se o *approximate set membership* (teste de membresia aproximado) como uma **solução probabilística** que permite responder consultas sobre um conjunto de forma aproximada, com o custo de resultados falsos positivos. Um resultado falso positivo ocorre com probabilidade ϵ, logo $0 < \epsilon < 1$.

2.1 Implementação de Dicionário Completo

Dicionários completos são estruturas de dados que apresentam uma entrada para cada elemento de um conjunto (isto é, há enumeração completa dos elementos).

Interface

Um dicionário completo 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. Também escolhe-se uma função de *hashing* h, $h : \mathcal{U} \rightarrow \{0, 1, ..., m\}$ 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 identificados individualmente. 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 $x \notin S$.

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 $\log_2 \binom{u}{n} = n \log_2 \frac{u}{n} + \theta(n)$. [1]

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 também $O(1)$ (como a implementação de uma lista ligada, que permite a inserção de novos elementos em tempo $\theta(1)$). [2] No entanto, elementos com comprimentos grandes 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*, introduzida 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 dos blocos na estrutura.

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

[1] Pretendo destrinchar melhor essa passagem

[2] citar fonte ou demonstrar

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 $x \notin S$.