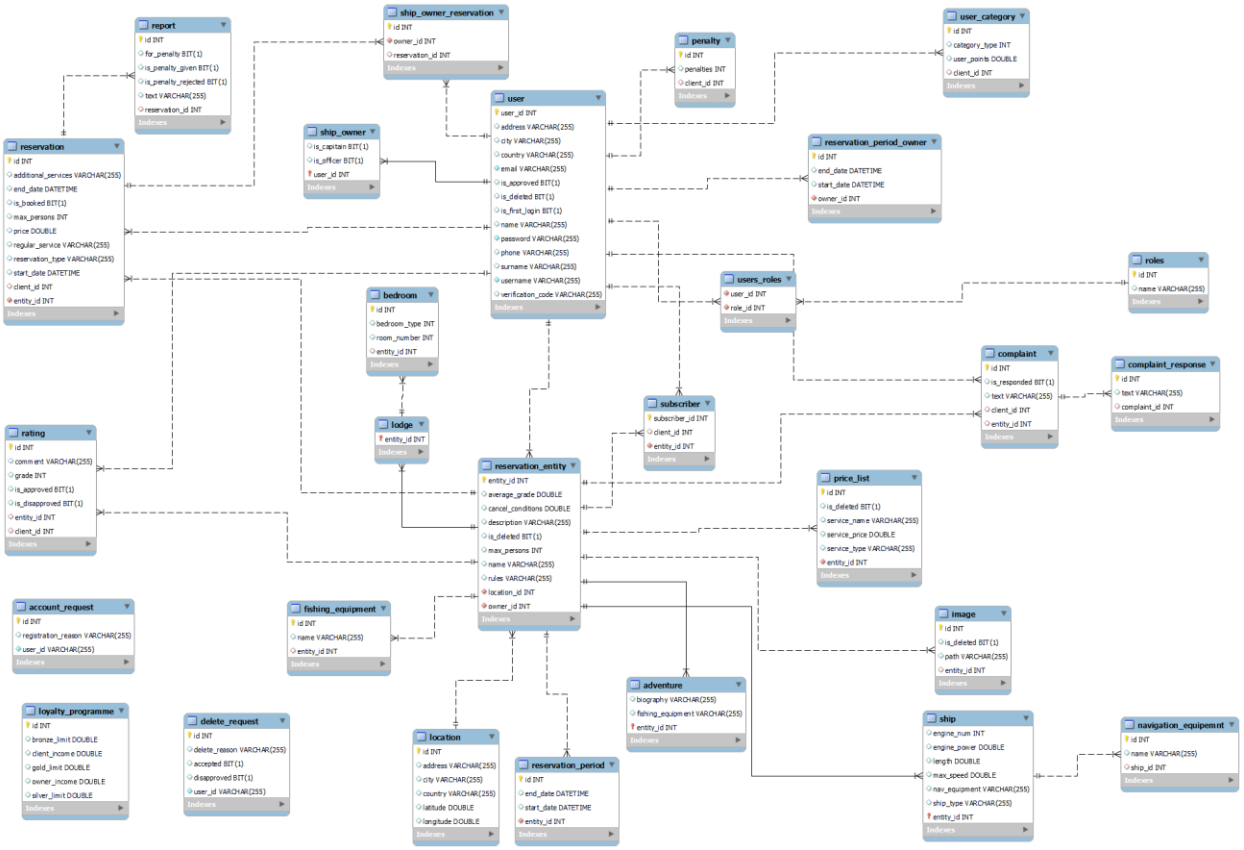
Skalabilnost

Dizajn šeme baze podataka(konceptualni)



Predlog strategije za particionisanje podataka

Glavna svrha ove aplikacije je rezervacija različitih entiteta (vikendica, brodova i avantura). Pored toga imamo i veliki broj čitanja entiteta iz baze, dodavanja i izmena entiteta u bazi. Radi bržeg čitanja i pisanja, potrebno je da čuvamo podatke o entitetima na više mašina.

Prvi predlog: Particionisanje podataka na osnovu ReservationEntity ID-a

- Raspodela podataka o entitetima na različite servere
- Kada nam je potreban entitet funkciji za heširanje se prosleđuje ID od tog entiteta i onda će ona znati gde da traži entitet za čitanje, kao i gde ga treba sačuvati prilikom dodavanja novog entiteta

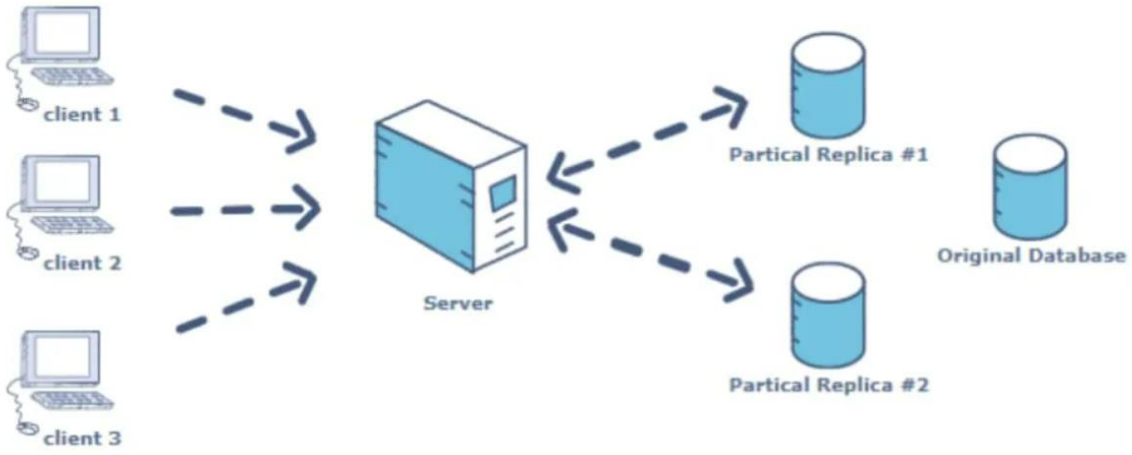
- Entiteti zauzimaju većinu ukupne količine podataka pa bi nam ovo donelo približno ravnomerno memorijsko zauzeće na svim serverima
- Mana ovog pristupa je: korisnici sistema bi često pretraživali entitete gde bi iz liste svih entiteta određenog tipa nalazili ono što ih zanima. Pri traženju svih entiteta iz baze, moralo bi se proći kroz svaki server da bi se prikupili ti podaci jer ne znamo na kojem je serveru koji tip entiteta. Zatim bi neki centralni server sve te podatke objedinio i poslao one koji odgovaraju datoj pretrazi. Ovo može veoma loše da utiče na performanse.

Drugi predlog: Particionisanje podataka na osnovu tipa entiteta i ReservationEntity ID-a

- Ukoliko bismo imali minimalno 3 servera, mogli bismo da rezervišemo određeni/e server/e za određeni tip entiteta i u okviru njih particionisali podatke na osnovu ReservationEntity ID-a
- Na ovaj način bismo rešili mane prethodnog pristupa a i dalje bismo uživali benefite koje donosi prvi pristup

Predlog strategije za replikaciju baze i obezbeđivanje otpornosti na greške

Repliciranje baze podataka igra krucijalnu ulogu u obezbeđivanju visokog nivoa dostupnosti podataka. Generisanjem više kopija kompleksnih skupova podataka i skladištenjem na više različitih lokacija dobićemo dva tipa skladišta, master (originalna baza podataka) i snapshot skladišta. Replikacija nam omogućava da svi korisnici imaju pristup istim podacima nesmetano i bez ikakvih nekonzistentnosti.



Naša aplikacija se pretežno zasniva na čitanju podataka, tako da je prirodno da imamo više pomoćnih servera koji će služiti samo da bismo iščitali određene podatke. Sam upis novih podataka može da obavlja glavni server baze podataka. U slučaju da glavni server nije dostupan, može da se odradi failover tj. da neki od pomoćnih server postane glavni

Predlog strategije za keširanje podataka

Keširanje podataka je implementirano u kodu. Za keširanje podataka smo koristili EhCache. Potrebno je konfigurisati cache i to smo učinili u ehcache.xml fajlu.

```
<config
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xmlns='http://www.ehcache.org/v3'
        xsi:schemaLocation="http://www.ehcache.org/v3
            http://www.ehcache.org/schema/ehcache-core-3.7.xsd">

    <persistence directory="cache-example/cache" />

    <cache-template name="default">
        <expiry>
            <ttl unit="seconds">15</ttl>
        </expiry>
        <resources>
            <heap>1000</heap>
            <offheap unit="MB">10</offheap>
            <disk persistent="true" unit="MB">20</disk>
        </resources>
    </cache-template>

    <cache alias="adventure" uses-template="default">
        <key-type>java.lang.Integer</key-type>
        <value-type>com.example.fishingbooker.Model.Adventure</value-type>
        <resources>
            <heap>2</heap>
        </resources>
    </cache>
```

Za svaki entitet (u servisima: AdventureService, LodgeService, ShipService) smo naznačili šta treba da se kešira, čita iz keša ili kada da se menja keš. Za to smo koristili anotacije @Cacheable, @CacheEvict i @CachePut. Primer:

```java
@Override
@Cacheable(value = "adventure", key = "'AdventureInCache'+#id")
public Adventure fetchById(Integer id) {
    return adventureRepository.findById(id).get();
}

@Override
@CacheEvict(value = "adventure", key = "'AdventureInCache'+#id")
public void deleteById(Integer id) {
    adventureRepository.deleteById(id);
}

@Override
@CachePut(value = "adventure", key = "'AdventureInCache'+#adventure.id")
public void updateAdventure(Adventure adventure) {
    adventureRepository.editAdventure(
            adventure.getName(),
            adventure.getDescription(),
            adventure.getDescription(),
            adventure.getMaxPersons(),
            adventure.getCancelConditions(),
            adventure.getFishingEquipment(),
            adventure.getId());
}
```

Okvirna procena za hardverske resurse potrebne za skladištenje svih podataka u narednih 5 godina

Pretpostavke:

- Ukupan broj korisnika aplikacije je 100 miliona
- Broj rezervacija svih entiteta na mesečnom nivou iznosi 2 miliona
- Sistem mora biti skalabilan i visoko dostupan
- 10% sistema čine vlasnici entiteta, koji u proseku imaju po 3 entiteta
- 7% rezervacija su akcije (brze rezervacije)
- Entiteti u proseku imaju 10 stavki u cenovniku
- Vikendice u proseku imaju po 4 sobe
- Entitet u proseku ima 3 perioda dostupnosti
- Entitet u proseku ima 1 definisanu akciju

- Svaki string u proseku sadrži 10 UTF-8 karaktera (10B)

Memorijsko zauzeće najzastupljenijih entiteta u sistemu (po torci):

- Location: 82B
- RegisteredUser: 94B
- Pricelist: 320B = 10 * 32B
- ReservationPeriod: 16B
- ReservationEntity: 112B = 56B + 4B (OwnerID) + 4B (LocationID) + 48B (ReservationPeriods)
- Bedroom: 16B = 12B + 4B (LodgeID)
- Lodge: 52B = 48B (4 sobe) + 4B (ReservationEntityID)
- Ship: 48B
- Adventure: 32B
- Reservation: 138B = 130B + 4B (ReservationEntityID) + 4B (ClientID)
- Image: ~500KB

Neophodni hardverski resursi za rezervacije na mesečnom nivou: 138B * 2 * 10^6 ~ 260MB

Neophodni resursi za čuvanje entiteta: 10m vlasnika/instruktora * 3 entiteta * (112 + 1/3 * (52 + 48 + 32)B + 2 slike * 500KB) ~ 30TB

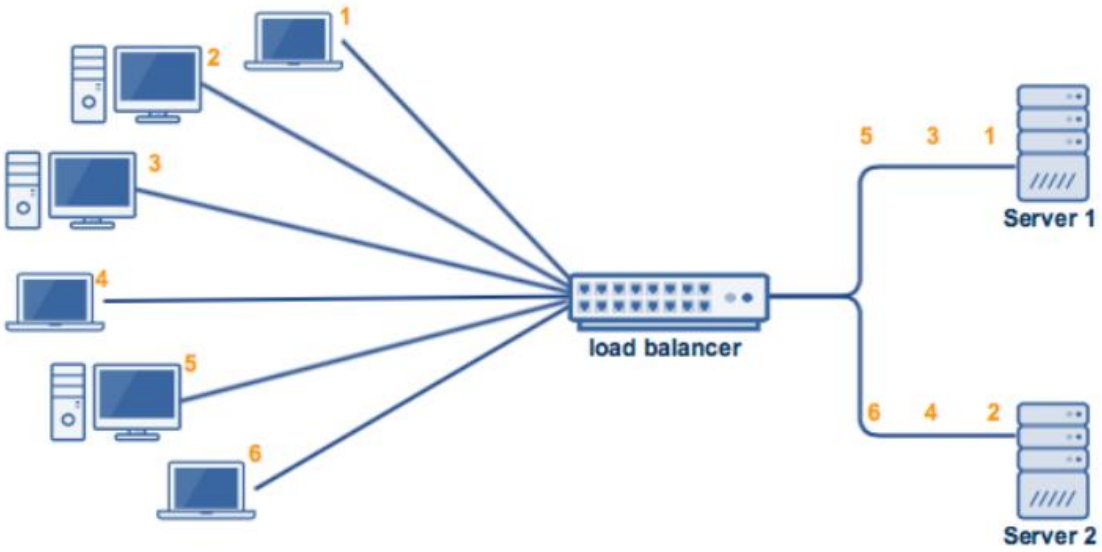
Neophodni resursi za čuvanje korisnika (klijenti i administratori): 90m * 94B = 8.5GB

Za period od 5 godina: 30TB + 8.5GB + 80MB ~ 30TB

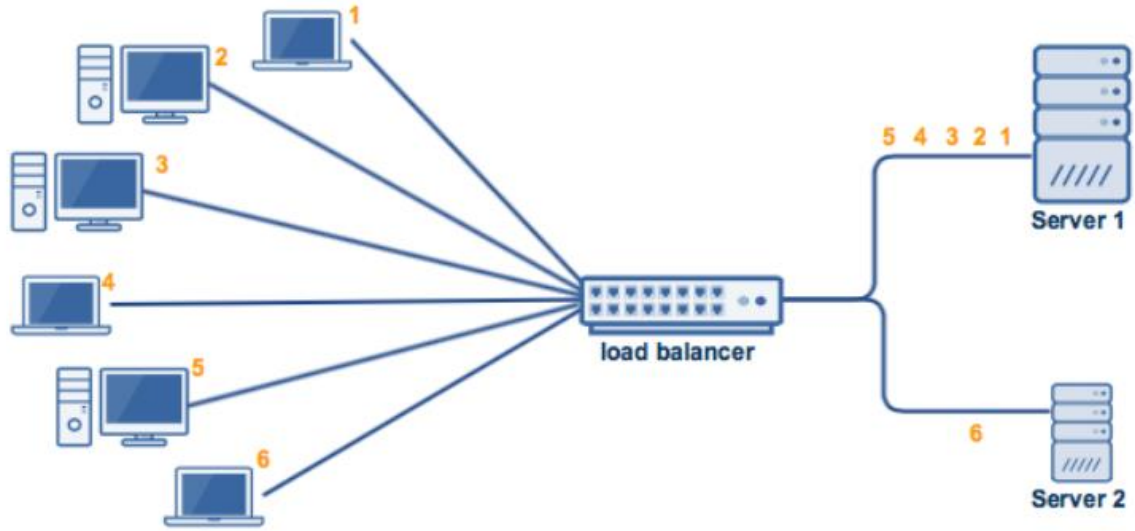
Predlog strategije za postavljanje load balansera

Postoje razne strategije i algoritmi za održavanje klijentskih zahteva širom serverskih grupa.

- Round Robin je jedan od najednostavnijih i najviše upotrebljavanih algoritama za load balancing. Klijentski zahtevi se šalju aplikativnim serverima u rotaciji. Npr. ukoliko imamo 2 servera: prvi klijentski zahtev se šalje prvom serveru, drugi klijentski zahtev drugom serveru, treći će se ponovo slati prvom serveru, i tako u krug. Ovaj algoritam ne uzima u obzir karakteristike servera već pretpostavlja da su svima karakteristike jednake i da su svi jednako dostupni.



- Weighted Round Robin predstavlja nadogradnju Round Robin algoritma kako bi se uzele u obzir različite karakteristike aplikativnih servera. Administrator dodeljuje težinu svakom serveru na osnovu kriterijuma po svom izboru kako bi rukovao saobraćajem između servera neke aplikacije.



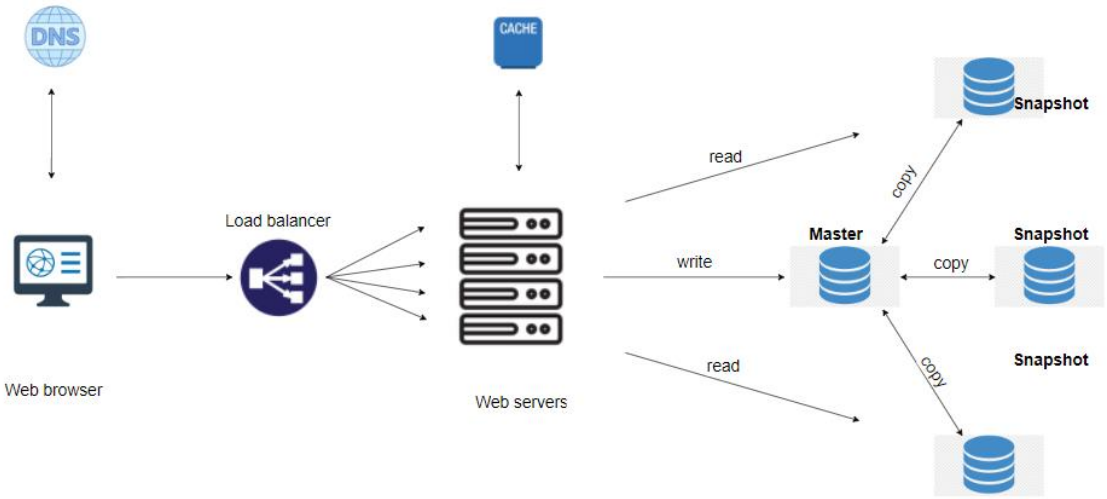
Predlog koje operacije korisnika treba nadgledatiu cilju poboljšanja sistema

Aktivnim posmatranjem našeg sistema možemo da uočimo nove strategije za poboljšanje sistema. Posmatrajući sistem imamo uvid kako sistem radi i možemo da uočimo određene nedostatke, ukoliko ih ima. U našem slučaju, od velikog značaja su podaci o broju rezervacija nekog entiteta ili avanture na osnovu kojih možemo da utvrdimo koji entiteti/avanture su traženiji pa primenjujemo odgovarajuće strategije za keširanja za entitete koji su „popularniji“.

Takođe, od značaja je i koje entitete klijenti najviše posećuju u okviru sajta, pa se na osnovu toga mogu unaprediti sistemi za preporuke za određene entitete i akcije vezane za njih.

Pored toga, mogu se pratiti i periodi kada klijenti vrše najveći broj rezervacija (oko praznika, neradnih dana, u koje doba dana/godine) i na osnovu toga dobiti informacije kada je najveća posećenost, odnosno saobraćaj aplikacije.

Kompletan crtež dizajna predložene arhitekture



Marija Petrović RA 53/2018

Sara Poparić RA 60/2018

Nikolina Pavković RA 56/2018