Filtry Blooma i systemy rozproszone

Krzysztof Kotuła

Seminarium z systemów rozproszonych 2008/09

Szkic prezentacji

1. Filtry Blooma
   - Wstęp
   - Budowa
   - Prawdopodobieństwo błędu
   - Kompresowane filtry Blooma
2. Krótko o haszowaniu
3. Zastosowanie w systemach rozproszonych
   - Wstęp
   - Sieci
   - Śledzenie ataków sieciowych
4. Filtry Bloomiera
   - Wstęp
   - Konstrukcja
   - Uwagi

Co to jest filtr Blooma?

- **Struktura danych** wynaleziona w 1970 roku przez Burtona Blooma
- Potrafi odpowiadać na pytanie, czy dany element należy do zbioru
- Jest probabilistyczny – **może się mylić**
- Jest generalizacją tablicy haszującej

Dozwolone operacje

- Wstawienie do zbioru
- Zapytanie o posiadanie danego elementu (w dalszej części prezentacji operacja ta nazywana będzie po prostu *zapytaniem*)
- W podstawowej wersji – **usuwanie zabronione**

Później przekonamy się, że konstrukcja filtrów Blooma pozwala też na inne sztuczki

Element losowości

- Struktura wykorzystuje losowe funkcje haszujące (wiele)
- Jeśli algorytm odpowie na zapytanie *nie*, to się **nie myli**
- Jeśli jednak powie *tak*, to **może kłamać**

W *ogólności* korzystanie z filtru Blooma nie uwalnia nas więc z jawnego przechowywania zbioru

Elementy składowe

- Tablica m bitów
- k losowych funkcji haszujących dających wartości z przedziału $\langle 0..m)$
- Pozycje w tablicy indeksowane są wartościami haszy

Parametry m i k oraz *jakość* funkcji haszujących istotnie wpływają na efektywność działania filtra.

Inicjacja i wstawianie

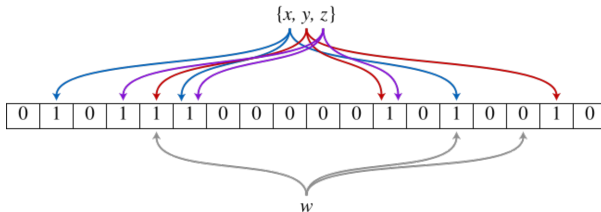
- Na początku filtr jest pusty – wszystkie bity są zgaszone
- Wstawienie elementu polega na zapaleniu wszystkich k bitów wyznaczonych przez funkcje haszujące obliczone na tym elemencie
- W wersji podstawowej bitów nigdy nie gasimy i nie przejmujemy się tym, że mogą zostać wielokrotnie ustawione
- W tym miejscu tracimy zdolność usuwania

Zapytanie

- Dla testowanego elementu obliczanych jest wszystkich k haszy
- Wyznaczone w ten sposób bity są testowane
- Jeśli co najmniej jeden z nich jest zgaszony – element **na pewno** nie należy do zbioru
- W przeciwnym wypadku – **nie wiadomo**, tzn. jest szansa, że element jest w zbiorze
- Gdy $k = 1$ mamy do czynienia z prostą tablicą haszującą

Dobry filtr Blooma mylić się będzie z małym prawdopodobieństwem

Przykład



Rysunek: źródło: Wikipedia.org

- $k = 3, \; m = 18$
- Kolorowe strzałki wyznaczają kolejne wartości haszy elementów $x, \; y, \; z$
- Próba odnalezienia elementu w kończy się niepowodzeniem
- Jeśli wyobrazimy sobie element u, którego trzy hasze wynoszą 1, dostajemy przykład błędu

Problemy z usuwaniem

- Jeden bit może zostać zapalony wielokrotnie, przy wstawianiu różnych elementów
- A zatem usuwając jeden element nie można po prostu zgasić odpowiadających mu bitów
- Inaczej algorytm zacząłby kłamać mówiąc *nie*, gubiąc tym samym jedną z kluczowych własności

Problem można jednak próbować obejść...

Usuwanie z drugim filtrem

- Mamy dwa filtry Blooma (niekoniecznie tego samego rozmiaru)
- Elementy wstawiamy do pierwszego filtra
- Usuwanie polega na wstawieniu obiektu do drugiego filtra
- Zapytanie sprawdza zatem, czy element został wstawiony oraz, czy nie został usunięty

Usuwanie z drugim filtrem – problemy

- Średnio dwukrotnie więcej wymaganej pamięci
- Średnio dwukrotnie dłuższy czas zapytań
- Drugi filtr jest znowu probabilistyczny – może kłamać twierdząc, że element był usunięty
- ...i najważniejsze – usuniętego elementu nie daje się ponownie wstawić!

Usuwanie z licznikiem

- Pomysł: skoro jeden bit gubi informację o tym, ile razy został zapalony – zastąpmy go licznikiem
- Zapalanie bitu to zwiększanie wartości o jeden, gaszenie – zmniejszanie o jeden
- Zapytanie polega na sprawdzeniu, czy wyznaczone k liczników ma wartości dodatnie
- Do pewnej granicy pozwala to więc spokojnie wstawiać i usuwać elementy

Usuwanie z licznikiem – problemy

- Dla każdej pozycji przeznaczona jest z góry określona liczba bitów – grozi to przepełnieniem licznika
- W efekcie – znowu tracimy informację, ile razy licznik został zwiększony
- W związku z tym nie można zmniejszać licznika ustawionego na maksymalną wartość (czyli taką, na jaką pozwala mu jego rozmiar)
- Osiągnięcie tego pułapu grozi szybkim *zapchaniem* filtra
- Stosując tą metodę dobrze jest znać oszacowanie maksymalnej liczby wstawień elementu
- ...i najważniejsze – tablica jest kilkakrotnie większa niż w wersji pierwotnej

Ciekawe własności

- Filtr zużywa stałą ilość pamięci (jeśli go nie skalujemy)
- Operacja wstawiania jest zawsze wykonywalna
- Filtr potrafi reprezentować zbiór pełny
- Złożoność obliczeniowa zapytania nie zależy od n
- Nie potrzeba dodatkowej pamięci na żadne wskaźniki :)
- Obliczanie k niezależnych funkcji aż się prosi o zrównoleglenie

Za niskie i stałe zużycie pamięci płacimy wzrostem prawdopodobieństwa błędu

Niestandardowe operacje

- Suma dwóch filtrów – OR dwóch wektorów bitowych
- Przecięcie dwóch filtrów – AND dwóch wektorów bitowych
- Dwukrotne zmniejszenie filtra – OR dwóch połówek wektora
- Powiększanie filtra i dopełnienie zbioru naturalnie nie są już takie proste :)

Motywacja

Konstruując filtr mamy pole do manewru na *dwóch* płaszczyznach:

- liczbie bitów w tablicy – za mało $\equiv$ duża szansa błędu, za dużo $\equiv$ nadmierne zużycie pamięci,
- liczbie funkcji haszujących – za mało $\equiv$ duża szansa błędu, za dużo $\equiv$ duża szansa błędu

Stosując starą dobrą *matematykę* wyprowadzimy teraz wartości optymalne :)

Postawienie problemu

- m – rozmiar tablicy (liczba bitów w filtrze)
- k – liczba funkcji haszujących
- n – liczba wstawionych elementów

Zadanie

Przy ustalonej liczbie n znaleźć wartości m i k minimalizujące prawdopodobieństwo błędu

Obliczenia

- Zakładamy, że funkcja haszująca wybiera każdy z równym prawdopodobieństwem
- $P(\text{dany bit zostanie zapalony przez daną funkcję}) = \frac{1}{m}$
- $P(\text{dany bit nie zostanie zapalony przez daną funkcję}) = 1 - \frac{1}{m}$
- $P(\text{pewien bit nie zostanie zapalony}) = (1 - \frac{1}{m})^{kn}$
- Co w przybliżeniu jest równe $e^{-kn/m}$
- $P(\text{pewien bit zostanie zapalony}) = 1 - (1 - \frac{1}{m})^{kn}$

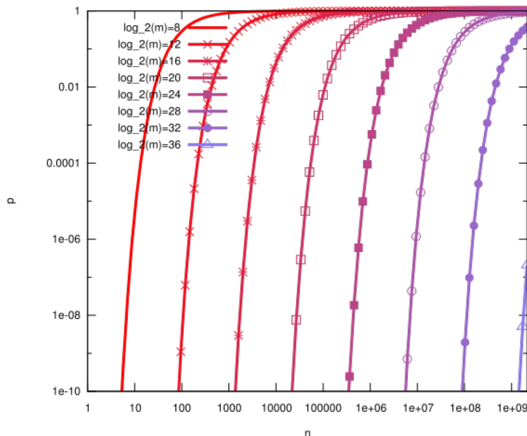
Obliczenia – cd.

- $P(\text{pewien bit zostanie zapalony}) = 1 - (1 - \frac{1}{m})^{kn}$
- Weźmy teraz element nie należący do zbioru
- Żeby filtr skłamał, wszystkie k bitów wyliczonych przez funkcje haszujące musi być zapalonych
- Szansa na to wynosi $P = (1 - (1 - \frac{1}{m})^{kn})^k$
- Stąd ostatecznie $P \approx (1 - e^{-kn/m})^k$
- Zgodnie z intuicją: szansa błędu rośnie wraz z liczbą wstawianych elementów i maleje wraz ze wzrostem rozmiaru tablicy bitowej
- Przy ustalonym n i m minimum tej funkcji wynosi $k = \frac{m}{n} ln2 \approx 0.693\frac{m}{n}$

Obliczenia – cd.

- Szansa błędu wynosi więc $P(k) = \frac{1}{2}^k \approx 0.6185^{m/n}$
- Wniosek – m musi być kilkakrotnie większy od n
- To w praktyce jest wykonalne :)
- $m = 8n \rightarrow P \approx 0.0214$
- $m = 12n \rightarrow P \approx 0.0031$
- $m = 16n \rightarrow P \approx 0.0005$
- Nie wolno zapominać przy tym, że k musi być całkowite :)

Podsumowanie



Rysunek: Błąd w zależności o n i m, k optymalne. Źródło: Wikipedia

Motywacja

Pomysł

Jeśli filtr wysyłany jest np. przez sieć, dlaczego nie spróbować go dodatkowo skompresować?

- W przedstawionej analizie braliśmy pod uwagę funkcje rozsiewające bity równomiernie
- Po wstawieniu dużego zbioru elementów – mamy praktycznie losowy wektor 0-1 (biały szum)
- Taki ciąg danych niestety słabo się kompresuje

Pokażemy teraz, jak obejść ten problem

Motywacja – cd.

- Podejście pierwotne: zminimalizować prawdopodobieństwo błędu względem k przy ustalonym m i n
- Ponieważ dysponujemy kompresją – uwalniamy m na rzecz nowego czynnika z
- Teraz mamy szansę zyskać na kompresji
- Pytanie, co na tym stracimy?

Postawienie problemu

- n – liczba wstawianych elementów
- m – liczba bitów w tablicy nieskompresowanej
- k – liczba funkcji haszujących
- z – żądany rozmiar tablicy skompresowanej (w bitach)
- p – prawdopodobieństwo, że dany bit jest zgaszony (bity są niezależne)

Zadanie

Dla ustalonej wartości n i z znaleźć wartości m i k minimalizujące prawdopodobieństwo błędu

Górne oszacowanie

Posługując się wcześniej obliczonymi wartościami mamy:

- $z = m$
- $k = \frac{m}{n} ln2$
- Zatem $P \leq 0.6185^{z/n}$, gdy $p = \frac{1}{2}$
- Naturalnie da się lepiej :)

Można nawet pokazać, że $p = \frac{1}{2}$ maksymalizuje błąd kompresowanego filtra!

Uzyskane wyniki

m/n	8	14	92
z/n	8	7.923	7.923
k	6	2	1
P	0.0216	0.0177	0.0108
$\approx$ 8 bitów na element			

m/n	16	28	48
z/n	16	15.846	15.829
k	11	4	3
P	0.000459	0.000314	0.000222
$\approx$ 16 bitów na element			

Źródło: *Compressed Bloom Filters*, Michael Mitzenmacher

Uzyskane wyniki – cd.

m/n	8	12.6	46
z/n	8	7.582	6.891
k	6	2	1
P	0.0216	0.0216	0.0215
Współczynnik błędu ≈ 0.0216			

m/n	16	37.5	93
z/n	16	14.666	13.815
k	11	3	2
P	0.000459	0.000454	0.000453
Współczynnik błędu ≈ 0.00045			

Źródło: *Compressed Bloom Filters*, Michael Mitzenmacher

Wydajność obliczeniowa

- Otrzymana struktura po spakowaniu zajmuje tą samą ilość pamięci
- Niestety bieżące korzystanie z niej wymaga ciągłej kompresji/dekompresji
- Żeby uniknąć ogromnego narzutu czasowego – stosujemy kompresję fragmentami
- Przy okazji niestety pogarszamy współczynnik kompresji

Uwagi

- W podanych przykładach operować będziemy na liczbach
- Pokażemy jak zbudować cały zbiór dobrych funkcji haszujących
- W praktyce warto jednak skorzystać z gotowych rozwiązań (np. algorytm *HMAC* z *OpenSSL*).

Oznaczenia

- A – zbiór pierwotny
- B – zbiór na który odwzorowujemy
- A więc funkcja haszująca będzie typu $h : A \to B$
- Naturalnie zachodzi $|B| \ll |A|$

Konstrukcja za pomocą macierzy

- Liczby ze zbiorów A i B traktujemy jako wektory bitów (ustalonych rozmiarów)
- Niech n i m będą odpowiednio długością wektorów z A i B
- Losujemy macierz zero-jedynkową H o rozmiarze $n \times m$
- Wtedy szukaną funkcją haszującą będzie $h(x) = aH$
- Można pokazać, że dla losowej funkcji $Pr(h(x) = h(y) | x \neq y) \leq \frac{1}{2^m}$

Konstrukcja za pomocą liczb pierwszych

- Bierzemy liczbę pierwszą $p \geq |A|$
- Wtedy $h(x) = ((ax + b) \mod p) \mod |B|$, gdzie $a, b \in Zp$ oraz $a \neq 0$
- Zbiór tak powstałych funkcji spełnia poprzednią własność...
- ...a nawet więcej: $P(h(x_1) = y_1 \& h(x_2) = y_2) = \frac{1}{n^2}$

Zasada filtrów Blooma

Gdziekolwiek lista lub zbiór użyte są, a pamięć priorytetem jest, zważ wykorzystanie filtrów Blooma, gdy kłamstwa jego drogi Ci nie zagrodzą

Dawne zastosowania

- Sprawdzanie pisowni (tj. reprezentacja słownika)
- Przechowywanie zbioru *niepoprawnych* haseł w UNIXie
- Operacja semi-join w rozproszonej bazie danych (przykład):
  - Baza A chce wysłać do bazy B listę miast, w których koszt życia wynosi > 50000
  - Baza B natomiast chce odesłać do A listę osób żyjących w takich miastach
  - Zamiast przesyłać listę miast z A do B – można wysłać filtr Blooma
  - W zamian B odsyła listę potencjalnych odpowiedzi, z których należy odrzucić błędy

Dawne zastosowania – cd.

- Filtr dla dziennika transakcji bazy danych:
  - W bazie trzymany jest zapis zmian z danego dnia
  - Chcąc odczytać rekord sprawdzamy najpierw, czy został dzisiaj zmodyfikowany
  - Zamiast sięgać do dziennika – można odpytać przygotowany filtr Blooma
  - Błąd spowoduje co najwyżej niepotrzebny odczyt z dziennika

Web cache w serwerach proxy

- Serwery proxy cache'ują strony internetowe
- Jeśli pewien serwer chce odczytać stronę i nie ma jej we własnej pamięci – odpytuje inne
- Normalnie serwery mogłyby wymieniać się listami adresów URL
- Używając filtrów Blooma zamiast takich list – oszczędzają pasmo
- Serwer trzyma filtr Blooma dla każdego innego serwera

Web cache w serwerach proxy – cd.

- Rozgłaszanie aktualnego filtra odbywa się w pewnych odstępach czasu
- Ponieważ cache zmienia się dynamicznie, często zdarzać się będą zapytania o wyrzuconą stronę
- W praktyce takich błędów będzie znacznie więcej niż tych wynikających z pomyłki filtra
- A zatem omylność filtrów Blooma prawie nie wpływa na wydajność serwerów

Lokalizacja obiektów w sieciach p2p

- Wykorzystanie filtrów Blooma jest w zasadzie identyczne, jak w serwerach proxy
- Dla dużych, zmiennych sieci zaleca się użycie rozproszonych tablic haszujących, które poprawiają skalowalność
- Błąd filtra znowu powoduje tylko zbędne zapytanie

Uzgadnianie zbiorów

- Peer B chce pobrać od peera A elementy, których on nie ma
- Jeśli A dysponuje zbiorem S_A, a B zbiorem S_B, to przesłany powinien zostać zbiór $S_A - S_B$
- Wystarczy, że B wyśle do A filtr Blooma reprezentujący S_B
- Kłamstwo filtra spowoduje niepełne uzgodnienie (część elementów nie zostanie wysłana)
- W sieciach p2p zbiory S_X mogą (i nawet powinny) składać się z kawałków plików

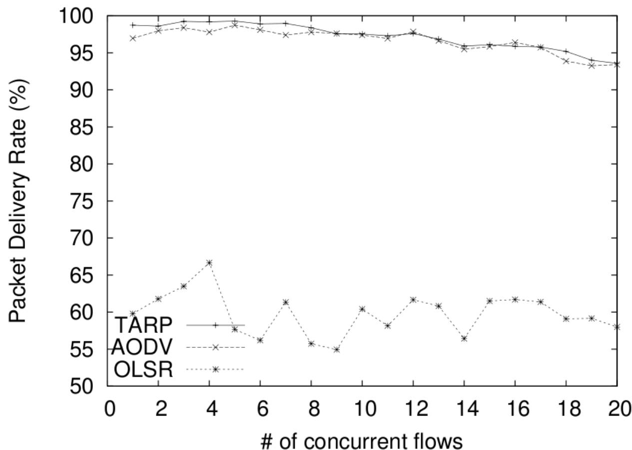
Routowanie zasobów

- Chcemy móc odnajdywać obiekty, które znajdują się niedaleko w sieci (w sensie liczby węzłów pośredniczących)
- Dla każdego połączenia i odległości (do wyznaczonego limitu d) tworzymy filtr Blooma
- W ten sposób (z pewnym błędem) możemy docierać do danego obiektu po najkrótszej ścieżce
- Błąd filtra skutkuje sięgnięciem po zasób odleglejszy niż d kroków

Routowanie zasobów – wyniki (objaśnienia)

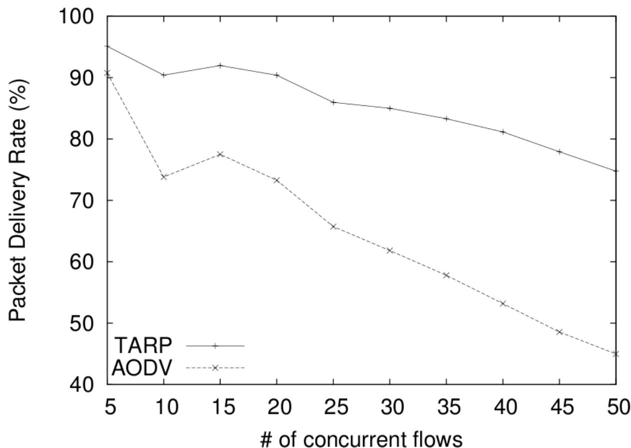
- Źródło: *Location Independent Compact Routing for Wireless Networks*, Robert Gilbert, Kerby Johnson, Shaomei Wu, Ben Y. Zhao, Haitao Zheng Department of Computer Science, University of California
- Badania wykonane na sieci wifi
- OLSR – protokół na bieżąco badający istniejące trasy
- AODV – protokół budujący i utrzymujący trasy routingu na bieżące potrzeby
- TARP – protokół posługujący się filtrami Blooma mniej-więcej jak wyżej

Routowanie zasobów – wyniki



Rysunek: 100 węzłów w sieci

Routowanie zasobów – wyniki cd.



Rysunek: 500 węzłów w sieci

Problematyka

Ataki hakerskie są obecnie na porządku dziennym. Główne powody to:

- Powszechny dostęp do internetu
- Duże natężenie ruchu w sieci utrudniające odnalezienie hakera
- *Spoofing* – możliwość fałszowania adresu źródłowego pakietu

...

Popularne metody ataku

- *Flooding* – *zapchanie* ofiary masową liczbą pakietów, pośrednio (poprzez komputery *zombie*) lub bezpośrednio
- Skanowanie – np. program *nmap*
- Wykorzystywanie błędów w systemach zdalnych – np. przez przepełnienie bufora

Osiągnięte rezultaty

- Metody ochrony przed *floodingiem* i skanowaniem sprawdzają w obrębie sieci lokalnych
- Obrona przed atakiem z zewnątrz wymaga znajomości źródła
- Szczególnie trudne do zidentyfikowania są ataki krótkie
- Rejestrowanie całego ruchu w sieci jest niewykonalne: przy łączu 1 Gbps w trakcie 1 min pełnego ruchu rejestrowanie 20-bajtowych nagłówków IP wymagałoby prawie 5 GB pamięci...
- ...tu wkraczają filtry Blooma

Cele

- Zbudować system monitorujący ruch w poszczególnych węzłach sieci
- Ograniczyć prawdopodobieństwo błędów

Architektura

- Do rejestracji wykorzystane zostaną wszelkie urządzenia sieciowe (tj. routery, switche, itp.)
- Każde z nich posiadać będzie ustaloną liczbę filtrów Blooma
- Dodatkowo w odpowiednich punktach rozmieszczone będą *detektory intruzów* – to one rozpoczynać będą śledzenie wybranych przez siebie pakietów
- Na podstawie zebranych informacje stworzone będą potencjalne trasy ataków
- Brak centralnego serwera, węzły są równorzędne

Liczba filtrów

- Co najmniej jeden na każdy interfejs sieciowy urządzenia
- Za mało – informacje będą szybko nadpisywane
- Za dużo – wyszukiwanie źródła będzie trwało zbyt długo, a prawdopodobieństwo błędu się powiększy
- Każdy filtr rejestruje jedynie ruch wychodzący lub przychodzący – pozwoli to odtworzyć trasę pakietu
- Ewentualnie dodatkowy filtr do śledzenia pakietów generowanych przez urządzenie (szczególnie, gdy jest *końcówką* sieci)

Wyznaczenie źródła pakietu

- Dany węzeł otrzymuje zapytanie o konkretny pakiet:
  - jego początkowe X bajtów (wielkość powinna być z góry ustalona),
  - unikalny identyfikator (np. numer),
  - szacowany odstęp czasu, jaki upłynął od zarejestrowania pakietu u sąsiada
- ...po czym odpytuje swoje filtry:
  - jeśli nie znalazł pakietu, kończy swoje zadanie
  - jeśli znalazł – odpowiada i przekazuje pytanie do odpowiednich sąsiadów

Wyznaczanie źródła pakietu – cd.

- Ze względu na omylność filtrów jedno urządzenie może dostać dwukrotnie zapytanie o ten sam pakiet
- Przed takim zapętleniem chroni numerowanie zapytań
- Dla bezpieczeństwa warto stosować szyfrowanie i uwierzytelnianie
- Należy też pamiętać, że pewne pola nagłówka IP ulegają zmianie podczas podróży – przy zapytaniu trzeba je ignorować
- Aby pokonać NAT, router będący jego wyjściem musi umieć prawidłowo przekazać zapytanie do podsieci

Kalibracja filtrów Blooma

- Filtry Blooma nasycony został 270000 pakietów z sieci lokalnej klasy C
- Zbadano również różne algorytmy liczenia skrótów jako bazy do tworzenia funkcji haszujących

Wyniki

		Sieć dostępowa	Ethernet 10BaseT	Fast Ethernet	Gigabit Ethernet
Przepustowość	[Mbps]	2	10	100	1000
	[kpps]	5,4	27	272	2717
Zasoby dla całych ramek		18 MB	89 MB	889 MB	8,9 GB
Zasoby dla filtrów		< 1MB	7,5 MB	60 MB	960 MB
Zysk [%]		95	92	93	89
Parametry filtrów	*m*	2^16	2^19	2^22	2^26
	k	8	13	11	17
	p	3,6e-3	9,4e-5	6,0e-4	7,0e-6

Rysunek: Wykorzystane zasoby przy 1 minucie rejestrowania 40 B nagłówków IP

Wnioski

- Żeby router mógł wytrzymać obciążenie obliczeniowe, wymagane są specjalizowane układy sprzętowe
- Jak również mnóstwo (szybkiej) pamięci dyskowej
- ...a to kosztuje $$$
- A zatem w najbliższym czasie nie grożą nam masowe aresztowania :)

Co to jest filtr Bloomiera?

- Rozszerzenie idei zwykłych filtrów Blooma
- Zaprojektowane przez Bernarda Chazelle'a w 2004 roku
- Implementują tablicę asocjacyjną (mapę) przy pomocy tradycyjnych filtrów
- Ponownie – mogą się mylić

Uwagi

- Prezentowane rozwiązanie jest semi-optymalne w sensie zużycia pamięci
- Z drugiej strony jest proste do wytłumaczenia
- Skonstruujemy tablicę mapującą obiekty na liczby 0 lub 1 (jeden bit)
- Rozwiązanie można łatwo rozszerzyć na większy przedział – dodając nowe filtry (Bloomiera), które razem złożą się w ciąg bitów kodujący kolejne liczby

Stan początkowy

- Dwa filtry Blooma: A0 oraz B0, obydwa puste
- Pierwszy pamięta obiekty mapujące się na 0, drugi na 1
- Nigdy nie będziemy wstawiać danego elementu do obydwu filtrów

Wstawianie

- Dodajemy element do odpowiedniego filtra, w zależności od tego jaką wartość mu przypiszemy (0 lub 1)
- Odpytujemy drugi filtr, czy zawiera obecnie wstawiany obiekt
- Jeśli odpowiedź brzmi *nie*, to kończymy pracę
- W przeciwnym przypadku mamy problem – w razie odpytania filtru Bloomiera o przydzieloną wartość obydwa filtry Blooma stwierdzą, że zawierają element

Wstawianie cd.

- Tworzymy więc... nowe filtry: A1 i B1 (znów mapujące obiekty odpowiednio na 0 lub 1)
- Patologiczny obiekt wstawiamy do odpowiedniego mu nowo powstałego filtra
- Element pozostaje naturalnie we właściwym starym filtrze
- Jeśli w pewnym momencie okaże się, że zarówno A1, jak i B1 stwierdzą, że posiadają pewien element (po uprzednim odpytaniu A0 i B0), znów mamy problem

Wstawianie cd.

- Tworzymy więc... nowe filtry: A2 i B2 (znów mapujące obiekty odpowiednio na 0 lub 1)
- (...)

Odpytywanie

- Żeby sprawdzić, na jaką wartość mapuje się dany element – odpytujemy filtry A0 i B0
- Jeśli żaden nie zawiera obiektu – nie ma go w tablicy
- Jeśli tylko jeden go zawiera – mamy odpowiedź
- Jeśli obydwa mówią *tak*, to znowu mamy problem...
- ...odpytujemy więc filtry A1 i B1, itd.

Złożoność pamięciowa

- Potencjalnie tworzymy nieskończony ciąg filtrów (jeśli po drodze nie zabraknie elementów)
- Wydawać by się więc mogło, że filtr Bloomiera pochłania masę pamięci
- Z drugiej strony – prawdopodobieństwo przerzucenia danego elementu poziom wyżej jest równe prawdopodobieństwu wystąpienia błędu w drugim filtrze przy wstawianiu
- To z kolei - jest niewielkie (zakładamy, że mamy *porządny* filtr)
- A więc szansa wejścia na kolejny poziom maleje w postępie geometrycznym
- Osiągając pewien pułap warto przejść na deterministyczną strukturę, bo trafi do niej niewielki odsetek mapy

Błąd

- Z definicji filtr Bloomiera myli się, gdy zwraca wartość dla klucza, którego nie ma w mapie
- Łatwo pokazać przykład dla podanej konstrukcji, który łamie tę własność
- Prawdziwy filtr Bloomiera wymaga trochę więcej wysiłku

Możliwe zastosowania

- W zasadzie to samo, co w przypadku zwykłych filtrów Blooma
- Mapowanie na większą liczbę wartości daje natomiast szersze pole do działania
- ...i szersze pole do pomyłek :)

- http://en.wikipedia.org/wiki/Bloom_filter
- http://www.cs.ucsb.edu/~ravenben/publications/pdf/tarp-mobishare06.pdf
- http://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf
- http://www.eecs.harvard.edu/~michaelm/NEWWORK/postscripts/cbf2.pdf
- http://www.cert.pl/PDF/secure2004/kruk-tobis.pdf
- http://www.ee.technion.ac.il/~ayellet/Ps/nelson.pdf