SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Bambus filter

Jelena Glasovac i Hrvoje Radoš

Voditelj: Mirjana Domazet-Lošo

Zagreb, mjesec, godina

**Sadržaj**

[1. Uvod 2](#_Toc199766890)

[2. Seminarski rad 4](#_Toc199766891)

[2.1 Opis bambus filtera 4](#_Toc199766892)

[2.1.1 Umetanje 5](#_Toc199766893)

[2.1.2 Pretraživanje 5](#_Toc199766894)

[2.1.3 Proširivanje 6](#_Toc199766895)

[2.1.4 Brisanje 6](#_Toc199766896)

[2.1.5 Kompresija 7](#_Toc199766897)

[2.2 Rezultati 7](#_Toc199766898)

[2.2.1 Vremenski utrošak umetanja 7](#_Toc199766899)

[2.2.2 Vremenski utrošak pretraživanja 8](#_Toc199766900)

[2.2.3 Memorijski utrošak izvođenja 10](#_Toc199766901)

[2.2.4 Lažni odgovori funkcije pretraživanja 11](#_Toc199766902)

[3. Zaključak 12](#_Toc199766903)

[4. Sažetak 13](#_Toc199766904)

[5. Literatura 14](#_Toc199766905)

# Uvod

Česti je problem u informatici pitanje nalazi li se neki podatak u nekom skupu. Naivno pretraživanje skupa podataka član po član ima vremensku složenost O(n), a zauzeće prostora ograničeno je time koliko je potrebno za pohraniti sve elemente u memoriju. To je izrazito neefikasno za situaciju u kojoj nije bitno koji su sve podatci u toj strukturi nego samo nalazi li se nešto u njoj. Rješenje za to nudi tip struktura podataka koje žrtvuju pouzdanost za brzinu i uštedu memorijskog prostora. One unose nesigurnost u obliku male šanse za lažan odgovor, a zauzvrat drastično smanjuju utrošeno vrijeme i memoriju.

Takve vjerojatnosne strukture (eng. AMQ, approximate membership query) podataka već se dugi niz godina koriste za mjerenje mrežnih podataka, sigurnosti, bioinformatici i dr. područjima [1] kako bi približno odredili nalazi li se neki podatak u skupu. Kroz povijest neke značajne strukture podataka koje pripadaju ovom tipu su Bloom filtri (eng. Bloom filters), kuku filtri (eng. Cuckoo filters) i bambus filtri (eng. Bamboo filters) čiju implementaciju se predstavlja u ovom seminaru. Prvo su predstavljeni bloom filtri kod kojih ekstremno brzom vremenu pretraživanja skupa i memorijskoj učinkovitosti doprinosi spremanje hasha nekog elementa umjesto samog elementa. [3] Sve od navedenih struktura podržavaju dvije osnovne funkcionalnosti: dodavanje elemenata i njihovo traženje. Funkcije pretraživanja vraćaju pozitivan odgovor ako je predmet vjerojatno u filtru, a negativan ako nije, a to čine na temelju uspoređivanja hasha predmeta s hashovima zapisanim u odgovarajućem pretincu filtera.

Napredak nad njima učinio je kuku filter koji omogućuje podršku brisanju podataka. Također, umjesto spremanja čitavog hasha sprema njegov otisak prsta (eng. Fingerprint) koji je kraći dio hash funkcije, a ostale podatke sprema u lokaciju otiska prsta u strukturi podataka. [4] Kuku filter može se zamisliti kao polje pretinaca (eng. Buckets) gdje svaki pretinac predstavlja jednu od dvije lokacije na koje element može biti spremljen. Ta se lokacija može izračunati iz hasha objekta, ali također alternativna se lokacija mora moći izračunati iz prve lokacije i spremljenog otiska prsta. Time dovodimo dvije lokacije u bijektivnu matematičku vezu međusobno. Prilikom dodavanja elementa u kuku filter, provjerava se ima li mjesta u pretincu na bilo kojoj od te dvije lokacije, a ako nema niti na jednoj, nasumičan se element iz nekog od ta dva pretinca izvadi, na njegovo se mjesto stavi predmet koji se umeće, a izvađeni predmet se pokuša umetnuti na svoju alternativnu lokaciju. Postupak se ponavlja dok svi elementi nisu uspješno relocirani ili dok se ne dogodi maksimalan broj izbacivanja elemenata što ukazuje na to da je kuku filter dosegao svoj maksimalan kapacitet.

Prilikom traženja nekog predmeta, sve što se treba učiniti jest pregledati oba moguća pretinca i ako je pronađen odgovarajući otisak prsta vratiti istinitu vrijednost, a u suprotnom vratiti neistinitu vrijednost.

Primijetimo da zbog toga što dva predmeta mogu imati iste otiske prstiju, ovaj postupak nije deterministički te je moguće dobiti istinit odgovor prilikom traženja nekog predmeta iako nije on bio dodan nego je bio dodan drugi predmet istog otiska prsta. Rezultat toga su lažni istiniti odgovori (false positives) koji se upravo događaju kada funkcija pretraživanja nalazi predmet koji nije u ovoj strukturi podataka. Lažni neistiniti odgovori (false negatives) nisu mogući, dakle možemo sa sigurnošću znati da predmet nije u strukturi podataka ako je odgovor funkcije pretraživanja neistinit.

Ključni problemi ove metode su da kako bi se proširio kuku filter mora se cijela struktura podataka ponovo izgraditi tj. Svi se hashevi moraju ponovo izračunati (eng. Rehashing) kako bi se dodalo još prostora za podatke. Problem se rješava u trećoj generaciji AMQ-a koja je bambus filter. Osnovne razlike su to da bambus filter podržava dinamičko mijenjanje veličine filtera s parcijalnim ponovnim računanjem hash funkcija i brisanje elemenata iz nje, pri čemu efikasnost nije jednako ovisna o količini podataka u strukturi.

U ovom seminaru cilj je detaljno predstaviti ideju bambus filtera, algoritam na kojem se temelje te demonstrirati vlastitu implementaciju bambus filtera po uzoru na radove Hangchenga Wanga u kojima je predstavljena originalna implementacija ove strukture podataka [1] [2] i s njom usporediti učinkovitost programa.

# Seminarski rad

## Opis bambus filtera

Bambus filteri vrsta su AMQ strukture podataka koja korisniku omogućuje tri osnovne funkcionalnosti. One su umetanje (eng. insert), pretraživanje (eng. lookup) i brisanje (eng. Delete). Interno, oni obavljaju još dvije funkcionalnosti, a to su proširivanje (eng. Expand) i kompresija (eng. Compress). Naš bambus filter slijedi isti koncept i podržava iste funkcionalnosti, a u nastavku opisana je njegova implementacija.

Struktura podataka koristi četiri sloja. Bambus filter dinamička je hash tablica unutar koje su spremljeni individualni kuku filteri zajedničkih karakteristika. Svaki kuku filter jedan je segment, a u njemu se nalazi neki broj pretinaca i u svakom pretincu ima mjesta za neki broj predmeta. Početna veličina hash tablice određena je implementacijom, a bitno je da se može povećati i smanjiti ovisno o broju podataka u njoj.

A diagram of a diagram

Description automatically generated

Slika 1. Shema strukture bambus filtera

Hash svakog predmeta može se podijeliti u tri dijela. Prvi dio je otisak prsta, drugi dio je indeks segmenta, a treći indeks pretinca. Primjerice, kod hasha veličine 32 bita, ako je otisak prsta veličine 16 bita, preostalih 16 bita su raspoređeni u dva skupa: indeks pretinca i indeks segmenta. U našoj je implementaciji veličina indeksa pretinca 10 bitova, a veličina indeksa segmenta 6 bitova. To znači da u svakom segmentu ima 210 pretinaca, a u početnom stanju bambus filter ima 26 segmenata što je ujedno i minimalan broj segmenata koji mogu biti u uporabi.

A diagram of a line with a red arrow

Description automatically generated

Slika 2. Podjela hasha na otisak prsta, indeks segmenta i indeks pretinca

### Umetanje

Umetanje podataka počinje računanjem hasha predmeta koji se umeće u filter. Iz tog se hasha zatim izdvaja otisak prsta i indeks segmenta. U odgovarajući segment umeće se otisak prsta na jednu od dvije moguće lokacije. Primijetimo kako indeks pretinca upućuje na samo jednu lokaciju. Druga lokacija računa se kao xor prve lokacije i otiska prsta koji je skraćen na duljinu indeksa pretinca. Ponavljanjem postupka može se ponovo iz druge lokacije dobiti prvu.  
Kao i kod kuku filtera, predmet se može ubaciti na bilo koju od te dvije alternativne lokacije, a ako niti na jednoj nema mjesta, izbacuje s nasumičan otisak prsta iz jedne od te dvije lokacije te se prebacuje na svoju alternativnu lokaciju. Ako niti ondje nema mjesta za izbačeni predmet, on uzima mjesto nekog drugog nasumičnog otiska prsta i postupak se ponavlja dok se ne dosegne maksimalan broj kuku izbacivanja. Ako je taj broj dosegnut, otisak prsta koji je ostao bez svog mjesta stavlja se u overflow segment koji je implementiran kao dinamička lista za svaki pretinac.  
Naravno, tu postoji mogućnost da se u isti pretinac overflow segmenta doda jako puno otisaka prsta, a ostali pretinci ostanu prazni što bi približilo vremensku učinkovitost listi, no u praksi to nije problem jer korištena implementacija hash funkcije osigurava ravnomjernu raspršenost elemenata po svim pretincima segmenta, a problem prenapučenosti filtera rješava se proširivanjem filtera pod određenim uvjetima proširivanja (eng. Expand conditions).  
Umetanje podataka ima vremensku složenost ograničenom maksimalnim brojem izbacivanja elemenata prilikom traženja, dakle ona ostaje O(1).

### Pretraživanje

Nove predmete u bambus filteru traži se jednostavnim postupkom. Prvo se izračuna hash funkcija i podijeli se u otisak prsta, indeks segmenta i indeks pretinca te se u segmentu s odgovarajućim indeksom pronađe pretinac s odgovarajućim indeksom. Ako je željeni otisak prsta unutar pretinca ili unutar odgovarajućeg pretinca overflow segmenta, funkcija vraća istinit odgovor, a u suprotnom provjerava se i alternativna lokacija koja se dobije ranije opisanim postupkom. Ako predmet nije niti ondje, funkcija vraća neistinit odgovor.  
Primijetimo da je pretraživanje također ima konstantnu vremensku složenost pod uvjetom da hash funkcija dobro raspršuje podatke i da ima dovoljno prostora u filteru da su predmeti u overflow segmentu svedeni na minimum što se postiže redovitim proširivanjem filtera.

### Proširivanje

Proširivanje segmenata događa se pod uvjetom da je zadovoljen uvjet proširivanja. Ono je zapravo dodavanje segmenata u niz. Jedan poziv funkcije proširivanja dodaje jedan segment u filter. Segmenti se proširuju slijedno, pri čemu se vraća na segment s indeksom 0 nakon što prođe jedna runda ekspanzije. Runda ekspanzije određena je kao n uzastopnih ekspanzija gdje je n broj segmenata u filteru u trenutku kada je pokazivač na sljedeći segment koji treba proširiti usmjeren u segment s indeksom 0.  
Na taj način svaka runda ekspanzije poveća ukupni broj segmenata 2 puta. Primijetimo da se time i povećava raspon mogućih indeksa segmenata. Nakon svake runde ekspanzije, stoga, veličina indeksa segmenata se povećava za jedan tako što se posljednji bitovi otiska prsta gledaju i kao prvi bitovi indeksa segmenta. Tako dolazimo i do maksimalnog broja segmenata koji mogu biti u bambus filteru koja ja 2m gdje je m zbroj brojeva bitova otiska prsta i indeksa segmenta.  
Prilikom svakog proširenja, stoga, treba se dio elemenata prebaciti iz segmenta u kojem se trenutačno nalaze u novonapravljeni segment kako bi ponovo odgovarali njihovi indeksi koji sada imaju više bitova. Te se element lako može pronaći zato što oni na svom posljednjem bitu otiska prsta imaju jedinicu.  
Sam postupak počinje tako što se napravi novi segment. Iz segmenta koji se proširuje i njegovog overflow segmenta izvade svi otisci prstiju koje treba preseliti u novonapravljeni segment te se oni ondje prebace, a zatim se elementi koji su ostali u overflow segmentu originalnog segmenta ponovo ubacuju u taj segment. Pokazivač na sljedeći segment koji treba proširiti povećava se za jedan, a ako je runda dovršena, umjesto toga pomiče se na segment s indeksom jedan.  
Jedino pitanje koje preostaje je kada se uopće treba proširiti filter. Rješenje predloženo u originalnoj implementaciji je izabrati broj k i proširiti filter svaki put kada je broj elemenata u filteru n\*k1 [2]. To smo i mi iskoristili kao uvjet proširivanja u našoj implementaciji.

### Brisanje

Brisanje se izvodi tako što se za element koji se želi obrisati izračuna hash, izdvoji otisak prsta, indeks segmenta i indeks pretinca. Pretraže se dva moguća pretinca u kojima se element može nalaziti i kada se pronađe otisak prsta koji odgovara traženom, on se obriše iz tog pretinca. Prilikom pretraživanja pretinca, prvo se pretražuje pripadajući pretinac u overflow segmentu, a tek onda sam pretinac kako bi se izbjegli prazni ili poluprazni pretinci čiji pretinci u overflow segmentu su popunjeni. Ako je više predmeta s istim otiskom prsta, valja primijetiti da ne znamo koji od ta dva brišemo ali budući da je otisak prsta i njegova pozicija u bambus filteru jedino po čemu pamtimo predmete, nije bitno koji obrišemo, a koji ostavimo. Nakon svakog brisanja, povjerava se uvjet kompresije koji je analog uvjeta proširivanja, ali u suprotnom smjeru.

### Kompresija

Suprotno od proširenja, kada se svakih k2 elemenata ukloni iz filtera, kako bi se izbjegla premala popunjenost segmenata, filter se kompresira. Kompresija je inverzan postupak od proširenja. Svi podatci se iz segmenta koji se uklanja moraju vratiti u segment čiji je indeks ostatak originalnog indeksa pri dijeljenu s dva. Nakon toga, segment se briše, pokazivač na sljedeći segment se pomiče na prvi manji segment, a ako nema prethodni segment, umjesto toga, vraća se na posljednji segment iz prethodne runde proširenja.

## Rezultati

U sljedećem dijelu, uspoređuje se napravljena implementacija koju se razmatra u ovom seminaru s referentnom implementacijom koju predlaže Hangcheng Wang u svom radu o o bambus filterima [2]. Za sve testove, četiri su testna slučaja u kojima se mijenja broj podataka koji se uzastopno dodaju i traže u bambus filteru. Riječ je o 103, 104, 105 i 106 podataka. Prikazana je usporedba utroška vremena umetanja, pretraživanja i utroška memorije prilikom rada programa.

### Vremenski utrošak umetanja

U ovom testu umeću se podatci u bambus filter. Brojač vremena pokreće se netom prije početka operacija ubacivanja, a gasi nakon njihovog završetka, a zatim se računa prosječan broj unosa u bambus filter po milisekundi.

Slika 3.1 Graf koji pokazuje ovisnost broja umetanja u milisekundi o broju ukupnih uzastopnih umetanja

Tablica 1. Podatci prethodno prikazani u grafu.

n = ukupan broj umetnutih podataka

m = broj umetanja po jedinici vremena u našoj implementaciji

mref = broj umetanja po jedinici vremena u referentnoj implementaciji [2]

|  |  |  |
| --- | --- | --- |
| **n** | **m /ms-1** | **mref /ms-1** |
| **103** | 13.53 | 30.22 |
| **104** | 11.64 | 23.82 |
| **105** | 15.14 | 15.20 |
| **106** | 9.46 | 11.02 |

Iako su brzine umetanja usporedive, referentna implementacija je nešto brža u većini ispitnih slučajeva. Predlažemo dva moguća objasnjenja za to; jedan je problem u čestim i dugotrajnim proširivanjima filtera, a drugi je korištenje malih vekrota (eng. Small vector) za implementaciju pretinaca.

Dodavanje novih segmenata obavljamo tako što sve elemente koji trebaju biti premješteni izbacujemo iz starog segmenta i dodajemo u novi, a zatim pretražujemo overflow segment i ponavljamo postupak na njemu. Te su operacije vremenski skupe u odnosu na jednostavno umetanje elemenata u bambus filter, dakle mogle bi biti razlog značajnom usporenju programa.

Drugi krivac za sporije umetanje podataka od referentne izvedbe mogao bi biti to što su u njoj pretinci implementirani kao cijeli broj (eng. Integer) koji se zatim bitovnim operacijama cjepka na otiske prstiju koji su u njemu spremljeni. Naša implementacija koristi male vektore koji su dinamička struktura podataka ograničene veličine. Zauzimaju memoriju samo kada je ona potrebna, brži su od običnih vektora jer se odjednom učitaju u priručnu memoriju (eng. Cache), no unatoč tome, rađenje bitovnih operacija nad cjelobrojnom varijablom je brže nego izvođenje operacije pisanja u mali vektor.

Jedan slučaj ispitivanja odskače od trenda nešto niže brzine umetanja, ali za njegovo objašjenje zahtjevalo bi dodatno testiranje programa i proučavanje procesa prevođenja.

### Vremenski utrošak pretraživanja

U ovom testu šalju se upiti za neki element koji može i ne mora prethodno biti dodan u bambus filter. Vrijeme se mjeri od trenutka prije početka prvog pretraživanja sve do kraja izvođenja uzastopnih operacija pretraživanja.

Slika 4.1 Graf koji pokazuje ovisnost broja pretraživanja u milisekundi o broju ukupnih uzastopnih pretraživanja

Tablica 2. Podatci prethodno prikazani u grafu.

n = ukupan broj pretraženih podataka

m = broj umetanja po jedinici vremena u našoj implementaciji

mref = broj umetanja po jedinici vremena u referentnoj implementaciji [2]

|  |  |  |
| --- | --- | --- |
| **n** | **m /ms-1** | **mref /ms-1** |
| **103** | 26.48 | 24.5 |
| **104** | 22.68 | 21.17 |
| **105** | 19.11 | 17.96 |
| **106** | 10.7 | 14.36 |

Ranije prikazani rezultati posebice su zanimljivi jer pokazuju poboljšanje u odnosu na referentnu implementaciju pri pretraživanju bambus filtera koji sadrže broj podataka reda veličine manjeg od 106. U slučajevima kada se često treba pretraživati, a rijetko dodavati elemente u bambus filter, moglo bi se uštedjeti vremena koristeći našu implementaciju. Upravo mali vektori kao način implementacije pretinaca je promjena koja bi mogla biti ključna za ovu razliku. Ako se i cijeli broj i mali vektor cijeli učitavaju u priručnu memoriju odjednom, približno iste brzine, korištenjem cijelog broja obavezujemo se na dodatne bitovne operacije kako bismo iz njih očitali željene podatke, dok su podatci u malm vektoru odmah spremni za uporabu. Na taj se način pri velikom broju ponavljanja operacije pretraživanja moglo bi se uočiti malo povećanje brzine u odnosu na originalnu implementaciju. Ipak, u bambus filterima koji su namijenjeni za sadržavanje velikog broja podataka, referenta implementacija pokazuje se boljom u odnosu na našu.

### Memorijski utrošak izvođenja

Posljednji test mjeri maksimalnu memorijsku potrošnju tjekom umetanja podataka u bambus filter.

Slika 5.1 Graf koji pokazuje ovisnost maksimalnog memorijskog utroška o broju elemenata dodanih u bambus filter

Tablica 2. Podatci prethodno prikazani u grafu.

n = ukupan broj umetnutih podataka

m = maksimalno zauzeće RAM-a u našoj implementaciji

mref = maksimalno zauzeće RAM-a u referentnoj implementaciji [2]

|  |  |  |
| --- | --- | --- |
| **n** | **M / MB** | **mref /MB** |
| **103** | 8 | 4 |
| **104** | 12 | 6 |
| **105** | 40 | 38 |
| **106** | 344 | 340 |

Gledajući u memorijsku potrošnju pri malim količinama umetnutih podataka, naša se implementacija čini dosta gorom od originalne. Kada se sagleda šira slika, doduše, možemo uočiti da razlika u memorijskoj potrošnji ne ovisi značajno o količini dodanih podataka. Razlika je fiksno između 2 i 6 MB koje naša implementacija dodatno koristi prilikom proširivanja filtera. Podatke premještamo u što je efektivno dodatan segment kako bismo ih lakše prebacili na njihove nove lokacije. Iako se razlika čini velika gledajući manji broj unosa u filter, ona brzo postaje zanemariva kada se filter korist za pohranu velikog broja podataka.

### Lažni odgovori funkcije pretraživanja

Vjerojatnosne strukture podataka karakteristične su po greškama u pretraživanju. Iz tog je razloga važno mjeriti šansu pogrešnog istinitog i neistinitog odgovora. Kada se provodi samo dodavanje i pretraživanje, lažno neistiniti odgovori u našoj implementaciji nisu mogući kao ni u originalnoj. Lažno istiniti odgovori su ono što se treba testirati prilikom razmatranja novih implementacija filtera. Oni ovise o izboru hash funkcije. Koristimo istu hash funkciju kao i originalna implementacija, dakle očekivani je rezultat da nemamo više od maksimalnog postotka pogrešnih odgovora predloženih u originalnom radu [2], a tu tvrdnju potvrđuju i ispitni slučajevi koji u prosjeku pronalaze manje od 1% pogrešnih odgovora.

# Zaključak

U ovom seminaru, predložili smo vlastitu implementaciju bambus filtera i usporedili ju s referentnom implementacijom predloženom u radu autora Hangchenga Wanga. Implementaciju bambus filtera koristimo za dodavanje nasumičnih nizova znakova i provjeravanje jesu li nasumični znakovi već u njemu, te isprobavanje iste funkcionalnosti na nasumičnim k-merima genoma bakterije Escherichia coli.

Testiranja su pokazala da se brzina obavljanja predviđenim funkcionalnosti može usporediti s originalnom, ali primjetne su određene razlike. Na temelju dosadašnjeg znanja, predlažemo objašnjenje da na vrijeme izvođenja vjerojatno utječu razlike u implementaciji pretinaca i dodatni vektori koje koristimo za prijenos podataka prilikom proširivanja filtera.

Korištenje malih vektora umjesto cijelog broja u koji se bitovnim operacijama upisuju otisci prstiju mogli bi negativno utjecati na brzinu umetanja podataka u filter, ali bi također mogli biti i odgovorni za veće brzine pretraživanja filtera koji ima red veličine 105 ili manje podataka u sebi.

Korištenje dodatnih segmenata za premještanje podataka prilikom proširivanja filtera, doduše, pokazuje se kao vremenski i memorijski utrošak. U sljedećoj inačici filtera, bilo bi poželjno razmisliti o alternativnim načinima za implementaciju iste operacije koji bi mogli pozitivno utjecati na vrijeme izvođenja operacije proširivanja i njezine memorijske zahtjeve.

Za daljnja poboljšanja, trebalo bi razmisliti i o načinima na koje se što više operacija može svesti na bitovne operacije budući da se one brže izvode, detaljnije proučiti optimizacije koje unosi prevoditelj i razmišljati o implementaciji na nižoj razini. Ipak, čitljivost koda je žrtva koju se u tom slučaju prinosi. Naša implementacija možda nije na istoj razini kao i originalna u većini slučajeva, ali može služiti kao dobra početna točka za proučavanje principa rada bambus filtera onima koji nisu dobro upoznati s temom i žele razumljivu i jednostavnu implementaciju.

# Sažetak

Bambus filteri predstavljeni su kao struktura podataka koja nudi prostorno i vremenski efikasno rješenje za problem pripadnosti skupu. Opisana je jedna moguća implementacija ove strukture podataka napravljena po uzoru na originalnu implementaciju Wanga i njezin princip rada.

Ta implementacija uključuje pet funkcionalnosti: umetanje, pretraživanje, proširivanje, brisanje i kompresiranje filtera, a temelje se na četveroslojnoj strukturi filtera i korištenjem različitih dijelova hasha za odrediti otisak prsta i lokaciju podatka unutar strukture podataka.

Naša je implementacija testirana umetanjem i traženjem nasumičnih znakova i k-mera *E. coli* na ispitnim primjerima redova veličina 103, 104, 105, i 106. Vremenski i prostorni utrošak uspoređen je s originalnom implementacijom, a prednosti i mane naših bambus filtera prikazani su i analizirani uz moguća objašnjenja kako je došlo do tih razlika.

Predložene su mogućnosti za daljnji razvoj i poboljšanja ove implementacije.

# Literatura

[1] Wang et al. Bamboo Filters: Make Resizing Smooth doi: 10.1109/ICDE53745.2022.00078

[2] Wang et al. Bamboo Filters: Make Resizing Smooth and Adaptive 10.1109/TNET.2024.3403997

[3] Fan et al. 2013. Cuckoo Filter: Better Than Bloom; https://www.cs.cmu.edu/~binfan/papers/login\_cuckoofilter.pdf • Fan et al. 2014.

[4] Cuckoo Filter: Practically Better Than Bloom; http://www.cs.cmu.edu/%7Ebinfan/papers/conext14\_cuckoofilter.pdf