Bloomfilter cascade

project algoritmen en datastructuren 3

Noah Van Steenbrugge

2025

Contents

algoritme

1.0

opbouwen

We noteren $BF_i^{(n)}$ en $C_i^{(n)}$ voor de bloomfilters en initiële categoriëen van cascadetrap n ($n = 0, 1, 2$ ...).

Zonder verlies van algemeenheid zal ik $n = 0$ gebruiken om het minder omslachtig te maken.

We bouwen 1 cascadetrap als volgt op:

$C_1 \Rightarrow BF_1(C_2, C_3, ..., C_n \Rightarrow C_1'$

$C_2 \Rightarrow BF_2(C_1', C_3, C_4, ..., C_n) \Rightarrow C_2'$

$C_3 \Rightarrow BF_3(C_1', C_2', C_4, C_5, ..., C_n) \Rightarrow C_3'$

...

$C_n \Rightarrow BF_n(C_1', C_2', ..., C_{n-1}') \Rightarrow C_n'$

Dan kunnen we de volgende cascadetrap doen met $C_1', C_2', ..., C_n'$.

Indien de categoriëen die we in een bloomfilter zullen steken allemaal leeg zijn, stopt ons algoritme.

Indien alle categoriëen behalve 1 leeg zijn na het afwerken van de volledige trap, stopt ons algoritme ook.

Na het stoppen zal er 1 niet-lege categorie overblijven.

classificiëren

Bij $BF_i^{(n)}$ als het antwoord “neen” is, dan weten we dat het sowieso in de bijhorende $C_i^{(n)}$ zit en dus in C_i.

Bij “ja” gaan we over naar de volgende bloomfilter.

Indien de volledige cascade werd afgewerkt en we overal het antwoord “ja” kregen, dan zit het in de laatste niet-lege categorie.

2.0

We kunnen beter doen omtrent opslag (voor grote verzamelingen).

Het is geen nieuw algoritme, maar een uitbreiding van algoritme 1.0.

opbouwen

Ook hier weer zonder verlies van algemeenheid $n = 0$.

Noem $C'_1, ..., C'_{i-1}, C_{i+1}, ..., C_n = \bar{C}_i$

Om $BF_i(\bar{C}_i)$ uit algoritme 1.0 te verkleinen zullen we $\bar{C}_i$ eerst filteren met een kleinere bloomfilter van C_i:

$\bar{C}_i \Rightarrow BF_{i_1}(C_i) \Rightarrow \bar{C}_i{}'$

Nu kunnen we de volgende bloomfilter opstellen:

$C_i \Rightarrow BF_{i_2}(\bar{C}_i{}') \Rightarrow C'_i$

Dus $BF_i^{(n)}$ wordt vervangen door 2 bloomfilters $BF_{i_1}^{(n)}$ en $BF_{i_2}^{(n)}$. Maar deze gebruiken samen minder bits (voor grote verzamelingen) en de grootte van de nieuwe categorieën $(C_1^{(n+1)}, C_2^{(n+1)}, ..., C_n^{(n+1)})$ zal gemiddeld hetzelfde zijn.

De rest van het opbouwen is analoog aan het vorige algoritme.

classificiëren

Indien $BF_{i_1}^{(n)}$ als antwoord "neen" geeft, skippen we $BF_{i_2}^{(n)}$ en gaan we direct naar $BF_{i+1_1}^{(n)}$ (of $BF_{1_1}^{(n+1)}$ indien de trap afgerond is).

Indien het anwoord "ja" is checken we $BF_{i_2}^{(n)}$, indien we daar "neen" krijgen zit het in $C_i^{(n)}$, dus in C_i, en stoppen we, indien "ja" gaan we naar de volgende bloomfilter.

Indien $\bar{C}_i{}^{(n+1)}$ leeg is (na filteren), maken we geen $BF_{i_2}^{(n)}$. In dit geval als het antwoord "ja" is bij $BF_{i_1}^{(n)}$ weten we al dat het element in categorie $C_i^{(n)}$ zit (en dus in C_i).

Ook hier als we steeds moeten overgaan naar de volgende filter en op het einde aankomen, wordt het element geclasfficiëerd in de overgebleven niet-lege categorie.

bits en hashfuncties

hashfuncties optimaal: $k = \frac{n}{m} \ln 2$ met $m = \#$ elementen en $n = \#$ bits.

Kans op false positives: $(1 - e^{\frac{-km}{n}})^k$

Kans bij optimale k: $(1 - e^{-\ln 2})^{\frac{n}{m} \ln 2} = (1 - \frac{1}{2})^{\frac{n}{m} \ln 2} = (\frac{1}{2})^{\frac{n}{m} \ln 2}$

Hoeveel moet n i.f.v. met m zijn om de kans $\leq x$ te maken?

$$(\frac{1}{2})^{\frac{n}{m} \ln 2} \leq x \Leftrightarrow 2^{\frac{n}{m} \ln 2} \geq \frac{1}{x} \Leftrightarrow \frac{n}{m}(\ln 2)^2 \geq \ln(\frac{1}{x}) \Leftrightarrow n \geq -m\frac{\ln x}{(\ln 2)^2}$$

Neem voor x telkens een negatieve macht van 2: $x = 2^{-p}$ met $p = 1, 2, 3$ ...

Dan wordt onze formule:

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

Bij een gelijkenis heb je:

$k = \frac{\frac{mp}{\ln 2}}{m} \ln 2 = p$

We krijgen:

$n = \frac{mk}{\ln 2}$ en $x = 2^{-k}$

Dus als we kans op false positives 1/32 willen en m = 1 MB (1 000 000 B) dan hebben we:

$k = 5$ en $n \geq \frac{5}{\ln 2}$ MB = 7.213... MB

We zullen dus nu een k meegeven aan de bloomfilter om de n mee te berekenen zodanig dat de kans op false positives kleiner of gelijk is aan 2^{-k}.

Waarom doe ik 2^{-k} als kans en geef ik de k mee om daaruit de n te halen i.p.v. meer vrijheid te geven aan de kans op false positives en daaruit de k en n te halen?

Omdat de afrondingen dan veel minder doorwegen. Nu heb ik afrondingen op mijn aantal bits dat makkelijk een paar miljoen kan zijn, terwijl ik geen afrondingen heb op mijn kleine k.

Bv. een afronding van 5.4 naar 5 bij de k zal meer verschil geven dan een afronding van 1234567.4 naar 1234567 bij de n, niet alleen doordat het getal daardoor relatief minder verkleint, maar ook omdat ik die n toch moet afronden naar een getal deelbaar door 8 (aangezien ik een `uint8_t*` gebruik voor de bloomfilter bits). Dus 1234567.4 wordt 1234567, maar dan 1234568 om deelbaar door 8 te worden, dus de eerste afronding maakt geen verschil.

benchmarks voor optimale k

Ik zal geen rekening houden met tijd, aangezien de opslagruimte van de files het belangrijkst is.

Om te zien of k per cascadetrap aan te passen iets helpt heb ik op het eenvoudig algoritme met de large.txt de volgende benchmarks uitgevoerd:

`k=3` voor de eerste cascadetrap:

`k` 57 MB, `k++` 58 MB, `k--` 57 MB

Conclusie, gebruik dezelfde k bij elke cascadetrap.

Hypothese, de optimale k is quasi hetzelfde, onafhankelijk de grootte van de verzameling.

algoritme 1.0

large.txt

`k=1` 53 MB, `k=2` 53 MB, `k=3` 57 MB, `k=4` 64 MB, `k=5` 73 MB

medium.txt

`k=1` 6.3 MB, `k=2` 7.0 MB, `k=3` 8.1 MB, `k=4` 9.5 MB, `k=5` 12 MB

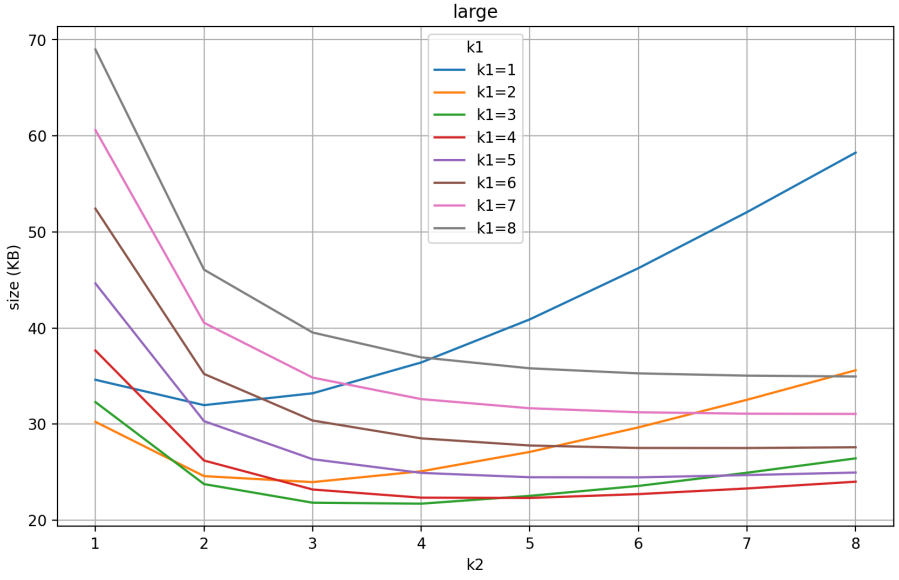
small.txt

`k=1` 128 KB, `k=2` 144 KB, `k=3` 168 KB, `k=4` 200 KB, `k=5` 236 KB

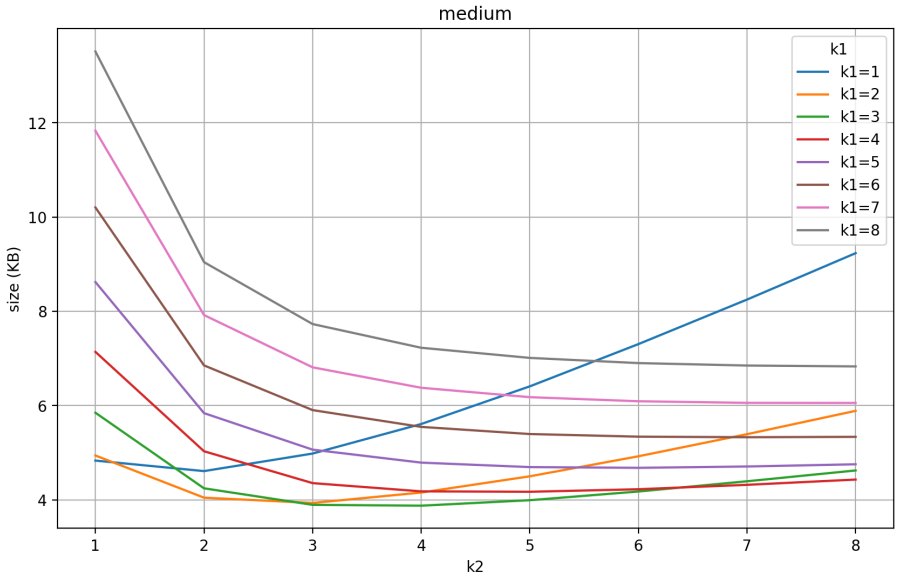
Conclusie, de optimale k voor dit algoritme is 1 en mijn hypothese klopt hoogstwaarschijnlijk.

algoritme 2.0

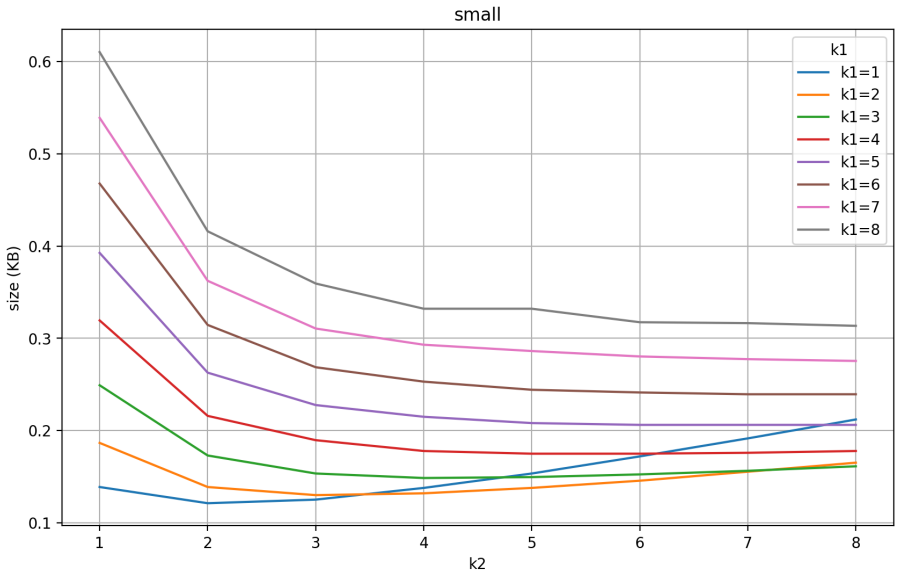
`k1` wordt gebruikt voor BF_{i_1} en `k2` voor BF_{i_2}.



benchmark large



benchmark medium



benchmark small

We zien dat (3, 4) het beste is voor large en medium en (1, 2) voor small.

Dit is geen tegenvoorbeeld voor de hypothese, aangezien ik “quasi” gebruikte en de size voor (3, 4) bij small scheelt amper met die voor (1, 2). Dus neem ik overal `k1=3` en `k2=4`.

bestandsformaat cascade

Header:

- 1 byte die het soort algoritme aangeeft (0 = algoritme 1.0, 1 = algoritme 2.0)
- aantal categoriëen (4 byte) (= aantal bloomfilters per trap)
- namen van categoriëen in juiste volgorde (1 byte lengte + 1 byte * lengte)

Cascade trap header: -

Elke bloomfilter binnen de trap:

- aantal hashfuncties in bloomfilter (1 byte)
- seeds voor hashfuncties (1 byte seeds * aantal hashfuncties)
- aantal bytes in bloomfilter (4 byte)
- bloomfilter bits (1 byte * aantal bytes)

Indien een bloomfilter leeg is, is er 1 byte aan 0-bits voor het aantal hashfuncties en dan ga je over naar de volgende bloomfilter (geen nutteloze 4 byte voor het aantal bytes in de lege bloomfilter).

Na de laatste cascadetrap 1 byte aan 1-bits en dan eindigen met de laatste niet-lege categorie naam (1 byte lengte + 1 byte * lengte).

limitaties

Voor de categorie namen is er een limitatie van 256 chars bij zowel `train` als `classify`.

Indien een ingegeven element bij `classify` in geen enkele categorie zit, dan zal deze geclassificeerd worden als een random categorie.

Voor de integers (aantal elementen, aantal bits ...) is er een limitatie van 32 bits.

Bv. voor het aantal bits in een bloomfilter met $k \leq 4$ (optimale k) kan je 744 261 117 elementen hebben in de bijhorende categoriëen ($\ln 2 \frac{2^{32}-1}{4}$). Je zal eerder tegen de RAM limiet botsen, dus 32 bits is meer dan voldoende.

benchmarks

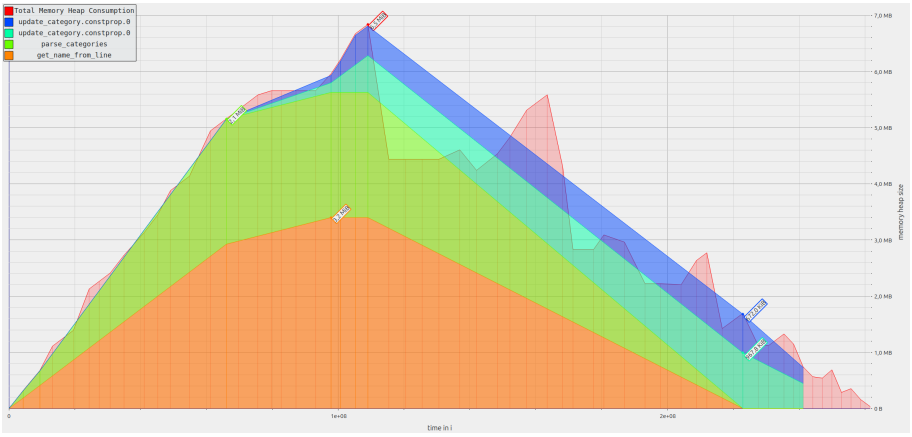
small

3.4 MB, 3 categories, 226 556 elements

algoritme 1.0

128 KB - 0.413s

6.5 MiB peak RAM

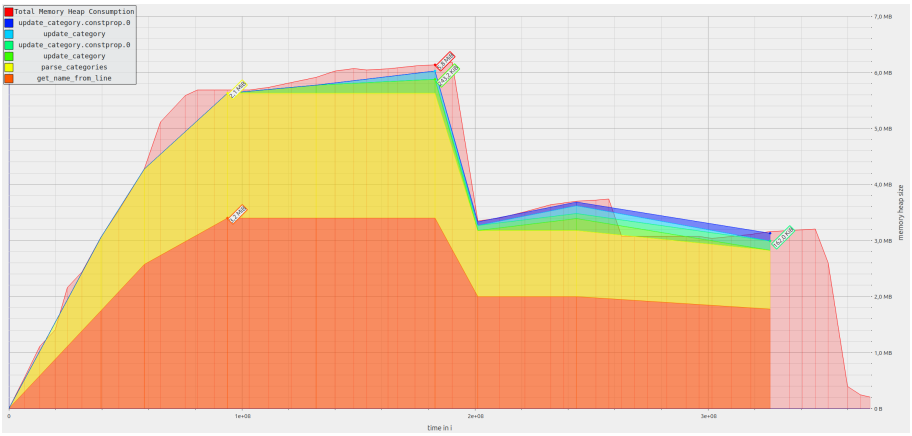


massif small 1.0

algoritme 2.0

156 KB - 0.410s

5.8 MiB peak RAM



massif small 2.0

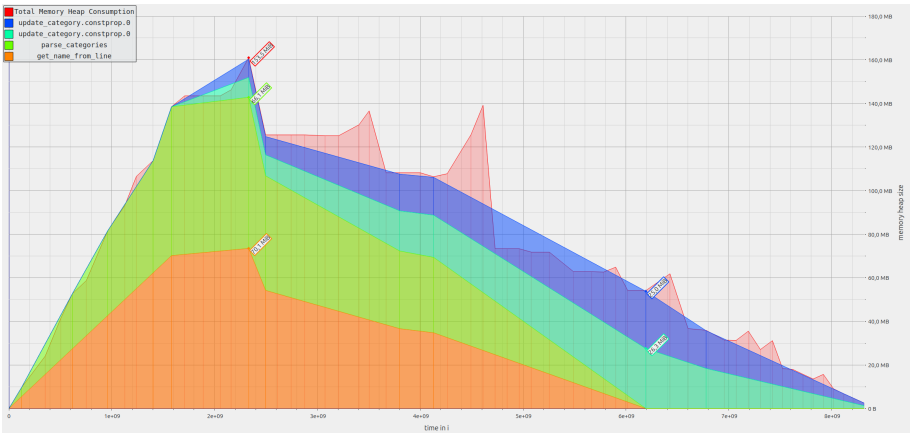
medium

73.5 MB, 6 categories, 4 898 932 elements

algoritme 1.0

6.3 MB - 1.360s

153.5 MiB peak RAM

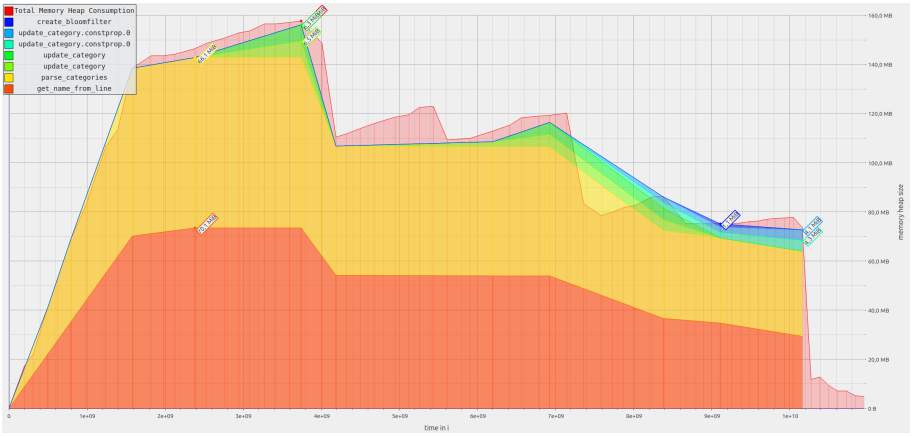


massif medium 1.0

algoritme 2.0

3.9 MB - 1.340s

150.4 MiB peak RAM



massif medium 2.0

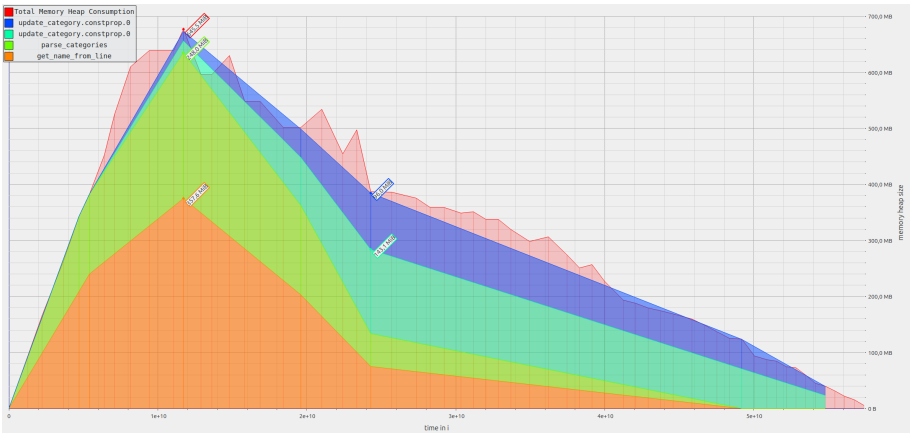
large

375.0 MB, 10 categories, 25 000 000 elements

algoritme 1.0

53 MB - 6.969s

645.5 MiB peak RAM

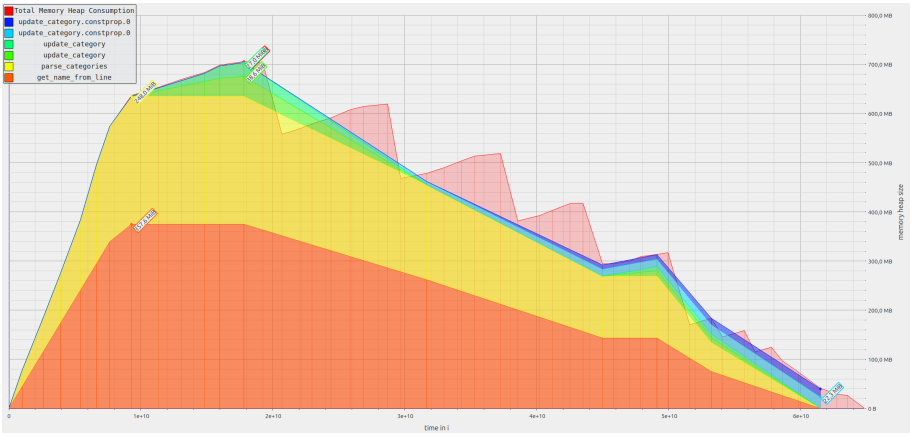


massif large 1.0

algoritme 2.0

22 MB - 6.432s

673.1 MiB peak RAM



massif large 2.0

bespreking

Je ziet dat algoritme 1.0 beter is voor de small, maar bij medium en large is algoritme 2.0 beter. Voor de small ligt algoritme 2.0 wel dicht bij algoritme 1.0 voor de optimale k, maar die gebruiken we niet voor het gemak. Indien je kleinere verzamelingen hebt, moet je maar algoritme 1.0 gebruiken met de flag `-a 0`.

Het RAM-gebruik voor beide algoritmes is ongeveer hetzelfde.

Plezant dat je de trappen mooi kunt zien in de figuren, vooral bij algoritme 2.0 op large. Bij algoritme 1.0 zijn er veel meer trappen, aangezien `k=1`, dus is het moeilijker om te zien.

conclusie

Ik heb mij geamuseerd :)