ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

IMPLEMENTACIJA TEHNIKA ZA POVEĆANJE BROJA PODRŽANIH KONKURENTNIH KORISNIKA VEB SAJTA

– Master rad –

Kandidat:

Janko Sokolović 2012/3142

Mentor:

doc. dr Zoran Čiča

Beograd, Septembar 2014.

SADRŽAJ

1. UVOD

Sa konstantnim porastom internet korisnika u svetu, sve više se skreće pažnja na održivost sistema sa aspekta visoke posećenosti. Inženjeri veb aplikacija imaju sve veći izazov pri dizajniranju rešenja kako bi sistem učinili održivim. Mnogi *startap* projekti, pokreću se od strane mladih entuzijista i vrlo često se u procesu razvoja zanemari strateško planiranje, već se odmah pristupi razvoju i kodiranju idejnog rešenja. Problem nastaje kada i ako aplikacija doživi veliki uspeh i postane *viralna*, ali tada je jako teško kontrolisati veliku posećenost i održavati sistem skalabilnim postaje težak zadatak. Značajnost dizajniranja softverskog rešenja jednog veb sajta, a da se od samog početka vodi računa o opsluživanju velikog broja konkurentnih korisnika, postaje suštinski važna stavka koju inženjeri moraju imati na umu. Ovaj rad je i nastao kao posledica jedog veb sajta koji je neočekivano brzo doživeo veliku popularnost, koja je sa sobom donela i velike probleme sa aspekta održivosti sajta. Susretnuti sa ovim problemima, autori su bili primorani da reaguju i rešavaju probleme dok je sistem bio u produkcijskoj fazi, a sve u svrhu izbegavanja smanjenja kvaliteta servisa ili u najgorem slučaju, vremena nedostupnosti sajta (eng. downtime).

Ovaj rad govori o opštoj i praktičnoj primeni tehnika koje omogućavaju veb aplikaciji ili servisu da nezavisno od hardverske konfiguracije, znatno poveća skalabilnost u vidu dozvoljenog broja konkurentnih korisnika, a da pritom kvalitet servisa ostane u zahtevanim granicama.

Rad je nastao kao posledica jedne veb aplikacije koja je za kratko vreme doživela veliku popularnost i samim tim veliku promenu sa tehničkog aspekta kako bi se ispratio brzo rastući trend. Broj posetilaca u početku nije prelazio 50-100 konkurentnih korisnika, međutim u kratkom roku taj broj se mnogostruko povećao i samim tim izazivao velika opterećenja na tada neoptimizovanoj aplikaciji. Ovaj servis je doživeo potpunu tehničku transformaciju i prošao je kroz razne faze optimizacije da bi sada uspeo da podrži i do 50 puta veću posetu bez smanjenja kvaliteta servisa.

U nastavku, najpre će se govoriti o opštim problemima sa kojima se susreću visokoposećeni sajtovi i veb servisi, kao i o najpopularnijim rešenjima kojima su savladane prepreke koje unosi veliki broj konkurentnih korisnika. U ovoj sekciji autor će obrazložiti i važnost sigurnosti sistema, kako je ovo jedan od najvažnijih aspekata jednog veb sajta.

Treće poglavlje, više će se fokusirati na same tehnike koje se koriste kako bi se postigao željeni cilj, odnosno veća skalabilnost i mogućnost opsluživanja znatno većeg broja korisnika bez smanjenja kvaliteta servisa. Govoriće se o skaliranju veb servera, optimizaciji baza podataka, CDN mrežama, kao i o najvažnijoj tehnici, keširanju podataka.

Četvrto poglavlje će predstaviti konkretan primer jednog ovakvog sajta koji je prošao kroz faze velikih promena, a sve u svrhu optimizacije i postizanja veće skalabilnosti. Detaljnije će biti opisane tehnike koje su korišćene, njihova implementacija kao i njihov ukupni doprinos da jedan visokoposećeni sajt ostane održiv i pod velikim pritiscima na serverskoj strani od strane velikog broja klijenata.

Nakon izlaganja svih korišćenih implementacija, autor će izložiti naredne korake koje ova veb aplikacija planira da inkorporira radi dalje održivosti sistema kao i optimizacije

neophodne za bolji kvalitet servisa. U petom poglavlju rada koji predstavlja zaključak, autor će zaključiti celokupnu priču i dati završnu reč o temi povećanja broja konkurentnih korisnika prilikom razvoja veb sajta.

2. PREGLED PROBLEMA SAJTOVA VISOKE POSEĆENOSTI

2.1. SKALABILNOST

Iako sama izrada veb aplikacija može delovati prilično prosto, izrada dobro postavljenih i operativnih aplikacija koje mogu da se skaliraju je veoma težak zadatak. Tehnike i tehnologije koje rade odlično na malim brojevima posetilaca, mogu potpuno da propadnu kada brojke krenu da rastu. Kako bi se izbeglo dodatno vreme i ulaganje dodatnog napora, najbolje je misliti o tome unapred.

Veb aplikacije koje opslužuju oko 10 miliona stranica u danu i obrađuju još nekoliko milion zahteva koji stižu putem Ajax poziva (ako uzmemo u pretpostavku da aplikacija koristi neku vrstu *API*-ja, o kojoj će biti reči kasnije) neophodno je razmišljati o skalabilnosti. Na primer, ako veb aplikacja, tj. sistem treba da opsluži 10 miliona zahteva u toku dana i imajući u vidu da svaki zahtev u proseku poziva oko 10 upita ka bazi podataka, lako se može zaključiti da komunikacija sa bazom podataka može da predstavlja usko grlo sistema. U navedenom primeru broj upita prema bazi iznosi preko 1,000 po sekundi, odnosno čak i više imajući u vidu da su vršna opterećenja sistema znatno veća od prosečnog opterećenja. Ovim je jasno da se mora posvetiti posebna pažnja komunikaciji sa bazom podataka kako bi se obezbedila skalabilnost čitavog sistema. Skaliranje i skalabilnost su jedna od najpopularnijih oblasti veb aplikacija i inženjeri dugi niz godina već diskutuju na ovu temu. Kada ljudi prave veb aplikacije koje su namenjene za veliki broj korisnika, skaliranje postaje problem: "Kako da podržimo sto korisnika? Hiljadu? Milion illi čak sto miliona?"

Iako je veoma puno pričano o ovoj temi, skaliranje još uvek nije jedinstveno definisano.[1]

2.1.1. PHP SKALIRANJE

Iako PHP sve više biva prihvaćen kao ozbiljan jezik, još uvek postoji veliki broj ljudi koji tvrdi da ovaj jezik ne može da se skalira. Očito, ovo nije slučaj budući da mnogi sajtovi od kojih su neki među najvećim na svetu, su kreirani baš u PHPu. Razlog zbog kojeg PHP jeste skalabilan dovodi do pitanja šta je skalabilnost zapravo. Mnogi ljudi kritikuju PHP i njegovu mogućnost skaliranja sudeći po nekim aspektima u kojima PHP pokazuje slabije performanse. Ali to ne znači da aplikacije koje koriste PHP ne mogu da budu skalabilne. Postoje tri osnovna kriterijuma skalabilnosti, i u nastavku će biti data analiza da li PHP ulazi u zahtevane granice jednog skalabilnog programskog jezika.

Jedan od kriterijuma, održavanje, se lako da razumeti. Svaki jezik može da se koristi tako da kreira dobro ili loše organizovan kod i samim tim lako ili teško struktuiran kod za održavanje. Samim tim što ima striktna strukturalna i stilska pravila, PHP aplikacije mogu lako biti održavane.

Porast skupa podataka, je drugi kriterijum skalabilnih veb sistema. Kod PHP veb aplikacija, podaci su potpuno odvojeni od procesiranja. Inženjeri ne moraju da brinu mnogo o količini podataka, jer će načini kreiranja upita uvek biti uniformni. PHP prepušta odgovornost

skupa podataka i njegovog rasta sloju baze podataka, što dozvoljava mogućnost proizvoljnog skaliranja.

Poslednji kriterijum je dozvoljavanje rasta saobraćaja. PHP ovo rešava kroz paradigmu kreatora PHP-a Rasmus Lerdorfa, koja glasi - *shared nothing architecture*, što znači deljeno-ništa-arhitektura. PHP se ponaša kao servis bez stanja kao što je HTTP tako što odgovornost stanja prepušta nižim slojevima. PHP proces se izvršava tako što se obrađuje samo jedan zahtev u trenutku i ne može da ima bilo kakvu korelaciju sa drugim procesima koji opslužuju druge zahteve. Procesi ne pamte informacije između zahteva, što je zapravo definicija servisa bez stanja (*stateless*). Ovaj princip izoluje svaki zahtev i ne deli ništa između drugih zahteva drugih procesa. Ukoliko nam je potrebno da zapamtimo informacije iz jednog zahteva i koristimo ih kasnije u drugom, ovu funkciju obavlja sloj ispod koji čuva ove podatke. Samim tim, mi uopšte ne moramo da koristimo isti server za obrađivanje narednih zahteva. Kako bismo obrađivali više zahteva, jednostavno možemo dodati više veb servera. Ovi principi nisu samo osobina PHPa već čitava filozofija REST (Representational State Transfer) servisa. Sa ispravnim dizajnom aplikacija, ovaj princip može da se primeni nezavisno od jezika implementacije. Ovo nije istina samo za neke PHP servise, kao što je čuvanje sesija, koje PHP kreira na hard disku, ali postoje određeni alati koji omogućavaju da i ovaj konkretan primer postane *stateless*.[1]

2.2. Sigurnost

Sigurnost predstavlja veoma važnu temu koja se tiče svih vrsta sajtova, a naročito visokoposećenih, budući da popularnost sajta implicira da će postojati veći afiniteti od strane zlonamernih napadača da obore sajt ili probaju da ukradu određene informacije. Iako sigurnost nije osnovna tema ovog rada, autor smatra da je bitno napomenuti neke od osnovnih principa napada i načina odbrane od istih radi povećanja bezbednosti sajta. Naravno, nijedan sistem nije neprobojan, ili perfektno zaštićen i to treba imati na umu.

2.2.1. MySQL Injection

Većina današnjih sajtova, pogotovo onih sa velikim brojem posetilaca, je dinamičke prirode. Uglavnom se više ni ne prave veb aplikacije koje su potpuno statične. To znači, da u pozadini, u jednom trenutku, sajt komunicira sa nekom vrstom baze podataka. Postoje razne vrste baza podataka i stalno nastaju nove, ali još uvek jedna od najpopularnijih jeste MySQL. SQL (SQL - *Structured Query Language*, strukturalni jezik upita) predstavlja jezik koji služi za obavljanje komunikacije nad relacionim bazama podatka. Baze podataka služe za skladištenje informacija kojima sajt dinamički pristupa. Kada korisnik potraži neku stranicu, najverovatnije je da je veb server u nekom trenutku, neke podatke potražio iz baze. Takođe, mnogi podaci mogu da dođu od korisnika, kao što je e-mejl, lozinka, komentar ili čak kraća priča u obliku teksta – blog. Sve ove operacije sa bazom se nazivaju *upiti*. Upit je, dakle, neka vrsta obraćanja bazi sa određenim ciljem, upisivanja, čitanja, brisanja ili menjanja u bazi podataka. Prilikom svakog od ovih upita, kreira se SQL upit kojim se zadaju komande ka bazi.

Na primer:

```
SELECT ime, prezime FROM korisnici WHERE email = 'marko.markovic@mail.com'
AND password = 'mojpassword'
```

Sa prethodnog upita, može se lako zaključiti, da je informacije e-mejl i lozinku, korisnik uneo sa klijentske strane. Ovo znači, da korisnik na neki način može da utiče na komandu koja se izvršava nad bazom. To predstavlja veliku opasnost i veliki potencijal za napadače da nanesu štetu čitavom sistemu.

Uzmimo na primer da neko dođe do saznanja (pretpostavkom ili drugim putem) da se tabela zove *korisnici*. Ukoliko ne postoji nijedan nivo zaštite, napadač bi lako mogao da *probije* u bazu tako što bi umesto svoje lozinke uneo sledeću konstrukciju

‘ OR 1=1--’

Ovo bi stvorilo sledeći upit

```
SELECT ime, prezime FROM korisnici WHERE email = ‘marko.markovic@mail.com’
AND password = ‘’ OR 1=1--’
```

Na ovaj način, napadač ne mora znati pravi password jer će uslov u upitu uvek biti ispunjen zbog 1=1 i na ovaj način je čitav sistem kompromitovan.

Drugi način je da unese nešto kao što je ‘; drop table korisnici --’

Ovo bi značilo da, iako je neuspešno ulogovan, izvršio bi se i drugi upit sekvencijalno, koji za cilj ima brisanje čitave tabele iz baze.

Ovde smo ukratko pokazali jedan od napada sa najozbiljnijim posledicama, ukoliko ne bi bilo zaštite.

Kako se zaštiti od SQL Injection-a? Postoje razne metode zaštite od ovog napada, a jedna od najbezbednijih jeste parametrizacija upita i uklanjanje zaštićenih ključnih znakova. Na taj način sistem zna da na određenom mestu očekuje parametar i tu se ne može naći drugi upit.

U nastavku je prikazan primer zaštite upita parametrizacijom u PHP jeziku.

```
<?php

$stmt=$db->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// upisivanje jednog reda
$name = 'one';
$value = 1;
$stmt->execute();

// upisivanje još jednog reda sa drugim vrednostima
$name = 'two';
$value = 2;
$stmt->execute();

?>
```

Prvom instrukcijom u ovoj skripti inicijalizuje se objekat *$stmt* izvršavanjem funkcije *prepare* sa odgovarajućim argumentom koji predstavlja string u kojem su referencirana dva polja koja očekuje ova instrukcija. Radi se o poljima koji kao prefiks sadrže dve tačke, odnosno *:name* i *:value*. Naredne dve instrukcije rade *binding* parametara, odnosno poziva se funkcija *bindParam* koja govori SQL upitu koje vrednosti da očekuje na pozicijama parametara *name* i

value. Naredne instrukcije predstavljaju prostu dodelu vrednosti promenljivama *$name* i *$value*, a instrukcija *$stmt->execute();* izvršava odgovarajući upit i ukoliko ne bude grešaka, u tabelu REGISTRY biće upisan novi red sa vrednostima "one" i "1" u kolone "name" i "value" respektivno. Naredne tri instrukcije predstavljaju analogni primer koji govori o tome da nema potrebe za ponovnom inicijalizacijom upita veće je moguće izvršiti proizvoljan broj upita kada je instrukcija kreirana.

Način na koji ovaj princip štiti sistem od SQL Inject napada, ogleda se u tome da sistem prepoznaje da su *name* i *value* dva parametra i oni ne mogu biti posebne instrukcije, samim tim svaki pokušaj unošenja ključnih reči biće ignorisan odgovarajući znakovima (eng. escape character) i u bazu će se upisati običan *string*, odnosno niz znakova.

2.2.2. XSS – CROSS-SITE SKRIPTOVANJE

Još jedan od napada sa kojim se autor susreo prilikom razvoja, a putem kog napadač može da nanese veliku štetu jeste XSS. Jedan od osnovnih primera ovog napada jeste sledeći: Uzmimo za primer sajt koji na naslovnoj strani prikazuje informacije koje konstantno unose korisnici. Nešto kao kratke poruke na svojim profilima. I pretpostavimo da Korisnik 1, koji je napadač, postavi komentar na profilnoj strani Korisnika 2. Međutim, Korisnik 1 neće uneti bilo kakav komentar, već nešto kao što je dato na sledećem primeru:

```
<script> while(1) alert(“Ovo je XSS napad!”); </script>
```

Ukoliko se ne izvrši uklanjanje potencijalno opasnog koda, u ovom slučaju, bi se kod svakog korisnika koji učita stranicu sa ovim komentarom, na klijentskoj strani izvršila skrpita, jer bi svaki pretraživač ovaj deo koda tumačio kao javascript kod, budući da je smešten u okviru <script> tagova. Ova akcija bi beskonačno puta izpisivala alert dijalog sa porukom „*Ovo je XSS napad!*“.

U nastavku su dati primeri PHP funkcija sa kojima je moguće izbeći ovaj napad i očistiti kod od pokušaja upisivanja malignih informacija.

1. ***strip_tags***

Strip_tags je php funkcija koja za ulogu ima uklanjanje svih html tagova u određenom stringu, Na ovaj način, ukoliko krajnji korisnik pokuša da ka serveru uputi maligni kod sa određenom javaskript funkcijom, kao izlaz *iz strip_tag()* metode, u bazu će se upisati samo kod bez odgovarajućih tagova i na taj način nikada neće biti izvršeno kod narednih klijenata. Treba imati na umu da ova funkcija ne radi zaštitu od SQL napada već samo od XSS pokušaja probijanja sigurnosti sistema. Primer ove funkcije je prikazan u sledećem kodu:

```
<?php
  echo strip_tags("Hello <b><i>world!</i></b>","<b>");
?>
```

Ovaj primer će na korisničkoj strani prikazati html sa sledećim sadržajem: Hello world**!** Može se primetiti kako je tekst ipak postao boldovan, a razlog zbog čega se ovo desilo jeste drugi parametar u funkciji *strip_tags*, odnosno lista html tagova koje se dozvoljavaju prilikom izvršavanja ove funkcije. Tekst nije postao *italic* zato što to predstavlja html tag koji nije prosleđen kao drugi parametar.

2. ***htmlentities*** i ***htmlspecialchars***

Ove dve funkcije razlikuju se od *strip_tags*, ali se takođe koriste za zaštitu od XSS napada. Drugim rečima, Ove metode ne skidaju u potpunosti tagove, već ih putem specijalnih html znakova, samo konvertuju u odgovarajuće kodove koje pretraživač ume da prepozna i korisniku ispiše odgovarajući znak. Jedan primer ovog znaka je < koji će se konvertovati u **<** (eng. less than)

```
<?php
$str = "A 'quote' is <b>bold</b>";

// Ispisuje: A 'quote' is &lt;b&gt;bold&lt;/b&gt;
echo htmlentities($str);

// Ispisuje: A &#039;quote&#039; is &lt;b&gt;bold&lt;/b&gt;
echo htmlentities($str, ENT_QUOTES);
?>
```

Na prethodnom primeru, može se videti način funkcionisanja ***htmlentities()*** funkcije. Kako se <b> tag prepoznaje kao html entitet, ova funkcija će razložiti posebne znakove u odgovarajući kodovane sekvence koje će se kasnije na pretraživaču reprodukovati kao pravi znak i korisniku će se prikazati u originalnoj formi, bez potencijalnih posledica.

Drugi primer uključuje i drugi parametar koji je dodatna opcija prilikom parsiranja stringova koji se validiraju, i zbog toga je čak i apostrof konvertovan u određeni niz karaktera koji će kasnije biti prikazan kao što je bio u originalu.

Metoda *htmlspecialchars* ima vrlo sličnu funkciju a razlika je samo u znakovima koje će prevoditi. *htmlspecialchars* je poznata po tome da je brža jer se izvršava samo u nekoliko slučajeva, odnosno radi konverziju mnogo manjeg broja znakova.

Takođe postoje mnoge biblioteke koje se bave ovom problematikom a jedna od najpoznatijih je HTML Purifier.

3. TEHNIKE ZA POSTIZANJE VEĆE SKALABILNOSTI

3.1. SKALIRANJE VEB SERVERA

Proširenje u vidu dodavanja novog hardvera, odnosno novih server mašina, je uvek opcija. Ali je potrebno pre toga voditi računa da pri eksponencijalnom širenju posete, ne dođe do potrebe eksponencijalnog dodavanja novih servera. Zapravo, potrebno je optimizovati sistem tako da je skalabilan, odnosno da veliko povećanje konkurentnih korisnika zahteva skromno proširenje u vidu dodavanja novih komponenti.

Jedan od trenutno najkorišćenijih veb servera na svetu je Apache httpd server, koji zauzima i preko pola tržišta. Apache dugi niz godina drži reputaciju pouzdanog httpd servera sa velikim brojem mogućnosti za konfigurisanje. Međutim, jedna od velikih mana Apache-a jeste njegova moć pri ogromnim količinama saobraćaja. Budući da je Apache *process-based* veb server, prilikom svakog zahteva ka serveru, Apache formira proces koji zauzima određenu količinu memorije. RAM memorija jedna je od najvažnijih komponenti servera koji treba da podrži veliki broj konkurentnih korisnika. Imajući ovo u vidu, za veliki broj zahteva, alociraće se velika količina RAMa i ukoliko ponestane memorije, server može početi da ispušta, tj. odbija upite. Ovo je razlog zbog kojeg su se pojavili mnogi "lakši", brži veb serveri, čije performanse pokazuju mnogo bolje rezultate od veb servera *Apache*.

Jedan od njih je Nginx. Noviji je na tržištu i to je razlog zbog kojeg još uvek mnogi visokoposećeni sajtovi strahuju da pređu na njega. Mnogi uporedni testovi pokazuju da se mnogo lakše nosi sa velikim brojem uzastopnih upita bez zagušivanja servera. Osnovni razlog je što je za razliku Apache-a koji se baziran na dodeljivanju processa za svaki upit, Nginx zasnovan na događajima (*event-based*), što znači da ne dodeljuje poseban proces svakom http zahtevu već postoji jedan glavni process i da kroz njega rešava sve konekcije sa korisnicima. Takođe je asinhron jer rešava više od jednog zahteva u trenutku vremena

3.2. SKALIRANJE BAZE PODATAKA

Baza podataka je skoro uvek usko grlo. Svi dinamični sajtovi koriste neku vrstu baze podataka, i ona, iako je pouzdan i moćan alat za čuvanje podataka, crpi velike hardverske kapacitete. Ukoliko baza ima više desetina miliona upisa, i koristi veliki broj upisa i čitanja, jako je bitno dobro struktuirati bazu na taj način da se upiti izvršavaju sa minimalnim memorijskim zauzimanjem. Potrebno je pravilno postaviti indekse nad kolonama koje se koriste prilikom spajanja tabela ili u *WHERE* uslovu.

Indeks nad kolonama predstavlja mapu svih vrednosti u određenoj koloni i omogućava mnogo brže pretraživanje i ovo je posebno značajno za tabele sa velikim brojem upisa. Uzmimo za primer upit kojim želimo da dobijemo sve korisnike koji su tinejdžeri.

```
SELECT name, age, email FROM users WHERE age > 12 AND age < 20
```

Primetimo da poslednji deo ove instrukcije govori SQL serveru filter po kojem želimo da dobijemo podatke. Ukoliko se ne koristi nikakav indeks, SQL upit će zahtevati mnogo vremena kako bi prošao kroz čitavu tabelu, i na svakoj poziciji na kojoj naiđe na broj godina koji ispunjava ova dva uslova, smatraće kao jedan od traženih rezultata. Međutim, ukoliko bi kolona *age* bila indeksirana, ovaj proces bi se izvršio mnogo brže, jer bi SQL server imao mapu svih brojeva i ne bi bilo potrebe da pretražuje čitavu tabelu već samo određeni deo koji sadrži brojeve od 13 do 19. Iako je indeksiranje kolona jako korisna alatka, potrebno ju je koristiti pažljivo budući da koliko stvara olakšanja prilikom čitanja, može stvoriti isto toliko poteškoća prilikom upisa, jer se za svaki novi red u tabeli mora proći i osvežiti indeks kako bi i novi upis bio indeksiran. Indeks takođe utiče na veličinu koju baza zauzima i zato treba voditi računa pri njihovoj upotrebi.

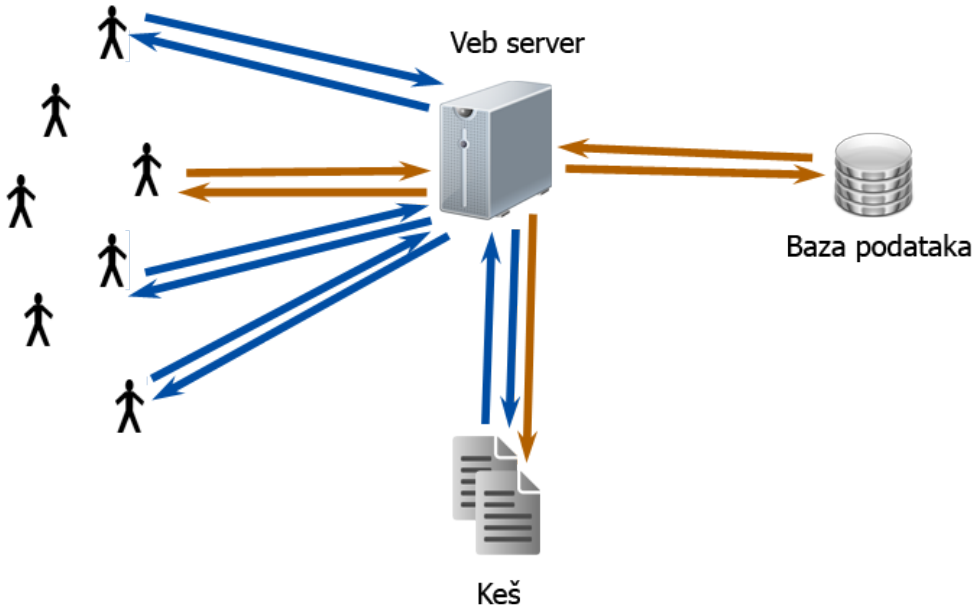
Jedno od mogućih rešenja jeste korišćenje više SQL servera. Dobra je opcija odvojiti upise i čitanja. Može postojati struktura sa jednom glavnom (master) bazom, koja je zadužena za prihvatanje upisa, kao i za održavanje sinhronizacije sa jednom ili višeslejv, sporednih SQL servera. Prilikom čitanja, što je najčešća situacija, koristi se jedan ili više servera koji služe samo za čitanje.

Druga opcija je koristiti neke od novijih baza podataka, koje su poznatije kao NoSQL baze. One ne spadaju u relacione sisteme baza podataka već koriste drugačije principe. NoSQL je skraćenica od *Not Only SQL database*. Ove nove baze podataka su nastale kao posledica tradicionalnih SQL baza čije performanse postaju ograničenje prilikom velikog broja pristupa, bilo čitanja ili upisivanja. Ove baze su mnogo brže i skalabilnije i koriste se u velikim sistemima koji analiziraju velike količine podataka. Postoji i mogućnost interoperabilnosti. Budući da su SQL baze još uvek poznate kao pouzdaniji izvori za skladištenje podataka, moguće je napraviti sistem u kojem se podaci smeštaju u MySQL bazu, dok na nivou ispred nje, ka klijentu, postoji NoSQL baza koja služi za brzo i dinamično opsluživanje visokog saobraćaja.

3.3. KEŠIRANJE

Ukoliko unapredimo vreme izvršavanja za 50%, koda koji se izvršava u 2% slučajeva je dobar rezultat. Ali je mnogo bolji rezultat unaprediti vreme izvršavanja za samo 10% koda koji se izvršava u 80% slučajeva.[3] Drugim rečima, prilikom razmišljanja o skaliranju, optimizacija keširanjem svodi se na ponovnu upotrebu podataka koji se često koriste. Isto to važi i za vreme izvršavanja koda. Keširanje podataka, ili generisanih stranica, predstavlja jedan od osnovnih principa povećanja moći opsluživanja neke veb aplikacije. Fraza keširanje potiče od procesorske keš memorije, koja je najbrža vrsta memorije i služi za skladištenje informacija koje se jako često upotrebljavaju i procesor njima može najbrže da pristupi. Odatle je i poteklo upotrebljivanje ovog termina za čuvanje podataka u svrhu čestog i brzog pristupa.

Kada se za podatke nekog veb servisa kaže da su keširani, misli se da su **privremeno** sačuvani u radnoj memoriji ili na disku u takvom formatu da su spremni za isporučivanje klijentima sa vrlo malo kašnjenja i vrlo malo obrađivanja. Princip ovakvog keširanja je prikazan na slici 3.1.



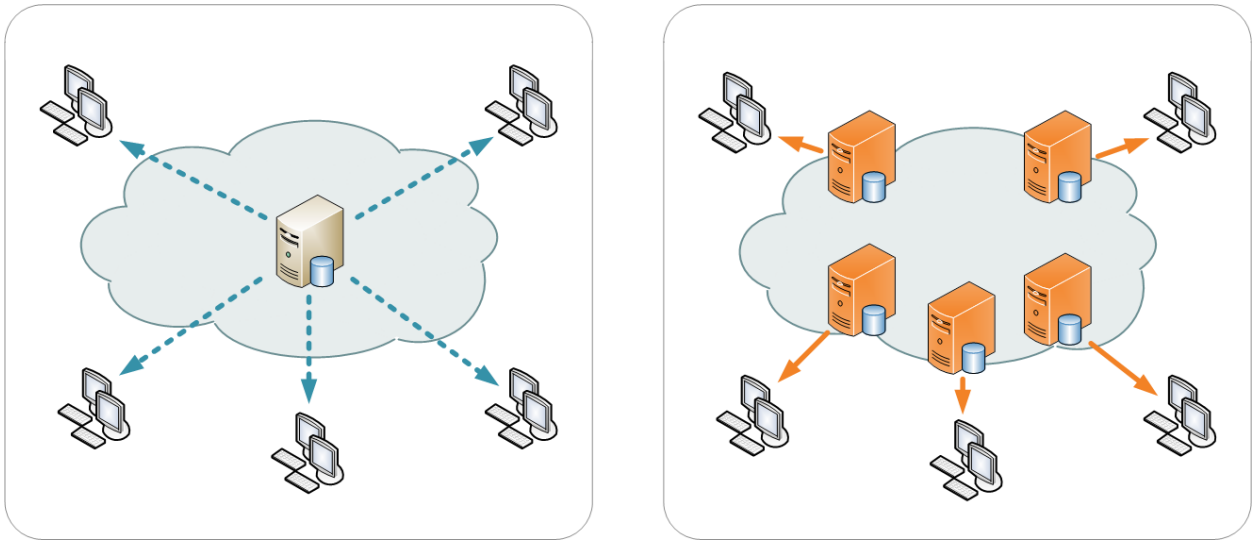
Slika 3.3.1 - Prikaz logike keširanja podataka i narednih pristupa

Na slici 3.1 možemo razlikovati dva tipa obrađivanja zahteva. Narandžastom bojom na slici je obeležen inicijalni zahtev. Kada ovaj zahtev stigne do servera, budući da je prvi put da se zatraži određena stranica, i samim tim ne postoji u kešu, server mora da se obrati bazi podataka i izvrši jako skupu operaciju kako bi formirao stranicu koju je korisnik zatražio. Dok server formira stranicu i šalje kao HTML kod ka klijentu u isto vreme se vrši čuvanje podataka u kešu. Ovo je na slici predstavljeno narandžastom linijom koja ide od servera ka kešu. Keš nema potrebe da vrati ništa veb serveru jer kod njega već postoje sveži podaci koji su upravo upisani u keš.

Plavom bojom obeleženi su zahtevi i odgovori koji dolaze neposredno nakon što je opslužen inicijalni korisnik. Ovi procesi su mnogo lakši jer ne koriste operacije sa bazom podataka i složene kompleksne kalkulacije, već veb server samo treba da za zahtevanu stranicu u kešu pronađe i pročita odgovarajuću kopiju koju nakon toga prosledi korisniku. Iskorišćenost keša se meri odnosom pogotka i promašaja, a jasno je da za što veći interval keširanja, kao i što veću posećenost u kratkom intervalu, iskorišćenost keširanja raste.

3.4. OPTIMIZACIJA KOMUNIKACIJE I CDN

CDN (*Content Delivery Network.)* takođe predstavlja jednu vrstu keširanja. Može postojati posebni hardverski uređaj koji služi samo za prosleđivanje statičnih podataka ili čak čitav skup distribuiranih servera koji na bazi lokacije vraćaju velikom broju korisnika statičan sadržaj koji se ne menja relativno često. Na ovaj način glavni server obrađuje samo neophodne podatke, kao što je na primer novi upis podataka. A narednih n korisnika koji potraže te podatke, sadržaj će dobiti od proksi servera koji brzo opslužuju statičke podatke koje kada postanu zastareli ponovo dobija od glavnog servera.



Slika 3.4.1. - Prikaz sistema bez CDN mreže (levo) i sa CDN-om (desno)

Na slici 3.2. možemo videti uporednu strukturu mreže kada se ne koristi, odnosno koristi CDN tehnologija. U prvom slučaju, sa leve strane, možemo videti da svi klijenti direktno zahtevaju sadržaj od servera. Na ovaj način sva težina procesorskih i memorijskih zahteva prenosi se na jednu mašinu koja je primorana da obrađuje ogroman broj zahteva, ukoliko govorimo o visokoposećenom veb sajtu. Sa druge strane vidimo distribuirano rešenje, odnosno CDN sistem. Distribuirane mašine imaju ulogu da čuvaju statične podatke odnosno podatke koji se retko kada menjaju, kao što su multimedijalne fajlove, javascript ili stilove, odnosno CSS. Na ovaj način glavni server je značajno rasterećeniji jer on zapravo komunicira sa CDN sistemom i to povremeno, kako bi osvežio podatke u slučaju da je došlo do nekih izmena. Još jedan veliki benefit CDN-a koji se primećuje sa slike 3.4.1. jeste taj što se za serviranje podataka korisnicima, koristi fizički najbliži izvor i na ovaj način se optimizuje kašnjenje usled ograničene brzine prenosa informacija kroz Internet infrastrukturu.

Jedan od osnovnih načina optimizacije je definitivno minimizacija broja HTTP zahteva. RTT (*Round Trip Time*) period, pored uspostavljanja konekcije, može drastično da poveća kašnjenje i uspori proces odziva a samim i umanje subjektivno korisničko iskustvo. S ovim na umu, potrebno je omogućiti interfejs sa što manjim brojem HTTP zahteva ka serveru. Imajući u

vidu da su multimedijalni sadržaji veoma česta pojava na stranici, možemo pretpostaviti da se na stranici nalazi čak nekoliko desetina slika različitih veličina. Pored toga što treba voditi računa o veličini slika, odnosno ne dozvoliti da se na strani učitavaju jako velike slike bez potrebe, pa da se smanjuju kroz razne atribute html elemenata ili css stilove, još je važnije smanjiti ukupan broj ovih zahteva. Kako bi se ovo ostvarilo, koristi se metod poznat kao *sprites*. Sprites metoda optimizacije se ogleda u tome da se velik broj slika, ili uglavnom sličica manjih veličina, kao što su ikonice i sl. postave na jednu sliku, a onda se na klijentu uz pomoć odgovarajućih css stilova, podese x i y koordinate na taj način da se korisniku prikazuje odgovarajuća slika. Na ovaj način, umesto npr. 10 http zahteva, klijent pravi samo jedan jedini zahtev, za koji dobija u odgovoru čak 10 slika. [2]

Takođe postoji i vremensko keširanje fajlova kao što su css i js. Dobra je praksa da se stilovi i skripte klijentske strane (javascript) postavljaju u izolovane fajlove i na taj način osim što se doprinosi boljoj arhitekturi sistema, takođe se dobija na performansama. Ukoliko imamo jedan veliki css fajl koji sadrži više hiljada linija, njega bismo morali prenositi kroz mrežu svaki put pri svakom pozivu stranice, ukoliko bi se nalazio direktno na stranici. Na ovaj način, kada se nalazi u potpuno odvojenom fajlu, t.j url strani, možemo pretpostaviti da se ovaj fajl neće menjati skoro, i s toga ga možemo čuvati na klijentskoj strani neki određeni predefinisani period. Ova tehnika se naziva keširanje fajlova na klijentu. Posebno je zahvalno napraviti određeni url poziv za css i js fajlove sa određenim parametrima koji klijentu govore o vremenu modifikacije fajla. Na ovaj način će čim fajl bude promenjen, klijent automatski zahtevati novu verziju fajla koju će potom čuvati kod sebe u pretraživaču. Primer ovakvog načina imenovanja fajlova se vidi u nastavku

```
<link rel="stylesheet" type="text/css" href="/css/styles.min.css?1395837833" />
```

Na ovom primeru možemo da primetimo da se radi o fajlu koji definiše stilove, odnosno css fajlu (type="text/css"), kao i iz atributa rel (rel = "stylesheet"), dok nam href atribut govori o destinaciji fajla.

Vodeća kosa crta, *sleš*, znači da se fajl traži na apsolutnoj putanji u odnosu na glavni direktorijum (*root folder*). Nakon toga sledi putanja i naziv fajla (css/styles.min.css), potom slede parametri iza znaka pitanja. Ovo je univerzalni primer adrese, a parametri su opcioni niz ključ-vrednost koji se šalje ka serveru radi specificiranja željenog sadržaja ili kao slanje podataka. U ovom slučaju ne postoji nakakav parametar kojem je dodeljena vrednost broja, već samo postoji niz cifara koji predstavljaju vreme u unix formatu i ova prezentacija vremena se odnosi na broj koliko sekundi je proteklo od 1. Januara 1970 godine. Takođe, može se koristiti i drugačiji vid forsiranja pretraživača da osveži fajl, kao što je menjanje adrese, ali je onda potrebno svaki put kada se izvrši promena, ponovo osvežiti i ovaj broj, odnosno adresu.

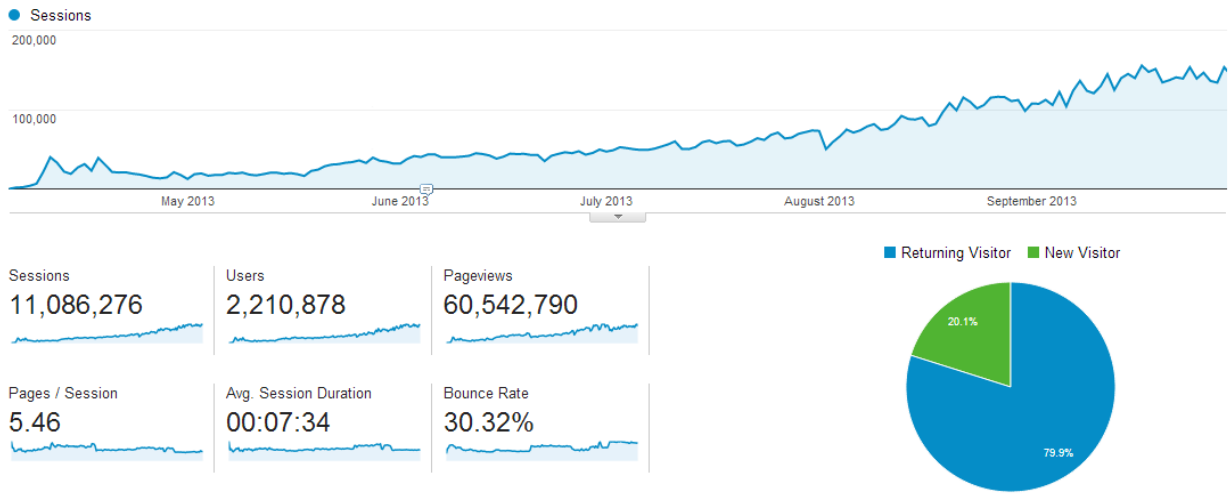
Primetili smo da postoje određeni statični fajlovi koji se razlikuju od glavnih stranica veb aplikacije koje su dinamične i potrebno ih je učitavati iznova posle svakog zahteva ka serveru.

Ovi fajlovi, multimedijalni ili css i js, mogu se takođe premestiti na druge lokacije, koje samo služe za izuzetno brzo služenje fajlova, i koje su često distribuirane na taj način da se klijentu najbrže dostavi zahtevani fajl. Na ovaj način, umesto da fajl pozivamo sa našeg servera, sve statične izvore možemo premestiti na CDN mrežu koja će ubrzati učitavanje stranica i olakšati opterećenje na serveru koji služi dinamični sadržaj.

Postoji i tehnika koja se zove kompresija u letu (*on-the-fly compression)*, koja može dodatno da smanji veličinu prenošene informacije sa minimalnim uticajem na CPU. Ova tehnika je veoma dobra za aplikacije bazirane na tekstu ili za sisteme sa ogromnom količinom sintakse kao što je html, xml i sl.

4. PRIMER SAJTA VISOKE POSEĆENOSTI I KORIŠĆENE TEHNIKE

Sajt koji će se koristiti kao primer u okviru ovog poglavlja napravljen je kao privatni projekat tokom pripreme ispita iz Internet programiranja, i lansiran je 6. aprila 2014. godine. Domeni ispovesti.rs i ispovesti.com usmeravaju na ispovesti.com. Ova veb aplikacija sada generiše i preko 30,000,000 pregleda stranica na mesečnom nivou, kako putem desktop računara tako i putem mobilnih i tablet uređaja. Ovaj portal podržava i *responsive design*, odnosno prilagođava se mobilnim uređajima, a od nedavno postoji *native* aplikacija za Android i Windows phone uređaje, a u planu je i iOS aplikacija.



Slika 4.1. - Prikaz statistike sajta ispovesti.com u prvih 6 meseci postojanja (Izvor - Google Analytics)

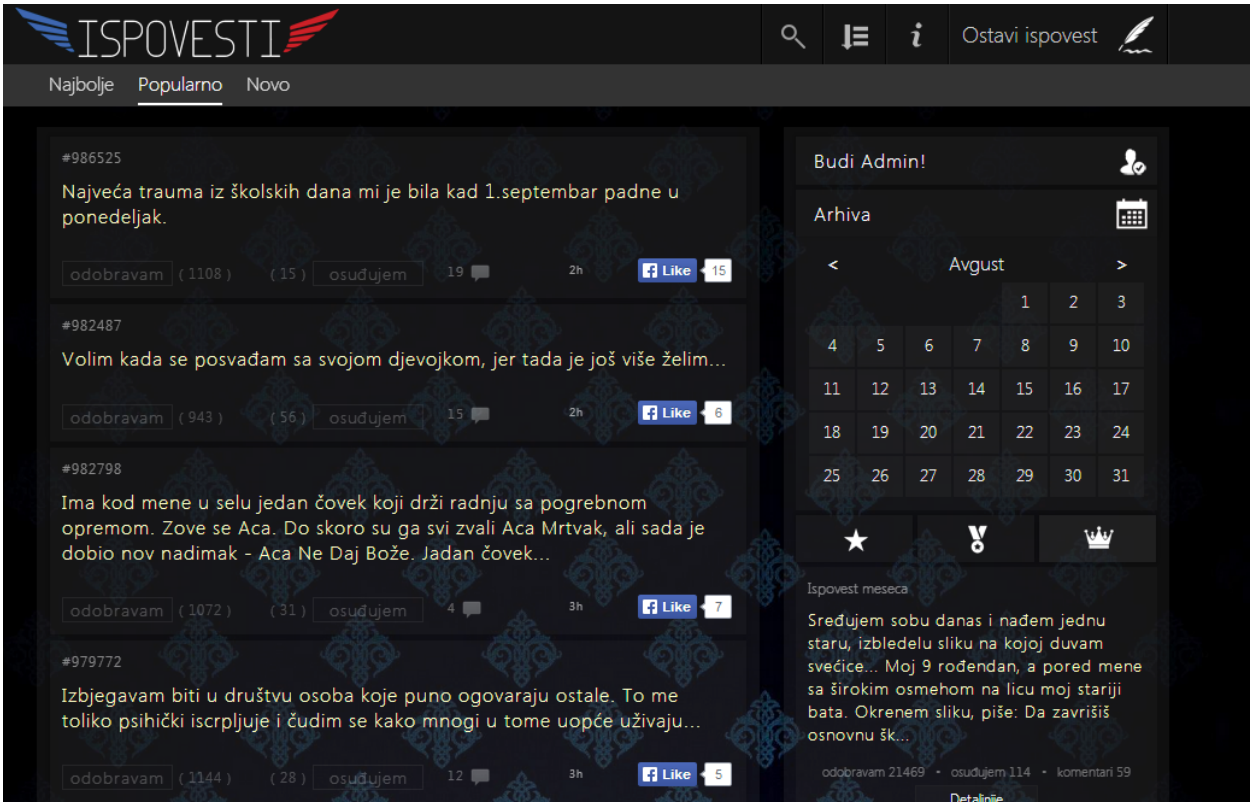
4.1. MOTIVACIJA

Prvi osnovni problem sa kojim se autor susreo bila je baza podataka. Naime, ogroman broj konkurentnih zahteva, od kojih svaki poziva upit ka bazi podataka, izaziva povlačenje resursa i otvaranje novih konekcija čiji je broj ograničen i na taj način se ceo sistem zagušuje. Bilo je neophodno podatke iz baze čuvati u nekoj vrsti memorije, privremeni period, kako bi se minimizovao broj direktnih upita sa bazom. Takođe je bilo potrebno optimizovati strukturu baze tako da se određeni upiti što lakše izvršavaju bez angažovanja velikih procesorskih kapaciteta.

Nakon određenog vremena, ni ovaj princip nije bio dovoljan, tako da je postalo neminovno podići optimizaciju na još viši nivo, jer se u trenucima najvećeg opterećenja, javljalo jako zagušenje usled ogromnog broja zahteva ka serveru. Korišćeni su i određeni principi kojima se smanjuje broj zahteva putem http-a kako bi se smanjio prenos od servera ka klijentu.

U najkasnijoj fazi su korišćene i tehnike kao što su APC, kojim se značajno dobija na brzini čitanja i pisanja keširanih podataka, budući da se sve smešta u operativnu memoriju, umesto na hard disk, kao što je bio slučaj do tada. Trenutno sajt ima posetu od preko 1,200,000 pregleda stranica dnevno, a u vrhuncu posete u toku dana, dostiže i preko 5,000 konkurentnih korisnika. Zabeleženi rekord je do sada više od 9,000 korisnika.

Na slici 4.1.1. prikazan je trenutni izgled sajta:



Slika 4.1.1. - Prikaz početne strane sajta ispovesti.com

4.2. TEHNIČKA POSTAVKA I OKRUŽENJE - SETUP

Kao što je spomenuto ranije, ovaj portal rađen je u skladu sa predmetom Internet Programiranje, tako da je postavka koja je korišćena, u skladu s tim, uz neke inovacije i dodatke koji su se kasnije pokazali od velike pomoći za rešavanje mnogih zadataka, a neki su čak bili i neophodni za opstanak sajta.

Od hardvera je korišćena prosečno snažna mašina sa dvojezgarnim Intel Xeon procesorom sa taktom 2.3GHz, sa 4 thread-a, 4GB RAM operativne memorije, 500GB HDD, i po 4MB keš memorije po jezgru. Mašina radi na operativnom sistemu Linux.

Kao veb server, korišćen je dobro poznati httpd Apache, 2.2, PHP je korišćen za skriptovanje na serverskoj strani, a od baza podataka izabran je MySQL relaciona baza sa MYISAM tabelama. Na frontendu je, očekivano, korišćen tipičan HTML/CSS i JavaScript. Korišćene su neke od Javascript biblioteka, a ponajviše JQuery. Iskorišćena je pogodna priroda AJAX poziva kako bi se optimizovali određeni zadaci. Za izradu sajta nije korišćeno nijedno gotovo rešenje (*framework)*, već je sve samostalno urađeno (*custom)*.

4.3. POČETNO REŠENJE

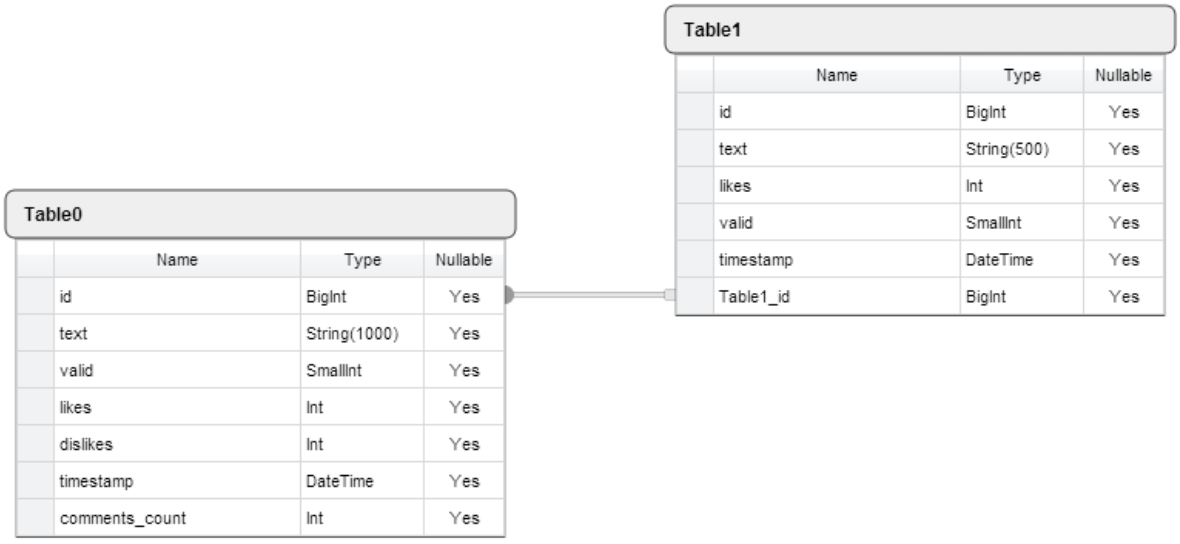
Na samom početku, ispovesti.com nije posedovao nikakav vid keširanja i za dovoljno mali broj korisnika nisu se pokazivali nikakvi znaci preopterećenja servera. Međutim, kako se sadržaj širio po socijalnim mrežama i raznim časopisima i dnevnim novinama, radio stanicama, pa čak i televizijskim kanalima, broj korisnika je naglo porastao i već na preko 100 jedinstvenih konkurentnih korisnika, dolazilo je do preopterećenja baze podataka prilikom velikog broja upita. Na ovaj način server je crpio previše operativne memorije i budući da je do tada korišćen običan vid deljenog hostovanja (*shared hosting*), dolazilo je do obaranja čitavog servera što je bilo nepodnošljivo za vlasnike drugih sajtova koji su hostovani na istoj mašini i sa kojima su deljeni svi resursi. Ovaj problem morao se rešiti u kratkom roku i tako je započet prvi vid optimizacije uvođenjem odgovarajućih indeksa u bazi i keširanjem pojedinih podataka.

4.4. STRUKTURA BAZE

Baza podataka se sastoji od 10 tabela, od kojih su većina pomoćnih, i samo nekoliko glavnih.

Naravno, osnovna tabela je tabela u kojoj se smeštaju ispovesti, nazvaćemo je Table0. U ovoj tabeli su smešteni meta podaci o određenoj ispovesti koju korisnici šalju putem forme, kao i njen sadržaj u vidu teksta. Ova tabela takođe sadrži broj pozitivnih i negativnih ocena određenog posta, odnosno priče, ispovesti, podatak o tome da li je validna, odnosno vidljiva na sajtu, ili je obrisana od strane administratora jer ne ispunjava odgovarajuće kriterijume po pitanju sadržaja. Naravno, tabela ima primarni ključ, odnosno ID kolonu u kojoj se smešta redni broj priče. Treba napomenuti da je svaki primarni ključ automatski i indeks i jedinstveni ključ. Jedinstveni ključ, najčešće označen kao ID, predstavlja osnovnu identifikaciju podatka u tabeli, odnosno jednog reda. On služi za jednoznačno identifikovanje željene informacije kojoj se želi pristupiti radi čitanja, izmene ili brisanja. Postoje tabele koje nemaju ID, a one obično služe kao vezivne tabele i tabele kod kojih tačno specifičan red nikada neće biti zatražen. Pojam indeksiranja kolona je obrazložen ranije u poglavlju 3.2.

Druga bitna tabela je tabela u kojoj se smeštaju komentari o određenoj priči, tj. ispovesti. Ova tabela, nazvaćemo je Table1, sadrži takođe svoj primarni ključ, tekst komentara, pseudonim autora, broj koji označava pozitivan ili negativan stav korisnika, vezivni strani ključ prema tabeli Table0 (glavnoj tabeli) i takođe validnu tabelu koja govori o tome da li je komentar objavljen na stranici ili je nevidljiv za korisnike. Naravno, obe tabele sadrže vreme upisa podataka. Postoji i posebna tabela koja čuva informacije o ispovestima koje su pristigle ali još uvek nisu obrađene, kao i mnoge tabele o korisnicima-administratorima koji odobravaju ispovesti, međutim one nisu predmet ovog rada jer nisu uticale na skalabilnost sistema budući da nikada nisu korišćene od velikog broja korisnika.



Slika 4.4.1. - Struktura baze podataka, osnovne dve tabele

Budući da su upiti u bazu rađeni na taj način da se povlače svi komentari određene ispovesti, neophodno je staviti index i na vezivnu kolonu u tabeli Table1 koja ukazuje na ID tabele Table1. Pored toga, budući da se pretraživanje vrši u zavisnosti od validnosti, odnosno nekada se povlače samo objavljene priče, potrebno je postaviti indexe na kolone po kojima se nešto filtrira, kao i one kolone po kojima se vrši sortiranje. Kao što je na primer kolona koja čuva podatak o vremenu, tj. timestamp.

Nakon uvođenja ovih indeksa, znatno je povećana brzina čitanja iz baze, jer sada tabele ne moraju da budu pročitane u kompletu već određeni indeksi, koji se ponašaju kao mape, direktno dolaze do zahtevanih podataka. Na primer, ako korisnik zatraži najbolje ispovesti danas (one priče koje imaju preko 1,000 pozitivnih ocena u toku dana), SQL server bi morao da pretraži ceo set postojećih podataka i za svaki upita da li je broj pozitivnih ocena (*likes)* veći od 1000. Koristeći indeks na koloni *likes*, može se prevazići ova poteškoća, tako što indeks omogućava postojanje tačno definisane mape po kojoj SQL server može odmah da locira podatke sa ovim osobinama. Na ovaj način je povećana skalabilnost budući da je svaki proces čitanja iz baze znatno ubrzan i na taj način konekcije i memorija su se znatno brže oslobađali.

4.4.1. *ORDER BY RAND* PROBLEM I REŠENJE

Postoje različiti načini sortiranja kada je u pitanju manipulacija sa MySQL bazama, koji mogu da dovedu do dodatnog povećanja vremena izvršenja upita, ali svi mogu donekle da se optimizuju uvođenjem odgovarajućih indeksa. Međutim, postoji i nasumično sortiranje, odnosno *order by rand(),* koje nije moguće optimizovati i koje je samo po sebi jako nestabilno. Uvođenje ovog načina sortiranja, izaziva velike poteškoće mysql serveru da se izbori sa selektovanjem podataka, pogotovo za tabele sa većim brojem upita. Ovo je jedna od retkih opcija, koju je najbolje izbegavati, a evo i obrazloženja zašto.

Jedan od testova ovog načina sortiranja, pokazao je da za tabelu od oko samo 5,000 upisa, ovaj način sortiranja oduzima preko 1400 milisekundi. Ovo se dešava iz razloga što MySQL prolazi kroz sve redove u tabeli i dodeljuje neki privremeni id, koji izvlači nasumično i ovaj proces oduzima velike procesorske resurse, pa samim tim nije pogodan za sisteme koji zahtevaju bilo kakvu veću skalabilnost. To se pokazalo i na primeru sajta ispovesti.com, koji je u početku koristio *order by* rand funckiju pri generisanju nasumičnih postova iz glavne tabele, koja je u tom trenutku brojala oko samo 20,000 upisa i ovaj način povlačenja podataka iz baze izazivao je pad servera u nekoliko navrata.

Rešenje ovog problema skalabilnosti leži u prebacivanju random logike na PHP ili bilo koji drugi jezik koji se koristi. Dakle, ukoliko je potrebno povući proizvoljan broj N podataka iz tabele, sortiranih po nasumičnom redosledu, potrebno je izvući određeni skup ID-jeva iz tabele, ili čak ceo skup ukoliko tabela nije previše velika, a potom kroz PHP skriptu putem funkcije *shuffle* izmestiti članove niza i odabrati prvih N, nakon toga upit u bazu izgledaće ovako:

main.php:

```
<?php
if ($case == 'random'){
	$rnd_file =. '/cache/random _array.txt';
	if ((time() - $interval) > filemtime( $rnd_file )){
		/* retrieve all ids from db */
		file_put_contents($rnd_file, implode(',' ,
		 qr_generate_ids_for_random()));
}
	$random_ids = explode( ',' , file_get_contents($rnd_file));
	shuffle($random_ids);
	$random_ids = array_slice($random_ids, 0, 10);
}
// query
if($case == 'random'){
	global $random_ids;
	$random_ids = implode(',' , $random_ids);
	$sql_str.=" AND `id` IN ( $random_ids )";
}
?>
```

S ovim na umu, upit će izgledati:

```
SELECT
      `field1`, `field2`,..
FROM
      `table`
WHERE
      condition1 AND condition2 AND `id` IN (3,51,212,561,5,11,...)
```

Iz prethodnog primera dela koda, vidimo da se najpre izoluje slučaj kada je u pitanju opcija random. Odnosno, ukoliko se korisnik odlučio za nasumično sortiranje, *if* uslov će biti ispunjen i pristupiće se obračunavanju za ovaj poseban slučaj. Najpre se određuje fajl u koji će se smeštati privremeni podaci o id-jevima iz glavne tabele. Ukoliko je proteklo određeno unapred definisano vreme, podaci se osvežavaju iz baze i smeštaju u ovaj fajl. Sledeći korak je dovlačenje podataka iz fajla uz pomoć funkcije *file_get_contents()* i metodom *explode()* od stringa pravimo niz.

Potom, budući da imamo sortirani niz, što u ovom slučaju ne želimo, izvršava se metoda *shuffle()* koja radi pseudoslučajnu raspodelu elemenata u nizu. Treba imati u vidu da je ovo suština rešenja, jer je ova metoda mnogo brža od sortiranja po random poretku koje nudi MySQL. Na ovaj način je random logika prebačena na PHP sloj. Nakon toga se izvršava instrukcija *slice()* koja uzima samo željeni broj elemenata, budući da se po jednoj stranici korisniku ispisuje samo 10 postova.

U drugom delu je prikazan način formiranja SQL upita uz pomoć ovog niza u PHP-u i rezultat koji ova skripta formira prilikom SQL upita, gde se jasno vidi da se povlače podaci koji su nesortirani, što je i bio početni cilj.

4.5. TEHNIKE OPTIMIZACIJE AGREGACIJOM KODA U POSEBNE FAJLOVE

4.5.1. SPRITES TEHNIKA

Još jedna veoma korisna i korišćena tehnika za optimizaciju jeste *sprites*. Sprites predstavlja način isporučivanja slika od servera ka klijentu. CSS sprites je tehnika uz pomoć koje možemo kombinovati više slika u jednu veću sliku i pomoću pravilnog pozicioniranja različitih delova slike, pomeranjem x i y koordinata putem css atributa, dobijamo željeni element. Primarni cilj korišćenja ovog principa jeste smanjenje broja HTTP zahteva ka serveru. Ideja se sastoji u tome da grupisanjem višestrukih slika u jednu jedinstvenu sliku možemo selektivno prikazivati delove koje želimo. Tehnika počinje grupisanjem više slika. Na ovaj način, dolazi do znatno bržeg učitavanja stranice kao i do manjeg opterećenja servera jer se smanjuje broj kružnih (*round trip)* putanja do servera i nazad, ka i ukupni overhed.

Takođe, Yahoo istraživanja su pokazala da spajanje više manjih slika u jednu veliku, dovodi do manje količine informacija i samim tim zbirni fajl, odnosno slika, zauzimaće manje kapaciteta od zbira pojedinačnih slika.

Uzmimo za primer sajt koji sadrži 10 iseckanih dugmića kao glavni navigacioni bar, a svako dugme ima grafički prikaz kada se pređe preko njega mišem. To je ukupno 20 fajlova kojji bi trebali da se prevuku sa servera samo za navigaciju. Imali bi 19 fajlova koji doprinose suvišnom i nepotrebnom zaglavlju. Ukoliko bismo imali 10,000 novih poseta na sajtu, 200,000 preuzimanja bi se potencijalno dogodilo. Potencijalno je zato što kod nekih sajtova da bi se prevukla slika koja se pojavljuje nakon što mišem pređemo preko dugmeta, potrebno je prvo i aktivirati, odnosno preći kursorom preko kako bi se od servera zahtevala nova slika. U svakom slučaju, minimalno 100,000 skidanja će biti zahtevano, što smanjuje broj dozvoljenih konekcija koje server može da podnese, i samim tim se stvara usko grlo.

Ukoliko se koristi CSS sprites tehnika, desiće se samo 10,000 prevlačenja i značajno manji protok biti potreban, omogućavajući sajtu da najefektivnije iskoristi propusni opseg dodeljen od hosting provajdera. Kada se poseti neki sajt, jedna HTTP konekcija se uspostavi između klijenta i servera. Kada je konekcija uspostavljena, pretraživač zahteva dokument koji ste tražili. Odnosno HTML stranicu. Nakon pristizanja fajla, on se čita od strane pretraživača kako bi se otkrilo šta je još potrebno da se skine kako bi se vizuelno stranica učitala u potpunosti. Primetite u nastavku kako određeni html tag ne dobija atribut background-image, već svaka individualna klasa.

```
#nav li a {background:none no-repeat left center}
#nav li a.item1 {background-image:url('../img/image1.gif')}
#nav li a:hover.item1 {background-image:url('../img/image1_over.gif')}
#nav li a.item2 {background-image:url('../img/image2.gif')}
#nav li a:hover.item2 {background-image:url('../img/image2_over.gif')}
```

Korišćenjem CSS sprites, možemo u velikoj meri da olakšamo ovaj primer. Umesto postojanja desetina različitih slika, pet osnovnih slika i isto toliko za slike prilikom *hover*-a, možemo kombinovati njih u jednu veliku sliku.

```
#nav li a {background-image:url('../img/image_nav.gif')}
#nav li a.item1 {background-position:0px 0px}
#nav li a:hover.item1 {background-position:0px -72px}
#nav li a.item2 {background-position:0px -143px;}
#nav li a:hover.item2 {background-position:0px -215px;}
```

Na ovom primeru može se primetiti da se u prvom slučaju koriste 4 različite slike, što znači da je potrebno 4 puta ostvariti konekciju sa serverom i zatražiti određeni element, dok je u drugom primeru učitana samo jedna slika, koja se kasnije korišćenjem *background-position* atributa pomera i na taj način prikazuju se različiti delovi slike za različite potrebe brojevi izraženi u pikselima predstavljaju rastojanje (*offset)* od zamišljenog koordinatnog početka koji se nalazi na gornjem levom vrhu slike.

Uz pomoć ovog primera, smanjuje se broj HTTP upita za 9, a takođe se dobija i na ukupnom broju bajta koji se prenose kroz mrežu.

Na slici 4.3. se jasno vidi prikaz sprites tehnike:

Slika 4.5.1.1. - Sve ikonice korišćene na sajtu su sažete u jednu sliku i na ovaj način se štedi pri prenosu podataka i broju HTTP zahteva

Na prethodnom primeru možemo uočiti nekoliko detalja. Najpre, vidimo da postoji preko 30 različitih ikonica koje su iskorišćene na sajtu, i time je ušteđen veliki protok koji bi izazvale odvojene slike u posebnim fajlovima. Takođe možemo da primetimo da su slike različitih veličina i oblika, ali treba imati na umu da HTML element na kojem se postavlja određena slika, mora imati odgovarajuću veličinu tako da se slika koja je namenjena za taj element pokazuje izolovana, u suprotnom, kada bi element bio preveliki, došlo bi do zahvatanja drugih susednih ikonica i ova tehnika ne bi imala poentu.

Za primer sa slike 4.5.1.1, ukoliko bismo hteli da prikažemo glavni logo u donjem levom uglu, potrebno je da html element koji sadrži ikonicu ima sledeći stil definisan u stilovima (*CSS*).

```
background: url(/images/isp-sprite-4.png) no-repeat -17px -243px
```

U ovom slučaju, *url* govori o lokaciji slike, *no-repeat* atribut znači da se slika neće ponavljati beskonačno puta po x i y osi ukoliko bude više prostora u html elementu od veličine slike, dok su poslednja dva parametra zapravo rastojanje od gornje leve ivice slike. -17px znači da je potrebno pomeriti sliku u levo za 17 piksela i na gore za 243 piksela kako bi bila prikazana tačno glavna ikonica sajta.

4.5.2. IZDVAJANJE .JS I .CSS FAJLOVA RADI KEŠIRANJA

Još jedan način da se smanji količina prenesenih podataka preko mreže i značajno poboljšaju performanse sajta jeste izolovanje javascript i css fajlova u posebne statične komponente. Na ovaj način doprinosi se modularnosti sajta i izdvajanju logičkih celina, mogućnosti ponovnog korišćenja, odnosno izbegavanje replikacije koda, kao i smanjenju broja upita i količine podataka koji se prenose preko Interneta. Naime, kada bi se sav javascript i css kod nalazio na strani koja se učitava, to bi u mnogome povećalo težinu stranice, pogotovo ako sajt koristi veliki broj animacija i stilova. Ukoliko bi se sav javascript kod izdvojio u posebne .js

fajlove, dobili bismo smanjenje količine podataka na stranici koju učitavamo, tako da svaki sledeći put učitavamo samo osnovne informacije i neophodan html, dok bi se .js fajl učitao samo prvi put, a narednim učitavanjima stranice, pretraživač bi čuvao ove statične fajlove u kešu i na taj način bi bili spremni za upotrebu. Takođe, u svakoj sledećoj stranici koju korisnik zatraži, koristiće se verovatno slične animacije ili css stilovi koji će biti odmah dostupni bez ponovnog traženja od servera. Fajlovi sa stilovima mogu dostići i do nekoliko hiljada linija, što važi i za javascript animacije, tako da se ovom tehnikom u mnogome dobija, a takođe ukoliko želimo da promenimo nešto na svim stranicam, prostim menjanjem na jednom mestu, dobijamo osvežene sve stranice.

Može se postaviti pitanje kako se rešava problem koji se može javiti u slučaju kada dođe do promene u css ili javascript fajlovima, a klijentov pretraživač još uvek učitava stare fajlove, odnosno kešira u svojoj memoriji. Na ovaj način može doći do deformacija na stranici ukoliko su menjani neki bitniji stilovi ili animacije, tako što će html stranica sadržati neke elemente koje neće biti sinhronizovane sa trenutnim stilovima i js animacijama.

Da bi se sprečila pojava navedenog problema, postoje razni načini, a jedan od najefikasnijih je prikazan sledećim primerom.

Izgled php skripte na serveru:

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="/css/styles.min.css?<?php echo filemtime('css/styles.min.css'); ?>" />
</head>
<body>
...
</body>
</html>
```

Izgled html koda koji stiže do klijenta:

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="/css/styles.min.css?1395837833" />
</head>
<body>
...
</body>
</html>
```

Primećujemo označeni deo u html kodu, koji zapravo predstavlja celobrojnu reprezentaciju trenutnog vremena, odnosno vremena kada je fajl poslednji put izmenjen. Ovo znači da će svaki put kada se promeni fajl, ovaj celi broj biti drugačiji i samim tim url link neće biti isti, a samim tim pretraživač neće moći da nađe keširanu verziju ovog fajla jer nema isti potpis i na taj način se forsira pretraživač da ponovo učita određeni fajl.

4.6. AJAX I JSON

AJAX (*Asynchronous Javascript and Xml*) je grupa povezanih veb tehnika, korišćenih na klijentskoj strani, kako bi se stvorila asinhrona veb stranica. Na ovaj način, stranice mogu da kontaktiraju server, šalju ili prihvataju podatke bez ikakvog menjanja sadržaja na displeju korisnika, odnosno, asinhrono. Klasičan način slanja zahteva jeste osvežavanjem stranice uz slanje određenih informacija putem forme, nakon čega se stranica regeneriše uz odgovarajuću poruku korisniku u zavisnosti od aktivnosti koja je odrađena. Primer ovog tipičnog korišćenja http protokola je zapravo svaki pristup bilo kojoj stranici po prvi put, pa čak i kada izlazimo na početnu stranicu Google pretraživača. Putem ajaxa, pored sofisticiranijeg načina prikazivanja i boljeg korisničkog iskustva, u velikoj meri se doprinosi smanjenju opterećenja servera.

Naime, Ajax omogućava sajtu da ne osvežava celu stranicu bespotrebno ispočetka, ukoliko samo određene informacije treba promeniti, već šalje zahtev putem javascript-a, ne menjajući sadržaj na stranici i čekajući odgovor, korisnik ne primećuje nikakvu vrstu ometanja pri korišćenju sajta. Kada klijent prihvati odgovor od ajax poziva, na nju reaguje u zavisnosti od toga kakav je odgovor dobio. Na primer, prilikom poziva ajaxa za postavljanje komentara na neki blog post, server može odgovoriti na razne načine - uspešno postavljen komentar, server nije odgovorio uz error 500, stranica nije pronađena - error 404, niste autorizovani da ostavljate komentare, itd..

Uz Ajax pozive, moguće je vratiti skoro bilo kakav vid podataka. Tako da, na primer, ako želimo da na određeni poziv (klik na neku priču), server vrati sve komentare, prilikom klika će se pozvati server, a potom će baza podataka ili keširani izvor informacija, izvući potrebne podatke i uz unapred definisan način razvojnog tima, vratiti tražene informacije klijentu.

Prilikom razvoja ispovesti.com, na početku, korišćena je tehnika infinite scroll paginacije, odnosno, nije korišćen klasičan vid raspoređivanja sadržaja na stranicama.

Infinite scroll je jedna vrsta elegantnog rešenja proširivanja sadržaja, tako što se putem javascript-a koji osluškuje događaj skrolovanja, pozove određena akcija kada *scrollbar* dostigne dno ili bude dovoljno blizu. Naime, poziva se ajaxom određeni servis koji u zavisnosti od određenih parametara, vraća novi sadržaj i prikazuje ga korisniku bez obnavljanja odnosno osvežavanja stranice, nadovezivanjem na postojeći set podataka. Najpre je to izvođeno klasičnim html odgovorom, odnosno, informacije koje je server vraćao prilikom upotrebe ajaxa, su bile dodatnih 10 ispovesti koje su bile raspoređene u html kodu i to zajedno uz sve komentare kojih može biti na desetine po jednoj ispovesti. Loša strana ovog pristupa, lako se ogleda u tome da nije optimizovana za visoko posećene stranice.

Ukoliko je broj komentara veliki, doći će do situacije da se prenosi enormna količina podataka kroz mrežu, a takođe se uz sve to prenosi ogromna količina html koda koji formira elemente i njihov raspored. Kako je ovaj problem postajao sve primetniji, jer je rastao broj korisnika, a samim tim i količina prenošenih podataka, dolazilo je do većeg opterećenja na

serveru. Da bi se ovaj problem rešio, bilo je potrebno koristiti drugi način prenosa podataka prilikom ajax poziva.

Korišćenjem JSON formata za strukturu i prenos podataka, znatno je olakšan ovaj aspekt sajta. Naime, pri svakom ajax pozivu, server obradi zahtev, izvuče neophodne podatke i prosledi samo suve informacije ka klijentu, a javascript je odgovoran da stvori i popuni html elemente i prikaže ih korisniku. U proseku, odnos html kod prema sadržaju iznosi oko 45-50%, što znači da se ovim putem može smanjiti količina prenesenih informacija za skoro 50% korišćenjem prenosa čistih informacija umesto celokupnog html koda. Još jedan pozitivan aspekat korišćenjem JSON ili XML-a jeste višestruko korišćenje istog izvora informacija, naime, ovi podaci jednom izvučeni, mogu se ponovo koristiti za druge svrhe, kao što su android aplikacija, windows phone aplikacija, mobilna verzija sajta, kao i uskoro ios aplikacija. Ovo zapravo znači da se stvara jedan pravi veb servis koji ima odgovarajući interfejs.

Prikaz pseudo koda koji koristi ovaj princip:

JavaScript:

```
function loadComments(isp_id) {

  $.post('/php/load_comments.php', {
            isp_id:isp_id
  }, function(data) {

      if(data != 'error') {
            // PORUKA ZA GREŠKU
      } else{
            var json = $.parseJSON(data);
            // LOGIKA ZA PARSIRANJE JSON-a I PRIKAZ KORISNIKU
      }
});
```

PHP:

```
<?php
if(isset($_POST['isp_id'])) {
      $isp_id = $_POST['isp_id'];
      if(is_numeric($isp_id)) {
      require_once '/connection.php';
      require_once. '/php/function/main.ispovesti.php';
      $response = handle_and_return_comments($isp_id);
      die($response);
      }
```

```
}
die('error');
?>

<?php

function handle_and_return_comments($isp_id) {
	$root_path = $_SERVER['DOCUMENT_ROOT'];
	$apc_key = 'all-comments-for-' . $isp_id;
	$activity_entry = "apc:caching-candidates";
	$cache_file = $root_path . '/cache/comments/main/all_comm_' . $isp_id .
'.html';
	$interval = 120;
	if(apc_exists($apc_key)) {
		$response = apc_fetch($apc_key);
		return $response;
	} else {
		if(file_exists($cache_file) && (filemtime($cache_file) > time() -
$interval)){
			$cached_comments = file_get_contents($cache_file);
			return $cached_comments;
		}
		require_once $root_path . '/php/database/query.ispovesti.php';
		$results = qr_comments($isp_id); //DATABASE ACCESS
		$results = json_encode(prepare_comments($results));
		$cached = false;
		if(apc_exists($activity_entry)) {
			$activity_array = apc_fetch($activity_entry);
			if(in_array($isp_id, $activity_array)) {
				apc_store($apc_key, $results, 120);
				$cached = true;
			}
		}
		if(!$cached) file_put_contents($cache_file, $results);
		return $results;
	}
}
```

```
?>
```

Prvi deo koda predstavlja javascript ajax poziv. Može se primetiti da se ne koristi bazični javascript pristup, već određena biblioteka koja poseduje uprošćene javascript funkcije. Ova biblioteka je dostupna svima i naziva se JQuery. Metodom *$.post* se poziva određena skripta na serverskoj strani, uz odgovarajuće parametre. U ovom slučaju taj parametar je zapravo *id* ispovesti za koju se potražuju komentari i koristiće se prilikom čitanja i selektovanja komentara za tu ispovest.

Postoje dve unutrašnje grane koje će se izvršiti u zavisnosti od povratne vrednosti servera, odnosno odziva. U slučaju greške, može se kreirati neka vrsta poruke upozorenja, dok u slučaju uspešnog prenosa podataka, kreiraće se odgovarajući html elementi i prikazati korisniku.

Drugi deo koda predstavlja PHP implementaciju i najpre je prikazan skripta pristupa, koja u okviru svog koda ima pozivanje eksternih funkcija kao što je *handle_and_return_comments()*. Ova funkcija ima određenu kompleksnost zbog sistema keširanja koji se koristi, a o kome će biti reči nešto kasnije u ovom radu. Treba izdvojiti liniju

```
$results = json_encode(prepare_comments($results));
```

koja govori o načinu slanja podataka, odnosno konverziji objekata u JSON objekte i slanje istih ka klijentu u ovom obliku. [5]

4.7. SQL UPITI

Optimizacija upita ka bazi jedna je od veoma bitnih optimizacionih tehnika zarad povećanja skalabilnosti sistema, u ovom slučaju veb aplikacije. Budući da relacione baze podataka i manipulacija sa njima zauzima veliku procesorsku moć, potrebno je svu komunikaciju sa bazom olakšati što je to više moguće. Neki od principa za olakšanje komunikacije sa MySQL bazom prilikom optimizacije ispovesti.com, navedeni su u nastavku.

Indeksi predstavljaju jednu od osnovnih tehnika koje se gotovo uvek koriste prilikom dizajniranja baza podataka. Njihova osnovna uloga jeste ubrzavanje SQL upita. SQL upit, koji se izvršava nad bazom, koji selektuje određeni set podataka, u normalnim uslovima ide sekvencijalno i ispituje svaki red u tabeli da li ispunjava željeni uslov. Indeksi rade drugačije, indeks predstavlja mapu koja služi da prilikom upita, SQL server mnogo lakše i brže dođe do rezultata i na taj način mnogo brže oslobađa procesorske kapacitete koje je zauzeo. Indeksi se koriste uglavnom za kolone koje se ispituju u WHERE uslovima ili prilikom sortiranja podatka. Takođe, veoma bitna uloga indeksa jeste prilikom spajanja tabela. Potrebno je da kolone koje se koriste za *join* operaciju budu indeksirane.

Međutim, iako je indeks odličan za ubrzavanje upita prilikom povlačenja podataka, treba imati u vidu da on može znatno da uspori upit koji upisuje nešto u tabelu, jer nije dovoljno samo dodati zapis na kraj već je neophodno osvežiti sve indekse. S ovim na umu, jasno je da se od indeksa najviše profitira kada određena aplikacija koristi veliki broj selektovanja podataka i manji broj upisa.

Treba izbegavati korišćenje upita formata SELECT *. Kada se radi selektovanje podataka iz neke tabele, loša je praksa da se koristi izvlačenje svih kolona korišćenjem džokera(*wildcard*) tj. *. Ovaj način selektovanja zahteva dodatno povećanje zaglavlja i stvara dodatno opterećenje nad SQL serverom i samim tim nad procesorom.

4.7.1. REDUDANTNE KOLONE

Prilikom optimizacije ispovesti.com iskorišćena je čak i ne tako popularna tehnika redudantnih informacija. Naime, kako je poznato da postoje tabele ispovesti (Table0) i Table1, tj. tabela komentara, jasno je da je lako moguće izvući broj komentara običnim SELECT upitom sa informacijom o jedinstvenom ključu iz tabele ispovesti. Međutim, kako bi se izbeglo spajanje tabela ili višestruki upiti, u glavnoj tabeli je dodata kolona komentari_count, koja sadrži informaciju koja nije nezavisna, odnosno postoji implicitno u tabeli komentari. Međutim, pošto je ovo podatak koji se često izvlači iz baze a jedini je koji ne postoji u glavnoj tabeli, postojao je određeni benefit kreiranjem dodatne kolone koja bez sabiranja i dodatnih upita, sadrži broj komentara koji postoje za određenu ispovest.

MySQL takođe ima i određene ključne reči koje omogućavaju dodatnu optimizaciju upita pri pretpostavci da je broj upita nad bazom veći. Ovo se odnosi na upite za upis i modifikaciju podataka, odnosno INSERT i UPDATE. Ukoliko nije neophodno da upis u bazu bude izvršen odmah kada je korisnik poslao zahtev, vrlo je zgodno koristiti opciju *DELAYED* za insert i *LOW PRIORITY* za UPDATE instrukcije. INSERT DELAYED će učiniti da sql server sačeka sa izvršavanjem drugih dugih i zahtevnih SELECT upita i tek kada se oslobodi pristup bazi, odradiće upis, a isto važi i za LOW_PRIORITY UPDATE. Ovaj princip se naročito koristi za logovanje određenih informacija za praćenje i monitoring sistema, ali je jako zahvalna i za druge slučajeve gde brzina upisa nije od presudnog značaja.

4.8. KEŠIRANJE KORIŠĆENJEM FAJLOVA

Jedan od najvećih koraka u unapređenju i optimizaciji nastao je prilikom uvođenja keširanja podataka koji se nalaze u sql bazi. Naime, iako jako pouzdana i perzistentna, baza bi trebalo da služi za čuvanje više negoza konstantno pregledanje informacija. S tim na umu, kada je posećenost sajta ispovesti.com počela značajno da raste, usko grlo u sistemu je predstavljao broj direktnih upita nad bazom i ograničeni broj konekcija, kao i procesorskog kapaciteta koji su ti upiti zahtevali.

Da bi se ovaj problem rešio, bilo je neophodno uvesti neki način privremenog čuvanja podataka iz baze i prikazivanja ovih podataka korisnicima sa što bržim i lakšim odzivom. Odatle je potekla ideja za keširanjem podataka. Zapravo, kada korisnik zatraži informacije od servera, najpre će se dohvatiti neophodni podaci iz baze podataka, zatim će se podaci obraditi i formirati odgovarajuće html strukture koje će se naknadno poslati korisniku na klijentskoj strani. Međutim, u istom trenutku dok korisnik prima stranicu koju je potražio, stranica se čuva u odgovarajućem fajlu koji jedinstveno može da se odredi tom stranicom, na primer, URL. Nakon što je korisnik učitao stranicu, postoji određena verovatnoća da će u kratkom periodu, još korisnika probati da zahtevaju istu stranicu. Korišćenjem ove informacije, intuitivno je jasno da je ponovno izvlačenje podataka direktno iz baze jako skupo sa memorijskog aspekta, već je najbolje samo učitati stranicu koja je sačuvana na određenoj lokaciji na disku, kao običan html kod i to prikazati korisniku. Na ovaj način znatno je povećana skalabilnost sistema i nakon

implementacije ove tehnike, maksimalni broj korisnika koje je sajt uspeo da podrži porastao je za preko 200%. Baza podataka više nije predstavljala usko grlo jer je pozivana samo jednom, a svi sukcesivni pozivi su dobijali odgovor iz keširanog fajla, spremnog za slanje.

```
<?php
if (file_exists($filename) && (time()- $interval) < filemtime($filename)) {
      readfile( $filename );
      exit();
} else{
      ob_start();
      // POVLAČENJE INFORMACIJA IZ BAZE PODATAKA - SKUPA OPERACIJA
}
// GENERISANJE STRANICE

$buff = ob_get_contents();
$file = fopen( $filename, 'w' );
fwrite( $file, $buff );
fclose( $file );
ob_end_flush();

?>
```

Prethodni kod predstavlja prvi metod keširanja korišćen na sajtu i veoma je prost. Zapravo, *if* uslov ispituje da li je vreme poslednje modifikacije fajla, ako fajl uopšte postoji, bilo pre manje od *$interval* sekundi. Ukoliko je to slučaj, funkcija *readfile* će pročitati sadržaj iz fajla sa putanjom *$filename* i napustiti dalje izvršavanje skripte.

U slučaju da fajl ne sadrži sveže podatke, odnosno da je promenjen ili kreiran pre više od *$interval* sekundi, stranica se ponovo generiše i ceo sadržaj se smešta u bafer, koji se kasnije na dnu skripte, nakon što je cela stranica izgenerisana, prazni u određeni fajl i na taj način naredni pokušaji pristupa ovoj stranici biće iščitani iz keširanog fajla.

Nakon toga otišlo se korak dalje jer je sajt po prirodi dinamičan i postoji ogroman broj stranica na koje se može otići da bi se dohvatio drugi sadržaj. S tim na umu, ukoliko je broj korisnika dovoljno veliki, i ukoliko pristupaju sa različitih uređaja, povlačiće se različite stranice i onda je iskorišćenost keširanja znatno manja, ukoliko svaki deseti korisnik poseti različitu stranicu, za broj korisnika koji je preko 1000, značiće da će se u kratkom periodu napraviti preko 100 sql upita. Za velike baze sa tabelama od preko milion upisa, može biti potencijalni problem.

Kako bi se mobilna verzija i desktop verzija, kao i sve razne stranice koje sadrže određene ispovesti, optimizovale i minimizirale broj upita u bazu, došlo se do ideje da se u pomoćnim keš fajlovima ne čuvaju čitave html strukture veb stranice, već samo korisni podaci. Dakle, prilikom povlačenja podataka iz baze, stvaraju se odgovarajući modeli, odnosno objekti,

koji se potom pakuju u JSON format, a zatim takvi snimaju u fajlove. Ovo dovodi do toga da je onda taj objekat moguće iskoristiti i na mobilnoj verziji i na aplikaciji i na desktop varijanti. Dakle, ako želimo bilo kojim putem da dobijemo 10 najnovijih ispovesti, lako možemo preko ovog servisa uz nekoliko parametara, od servera dobiti JSON format niza od 10 ispovesti i svih njihovih informacija. U zavisnosti od toga kojim putem smo ovaj servis zvali (aplikacija, pretraživač, itd.) na odgovarajući način će se ovi podaci parsirati i od njih stvarati elementi za prikaz korisniku. Ukoliko je slučaj sa desktop računarom, odnosno pristup sajtu putem pretraživača, javascript će biti zadužen za prevođenje JSON niza 10 najnovijih ispovesti u odgovarajući niz objekata. Takođe je moguće koristiti te objekte i na drugim stranicama na sajtu gde se spominju te priče. Ovim je drastično porasla iskorišćenost keširanih informacija i u velikoj meri je dobijeno na skalabilnosti.

4.9. APC

Iako je keširanje dovelo do velikog poboljšanja sa aspekta povećanja skalabilnosti i broja korisnika koje server može da opsluži, ova tehnika je ipak ograničena i za ogroman broj upita u kratkom intervalu, može doći do znatne degradacije u kvalitetu servisa i brzini odziva. Naime, kada se radi veliki broj upisa i čitanja iz fajlova, u kojima se keširaju podaci, dolazi do sporijeg obrađivanja informacija, budući da je hard disk najsporiji medijum na kojem mogu da se čuvaju podaci. Istraživanja su pokazala da je RAM (*Random Access Memory*) memorija čak do 100,000 puta brža od hard disk-a. Ovo svakako otvara vrata za velikim unapređenjem i poboljšanjem u smislu brzine pristupa i serviranja keširanih podataka, a samim tim i velikim povećanjem skalabilnosti čitavog sistema.

APC, odnosno *Alternative PHP Cache,* predstavlja frejmvork za keširanje *bytecode*-a, odnosno kompajliranog koda u deljenu memoriju, odnosno RAM. Na ovaj način se u velikoj meri smanjuje parsiranje i pisanje i čiatanje sa diska. APC takođe pruža podršku za keširanje korisničkih podataka. Za aplikacije koje sadrže velike količine koda, kao što su one generisane od strane nekih gotovih alata (*framework*), kao što su Drupal ili Joomla, ovo je odlična prilika da se smanji vreme potrebno za kompajliranje i parsiranje PHP skripti. APC je korišćen čak i od strane Facebooka, najveće društvene mreže koja je čak radila na unapređenju ovog sistema, a i PHP-a uopšte. APC je postao toliko moćan i korišćen alat da će se u budućim verzijama uključivati u osnovno PHP okruženje. Postavlja se pitanje, šta ako u međuvremenu dođe do izmene fajla, a u keš memoriji bude sačuvan kompajliran stari fajl, i na taj način može doći do neželjenog rada aplikacije. APC se o ovome brine tako što proverava vreme poslednje izmene fajla, i ukoliko je php skripta menjana u međuvremenu, odnosno u kešu se nalazi kompajliran kod koji nije ažuran, APC će to prepoznati i ponovo učitati fajl u memoriju.

Kao što smo videli, keširanje php skripti, odnosno operativnog, kompajliranog php koda u memoriji, može i do 3 puta da ubrza proces serviranja sadržaja jer server ne mora uvek da čita, parsira i kompajlira fajl pre nego što ga izvrši. Međutim, ovo nije jedina moć koju poseduje APC, a u nastavku će biti objašnjeno i zašto.

Keširanje podataka u fajlovima, dovelo je do velikog unapređenja i težina procesa sa baze je prebačena na hard disk čime je mnogo olakšano serviranje podataka krajnjim korisnicima. Ukoliko u jednom trenutku 500 korisnika pokuša da dohvati isti sadržaj, samo prvi će zapravo podatke dovlačiti direktno iz baze, a narednih 499 će dobiti keširane podatke iz fajla. Naravno

sve ovo važi pod uslovom da su upiti izvršeni u vremenskom intervalu manjem od predefinisanog vremena keširanja.

Kao što je već pomenuto, ovaj pristup ima tu manu da podatke čuva u fajlovima, odnosno na hard disku, koji je poznat kao veoma spor medijum. Stoga, APC uvodi keširanje korisničkih podataka direktno u RAM memoriji i na ovaj način u velikoj meri ubrzava ovaj proces, a samim tim poboljšava performanse čitavog sistema. Uz pomoć određenih direktiva, kao što su apc_store i apc_fetch, moguće je lako čuvati određene podatke, u vidu ključ-vrednost (*key-value)* principa. Prilikom upisivanja podataka u apc keš, potrebno je navesti i vreme koje će određeni podaci provesti u kešu pre nego što se obrišu i oslobodi memorija. Primer upisa i čitanja iz memorije je dat na sledećem prikazu:

```
<?php
$cache_interval = 120; //seconds
if ($some_data = apc_fetch('some_key')) { // serve fresh data from cache
 echo $some_data;
 echo "Cached! " . $data;
} else { // get data, serve, and add to cache for $cache_interval
 $some_data = "This is some data.";
 echo "Not cached! " . $some_data;
 apc_add('some_key', $some_data, $cache_interval);
}
?>
```

Na prethodnom primeru vidimo da se najpre vrši inicijalizacija vrednosti intervala keširanja. Nakon toka, radi se ispitivanje da li određeni podatak sa ključem *some_key* postoji u APC memoriji, ukoliko postoji, *apc_fetch* će vratiti ovu vrednost, i moći će odmah da se koristi u narednoj instrukciji što je i prikazano na primeru. Ukoliko vrednost za ovaj ključ ne postoji u memoriji, funkcija *apc_fetch* vratiće logičku vrednost *false* i samim tim ući će se u *else* blok. U *else* bloku se najpre inicijalizuje vrenost promenljive *$some_data*, a potom se ispisuje, nakon čega se ova vrednost smešta u apc memoriju funkcijom *apc_add*(), radi naredne upotrebe.

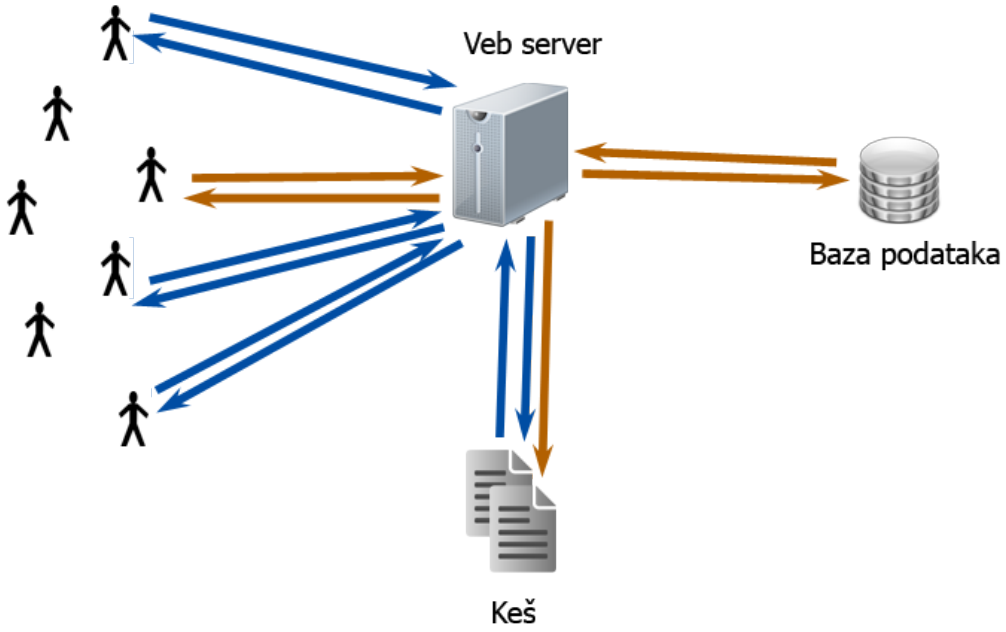
Ovo je primer običnog upisa i čitanja iz memorije putem APC-a i znatno je brži od keširanja u fajl. Ova primena keširanja podataka na portalu ispovesti.com je imala znatne rezultate budući da je brzina sajta porasla značajno čak i za velike količine posetilaca, a takođe je maksiman broj konkurentnih korisnika porastao za oko 120%. Naravno, APC keširanje korisničkih podataka ne mora da se zaustavi samo na običnim promenljivama, već se u ključ-vrednost mogu smeštati i nizovi kao i druge vrste objekata. Konkretno, najviše je korišćen običan JSON format objekata koji sadrže sve potrebne informacije za generisanje određenih stranica. Primer korišćenja APC korisničkog keširanja na sajtu ispovesti.com:

```
<?php
. . .
$apc_prefix = "index:";
$interval = 90;
$cache_entry = $apc_prefix . $sort . '-' . $page;
if(apc_exists($cache_entry)) {
      $content = apc_fetch($cache_entry);
die($content);
}
ob_start();

...
// page
...
//end of page
...

$buff = ob_get_contents();
apc_add($cache_entry, $buff, $interval);
ob_end_flush();
?>
```

Kod koji je dat kao primer radi istu operaciju kao i onaj opisan kada je u pitanju keširanje sa fajlovima. Najpre se inicijalizuje vrednost intervala keširanja na 90 sekundi, kao i prefiks radi prepoznavanja entiteta koji se kešira. Nakon toga se kreira jedinstveni $*cache_entry* koji će se kasnije koristiti za povlačenje informacija. U narednom koraku se ispituje da li postoji određena vrednost u APC kešu, za odgovarajući ključ, a u slučaju da ova funkcija *apc_exists()* vrati istinu, iz ove pozicije u memoriji se čita sadržaj i prikazuje se korisniku uz prekidanje izvršavanja skripte, metodom *die($content);* Ukoliko ovo nije slučaj, i interval je istekao, ili ova stranica nikada nije bila keširana, prelazi se u *else* blok u kojem se generiše stranica ispočetka i smešta u bafer, a na kraju PHP skripte, se bafer prazni u promenljivu *$buff* koja se nakon toga upisuje u APC keš funkcijom *apc_add()*. Na kraju se bafer oslobađa funkcijom *ob_flush()*.

4.10. KEŠ STAMPEDO PROBLEM

Keš stampedo, odnosno *dog-piling* je problem koji se javlja u skoro svim sistemima koji koriste neke vrste keširanja da bi olakšali pristup bazi podataka, odnosno smanjili broj teških upita ka bazi. Slika 4.4. opisuje situaciju i problem stampeda.



Slika 4.10.1. - Prikaz situacije kada nastupa keš stampedo

Kako bismo mogli da shvatimo kako se ova situacija dešava, zamislimo veb server koji koristi APC keš neko vreme, a kada istekne zadati period, sledeći zahtev će raditi upit nad bazom, prazniti keš i po izvršavanju čitanja iz baze, ponovo napuniti keš. Kada postoji veliko opterećenje određene stranice, sistem će biti u mogućnosti da odgovori dokle god se ovaj skup podataka nalazi u kešu, tako što će se zahtevi obrađivati serviranjem ovih podataka. Ovaj način minimizira skupe operacije kao što su očigledno one sa bazom.

Ukoliko je opterećenje relativno nisko, kada se desi keš promašaj (zahtev ne naiđe na ažurne podatke u kešu), sistem će raditi bez problema kao i pre, jer ne postoji veliki broj zahteva za istom operacijom. Međutim, kada je opterećenje veliko, odnosno broj upita za istom stranicom znatno veći, u kratkom vremenskom periodu, može doći do dovoljno konkurentnih zahteva, tako da će veći broj thread-ova na serveru probati da formiraju istu stranicu simultano. Ako je ovaj pritisak preveliki, ovo može dovesti do zagušenja u tolikoj meri da se sistem uruši, time što će se iscrpeti svi resursi. Ovo će značiti da stranica nikad neće biti u potpunosti učitana i samim tim nikada neće biti upisana u keš memoriju i svi naredni pokušaji će praviti novi thread koji radi isto. Kod u nastavku bolje opisuje ovaj problem.

```
<?php
```

```
// Povlačenje podataka iz baze
$data = $cache->get ("cached_item");

// Provera da li su podaci pravilno dovučeni
if ($cache->getResult () == Cache::NotFound) {
 // Nije pronađeno. Generiši podatke (skupa operacija)
 $data = generateData();

 // Sačuvaj podatke u keš. Isticanje za 30
 $cache->set ("cached_item", $data, 30);
}
// Ispiši podatke
print $data;

?>
```

Na ovom primeru vidimo kako dolazi do problema keš stampeda. Naime, u trenutku kada keš ne postoji u ažurnom stanju, potrebno je da izvršimo kompleksne i skupe operacije sa stanovišta resursa. Na ovaj način, funkcija *generateData();* može da potraje dugo, i u tom vremenu će se desiti još mnogo zahteva za istom stranicom, a pošto još uvek nije završen ovaj proces niti je bilo šta smešteno u keš za ovaj ključ, još veliki broj istih upita će se inicirati i tako da će se upiti međusobno ukočiti, te se nikada neće zaustaviti i izgenerisati stranica.

Postoje više metoda rešavanja ovog problema, koji su opisani narednim odeljcima.

4.10.1. Odvajanje procesa ažuriranja

Umesto da postoji update na veb serveru koje se izvršava na zahtev, može se podesiti poseban proces koji osvežava i ažurira keš. Na primer, kada se radi o statističkim podacima koji trebaju biti obnovljeni svakih 5 minuta, ali im treba oko minut kalkulacije, poseban proces, na primer cronjob, može raditi računanje ovih podataka i osvežavanje keša. Na ovaj način, podaci su uvek dostupni, a i ako nisu, nema potrebe brinuti o stampedu jer klijenti nikada ne generišu podatke, već je to odrađeno preko posebnih procesa. Ovo radi odlično za globalne podatke koji nisu neophodni da se računaju u letu (*on the fly*). Međutim, ova metoda nije mnogo zahvalna za situacije kao što su liste prijatelja, komentari, korisnički objekti itd.

4.10.2. Zaključavanje

Umesto da poseban proces pravi ažuriranje, može se koristiti normalni tok keširanja: 1. proveriti keš, 2. ako ne postoji, generisati. Međutim, potrebno je dodati posebna merenja kako bi se zaobišla mogućnost javljanja stampeda. Ovo je primer koda:

```
<?php
function get($key) {
```

```
  $result = $this->_mc->get($key);
  if ($this->_mc->getResultCode() == Memcached::RES_SUCCESS) return $result;
  // Pokušaj za dobijanje ekskluzivnog ključa
  $this->_mc->add ("stampede_lock_".$key, "locked", 30);
  if ($this->_mc->getResultCode() == Memcached::RES_SUCCESS) {
    $this->_setResultCode (Cache::GenerateData);
    return false;
  }
  $this->_setResultCode (Cache::NotFound);
  return false;
}

?>
```

Ovaj kod radi sledeće:

Proverava da li je keš prisutan, ako jeste, vratiće tražene podatak. Ako nije, probaće da napravi ekskluzivni ključ. Ekskluzivni ključ predstavlja dozvolu za pristup određenom fajlu ili memoriji radi pisanja, ali samo jednom procesu, kako ne bi došlo do konflikta. Ovo će uspeti ako i samo ako neko to nije već uradio pre nas, jer je "add" proces koji će uspeti samo jednom, iako veliki broj procesa pokuša isto. Kada dobijemo pristup ključu, vratićemo generisane podatke, ako ne dobijemo pristup ključu, vraćamo NOT_FOUND odgovor. Primećujemo da samo jedan proces vraća generisane podatke, a svi ostali NOT_FOUND. Zadatak na klijentu je da ovu povratnu vrednost pravilno parsira proveravajući povratnu vrednost i prikaže korisniku odgovor. Ukoliko je ključ uspešno dobijen pristupiće se generisanju podataka. Ova procedura je kompleksna i skupa i zbog toga je potrebno postaviti TTL. Takođe je bitno da ovo zaključavanje dobije i određeno vreme života. Takođe treba imati na umu da TTL bude dovoljno veliko da bi se podaci zapravo generisali, ukoliko je potrebno oko 20 sekundi, potrebno je postaviti vreme na barem 30 sekundi. Nakon isteka ovog vremena, ključ će automatski biti oslobođen.

4.10.3. AŽURIRANJE NA PRVOM PRISTUPU - LAŽNO AŽURIRANJE

Ovaj pristup je pogodan u slučaju da vreme koje se čeka za osvežavanje podataka nije preterano dugo budući da ovaj princip odlaže ažuriranje podataka za onoliko vremena koliko je potrebno da se izvrše kompleksne operacije sa bazom.. U svakom slučaju ovo je veoma elegantan trik za rešavanje stampedo problema pri keširanju.

Naime, kada korisnik zatraži podatke, server će mu u najvećem broju slučajeva plasirati keširane informacije iz nekog od vida sistema keširanja o kojima je bilo reči. Međutim, kada podaci postanu neažurni, prvi sledeći korisnik će zatražiti nove podatke koji verovatno moraju da kontaktiraju bazu. Ukoliko je ovaj proces skup i dug, što je slučaj u većini situacija, pa tako i kod ispovesti.com, proces može da potraje, a samim tim može doći do keš stampeda. Međutim, kako bismo bili sigurni da naredni korisnici neće zahtevati izvršavanje istog procesa, problem rešavamo tako što će prvi korisnik koji započne ažuriranje keša, samo osvežiti postojeći keš,

odnosno produžiti mu trajanje i tako osigurati da naredni korisnici pri ispitivanju da li je keš svež i ažuran, dobiju lažnu informaciju i koriste keširane podatke iako nisu najnoviji. Ukoliko se koristi keširanje podataka u fajlovima ovo se može ostvariti na vrlo prost način funkcijom *touch()*.

```
<?php
$filename = $_SERVER['DOCUMENT_ROOT']."/cache".$id.".cache";
$interval = 120; // vreme do kada je fajl ažuran
if (file_exists( $filename ) && (time() - $interval) < filemtime(
$filename ) ) {
      // ukoliko je keš fajl ažuran, čitaj direktno
      readfile( $filename );
      exit();
} else{
      // ukoliko keš fajl nije ažuran, dovuci podatke iz baze ali       //
prethodno promeni vreme fajla za sledeće zahteve korisnika
      touch($filename);
      get_data_from_database();
      ...
}

?>
```

Najpre se inicijalizuje jedinstvena vrednost, odnosno putanja fajla u kojem će biti smešteni keširani podaci. Superglobalna promenljiva *($_SERVER['DOCUMENT_ROOT'])* ukazuje na glavni direktorijum sajta (*root folder)*. Nakon toga se inicijalizuje vrednost intervala keširanja na 120 sekundi. Naredni deo koda ispituje da li je fajl modifikovan pre više ili manje od 120 sekundi. Ukoliko je menjan u vremenu t manje od 120s, podaci su relativno sveži i korisniku se mogu proslediti u ovom obliku. Nasuprot tome, ukoliko fajl ne postoji, ili je t > 120s, pristupiće se ponovnom generisanju podataka i ova operacija može potrajati dugo i crpeti velike resurse. Kako bi se izbegao stampedo efekat, najpre će se iskoristiti funkcija *touch($filename)* kako bi svi naredni korisnici naišli na osvežen fajl i samim tim povukli stare podatke, dok se novi ne generišu. Čim se podaci izvuku iz baze podataka, fajl će se osvežiti ponovo tako da neće biti potrebe da se čeka još dodatnih 120 sekundi kako bi korisnici imali najnoviji sadržaj.

4.11. PROBLEM ENORMNO ČESTIH ZAHTEVA

Na mnogim društvenim mrežama, a i drugim visokoposećenim dinamičkim sajtovima, gde korisnici imaju mogućnost interakcije sa sadržajem, vrlo je verovatno da će određeni procesi biti veoma često korišćeni. Ovo se najviše odnosi na neku vrstu izražavanja mišljenja u vidu akcija - sviđa mi se, ne sviđa mi se (ili poznatije kao *like*).

Ista situacija dešava se i na sajtu ispovesti.com. Broj upisa komentara i novih postova, iznosi oko nekoliko hiljada dnevno. Međutim, broj akcija kao što su "sviđa mi se", odnosno u ovo slučaju - "odobravam" na istom sajtu iznose i do 150,000 dnevno. Ovo znači da se dnevno napravi 150,000 ajax poziva ka serveru koji zahtevaju izvršavanje jedne akcije odnosno ažuriranja podataka. I kao što je moguće izvršiti sve upise komentara i postova direktnim INSERT upitima (odnosno INSERT DELAY) bez problema, ovaj broj direktnih upita u bazu bi izazvao velike devijacije u pogledu performansi sistema.

Postoje verovatno mnoga rešenja za ovakav vid problema, a u ovom radu će se opisati jedan koji je ujedno i korišćen na ranije spomenutom portalu ispovesti.com

javascript:

```
function add_like(post_id) {
    $.post("/php/like_add.php", {
      id: post_id
    }, function (data) {
      if (data == "success") {
            // handle the response value and update the UI
      }
    })
}
```

Ovaj javascript kod je uprošćena verzija originala i za cilj ima samo da prikaže način pozivanja određene skripte na serveru putem ajax poziva. Naravno, pre samog pozivanja postoje određene provere validnosti podataka. Takođe, prilikom uspešnog odziva sa porukom "success" izvršava se određeno ažuriranje korisničkog interfejsa koje korisniku daje do znanja da je akcija uspešno izvršena.

like_add.php:

```
<?php
      require '../cron/lib/functions.php';
      $root_path = $_SERVER['DOCUMENT_ROOT'];
      if(isset($_POST['id'])) {
            $id = (int)$_POST['id'];
            $cache_entry = "like-add-cache-array";
            if(true == log_user_action($cache_entry, $id, 1))
            die('success');
      }
```

```
	die('error');
?>
```

Kod iznad predstavlja skriptu koju će ajax akcija pozvati ukoliko ne dođe do grešaka u izvršavanju samog poziva. Najpre se učitavaju određene funkcije koje su objašnjene u nastavku i inicijalizuje se vrednost *root path*-a, odnosno osnovne putanje. Nakon toga postoji provera vrednosti promenljive ID koja govori o identifikatoru ispovesti kojoj treba da se izvrši inkrementiranje broja pozitivnih ocena (*like).* Nakon toga se komandom *id = (int)$_POST['id'];* kastuje id kao celi broj kako bi se dodatno zaštitili od pokušaja injektovanja nevalidnih podataka. Inicijalizuje se određeni naziv ključa za keširanje i potom se koristi funkcija *log_user_action()* kako bi se upisala željena akcija. Postoji veliki broj akcija ali ćemo se zadržati samo na akciji *like* radi lakšeg objašnjenja koda.

functions.php:

```
<?php
function log_user_action($cache_entry, $id, $type) {
	$key = "0";
	switch($type) {
		case 1:
			$key = "ids"; //ispovesti
			break;
		case 2:
			$key = "cids"; //komentari
			break;
		case 3:
			$key = "my_id";
			break;
	}
	if(session_id() == '') session_start();
	if(isset($_SESSION[$key])) {
		if(in_array($id, $_SESSION[$key])) {
			return false;
		} else {
			array_push($_SESSION[$key], $id);
		}
	} else {
		$_SESSION[$key] = array($id);
	}
```

```
        require_once 'apc_mutex.php';
        $mutex = "mutex-" . $cache_entry;
        // dohvatanje locka
        apc_get_mutex($mutex);
        if(apc_exists($cache_entry)) {
                $id_array   = apc_fetch($cache_entry);
                $id_array[] = $id;
        } else {
                $id_array = array($id);
        }
        // čuva niz u cache-u
        apc_store($cache_entry, $id_array, 800);
        // puštanje locka
        apc_release_mutex($mutex);
        return true;
}

?>
```

U prethodnom delu koda opisana je funkcija logovanja aktivnosti korisnika. Naime, najpre se uz pomoć *switch* instrukcije ispituje tip akcije, u ovom slučaju *like*. Nakon toga se startuje sesija komandom *session_start* ukoliko već nije postojala do ovog trenutka. Potom se prostim *if* uslovom ispituje da li je već ovaj korisnik (sa određenom sesijom) već probao da oceni ovu ispovest, jer nije dozvoljeno povećavati ocenu određenom postu proizvoljni broj puta, već samo jednom. Ukoliko nije, komandom *array_push*() se smešta element (u ovom slučaju id ispovesti) u niz ispovesti koje je ovaj korisnik ocenio. Nakon ovog dela, pristupa se upisu određenog *id*-ja u apc niz u kojem se keširaju ovi podaci, koji će na određeno vreme biti upisan u bazu u jednom mahu. Dakle, najpre se komandom *if(apc_exists($cache_entry))* proverava da li postoji određeni niz ili ne u samom apc kešu. Ukoliko je uslov istinit, najpre se povlači dati niz iz keša, a potom se smešta dodatni element komandom $array[] = $var; [4]**.** Ovo je atipična i veoma korisna komanda koja je dostupna u PHP-u i označava dodavanje promenljive *$id* u niz *$id_array*. U suprotnom, stvara se novi niz sa postojećim elementom *$id*. Potom se čuva niz u kešu, na 800 sekundi i otpušta se ekskluzivni ključ kako bi bio slobodan za druge procese.

Cron job:

```
<?php
        $path = $_SERVER['DOCUMENT_ROOT'] . '/';
        include($path . 'connection.php');
        include($path . 'cron/lib/functions.php');
```

```
    // maksimalno dozvoljeno vreme izvršavanja
    set_time_limit(2 * 60);
    $cache_entry = "like-add-cache-array";
        $cacheconf   = $path . 'cache/query/like_conf.cache';
        $c1 = process_crone($cache_entry, $cacheconf, 3);
        if($c1) die('success');
        die('error');
?>
```

Prethodni kod predstavlja Cron job koji se izvršava radi upisa podataka u bazu. Već smo rekli da su akcije kao što su ocenjivanje ispovesti veoma česte i zato je najbolje rešenje skupiti što više ovih informacija na što optimalniji način, a potom odjednom sve upisati u bazu. Ovaj upis se odrađuje Cron job-om. Cron job je proces koji je moguće programirati na Linux operativnim sistemima, a osnovna mu je funkcija unapred zadato izvršavanje određenog fajla, koje može da se ponavlja proizvoljan broj puta. Naime, ovim fajlom najpre se učitavaju fajlovi connection.php (konektovanje sa bazom podataka) i functions.php (koji sadrži potrebne funkcije). Postavlja se ograničenje u vidu vremena izvršavanja upita na 120 sekundi. Fajl like_conf.cache služi samo za logovanje informacija o broju izvršenih upita i uspešnosti izvršavanja.

```
<?php
        function process_crone($cache_entry, $log_file, $type) {
        date_default_timezone_set('Europe/Belgrade');
        require_once 'apc_mutex.php';
        $mutex = "mutex-" . $cache_entry;
        // dohvatanje locka
        apc_get_mutex($mutex);
        if(apc_exists($cache_entry)) {
                $contents = apc_fetch($cache_entry);
        } else {
                apc_release_mutex($mutex);
                return false;
        }
        apc_delete($cache_entry);
        // puštanje locka
        apc_release_mutex($mutex);
        $total = 0;
```

```
	switch($type) {
		case 1:
			$total = update_like($contents);
			break;
		case 2:
			$total = update_dislike($contents);
			break;
	}

	$log_data = "cron : (".date("Y-m-d H:i:s").") > ".$total."  queries
executed.\n";
	file_put_contents($log_file, $log_data, FILE_APPEND);
	return true;
}

?>
```

Na početku prethodnog fajla, podešavaju se najpre određene meta informacije radi logovanja podataka. Potom se dohvata ekskluzivni ključ nad određenom apc keš memorijom i čitaju se podaci iz niza, nakon čega se niz prazni, a ključ oslobađa. Nakon toga u zavisnosti od tipa akcije, izvršava se odgovarajuća *update* funkcija. Potom se vrši logovanje u ranije pomenuti fajl.

```
<?php

function update_like($data) {
$sql = "UPDATE `main_ispovesti_table` SET `likes`=`likes`+:num
	WHERE ispovesti_id=:id";
	return execute_update($sql, $data);
}
?>
```

Prethodna funkcija, *update_like* predstavlja inicijalizaciju SQL upita koje će se izvršiti. Nakon toga se poziva funkcija *execute_update()* koja je opisana u nastavku.

```php
<?php

function execute_update($sql_string, $data) {
	global $db;
	$arr  = $data;
	$arr  = array_count_values($arr);
	$count = count($arr);
	$key  = 0;
	$value = 0;
	$stmt = $db->prepare($sql_string);
	$stmt->bindParam(':id', $key, PDO::PARAM_INT);
	$stmt->bindParam(':num', $value, PDO::PARAM_INT);
	foreach($arr as $key=>$value) {
		$stmt->execute();
	}
	return $count;
}
?>
```

Funkcija *execute_update()* radi već poznatu operaciju. Najpre se iz globalnog polja (*scope)* dohvata promenljiva *$db* koja je inicijalizovana prilikom pozivanja *connection.php.* Nakon što je pristup bazi omogućen, radi se priprema SQL upita metodom *$db->prepare(),* a potom se vrši dodeljivanje odgovarajućih vrednosti svojim parovima, odnosno promenljivama. Nakon ovoga, se za svaki par ključ-vrednost radi izvršavanje upita, pri čemu je ključ zapravo id ispovesti dok je value broj ocena koji se upisuje. Na ovaj način smo sprečili da se za svaku pojedinačnu akciju izvrši posebni upit već se prvo akumuliraju a potop upišu svi odjednom.

5. BUDUĆA UNAPREĐENJA ZA POVEĆANJE SKALABILNOSTI

Optimizacija sistema zapravo nikada ne dostiže kraj. Svakodnevno dolaze nove tehnologije, nova rešenja, nove tehnike za sprečavanje potencijalnih problema ili unapređenje postojećih solucija. Osim toga, za rastuće sisteme potrebno je kontinualno praćenje performansi korišćenih rešenja i pokušavanje drugih rešenja jer svaki sistem je specifičan i ne funkcionišu sve solucije podjednako kod svih aplikacija.

Što se tiče sajta ispovesti.com, planirana su mnoga unapređenja, kako sa korisničkog iskustva, tako i sa serverske strane radi unapređenja trenutnog okruženja. Inovacije koje predstoje, od kojih je najveća uvođenje korisničkih naloga, dovešće sa sobom i veliki broj izazova koji neće biti rešivi trenutnim optimizacionim tehnikama, a isto tako, povećanje korisnika dovodi u pitanje održivost trenutnog okruženja na duže staze, a kao i kod svakih problema, uvek je najbolje pokušati predvideti i reagovati na vreme, nego rešavati problem kada je on već uzeo maha.

Jedan od planiranih optimizacionih aspekata predstavlja promena trenutnog veb servera, odnosno httpd-a. Apache, iako pouzdan već dugi niz godina i sa velikim udelom na tržištu, polako napušta prvo mesto zbog svojih nedostataka prilikom velikih naleta opterećenja. Zbog svoje inertnosti i povlačenja velikih procesorskih kapaciteta stalnim stvaranjem novih procesa za svaki zahtev, Apache vrlo brzo dovede sistem do zasićenja, a kada mašina ostane bez operativne memorije i počne da koristi swap mehanizam, ceo sistem se urušava.

NginX se već nekoliko godina pokazao kao veoma pouzdan veb server i veliki broj poznatih sajtova koristi ovu platformu za opsluživanje velikog broja korisnika. NginX ne povlači proces prilikom svakog zahteva, već radi po principu događaja tj. *event-based* princip. O ovome je bilo više reči na početku ovog rada.

Pored promene veb servera, tim ispovesti.com smatra da, iako pouzdana, MySQL baza neće biti dovoljna u budućnosti da opsluži velik broj korisnika ukoliko on nastavi da raste. Samim tim, potrebno je osmisliti mehanizam koji će pratiti porast korisnika sa lakoćom uz odgovarajuće sisteme keširanja. Uvođenjem NoSQL baze kao jedan virtuelni sloj iznad postojeće MySQL baze može se znatno povećati skalabilnost sistema ako uzmemo u obzir da je najveći broj pristupa sajtu zapravo čitanje iz baze, dok je neuporedivo manji broj upisa. NoSQL baze su odličan alat za ovakvu situaciju jer rade na drugačijem principu, nisu struktuirane i poznate su po svojoj brzini. Podaci se čuvaju u vidu ključ-vrednost (*key-value*) i na ovaj način, NoSQL baza može da povremeno komunicira sa MySQL bazom radi održavanja ažurnih podataka, umesto što je trenutno MySQL opterećen svakim zahtevom koji nema svoju keširanu kopiju.

Pored ovih tehnika, ispovesti.com će sa velikom verovatnoćom u budućnosti koristiti usluge CDN sistema. CDN predstavlja odličan mehanizam za služenje statičkih podataka. Korisnici svakodnevno učitavaju slike, javascript, css i druge fajlove koji se retko menjaju. Ovi podaci nisu dinamički kao što je sam sadržaj, a bez sumnje zahtevaju veliki propusni opseg i konstantno opterećuju mrežu a i sam server koji mora da isporuči ogromne količine megabajta u kratkom vremenskom intervalu. Još jedna prednost CDN sistema je što iskorišćavaju informaciju

lokacije korisnika. I pošto se ovaj sistem sastoji od fragmenata distribuiranih sistema širom sveta, korisnicima se sadržaj služi sa najbliže i optimalne lokacije tako da se i u ovom slučaju dobija na brzini učitavanja stranice.

Još jedan potencijalni korak u ne tako bliskoj budućnosti bi moglo biti i uvođenje drugih mašina, odnosno horizontalno skaliranje. Ovo bi bio veliki poduhvat budući da bi bilo potrebno instalirati i organizovati sisteme koji bi održavali podatke ažurnima na obe mašine, kao i šeme balansiranja radi raspodele opterećenja

6. ZAKLJUČAK

Autor ovog rada se nada da nakon upoznavanja sa ovom temom čitaoci imaju jasnu sliku o problemima sa kojima se susreće jedan sajt sa visokom posetom. Naravno, ovo je relativan pojam ali kada je skalabilnost sistema u pitanju, uvek treba pretpostaviti od početka da će posećenost biti velika i sa tim na umu treba dizajnirati sistem. Jer ukoliko je aplikacija već posatala aktivna i dostupna, veoma je teško i kompleksno uvođenje velikih promena, a pogotovo ako te promene obuhvataju kompletan redizajn sistema. Sistem treba da ima mogućnost da podrži mnogostruko veći broj zahteva nego što postoji u prosečnim trenucima, jer vršna opterećenja, odnosno impulsi velikog saobraćaja, mogu da budu i veoma kratki, a dovoljni da čitav sistem počne da ulazi u fazu iz koje ne može da se povrati i kada dođe do zagušenja, verovatno je da će se očekivati downtime, odnosno vreme nepružanja servisa, što ne utiče dobro ni na jednu aplikaciju, a pogotovo ako ta aplikacija ima bitne operacije kao što su servisi banaka.

Kao što se može videti iz rada, jedan od najbitnijih aspekata prilikom razmišljanja o skaliranju sistema predstavljaju sistemi keširanja, jer se može primetiti da su oni doprineli najvećem povećanju konkurentnih korisnika i ubrzanom radu čitavog sistema. Težište je sa baze podataka preneto najpre na fajlove u kojima su keširani podaci, a kasnije i u operativnu memoriju što je znatno ubrzalo čitanje i pisanje i samim tim otvorilo vrata za dodatno povećanje kapaciteta čitave aplikacije.

Naravno, veliki benefit pronađen je i u načinima prebacivanja fajlova i velikog dela logike na stranu klijenta, koji se keširaju u pretraživaču, dok je do korisnika stizala minimalna količina informacija u vidu JSON objekata, a kasnije kroz javascript alate, parsiranjem JSONa dobijaju se odgovarajući nizovi koji se kasnije pakuju u html formate i prikazuju korisniku.

Autor bi, ponesen svojim iskustvom, savetovao sve čitaoce koji planiraju ili se upuštaju u neku vrstu startap projekta, da u velikoj meri planiraju sistem i dizajn pre nego počnu sa samom implementacijom, jer jer skalabilnost nešto na šta treba misliti od samog početka, a nikako kada je sistem već dostupan krajnjim korisnicima.

LITERATURA

[1] O'Reilly, "*Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications*", O'Reilly media 2006.

[2] Martin L. Abbott, Micael T. Fischer "*Scalability Rules: 50 Principles for Scaling Web Sites*", AKF Consulting, 2011.

[3] Theo Schlossnagle, "*s*", Sams Publishing, 2006.

[4] *Understanding* Official PHP website [Online], Available: http://php.net/