Struktur Data Probabilistik:
Saringan Bloom

Jika kita memiliki rak buku yang dilindungi kaca, itu akan melindungi buku-buku dari debu dan serangga (termasuk rayap), tetapi kita tentu harus membayar lebih banyak waktu untuk mengakses buku-buku tersebut saat membutuhkannya. Karena kita terlebih dahulu perlu menggeser atau membuka kaca dan setekah itu baru dapat memperoleh buku. Di sisi lain, jika itu adalah rak buku terbuka maka akan memberi akses lebih cepat tetapi kita akan kehilangan perlindungan terhadap buku. Demikian pula, jika kita mengatur buku-buku dalam urutan leksikografis berdasarkan namanya, kita dapat dengan mudah mencari buku jika tahu namanya. Tetapi jika rak buku memiliki tinggi dengan ukuran berbeda dan mengatur buku berdasarkan ukurannya, itu akan terlihat bagus, tetapi dapatkah kita menemukan buku dengan cepat dan tergesa-gesa? Kemungkinan besar tidak.

Struktur data tidak ada bedanya. Ini seperti rak buku aplikasi di mana kita dapat mengatur data yang dimiliki. Struktur data yang berbeda akan memberikan fasilitas dan manfaat yang berbeda. Untuk menggunakan kekuatan dan aksesibilitas dari struktur data dengan benar, kita perlu mengetahui kelebihan dan kekurangan dari penggunaannya.

Struktur data *mainstream* seperti List, Map, Set, Tree, dll. banyak digunakan untuk mencapai hasil tertentu berkaitan dengan ada atau tidaknya data, mungkin bersama dengan jumlah kemunculannya dan semacamnya. Struktur data probabilistik akan memberikan efisiensi memori, hasil yang lebih cepat, bentuk hasil lebih ke 'mungkin' daripada 'pasti'. Tampaknya tidak intuitif untuk menggunakan struktur data probabilistik untuk saat ini, tetapi tutorial ini akan mencoba meyakinkan bahwa jenis struktur data ini memiliki tempat pemanfaatan khusus dan kita mungkin menemukannya berguna dalam skenario tertentu.

Dalam posting ini, kita akan mendiskusikan tentang salah satu struktur data probabilistik paling populer yang disebut ***Bloom filter***.

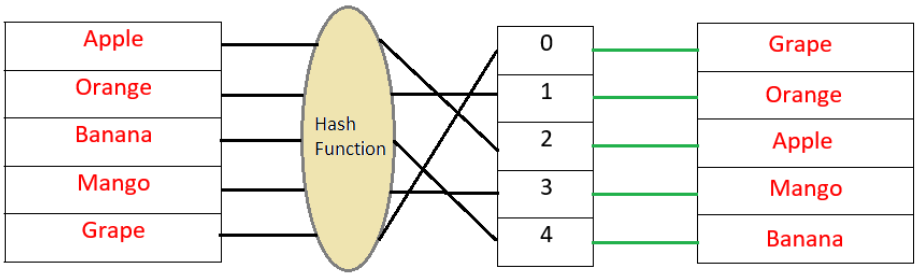
Saringan Bloom

Bagaimana cara kerja dari *hash tables*?

Pada saat kita memasukkan suatu data baru ke dalam array atau List sederhana, maka indeks dimana data ini akan disisipkan tidak ditentukan dari nilai yang dimasukkan. Artinya tidak ada relasi langsung antara ‘key(index)’ dan ‘value(data)’. Akibatnya, jika kita perlu mencari suatu nilai (*value*) di dalam array maka kita harus mencarinya dalam semua indeks. Pada hash tables, kita harus menentukan ‘key’ atau ‘index’ dengan melakukan *hashing* terhadap ‘value’. Kemudian value ini diletakkan di dalam index tersebut dalam bentuk List misalnya. Itu berarti ‘key’ ditentukan dari ‘value’ dan setiap kali kita perlu memeriksanya jika value eksis (ada) di dalam List yang baru saja dihash valuenya dan mencari pada key tersebut. Tabel hash ini sangat cepat dan waktu pencarian yang dibutuhkan hanya *O(1)* dalam notasi Big-O.

0	Apple
1	Orange
2	Banana
3	Mango
4	Grape

Array



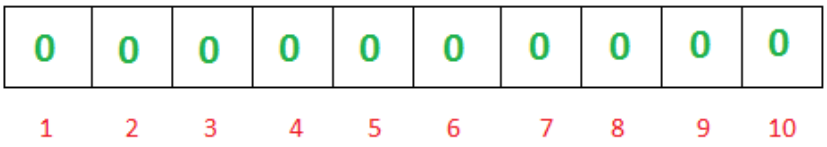
Hash Table

Sekarang, mari pertimbangkan bahwa kita memiliki daftar besar kata sandi (*password*) yang lemah dari sisi keamanan dan disimpan di beberapa server jarak jauh (*remote*). Tidak mungkin memuatnya sekaligus ke dalam memori / RAM karena ukurannya. Setiap kali pengguna memasukkan kata sandi maka dilakukan pemeriksaan apakah itu salah satu kata sandi yang lemah dan jika Ya maka sistem akan memberikan peringatan untuk mengubahnya menjadi sesuatu yang lebih kuat. Apa yang dapat dilakukan? Karena kita sudah memiliki daftar kata sandi yang lemah, kita dapat menyimpannya di tabel hash atau sesuatu seperti itu dan setiap kali diperlukan mencocokkan, maka dapat memeriksanya jika kata sandi yang diberikan memiliki kecocokan. Pencocokan mungkin cepat tetapi biaya pencarian pada disk atau melalui jaringan pada server jauh akan membuat proses ini lambat. Jangan lupa bahwa kita harus melakukannya untuk setiap kata sandi yang dimasukkan oleh setiap pengguna. Bagaimana kita dapat mengurangi *overhead* ini?

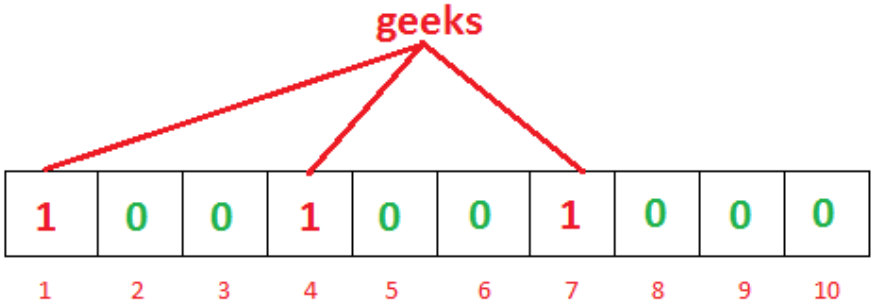
Well, Bloom filter dapat menyelesaikan masalah di atas. Caranya? Kita akan tahu jawabannya setelah memahami bagaimana suatu *bloom filter* bekerja. OK?

Secara definisi, Bloom filter dapat digunakan untuk memeriksa apakah suatu value **'mungkin ada dalam himpunan'** (*possibly in the set*) atau **'pasti tidak dalam himpunan'** (*definitely not in the set*). Perbedaan tajam antara **'mungkin** dan **'pasti tidak'** sangatlah penting sekali di sini. Kalimat 'mungkin ada di dalam himpunan*'* ini secara pasti mengapa dinamakan **probabilistik**. Menggunakan kata-kata smart itu berarti bahwa ***false positive*** adalah mungkin *(dapat terjadi dimana salah berpikir bahwa elemen tersebut positif)* tetapi ***false negative*** tidaklah mungkin. *Don't be impatient*, kita akan melihat apa sesungguhnya maksud kalimat dalam paragraf ini.

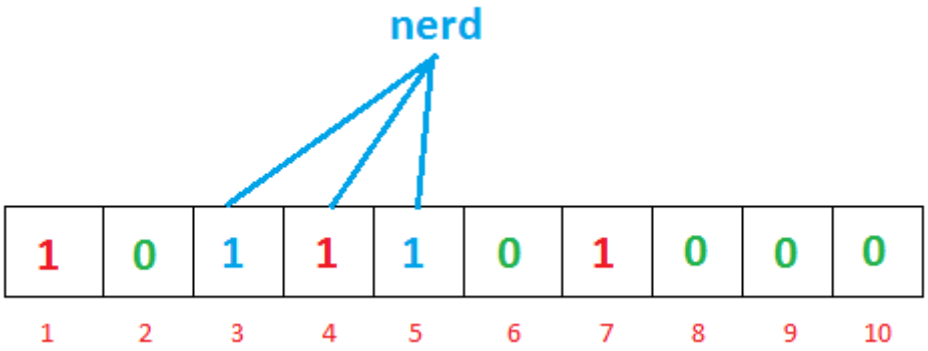
Pada dasarnya *bloom filter* terdiri dari suatu vektor bit atau daftar bit *(suatu list yang mengandung hanya nilai bit 0 atau 1-bit)* dengan panjang **m** yang pada awalnya semua nilai diset ke 0, sebagaimana ditunjukkan di bawah ini.



Untuk menambahkan suatu item ke dalam *bloom filter*, kita mengumpankan (*feed*) *inputan* tersebut ke **k** fungsi hash berbeda dan mengset (mengubah) bit ke nilai '1' pada posisi hasil dari fungsi hash. Seperti dapat dilihat, dalam hash tables kita menggunakan suatu fungsi hash tunggal dan sebagai hasilnya mendapatkan hanya sebuah indeks tunggal sebagai output. Tetapi dalam kasus *bloom filter*, akan digunakan banyak fungsi hash yang akan memberikan kita banyak indeks.



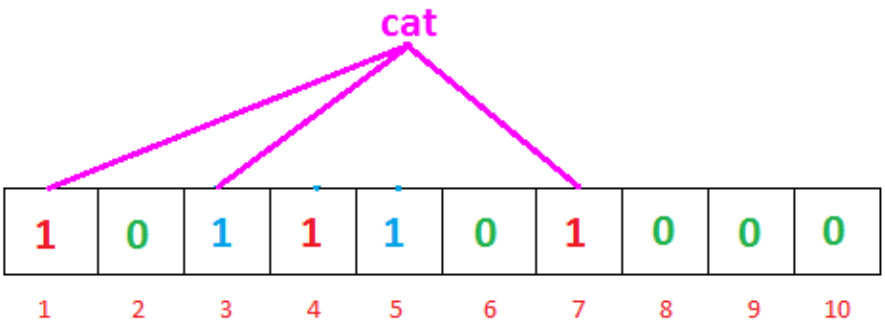
Pada gambar di atas, untuk input 'geeks' yang diberikan, 3 fungsi hash yang diterapkan memberikan 3 output berbeda: 1, 4 dan 7. Ketiga posisi tersebut ditandai (diberi nilai 1).



Saat ada input lain, misalnya teks 'nerd', fungsi hash memberikan kita hasil posisi 3, 4 dan 5 (misal). Kita dapat memperhatikan bahwa indeks '4' sudah ditandai oleh input 'geeks' sebelumnya. Tahan gagasan anda, inilah titik yang menarik dan akan segera didiskusikan.

Dua input yang diberikan telah mendiami vektor bit, sekarang kita akan memeriksa kehadiran suatu *value* di dalam vektor bit. Bagaimana ini dapat dilakukan?

Gampang. Sama persis seperti telah dilakukan terhadap hash table. Kita akan men-hash 'input yang dicari' dengan 3 fungsi hash yang sama dan melihat apa indeks yang dihasilkan.



Misal pada pencarian teks ‘cat’, fungsi hash mungkin memberikan posisi 1, 3 dan 7. Dan kita dapat melihat bahwa semua indeks telah ditandai dengan 1. Ini berarti dapat dikatakan bahwa *“mungkin ‘cat’ telah dimasukkkan dalam list tersebut”*. Kenyataannya? ‘cat’ tidak ada di dalam vektor. Jadi, apa yang salah? Sebetulnya tidak ada yang salah. Inilah yang disebut kasus **‘false positive’**. ***Bloom filter*** sedang memberitahukan kita bahwa terlihat bahwa mungkin ‘cat’ sudah disisipkan sebelumnya, karena indeks sebagai posisi hasil ‘cat’ ternyata sudah ditandai (lebih dulu oleh data berbeda).

So, jika kasusnya demikian, bagaimana ini bermanfaat? Well, mari kita perhatikan jika ‘cat’ memberikan output 1, 6, 7 bukan 1, 3, 7, apa yang akan terjadi kemudian? Kita dapat melihat bahwa di antara 3 indeks, posisi 6 bernilai ‘0’ yang berarti tidak ditandai oleh suatu input sebelumnya. Itu berarti dengan jelas ‘cat’ tidak pernah dimasukkan sebelumnya, jika sudah ada maka tidak ada kesempatan 6 bernilai ‘0’, ya kan? Itulah bagaimana *bloom filter* dapat memberitahukan **‘kepastian’** jika suatu data tidak berada di dalam list tersebut.

So, secara singkat:

- Jika kita mencari suatu nilai dan melihat indeks hasil hash untuk nilai ini adalah ‘0’ maka nilai itu dengan pasti tidak ada di dalam list tersebut.
- Jika semua indeks hasil hash memberikan bit ‘1’ maka ‘boleh jadi/mungkin’ nilai yang dicari ada di dalam list itu.

Apakah itu mulai masuk akal? baru sedikit?

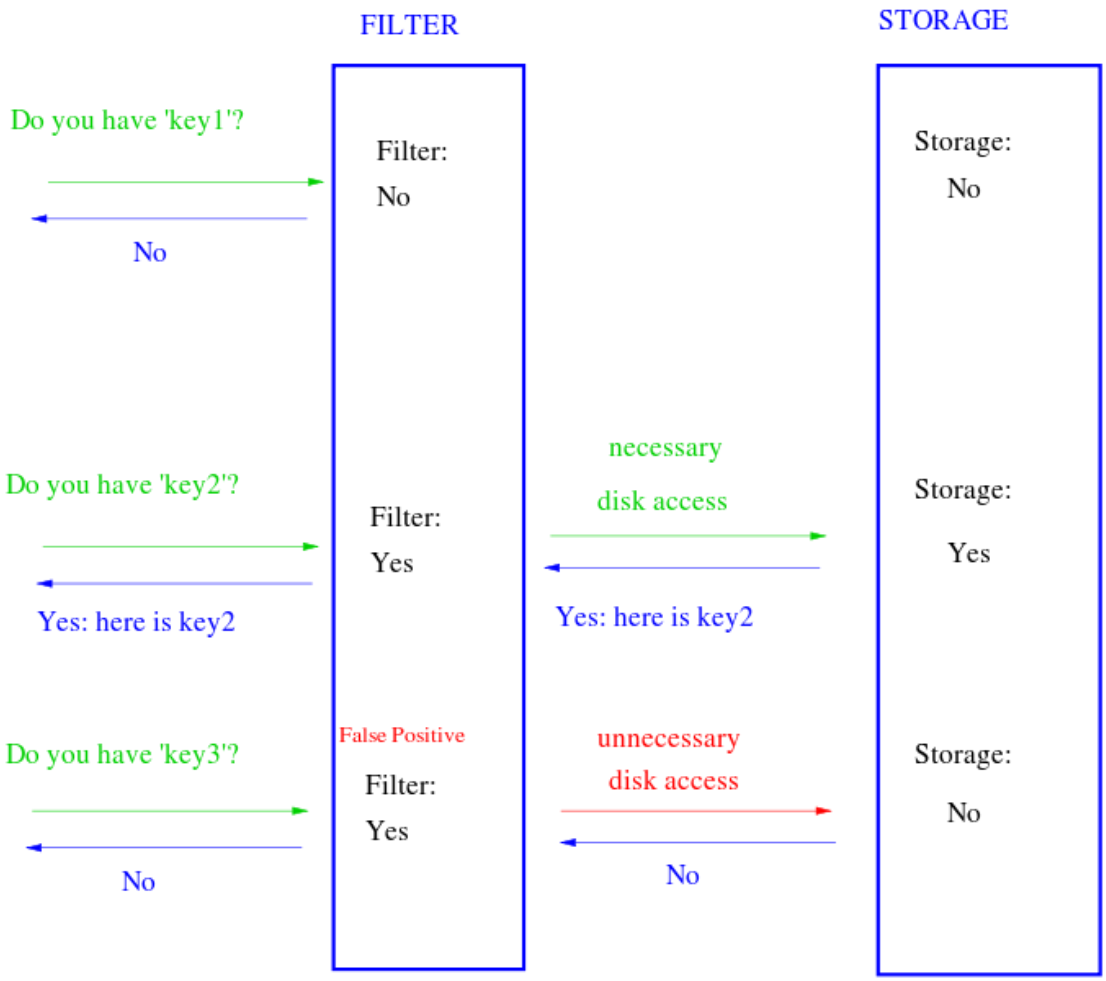
Baiklah, sekarang kita kembali ke contoh ‘password’ yang dibicarakan di awal tulisan ini. Jika kita mengimplementasikan pemeriksaan password lemah (*weak password checking*) dengan jenis bloom filter ini, kita dapat melihat bahwa pada awalnya kita akan menandai bloom filter dengan daftar passwords kita yang akan memberikan vektor bit dengan beberapa indeks ditandai sebagai ‘1’ dan sisanya dibiarkan sebagai 0. Karena ukuran dari bloom filter tidak akan sangat besar dan berukuran tetap, maka itu dapat dengan mudah disimpan dalam memory dan juga pada sisi client jika diperlukan. Itulah mengapa bloom filter sangat efisien dalam pemanfaatan ruang. Suatu hash table memerlukan ukuran berubah-ubah berdasarkan pada data input, sedangkan bloom filter dapat bekerja baik dengan ukuran tetap.

So, setiap kali seorang pengguna memasukkan passwordnya, kita akan mengumpan password itu ke fungsi hash tertentu dan memeriksa (membandingkan dengan) vektor bit. Jika password tersebut cukup kuat, maka bloom filter akan menunjukkan kita bahwa password pasti tidak berada dalam “daftar password lemah” (nilai bit 0) dan kita tidak harus melakukan query apapun lagi. Tetapi jika password tersebut terlihat lemah dan memberikan suatu hasil ‘positive’

(mungkin berupa *false positive*) kita kemudian akan mengirimkannya ke server dan memeriksa daftar aktual untuk konformasi.

Sebagaimana dapat dilihat, sebagian besar dari waktu tidak perlu membuat request ke server atau membaca dari disk untuk memeriksa daftar tersebut sehingga ini sangat signifikan meningkatkan kecepatan dari aplikasi. Jika suatu sistem tidak ingin menyimpan vektor bit pada sisi client, kita masih dapat memuatnya dalam memory server dan itu akan setidaknya menghemat beberapa waktu *disk lookup*. Juga pertimbangkan bahwa jika bloom filter memiliki false positive rate sekitar 1% (akan dibicarakan mengenai error rate secara rinci nanti), ini berarti antara *costly round-trips* ke server atau disk, hanya 1% dari query akan dikembalikan dengan hasil false, selain itu 99% tidak akan sia-sia.

Tidak jelek bukan?



Simulasi visual yang bagus mengenai bagaimana bloom filters bekerja

Operasi Saringan Bloom

Saringan Bloom dasar mendukung setidaknya dua operasi, yaitu: **test** dan **add**.

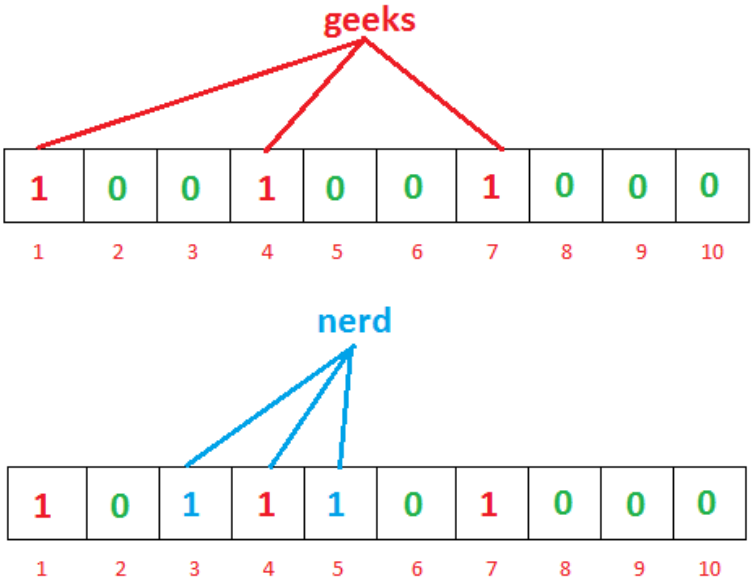
- **Test** digunakan untuk memeriksa apakah elemen yang diberikan berada di dalam himpunan atau tidak.
- **Add** menambahkan suatu elemen ke himpunan tersebut.

Sekarang sedikit kuis untuk anda.

Berdasarkan pada apa yang telah didiskusikan sejauh ini, adakah mungkin untuk **menghapus** suatu item dari bloom filter? Jika yes, maka bagaimana?

Ambillah istrihat 2 menit dan pikirkanlah solusi masalah di atas!

Dapat gagasan? Tidak ada? Mari kita diskusikan. Di bawah ini adalah vektor bit setelah bloom filter menerima masukan ‘geeks’ dan ‘nerd’ di dalamnya (seperti sebelumnya).



Sekarang kita ingin menghapus ‘geeks’ dari dalam vektor bit. Jika kita menghapus bit 1 pada posisi indeks 1, 4, 7 dari vektor bit, karena ditandai oleh 'geeks', dan mengubahnya menjadi '0', apa yang akan terjadi? Anda dapat dengan mudah melihat itu, lain kali jika kita mencari ‘nerd’, karena indeks ‘4’ akan menunjukkan ‘0’, pasti akan memberikan output bahwa ‘nerd’ tidak ada dalam daftar, meskipun sebenarnya ada. Itu berarti penghapusan tidak mungkin tanpa memperkenalkan negatif palsu (*false negatives*).

Jadi, apa solusinya?

Solusinya adalah dengan tidak dapat menggunakan operasi Hapus di bloom filter sederhana (dasar) ini. Tetapi jika kita benar-benar perlu memiliki fungsionalitas Penghapusan maka dapat menggunakan variasi filter bloom yang dikenal sebagai '***couting bloom filter***. Idenya sederhana. Alih-alih menyimpan sedikit nilai 1, kita akan menyimpan nilai integer. Vektor bit akan berubah menjadi vektor integer. Ini akan menambah ukuran dan biaya lebih banyak ruang untuk memberikan fungsionalitas Penghapusan. Alih-alih hanya menandai nilai bit ke '1' saat memasukkan nilai, kita harus menambah nilai integer dengan 1. Untuk memeriksa apakah suatu elemen ada di dalam vektor, cukup periksa (*test*) apakah nilai posisi indeks yang sesuai dengan nilai hasil hashing lebih besar dari 0.

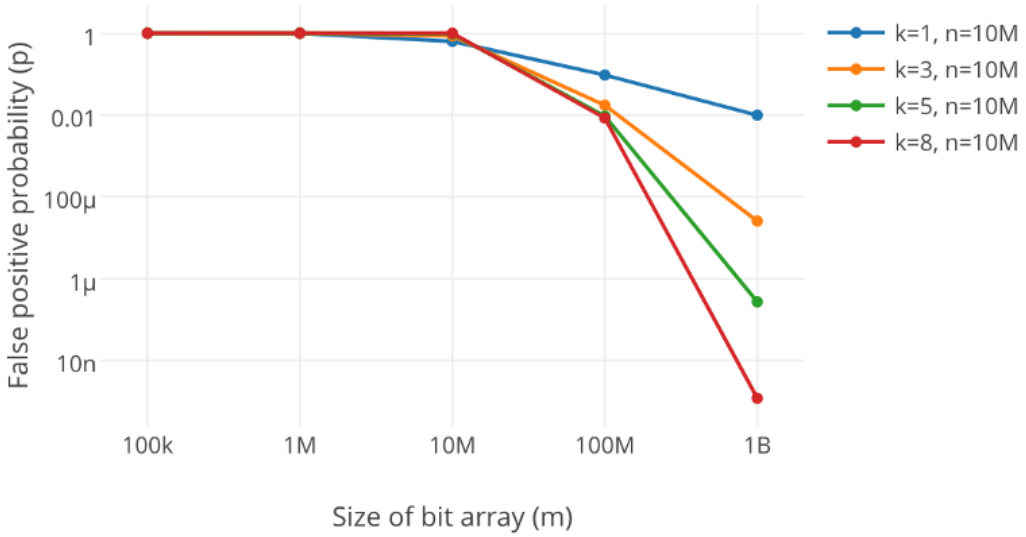
Jika Anda mengalami kesulitan untuk memahami bagaimana ‘counting bloom filter’ dapat memberikan fitur 'penghapusan', sangat disarankan Anda mengambil pena dan kertas dan mensimulasikan bloom filter ini sebagai *counting filter* dan kemudian mencoba penghapusan pada vektor itu. Semoga Anda dapat memperolehnya dengan mudah. Jika gagal, coba lagi. Jika

gagal lagi maka silakan tinggalkan komentar dan kami akan mencoba untuk menggambarkannya.

Ukuran Saringan Bloom dan Jumlah Fungsi Hash

Kita sudah memahami bahwa jika ukuran filter bloom terlalu kecil, segera semua bidang bit akan berubah menjadi '1' dan kemudian filter bloom akan sering mengembalikan 'false positive' untuk setiap input. Jadi, ukuran filter bloom adalah keputusan yang sangat penting untuk dibuat. Filter yang lebih besar akan memiliki lebih sedikit false positive. Jadi, kita dapat menyetel filter bloom berdasarkan seberapa tepat dibutuhkannya berdasarkan 'tingkat kesalahan positif palsu'.

Parameter penting lainnya adalah 'berapa banyak fungsi hash yang akan digunakan'. Semakin banyak fungsi hash yang digunakan, semakin lambat filter bloom. Namun, jika kita memiliki bllom filter terlalu sedikit, maka mungkin menderita terlalu banyak *false positive*.



Dapat terlihat jelas dari grafik di atas bahwa meningkatkan jumlah fungsi hash, **k**, akan secara drastis mengurangi tingkat kesalahan, **p**.

Kita dapat menghitung tingkat kesalahan positif palsu, p, berdasarkan ukuran filter, m, jumlah fungsi hash, k, dan jumlah elemen yang dimasukkan, n, dengan rumus:

$$p \approx (1 - e^{\frac{-kn}{m}})^k$$

Sepertinya WTF? Jangan khawatir, sebagian besar kita sebenarnya perlu memutuskan seperti apa m dan k kita. Jadi, jika kita menetapkan nilai toleransi kesalahan p dan jumlah elemen n sendiri, kita dapat menggunakan rumus berikut untuk menghitung parameter ini:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

$$k = \frac{m}{n} \ln 2$$

Poin penting lain juga perlu disebutkan di sini. Karena tujuan utama menggunakan filter bloom adalah untuk mencari lebih cepat, kita tidak dapat menggunakan fungsi hash yang lambat, bukan? Fungsi hash kriptografis seperti Sha-1, MD5 tidak akan menjadi pilihan yang baik untuk filter bloom karena agak lambat. Jadi, pilihan yang lebih baik dari implementasi fungsi hash yang lebih cepat adalah murmur, seri fnv hash, hash Jenkins dan HashMix.

Aplikasi

Filter Bloom adalah tentang menguji Keanggotaan dalam satu set. Contoh klasik menggunakan filter bloom adalah mengurangi pencarian disk (atau jaringan) yang mahal untuk kunci yang tidak ada. Seperti yang dapat kita lihat bahwa filter bloom dapat mencari kunci dalam waktu konstan O (k), di mana k adalah jumlah fungsi hash, akan sangat cepat untuk menguji tidak adanya kunci..

Jika elemen tidak ada dalam filter bloom, maka kita tahu pasti kita tidak perlu melakukan pencarian mahal. Di sisi lain, jika ada di filter bloom, kita melakukan pencarian, dan kita bisa berharap gagal di sebagian waktu (tingkat positif palsu).

Untuk beberapa contoh yang lebih konkret:

- Anda telah melihat dalam contoh yang diberikan bahwa kita dapat menggunakannya untuk memperingatkan pengguna akan kata sandi yang lemah.
- Anda dapat menggunakan filter bloom untuk mencegah pengguna mengakses situs jahat.
- Daripada membuat query ke database SQL untuk memeriksa apakah ada pengguna dengan email tertentu, Anda dapat menggunakan filter bloom untuk pemeriksaan pencarian yang tidak mahal. Jika email itu tidak ada, bagus! Jika memang ada, Anda mungkin harus membuat permintaan tambahan ke database. Anda dapat melakukan hal yang sama untuk mencari jika ‘Nama pengguna sudah diambil’.
- Anda dapat menyimpan filter bloom berdasarkan alamat IP pengunjung di situs web Anda untuk memeriksa apakah pengguna ke situs web Anda adalah 'pengguna kembali' atau 'pengguna baru'. Beberapa nilai positif palsu untuk 'pengguna yang kembali' tidak akan menyakiti Anda, bukan?
- Anda juga dapat membuat pemeriksa ejaan dengan menggunakan filter bloom untuk melacak kata-kata kamus.
- Ingin tahu bagaimana Medium menggunakan bloom filter untuk memutuskan apakah pengguna sudah membaca posting? Baca artikel mengagumkan yang menggetarkan pikiran ini.

Apakah Anda masih berpikir bahwa Anda tidak akan pernah membutuhkan filter bloom? Ya, kita tidak akan menggunakan semua algoritma yang telah dipelajari dalam kehidupan sehari-hari. Tapi mungkin suatu hari nanti itu mungkin menyelamatkan kita. Siapa tahu? Mempelajari hal baru tidak pernah menyakitkan, bukan?