Napredni algoritmi i strukture podataka

Token Bucket, Tipovi, Kompresija podataka

Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

Ograničenje stope pristupa — uvod

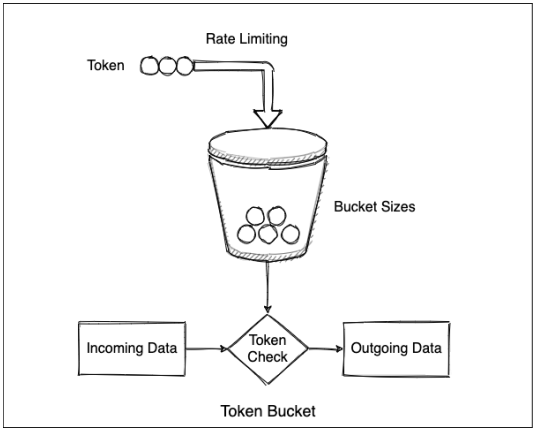
- U računarskim mrežama, ograničenje brzine/stope (rate limiting) se koristi za kontrolu brzine/stope zahteva poslatih ili primljenih od strane kontrolera mrežnog interfejsa
- **Rate limiting** je procedura koja nam omogućava kontrolu brzine kojom korisnici mogu da šalju zahteve sistemu
- Jedna od opcija je da definišemo koliko puta u jedinici vremena je dozvoljeno poslati zahtev sistemu
- Rate limiting se uglavnom koristi za zaštitu servera od neželjenih rafala, zlonamernih napada

Rate limiter kod sistema za skladištenje podataka

- Neki sistemi za skladištenje velike količine podataka su implementirali ovaj mehanizam
- To ih nekada izdvaja od drugih sličnih sistema i zato su čest izbor korisnika
- **RocksDB**, na primer, direktno podržava ovaj mehanizam, i zato je ponekad češći izbor od recimo **LevelDB**

Token bucket

- Ovo je najjednostavniji algoritam za ograničavanje brzine pristupa
- Jednostavno pratimo broj zahteva napravljenih u zadatom vremenskom intervalu
- Zbog svoje jednostavnosti, dosta se koristi
- Google cloud koristi ovaj algoritam (ili je koristio), za Task Queue opciju koja se nudi koirsnicima kao usluga
- Jednostavno se implementira i lako može da se poveže sa velikim brojem različitih slučajeva korišćenja



(What is Token Bucket and Leaky Bucket algorithms)

Algoritam

- Za svaki zahtev korisnika treba:
  - Proveriti da li je proteklo vreme nakon kog treba da resetujemo vremenski brojač
  - Ako vreme nije isteklo, treba proveriti da li korisnik ima dovoljno preostalih zahteva da obradi dolazni zahtev
  - Ako korisniku nije preostalo slobodnih zahteva, trenutni zahtev se odbacuje uz nekakvu poruku
  - U suprotnom, smanjujemo brojač za **1** i vršimo obradu dolaznog zahteva
  - Ako je vreme proteklo, tj. razlika resetovanog vremena i trenutnog vremena je veća od definisanog intervala, resetujemo broj dozvoljenih zahteva na unapred definisano ograničenje, i definišemo novo vreme resetovanja

Primer

Pimer: 3 zahteva/min:

- REQ **11:01:20** (BUCKET [11:01:05, 3]) => BUCKET [11:01:05, 2] + OK
- REQ **11:01:25** (BUCKET [11:01:05, 2]) => BUCKET [11:01:05, 1] + OK
- REQ **11:01:30** (BUCKET [11:01:05, 1]) => BUCKET [11:01:05, 0] + OK
- REQ **11:01:35** (BUCKET [11:01:05, 0]) => FAIL
- REQ **11:03:00** (BUCKET [11:01:05, 0]) => BUCKET [11:03:00, 2] + OK
- Uradili smo update vremena i broja tokena, smanjimo broj tokena za **1** i pustimo zahtev

Gde čuvamo limiter?

- Za aplikativne stvari, obično u memoriji ili nekom sistemu koji čuva podatke u memoriji zbog brzine
- Pošto mi pravimo sistemsku stvar, i pravimo storage engine - pa možemo čuvati u našem sistmu :)
- **Token bucket struktura treba da se čuva u sistemu kao bilo koji drugi podatak, sistem radi s njim pozivom GET i PUT operacija**
- Svaki korisnik može da bude **ključ**, dok **vrednost** može da sadrži vremensku odrednicu i broj tokena (a možete podrazumevati i da sistem ima samo jednog korisnika)

Tipovi

- Ako pogledamo našu strukturu podataka do sada, vidimo da čuvamo ključ kao string, a vrednost kao niz bajtova — opšti oblik
- Ovo korisniku daje mogućnost da u taj niz bajtova smesti **šta god hoće**
- Ako korisnik radi sa recimo HyperLogLog, Bloom Filter, Count-min sketch, SimHash strukturama, on može da ih serijalizuje u niz bajtova i sačuva u sistemu pozivom PUT operacije
- Međutim, neki sistemi kao što su **Redis** i **Riak** omogućavaju out-of-the-box rad sa HyperLogLog, Bloom Filter, Count-min sketch itd.
- Posao serijalizacije, deserijalizacije i implementacije algoritama za odgovarajuće strukture se prebacuje sa korisnika na sam sistem — znatno pojednostavljen rad sa sistemom iz perspektive korisnika

Primer

- Ovako izgleda rad sa HLL strukturom u Redis-u kroz njihov CLI tool

```
> PFADD bikes Hyperion Deimos Phoebe Quaoar
(integer) 1
> PFCOUNT bikes
(integer) 4
```

- **Istu stvar želimo da uradimo i u našem sistemu**
- Za svaki tip treba da definišete i implementirate operacije koje taj tip podržava
- Te operacije će se u pozadini oslanjati na već implementirane PUT, GET i DELETE operacije
- Recimo, sistem bi dodavanje elementa u HLL pod nazivom *bikes* implementirao ovako:
  - Pozovi GET za *bikes* ključ
  - Deserijalizuj dobijeni niz bajtova u HLL strukturu
  - Dodaj element u HLL strukturu
  - Serijalizuj izmenjenu HLL strukturu u niz bajtova
  - Taj niz bajtova sačuvaj pod *bikes* ključem pozivom PUT operacije

Kodiranje varijabilne dužine

- Problem sa kodiranjem fiksne dužine jeste taj da će za svaki npr. int64 broj uvek biti potrebno 64 bita tj. 8 bajtova, iako to nije uvek neophodno
- Na primer za $(7)_{10} = (111)_2$ je potrebno samo tri bita tj. jedan bajt, ostatak popunjavamo nulama
- Kako bi se taj problem prevazišao, mogu se koristiti PutVarint i PutUvarint funkcije iz binary paketa koje će smanjiti dužinu niza serijaizovanih bajtova

Kako radi kodiranje varijabilne dužine?

- Kada deserijalizujemo niz bajtova serijalizovan na ovaj način, nekako moramo da znamo gde jedna vrednost završava, a druga počinje
- To se postiže tako što prvi bit svakog bajta označava da li smo došli do poslednjeg bajta te vrednosti (1 - nismo, 0 - jesmo)
- U preostalih sedam bitova čuva se vrednost
- Na primer, $(300)_{10} = (100101100)_2$, što je ukupno devet bitova, dakle potrebno nam je dva bajta za serijalizaciju

```
10101100 00000010
↑        ↑
msb is 1 msb is 0 so we stop reading here
```

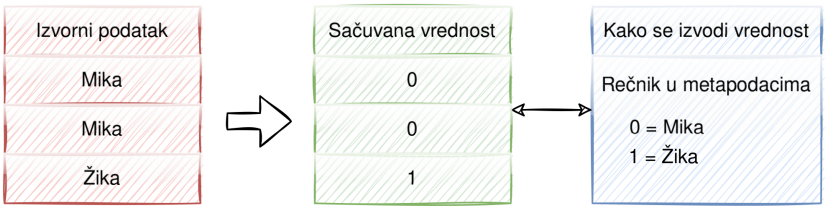
- Kada se vrednost čita, prvi bit svakog bajta se odbacuje, dok se ostali konkateniraju kako bi se dobila vrednost

Primer

```
buf := make([]byte, 3)
n := binary.PutUvarint(buf, 10)
fmt.Printf("bytes written: %d\n", n)
// bytes written: 1
n = binary.PutUvarint(buf[1:], 300)
fmt.Printf("bytes written: %d\n", n)
// bytes written: 2
num, n := binary.Uvarint(buf)
fmt.Printf("number read: %d\n", num)
fmt.Printf("bytes read: %d\n", n)
// number read: 10
// bytes read: 1
num, n = binary.Uvarint(buf[1:])
fmt.Printf("number read: %d\n", num)
fmt.Printf("bytes read: %d\n", n)
// number read: 300
// bytes read: 2
```

Dictionary Encoding

- Ako se podaci često ponavljaju (i zauzimaju veliki broj bajtova), možemo da iskoristimo ovaj algoritam (Dictionary Encoding with Bit-Packing)
- Svaku vrednost zamenjujemo malim celim brojem i čuvamo mapiranje u metapodacima – deskriptor, zauzimanje što manje mesta podacima
- ALI kada želimo da pročitamo komprimovane podatke, moramo da konsultujemo deskriptor



Zadaci

- Implementirati Token Bucket algoritam, za početak konfiguracione elemente kao i same podatke čuvati u memoriji
- Nakon toga, proširiti implementaciju sa prethodnih vežbi tako da se Token Bucket algoritam integriše u vaš sistem
- Sva podešenja vezano za algoritam treba da budu specificirana kroz konfiguracioni fajl
- Podatke vezano za Token Bucket čuvati u vašem sistemu
- **Ključ** treba da bude identifikator korisnika ili neka rezervisana vrednost ključa ako podrazumevate da imate samo jednog korisnika
- **Vrednost** treba da bude serijalizovana struktura koju čine timestamp poslednjeg resetovanja + broj preostalih tokena