Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nejc Tomažič

Primerjava uporabe in učinkovitosti sodobnih programskih vmesnikov porazdeljenega predpomnilnika

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Z razvojem porazdeljenih sistemov se je pojavila potreba po porazdeljenih predpomnilnikih, ki omogočajo sočasen dostop do podatkov iz več vozlišč. Programski vmesniki porazdeljenega pomnilnika rešujejo nekatere težave pri uporabi le-teh ter povečajo njegovo učinkovitost. V diplomski nalogi najprej predstavite porazdeljene sisteme, porazdeljene predpomnilnike ter vlogo vmesnikov porazdeljenega predpomnilnika. Za tri izbrane sodobne vmesnike (Memcached, Infinispan in Hazelcast) predstavite njihovo uporabo ter primerjajte njihovo učinkovitost dostopa do podatkov, izvedbe preprostih in agrigiranih povpraševanj. Rezultate pregledno predstavite ter strnite zaključke.

Zahvaljujem se mentorju, viš. pred. dr. Igorju Rožancu, za mentorstvo, nasvete in odlično komunikacijo.
Ani, ki me je podpirala in mi stala ob strani. Mateju za pomoč in motiviranje med študijem ter Neži za pomoč pri oblikovanju diplomskega dela.
Posebno zahvalo namenjam staršem in sestri. Omogočili ste mi študij in me podpirali pri vseh mojih odločitvah.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DC	Distributed Cache	porazdeljeni predpomnilnik
SQL	Structured Query Language	strukturirani povpraševalni jezik
DB	Database	podatkovna baza
ECM	Embedded Cache Manager	upravljalec vgrajenega predpomnilnika
P2P	Peer to Peer	vsak z vsakim
NoSQL	Not Only SQL/Non Relational	ne-relacijska baza
LRU	Least Recently Used (algorithm)	najdlje neuporabljen (algoritem)
LFU	Least Frequently Used (algorithm)	najmanj pogosto uporabljenega (algoritem)

Povzetek

Naslov: Primerjava uporabe in učinkovitosti sodobnih programskih vmesnikov porazdeljenega predpomnilnika

Avtor: Nejc Tomažič

V diplomskem delu je predstavljena uporaba in primerjava modernih programskih vmesnikov porazdeljenega predpomnilnika. Namen dela je pomoč programskim arhitektom pri izbiri vmesnika za izbrano programsko rešitev.

Na trgu je danes na voljo mnogo različnih vmesnikov porazdeljenega predpomnilnika, ki so si med seboj različni po namenu uporabe, združljivosti z ostalimi kompomentami sistema in zahtevnosti. V okviru pričujočega dela si izberemo tri izmed najpogosteje uporabljenih vmesnikov, jih analiziramo, uporabimo in ovrednotimo. Ti vmesniki so Memcached, Infinispan in Hazelcast. Na primeru aplikacije za hranjenje podatkov o vozilih predstavimo uporabo navedenih vmesnikov ter opišemo navodila za vzpostavitev osnovnega delovanja v praksi.

Z izvajanjem simulacij na praktičnih primerih vmesnike primerjamo in prikažemo, kateri izmed njih je primernejši za uporabo glede na začetne zahteve aplikacije. Primerjavo izvedemo za branje podatka po naključnem ključu, poizvedbo po atributu in agregacijo poizvedbe po atributu.

Ključne besede: porazdeljeni predpomnilnik, porazdeljeni sistemi, gruča, vozlišče, predpomnjenje.

Abstract

Title: Efficiency and use case comparison of modern distributed cache solutions

Author: Nejc Tomažič

This BSc thesis paper presents comparison and the use of modern distributed caching solutions. The purpose is to help software architects with a decision for which distributed caching interaface they should decide, when designing new software solutions.

Nowadays there are several different solutions on the market, but their purpose, complexity and compatibility with other system components can differ greatly. In this paper we choose three most commonly used interfaces, to analyze, integrate and evaluate. These interfaces are Memcached, Infinispan and Hazelcast. By developing a simple web application for storing vehicle data we present the usage of these interfaces, compare them and demonstrate integration of each.

With simulations of practical examples, we compare them and suggest which is the best choice for a particular use. Comparison is made to read the values by random key, querying by attribute and aggregating query results by specific attribute.

Keywords: distributed cache, distributed systems, cluster, node, caching.

Poglavje 1

Uvod

V preteklosti je bila večina poslovnih aplikacij razvitih z aplikacijsko arhitekturo monolita [1]. To pomeni, da je bil program v celoti razvit kot samostojen kos programske opreme in ni bil odvisen od drugih programskih delov. Tak program je bil zagnan na enem sistemu, se tam izvajal in bil sam sebi zadosten. Če je začel delovati počasneje, se je pogosto fizično nadgradilo sistem, kjer se je program izvajal, ali pa vzporedno zagnalo več inačic aplikacije. Za monolitne aplikacije se najpogosteje uporablja vertikalno razširjanje sredstev (angl. vertical scalability) v obliki nadgradnje strojne opreme (dodatni trdi disk, večji pomnilnik, hitrejši procesor) [9]. Nadgrajevanje monolitnih aplikacij in razširjanje sredstev zanje zato pogosto vodi v nedostopnost aplikacije in visoke stroške.

S porastom spletnih aplikacij, kjer število uporabnikov narašča iz dneva v dan in se dostopnost sistema pričakuje v vsakem trenutku, je monolitna arhitektura počasi začela izgubljati na priljubljenosti med načrtovalci programske opreme. Pogosto ze izkaže, da v poslovnih aplikacijah izbira monolitne arhitekture lahko pomeni tudi izgubo prihodka. Do tega lahko pride, če so bile pri načrtovanju podcenjene zahteve. Naknadno vertikalno razširanje zmogljivosti lahko celo privede do nedostopnosti aplikacije, ali vsaj zahteva veliko dodatnega dela, da sistem ostane dostopen tudi med nadgradnjo [10].

Da bi se izognili negativnim vplivom monolitne arhitekture na delovanje

aplikacije, so se v prejšnjem desetletju začele pojavljati reštive, pri katerih je mogoče nadgraditi sistemska sredstva brez večjih posegov v delovanje aplikacije, ki teče na njih. Pojavila se je želja po horizontalni razširljivosti sistemov (angl. horizontal scalability), kar pomeni, da se je na primer namesto nadgradnje obstoječega fizičnega strežnika raje dodalo še enega. Vse pogosteje uporabljeni so postali t.i. **porazdeljeni sistemi** (angl. distributed systems), kjer so naloge aplikacije porazdeljene med več sistemi (vozlišči) in vsak izmed teh sistemov skrbi le za svoj del nalog, skupaj pa sistemi navadno sestavljajo gručo vozlišč (angl. cluster). Da je takšno porazdeljevanje in razširjanje mogoče, je potrebno aplikacijo načrtovati in razviti drugače.

Da je aplikacija zadostila zahtevam horizontalnega razširjanja in porazdeljenega procesiranja, je morala biti vsaka logična enota aplikacije razvita kot svoj ločen del, ki z ostalimi deli aplikacije komunicira na dogovorjeni način. Ena izmed najbolj razširjenih metod načrtovanja porazdeljenih sistemov je **arhitektura mikro storitev** (angl. microservices architecture). Pri slednji je vsak del aplikacije (storitev) mogoče ločeno postaviti na svoj strežnik ali več njih, imenovanih vozlišča, ki skupaj tvorijo gručo. Arhitektura mikro storitev rešuje mnogo težav klasične monolitne aplikacije, a prinaša druge izzive. Klasična aplikacija lahko podatek, do katerega pogosto dostopa, preprosto shrani v lokalni predpomnilnik strežnika, na katerem se izvaja in ta je dostopen kadarkoli in kateremukoli delu aplikacije. V porazdeljenih sistemih to ni tako preprosto. Aplikacija v obliki mikro storitev se namreč lahko hkrati izvaja na več fizično ločenih strežnikih, saj je vsaka storitev postavljena ločeno. To pomeni, da storitve nimajo neposrednega dostopa do istega predpomnilnika [2].

Pogosto naletimo na potrebo, ko dve storitvi takšne aplikacije potrebujeta isti podatek. Če je ta podatek na voljo le v podatkovni bazi, se kmalu soočimo s težavo **tekmovanja storitev za sredstva** – v tem primeru za branje določenega podatka, ki je morda celo isti za obe storitvi. Torej dvakrat beremo isti podatek in aplikacija čaka sama sebe. Logična izboljšava je, da si ta podatek storitev shrani v lokalni predpomnilnik, kot je to mogoče

v monolitni arhitekturi. S tem dosežemo, da bo vsaka storitev le enkrat prebrala isti podatek, vendar pa ta podatek zdaj hranimo na več mestih: v podatkovni bazi, v predpomnilniku vozlišča prve storitve in v predpomnilniku vozlišča druge storitve. To pomeni, da potrebujemo na vsakem vozlišču veliko predpomnilniškega prostora, kar pa v resnici niti ni največja težava te rešitve. Ta je stanje, ko se v predpomnilniku nahajajo neveljavni podatki (angl. stale data).

Predstavljajmo si, da prva storitev podatek spremeni. Spremenjeno vrednost zapiše v podatkovno bazo in nadomesti staro v predpomnilniku svojega vozlišča. Druga storitev za spremembo podatka ne ve in še vedno uporablja staro vrednost, shranjeno v predpomnilniku svojega vozlišča. Aplikacija lahko pride na ta način v **nedovoljeno stanje** in pravilno delovanje je lahko ogroženo. S hranjenjem podatka v več fizično ločenih predpomnilnikih smo si v primeru takšne uporabe nakopali veliko težav, katere nimajo vedno preproste rešitve [11].

Zgoraj opisan scenarij je še toliko bolj izrazit v arhitekturi mikro storitev, kjer ena izmed mikro storitev pogosto teče na več vozliščih gruče hkrati. Ker so se potrebe deljenja podatkov med mikro storitvami večale in so prej pravilo kot pa izjema, so se na trgu začeli pojavljati programski vmesniki, razviti z namenom reševanja omenjenega problema. Združujejo se pod skupnim imenom – **vmesniki porazdeljenega predpomnilnika** (angl. distributed cache systems) [4].

Ti vmesniki nudijo razvijalcu programske opreme preproste mehanizme za hranjenje podatkov v predpomnilniku enega vozlišča, kjer teče en primerek storitve aplikacije, na voljo pa je tudi vsem ostalim primerkom storitve, ki so del iste aplikacije. Omogočajo nam nastavljanje veljavnosti podatka, določanje količine hranjenih podatkov, algoritma odstranjevanja elementov, ko zmanjkuje prostora v predpomnilniku ter še mnogo drugih uporabnih atributov.

Ker vsi vmesniki porazdeljenega predpomnjenja niso enaki, delujejo na različen način ter zahtevajo različne načine postavitve in integracije. Zato

izbira pravega ni samoumevna. Programski arhitekt mora torej pri načrtovanju aplikacije presoditi, ali potrebuje porazdeljeno predpomnjenje. Če ga, v kakšni obliki ter s kakšnim namenom. Ob izbiri porazdeljenega predpomnjenja mora izbrati najprimernejši vmesnik na trgu in ga pravilno vgraditi v svojo aplikacijo.

Cilj diplomskega dela je olajšati odločitve ter prihraniti čas načrtovalcu programa pri izbiri vmesnika za porazdeljeno predpomnjenje. V nadaljevanju bodo podrobneje predstavljeni najpogostejši primeri uporabe in tudi kratki vodiči za vzpostavitev osnovnega delovanja vsakega izmed treh izbranih vmesnikov.

Diplomsko delo sestavljajo naslednja poglavja:

- **Uvod**, v katerem se seznanimo s problemom in rešitvijo, ki je obravnavana v tem diplomskem delu.
- **Pregled področja**, kjer predstavimo porazdeljene sisteme in porazdeljeni predpomnilnik.
- **Primerjava vmesnikov porazdeljenega predpomnilnika**, v katerem določimo vmesnike za primerjavo, določimo ocenjevalne parametre in predstavimo simulacijo.
- **Implementacija**, kjer opišemo aplikacijo, ki je bila razvita za izvajanje simulacije in predstavimo implementacije izbranih vmesnikov.
- **Rezultati**, kjer primerjamo in predstavimo rezultate simulacij.
- **Sklep**, v katerem podamo sklepno misel, podamo smiseln zaključek in predlagamo možnosti nadgradnje.

Poglavje 2

Pregled področja

2.1 Porazdeljeni sistemi

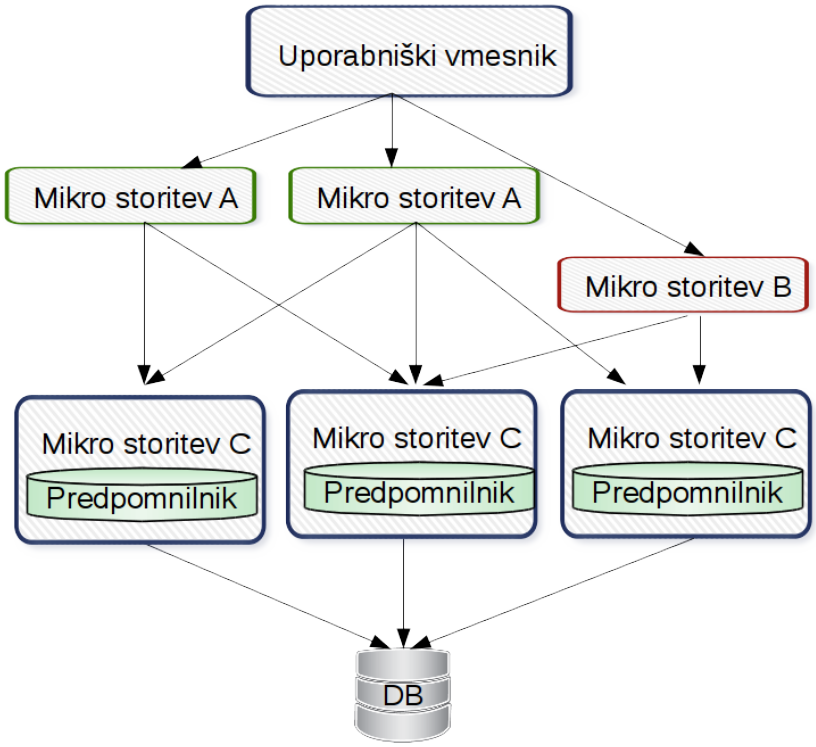
Ker je porazdeljeno predpomnjenje navadno uporabljeno v porazdeljenih sistemih, bomo najprej na kratko opisali, kaj so porazdeljeni sistemi.

Porazdeljen sistem je skupina entitet, kjer je vsaka izmed entitet avtonomna, programabilna, asinhrona in včasih tudi nedosegljiva drugim. Entitete so med seboj povezane s komunikacijskim medijem, ki pa ni nujno zanesljiv. Zato morajo biti entitete med seboj asihnhrone, da ne čakajo ena na drugo v primeru, ko komunikacija med njimi ne deluje [13].

Entitete so v aplikacijski arihtekturi navadno računalniki (v strokovni literaturi imenovani vozlišča), ki izvajajo določeno nalogo, komunikacijski medij pa je lokalno ali javno komunikacijsko omrežje oziroma internet. Skupaj tvorijo gručo vozlišč.

Porazdeljeni sistemi so infrastruktura za porazdeljeno procesiranje. Najbolj razširjen koncept v zadnjih desetih letih so **mikro storitve**. V arhitekturi mikro storitev (angl. microservices architecture) je vsaka storitev odgovorna za procesiranje svoje naloge, ki je v teoriji neodvisna od ostalih storitev. Vsaka izmed storitev je horizontalno razširljiva in lahko teče vzporedno na enemu ali več vozliščih [5].

Na sliki 2.1 vidimo aplikacijo, katere osnovni gradniki so uporabniški



Slika 2.1: Arhitektura mikro storitev.

vmesnik, tri različne storitve (**A**, **B** in **C**) ter podatkovna baza. Storitev **A** se izvaja paralelno – porazdeljeno na dveh entitetah, storitev **B** na eni, storitev **C** pa na treh. Skupno je vseh izvajanih entitet šest in najpogosteje bi se v praksi vsaka izmed njih izvajala na ločenem vozlišču gruče (bodisi virtualnemu ali fizičnemu).

2.2 Porazdeljeni predpomnilnik

Porazdeljeni predpomnilnik je nadgradnja klasičnega lokalnega predpomnilnika računalnika oziroma strežnika. Porazdeljeni predpomnilnik nam da možnost, da si povezana gruča vozlišč deli predpomnilnike. Tak sistem iz vidika enega vozlišča deluje kot en sam velik predpomnilnik. Podatek, ki je vne-

šen v predpomnilnik na enemu od vozlišč, je sčasoma na voljo vsem ostalim vozliščem, shranjen pa je le v enem fizičnem pomnilniku. Če nam vmesnik omogoča, lahko nastavimo **podvajanje podatkov** (angl. replication), s čimer dosežemo, da je podatek shranjen na več vozliščih hkrati. To je uporabna lastnost, saj s tem preprečimo izgubo podatka iz porazdeljenega predpomnilnika, v primeru da vozlišče, na katerem se je podatek nahajal, zapusti gručo [4].

Porazdeljeno predpomnjenje je danes prisotno že v več porazdeljenih sistemih in ga včasih uporabljamo nevede, sploh če je aplikacija postavljena v okolju, ki nam je na voljo kot storitev. Primer je Google App Engine [3].

2.2.1 Prednosti porazdeljenega predpomnilnika

Bistvene prednosti pri uporabi v porazdeljenih sistemih:

Skladnost (angl. consistency): predpomnjeni podatki so skladni med vsemi vozlišči v gruči, ne glede na to, katero izmed vozlišč je podatek vneslo oziroma spremenilo.

Visoka dosegljivost (angl. high availability): predpomnjeni podatki preživijo nadgradnje aplikacij na vozliščih v gruči ter do neke mere odstranjevanje in dodajanje vozlišč v gručo.

Razbremenjenost osnovnega podatkovnega vira: če v predpomnilniku hranimo podatke iz trajnega predpomnilnika, se število dostopov neposredno na osnovni vir podatkov v trajnem pomnilniku drastično zmanjša. Podatek je namreč prebran enkrat, zapisan v predpomnilnik in nato na voljo vsem ostalim vozliščem v gruči.

2.2.2 Osnovne lastnosti porazdeljenega predpomnilnika

Čas poteka (angl. expiration) oziroma čas veljavnosti: najpogosteje se nastavi *drsni čas*. Ta določa, po koliko časa od zadnjega dostopa do

dotičnega podatka naj bo ta odstranjen iz njega. S tem je zagotovljeno, da so podatki v predpomnilniku tisti, ki jih program res potrebuje.

Druga pogosta nastavitev pa je *absolutni čas veljavnosti*, ki se šteje, odkar je bil podatek prvič vnešen v predpomnilnik. Absolutni čas se navadno nastavi, ko je podatek trajno zapisan na drugi lokaciji (v podatkovni bazi) in obstaja možnost, da se podatek na drugi lokaciji spremeni, predpomnilnik pa za to ne ve. Za takšen podatek v predpomnilniku pravimo, da je zastarel.

Izselitev (angl. eviction): ker podatki v porazdeljenem predpomnilniku najpogosteje niso trajni in so hranjeni le v predpomnilnikih posameznih vozlišč z omejeno kapaciteto, je treba poskbeti za izseljevanje podatkov iz predpomnilnika, ko je ta meja dosežena. Meja se nastavi v obliki količine predpomnilnika ali največjega števila posameznih podatkov. Priporočena je nastavitev količine predpomnilnika, kajti tako lažje zagotovimo, da predpomnilnik sistema ne bo postal preveč zapolnjen. Za izseljevanje podatkov se najpogosteje uporabljata algoritma najdlje neuporabljenega (angl. Least Recently Used – LRU) in najmanj pogosto uporabljenega (angl. Least Frequently Used– LFU). Pri LRU algoritmu je iz predpomnilnika odstranjen podatek, do katerega že najdlje časa ni nihče dostopal. Pri drugem (LFU) pa se odstrani podatek, ki se statistično uporablja najmanj.

Zavedanje relacij med podatki (angl. caching relational data): kadar v predpomnilniku hranimo relacijsko povezane podatke, je odlična lastnost vmesnika, da se tega zaveda. Če se tega zaveda vmesnik predpomnilnika, je to ena stvar manj, za katero mora skrbeti razvijalec, ki upravlja in obdeluje te podatke.

Sinhronizacija predpomnilnika s podatkovno bazo (angl. database synchronization): ta funkcionalnost pride do izraza, kadar je predpomnilnik uporabljen za hranjenje podatkov, ki so sicer trajno shranjeni

na drugi lokaciji (naprimer v podatkovni bazi ali na trdem disku). Omogoča namreč, da je vmesnik obveščen o spremembi podatka v trajnem pomnilniku. Vmesnik po obvestilu dotičen podatek izseli iz predpomnilnika, ali pa pa ga ponovno prebere iz trajnega pomnilnika.

Branje skozi (angl. read-through): tak predpomnilnik omogoča branje zahtevanega podatka neposredno iz podatkovne baze, če ni bil najden v predpomnilniku.

Pisanje skozi (angl. write-through): tak predpomnilnik omogoča pisanje podatka neposredno v podatkovno bazo, ko je ta vnešen ali posodobljen v predpomnilniku.

Podpora poizvedbam (angl. querying): omogoča izvajanje poizvedb nad podatki v predpomnilniku.

Podvajanje podatkov (angl. replication): omogoča, da je podatek v predpomnilniku hkrati na več vozliščih gruče, med katerimi je porazdeljen. V primeru, da eno od vozlišč zapusti gručo, bodo podatki, ki jih je hranilo to vozlišče, še vedno na voljo, saj bo obstajala kopija na enem ali več drugih vozliščih.

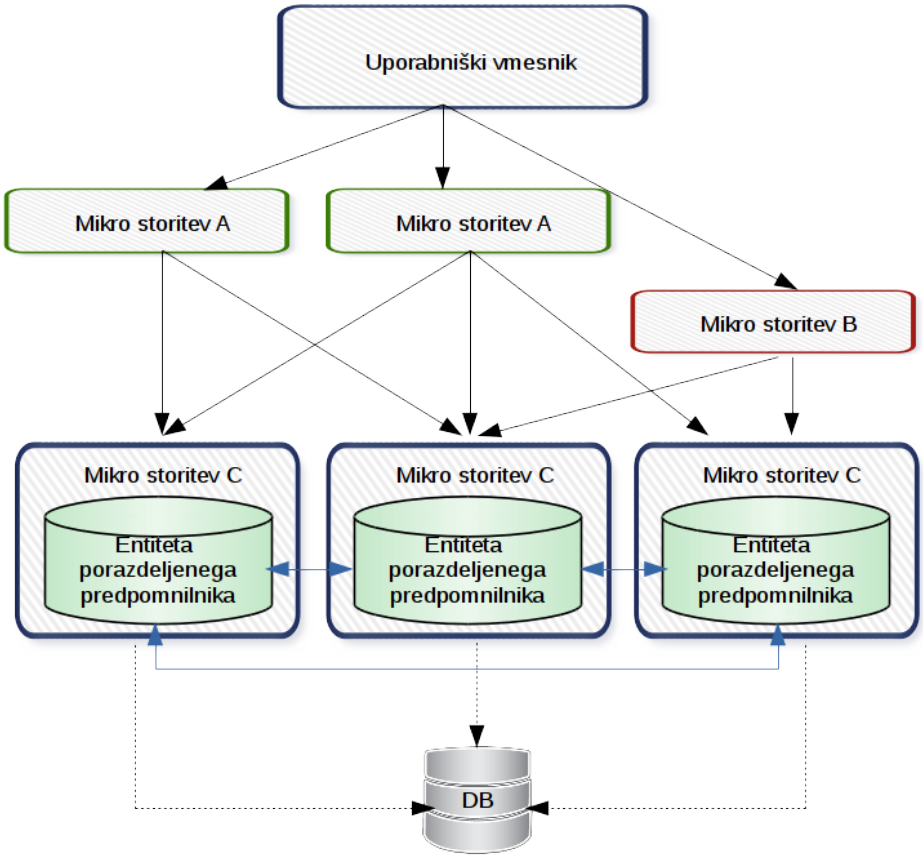
Razširljivost in visoka dosegljivost (angl. scalability and high availability): je lastnost, ki dovoljuje dodajanje in odstranjevanje vozlišč predpomnilnika medtem ko je v uporabi, uporabniki pa ga lahko medtem nemoteno uporabljajo.

2.2.3 Delitev porazdeljenih predpomnilnikov

Delovanja različnih vmesnikov porazdeljenega predpomnilnika se med seboj razlikujejo, vseeno pa obstajata dve bolj izraziti skupini. Omembe vredno je tudi to, da nekateri vmesniki podpirajo oba načina delovanja.

Vgrajeni (angl. embedded): predpomnilnik je vgrajen v aplikacijski strežnik (vozlišče gruče) skupaj z aplikacijo, ki se izvaja paralelno na več

vozliščih gruče. Ta način delovanja je primeren za hitro procesiranje zahtevkov, ki za izvedbo zahtevajo dostopnost do določenega podatka iz vseh vozlišč. Ker zahtevek lahko prispe na katerokoli izmed vozlišč, je pomembno, da je dostop do podatka hiter in da je vrednost podatka skladna med vsemi vozlišči. Če želimo predpomnilnik razširiti, to navadno pomeni novo vozlišče v obliki aplikacijskega strežnika, kjer se bo izvajala tudi dodatna inačica aplikacije. Dostop do podatkov je navadno zelo hiter. Pri izbiri dobrega algoritma in učinkovitega porazdeljevanja zahtevkov vozliščem je zahtevani podatek z veliko verjetnostjo namreč že v predpomnilniku vozlišča, ki je bil določen za obdelavo zahtevka, ki ta podatek potrebuje. Slika 2.2 prikazuje v vozlišča *storitve C* vgrajeni predpomnilnik.



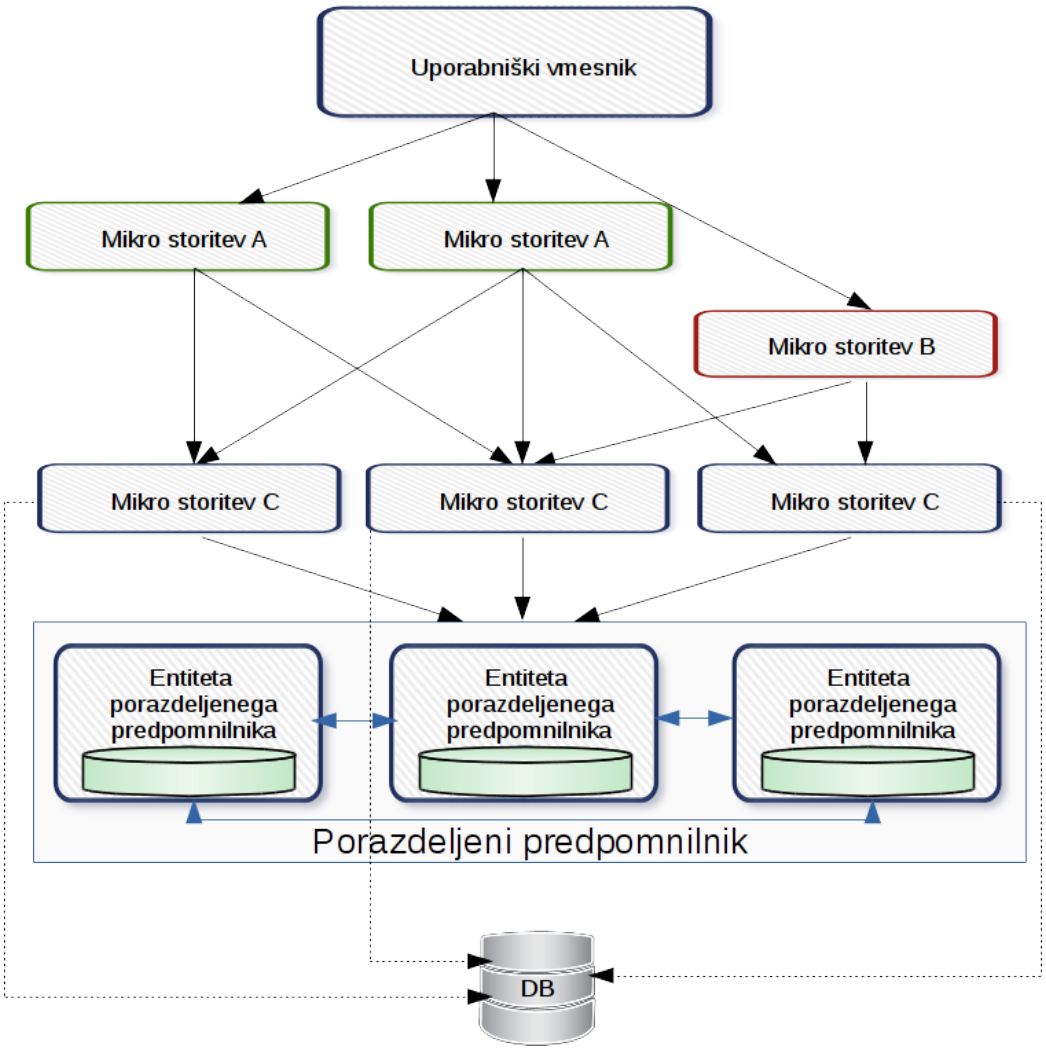
Slika 2.2: Arhitektura mikro storitev z vgrajenim porazdeljenim predpomnilnikom.

Strežnik/odjemalec: predpomnilnik je porazdeljen na namenskih strežnikih neodvisno od aplikacijskih strežnikov, ki ta predpomnilnik uporabljajo. Ta način je navadno uporabljen v aplikacijah, ki imajo opravka z večjimi količinami podatkov, sam proces obdelave pa ni nujno tako zahteven. Razširjanje predpomnilnika je v tem načinu veliko bolj neodvisno od aplikacije in aplikacijskih strežnikov. V primeru, ko naši aplikacijski strežniki opravljajo svoje delo dovolj hitro, primanjkuje pa predpomnilnika, se v gručo porazdeljenega predpomnilnika enostavno doda novo vozlišče.

Čas dostopa do podatka je tu nekoliko daljši, saj je podatek praviloma

na drugi lokaciji v omrežju kot pa aplikacija, ki podatek potrebuje. To težavo nekateri vmesniki rešujejo s t. i. **bližnjim predpomnilnikom** (angl. near cache), ki je vgrajen v vsako vozlišče gruče aplikacijskih strežnikov.

V tem načinu so strežniki, ki predpomnijo podatke, lahko člani gruče (vozlišča), ki med seboj komunicirajo z uporabo dogovorjenega protokola (najpogosteje P2P – vsak z vsakim), ali pa se drug drugega ne zavedajo. Ti vmesniki za pravilno delovanje potrebujejo tudi namensko programsko opremo odjemalca. Odjemalec namreč hrani seznam strežnikov, nato pa se na podlagi algoritma odloča, na katerega od strežnikov bo shranil določen podatek. Slika 2.3 prikazuje predpomnilnik tipa strežnik/odjemalec, kjer vsaka izmed *storitev C* uporablja odjemalca, ki komunicira s strežniki v predpomnilniku.



Slika 2.3: Arhitektura mikro storitev s porazdeljenim predpomnilnikom tipa strežnik/odjemalec.

2.3 Uporaba porazdeljenega predpomnilnika

Porazdeljeni predpomnilnik je največkrat uporabljen za naslednja namena:

Predpomnilnik podatkovni bazi: podatkovna baza se v večini primerov uporabe nahaja na trdem disku podatkovnega strežnika. Branje podat-

kov iz trdega diska je počasnejše kot branje iz predpomnilnika, hkrati pa je omejeno največje število sočasnih povezav na bazo. Dostop do podatkov v bazi tako hitro postane ozko grlo aplikacije.

Shranjevanje podatkov: porazdeljeni predpomnilnik lahko dobro služi tudi kot nerelacijska (NoSQL) podatkovna baza. Mnogo vmesnikov porazdeljenega predpomnilnika ima možnost trajnega shranjevanja vsebine, katero je mogoče ob svežem zagonu naložiti v predpomnilnik.

Če se vrnemo k sliki 2.1, vidimo, da storitev **C** dostopa do podatkovne baze. Podatkovna baza se v praksi navadno nahaja na fizično ločeni lokaciji od gruče aplikacijskih strežnikov, kjer so postavljene entitete naših storitev. Vsak dostop do podatkovne baze zahteva določen čas, prenos vsebine nazaj do aplikacijskega strežnika pa tudi pasovno širino na povezavi med njima. Če vse entitete storitve **C** potrebujejo isti podatek, med njimi pride do tekmovanja za sredstva, v tem primeru za dostop do podatkovne baze in dotičnega podatka.

Opisana težava je šolski primer, ki ga rešuje porazdeljeni predpomnilnik. Če med aplikacijske strežnike storitve **C** in podatkovno bazo vpeljemo porazdeljeni predpomnilnik, lahko drastično zmanjšamo število dostopov do podatkovne baze.

Slika 2.3 prikazuje novo shemo arhitekture z vpeljanim porazdeljenim predpomnilnikom.

Vsaka entiteta porazdeljenega predpomnilnika na sliki 2.3 je navadno ločen strežnik, ali pa je del aplikacijskega strežnika, na kateri je postavljena mikro storitev **C**. V tem primeru je porazdeljeni predpomnilnik sestavljen iz gruče treh vozlišč. Kadar storitev **C** potrebuje nek podatek, naredi poizvedbo po njem v porazdeljenem predpomnilniku. Če ga tam ne najde, naredi poizvedbo neposredno na podatkovno bazo, rezultat poizvedbe (najden podatek) pa shrani v porazdeljeni predpomnilnik. Mnogo vmesnikov porazdeljenega predpomnilnika to počne avtomatično in se uporabnik ne zaveda, da je bila poizvedba preusmerjena na podatkovno bazo – lastnost *branja skozi*.

2.4 Razvrščanje podatkov

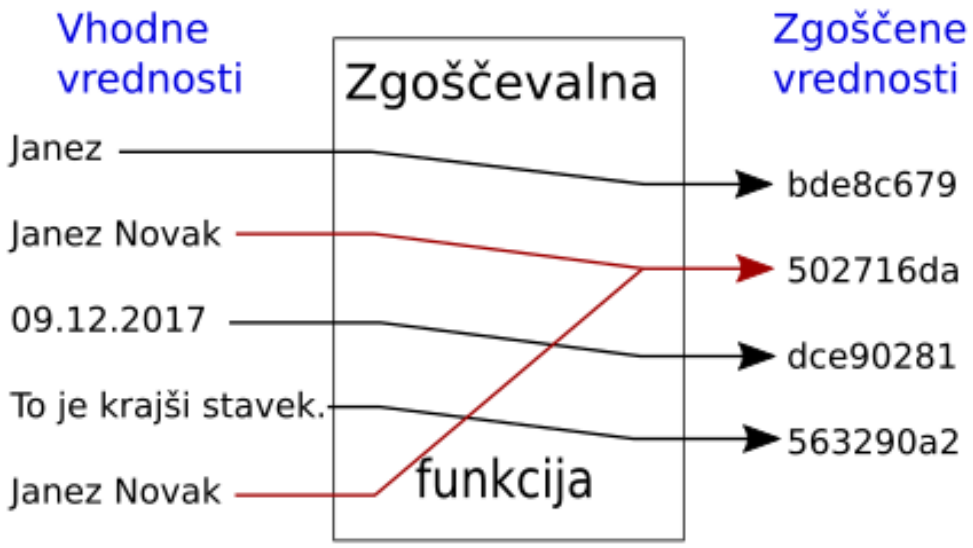
Ker so podatki v porazdeljenem predpomnilniku porazdeljeni na različnih vozliščih gruče, mora imeti vmesnik način za odločanje, na katero izmed vozlišč bo poslal zahtevek za hranjenje ali poizvedbo določenega podatka. Za to odločanje je največkrat uporabljen **algoritem doslednega razvrščanja** (angl. consistent hashing algorithm), ki je opisan v razdelku 2.4.3. Ta na podlagi **zgoščene vrednosti ključa** podatka enolično določa vozlišče, na katerega moramo zahtevek poslati. Za izračun te vrednosti se uporablja ena izmed mnogih zgoščevalnih funkcij.

2.4.1 Zgoščevalna funkcija

Zgoščevalna funkcija podatek poljubne dolžine pretvori v podatek fiksne dolžine – zgoščeno vrednost, pri tem pa v največji možni meri ohranja naslednje lastnosti [39]:

- Determinizem: ista vhodna vrednost funkcije vrača enak rezultat.
- Enakomerno porazdeljenost: verjetnost vsake možne zgoščene vrednosti je čimbolj enaka.
- Definiran razpon zgoščenih vrednosti: zaželjeno je, da je razpon zgoščenih vrednosti vnaprej definiran.
- Normalizacija vhodnih parametrov: ta omogoča, da vhodni podatek normaliziramo do določene stopnje. To pomeni, da lahko določene anomalije oziroma razlike v vhodnih podatkih ignoriramo pri izračunu zgoščene vrednosti.
- Enosmernost: iz zgoščene vrednosti ni mogoče rekonstruirati originalne vrednosti iz katere je bila izračunana.

Slika 2.4 prikazuje delovanje zgoščevalne funcije. Poudarek je na tem, da se vrednost *Janez Novak* v obeh primerih pretvori v isto zgoščeno vrednost.



Slika 2.4: Zgoščevalna funkcija.

Seznamu vseh možnih zgoščenih vrednosti pravimo **tabela zgoščenih vrednosti**, za katero je zaželjeno, da je razpon vrednosti znan. Če bi imeli še vrednost *janez novak* (mali začetnici), bi z normalizacijo lahko dosegli, da se tudi ta pretvori v isto zgoščeno vrednost.

2.4.2 Uporaba zgoščevalne funkcije za določanje ciljnega vozlišča

Predstavljajmo si, da oštevilčimo vozlišča s števili $0, 1, 2, 3, \ldots, n-1$, kjer je n število vozlišč. Želimo shraniti podatek s ključem in vrednostjo (k, v) v predpomnilnik. Za izbiro vozlišča za shranjevanje podatka lahko uporabimo preprosto funkcijo $hash(k) mod(n)$, pri čemer je $hash()$ zgoščevalna funkcija, ki z zgoščeno vrednostjo ključa po modulu skupnega števila vozlišč določi izbrano vozlišče.

Navedena funkcija delo opravlja popolnoma zadovoljivo, vendar ni skladna z eno od glavnih lastnosti porazdeljenih predpomnilnikov – z razširja-

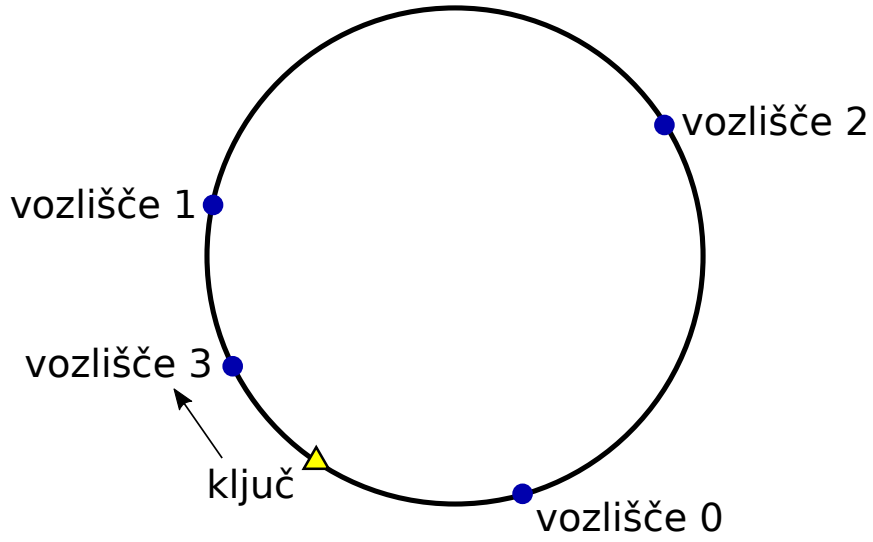
njem. Če uporabljamo to preprosto funkcijo in v gručo vozlišč porazdeljenega predpomnilnika dodamo novo vozlišče, se funkcija spremeni v $hash(k) mod(n+1)$. To posledično pomeni, da moramo premakniti skoraj celoten delež $(\frac{n}{n+1})$ podatkov na drugo lokacijo. To je seveda časovno zahtevno, hkrati pa ustvarja ogromno prometa na povezavi med vozlišči. Zaradi te pomanjkljivosti se v sistemih za porazdeljeno pomnjenje uporablja izboljšan **algoritem doslednega razvrščanja** (razdelek 2.4.3) [7].

2.4.3 Algoritem doslednega razvrščanja

Da bi odpravili pomanjkljivost zgornje rešitve, želimo doseči, da bo število podatkov, ki morajo biti premaknjeni ob spremembi števila vozlišč, čim manjše. To težavo zelo dobro rešuje algoritem doslednega razvrščanja, ki ga je kot prvi razvil profesor David Krager [7].

Najpreprosteje si je algoritem predstavljati tako, da vozlišča v gruči razporedimo na krožnico, ki predstavlja vse možne rezultate zgoščevalne funkcije. Izračunano zgoščeno vrednost poiščemo na krožnici ter najdemo najbližje vozlišče v smeri urinega kazalca, kot je prikazano na sliki 2.5. Najdeno vozlišče je izbrano za pomnjenje podatka s tem ključem. Ob umestitvi novega vozlišča v gručo je z uporabo tega algoritma navadno potrebno prestaviti le majhen delež $(\frac{1}{n+1})$ podatkov.

Opisan postopek je zelo posplošena razlaga algoritma, kajti s takšnim razvrščanjem bi nekatera vozlišča lahko postala preobremenjena, druga pa neizkoriščena. Algoritem v praksi krožnico z zgoščenimi vrednostmi razdeli na mnogo particij, kjer vsaka pripada točno določenemu vozlišču. S tem dosežemo, da so podatki zelo enakovredno razvrščeni med vozlišči. S tem se izognemo vročim točkam (veliko pogosto uporabljenih ključev na isti lokaciji) in preobremenjenosti, oziroma neizkoriščenosti, posameznih vozlišč [6].



Slika 2.5: Dosledno razvrščanje.

2.5 Pogosto uporabljeni vmesniki na trgu

Ker je vmesnikov mnogo, so na spodnjem seznamu le nekateri, ki so pogosto uporabljeni v poslovnih aplikacijah.

- Memcached [16]
- Infinispan [14]
- Hazelcast [15]
- Redis [17]
- Oracle Coherence [18]
- Ehcache [21]
- Apache Ignite [19]

- NCache [20]
- AppFabric [22]
- Cacheonix [12]
- Aerospike [23]
- GigaSpaces [24]
- Riak KV [25]

Veliko zgoraj naštetih vmesnikov so odprtokodne rešitve, nekateri se zato aktivno več ne razvijajo, drugi pa iz meseca v mesec dobijo še kakšno novo izvedenko. Ta seznam je torej zelo okviren in se lahko spreminja in dopolnjuje ves čas.

Poglavje 3

Primerjava vmesnikov porazdeljenega predpomnilnika

3.1 Določitev vmesnikov

Ker je vseh vmesnikov občutno preveč za neposredno analizo, se bomo osredotočili na tri, ki so med bolj sodobnimi in pogosto uporabljeni:

- Memcached [16]
- Infinispan [14]
- Hazelcast [15]

3.1.1 Memcached

Memcached je odprtokodna rešitev tipa **odjemalec/strežnik**, razvita v programskem jeziku C. Je eden izmed starejših (prva inačica leta 2003), preprostejših vmesnikov. Namenjen je hranjenju parov ključ-vrednost manjših velikosti [16].

Za delovanje nujno potrebuje namenskega odjemalca, ki skrbi za razvrščanje in dostopanje do podatkov, saj strežniki ne vedo eden za drugega.

Odjemalcu je potrebno ob zagonu podati seznam strežnikov, za porazdeljevanje in dostop do podatkov pa nato skrbi odjemalec. Za porazdeljevanje uporablja algoritem doslednega razvrščanja (razdelek 2.4.3). Memcached ne omogoča dinamičnega dodajanja oziroma odstranjevanja predpomnilnih strežnikov. Ob teh operacijah torej zahteva ponovni zagon odjemalca, kar pa navadno pomeni ponovni zagon aplikacije. Ker je Memcached odprtokodna rešitev, obstajajo na trgu rešitve, ki to pomanjkljivost odpravljajo. Primer je Amazon ElastiCache z možnostjo avtomatičnega odkrivanja Memcached vozlišč v gruči (angl. node auto discovery) [37].

Memcached nasploh velja za zelo stabilno rešitev in je mnogokrat prva izbira za predpomnjenje v poslovnih aplikacijah, kjer so zahteve po velikosti predpomnilnika znane vnaprej.

3.1.2 Infinispan

Infinispan je prav tako odprtokodna rešitev, ki pa podpira oba načina delovanja – vgrajeni in odjemalec/strežnik. Razvit je v programskem jeziku Java. Prva inačica je izšla leta 2009, vendar pa gredo začetki Infinispana vse do leta 2004. Izhaja namreč iz vmesnika **JBoss Cache** [26], ki je bil v osnovi razvit za interne potrebe projekta JBoss Application Server in ga je Infinispan tudi v celoti nadomestil [8].

Vmesnik podpira transakcije, obveščanje o spremembah z dogodki, poizvedbe, porazdeljeno procesiranje in mnogo integracij z drugimi storitvami, ki izkušnjo še izboljšajo. Če želimo Infinispan uporabljati v porazdeljenem načinu, ta za komunikacijo uporablja knjižnico JGroups. Ta skrbi za zanesljivo komunikacijo med vozlišči, za celotno gručo in posamezna vozlišča. Infinispan že vsebuje osnovno konfiguracijo JGroups knjižnice in postavitev gruče je z njim hitra in preprosta. Podpira tudi odstranjevanje in dodajanje vozlišč v gručo brez izgube podatkov ali ponovnega zagona aplikacije, kar pride do izraza predvsem v načinu odjemalec/strežnik.

3.1.3 Hazelcast

Hazelcast je prav tako v programskem jeziku Java razvita odprtokodna rešitev. Podjetje Hazelcast, Inc pa ponuja tudi **poslovno različico** (angl. enterprise edition), ki vsebuje še nekaj dodatnih funkcionalnosti in strokovno podporo pri integraciji [15].

Vsa vozlišča v gruči Hazelcast vozlišč so si enakovredna in med seboj komunicirajo. Samodejo poskrbijo za glavno vozlišče, ki skrbi za porazdeljevanje podatkov in v primeru odpovedi tega vozlišča njegovo mesto nadomesti drugo vozlišče. Dodajanje in odstranjevanje vozlišč je lahko opravljeno popolnoma brez zavedanja aplikacije, ki Hazelcast uporablja za predpomnjenje. Že v osnovni konfiguraciji je nastavljeno podvajanje podatkov, tako da ob odpovedi enega izmed vozlišč ni izgubljen noben podatek. To nastavitev pa je možno nastaviti tako, da lahko odpove tudi več vozlišč hkrati. Ima vse lastnosti, ki jih premore Infinispan, opcijsko pa omogoča tudi trajno hranjenje podatkov.

3.2 Določitev ocenjevalnih parametrov

Izbrani vmesniki si med seboj niso popolnoma enakovredni kar se tiče funkcionalnosti, zato smo morali primerjavo delno prilagoditi. Memcached namreč ne podpira poizvedb in agregacij nad hranjenimi podatki, želeli pa smo testirati tudi to, v poslovnih aplikacijah pogosto zahtevo. V ta namen smo postavili SQL podatkovno bazo, ki služi kot osnovni vir podatkov in je v primeru vmesnika Memcached uporabljena za izvajanje poizvedb in agregacij. Za Hazelcast in Infinispan pa je služila kot osnovni vir podatkov v testih, kjer predpomnilnik ni predhodno napolnjen.

V tej analizi se bomo osredotočili predvsem na zahtevnost vzpostavitve delujočega porazdeljenega predpomnilnika in na branje podatkov.

3.2.1 Branje podatka po ključu

- S **praznim predpomnilnikom** na začetku testa: zgrešena poizvedba je preusmerjena na MySQL podatkovno bazo. Vrnjen podatek je nato shranjen v predpomnilnik, kjer je na voljo za bodoče poizvedbe.
- S **predhodno napolnjenim predpomnilnikom**: poizvedba na podatkovno bazo ni potrebna.
- S **praznim predpomnilnikom** na začetku testa: zgrešena poizvedba je preusmerjena na MySQL podatkovno bazo. Vrnjen podatek je nato shranjen v predpomnilnik, ki pa **nima dovolj kapacitete za vse vrednosti** in mora poskrbeti za izseljevanje starih podatkov.

Merili bomo:

- povprečen čas trajanja zahtevka,
- čas trajanja izvajanja programa za različno število zahtevkov in
- število zadetkov v predpomnilniku.

3.2.2 Poizvedba po atributu

Poizvedbe bodo v primeru vmesnikov Infinispan in Hazelcast izvedene na predhodno napolnjenem predpomnilniku. Pri vmesniku Memcached bo poizvedba opravljena nad podatki v podatkovni bazi MySQL, rezultat pa pod ključem poizvedbe shranjen v predpomnilnik, od koder je vrnjen v morebitnih ponovitvah poizvedbe.

Merili bomo:

- povprečen čas trajanja poizvedbe ter
- čas trajanja izvajanja programa za različno število poizvedb.

3.2.3 Poizvedba po atributu in agregacija

Poizvedbe z agregacijo bodo v primeru vmesnikov Infinispan in Hazelcast izvedene na predhodno napolnjenem predpomnilniku. Pri vmesniku Memcached bo poizvedba z agregacijo opravljena nad podatki v podatkovni bazi MySQL, rezultat pa pod ključem poizvedbe shranjen v predpomnilnik, od koder je vrnjen v morebitnih ponovitvah poizvedbe.

Merili bomo:

- povprečen čas trajanja poizvedbe,
- čas trajanja izvajanja programa za različno število poizvedb.

3.3 Predstavitev simulacije

Vsi testi bodo opravljeni na gruči štirih enakovrednih aplikacijskih strežnikov. Za Infinispan in Hazelcast bomo uporabili v aplikacijski strežnik vgrajen način predpomnjenja. Ker Memcached ne podpira vgrajenega načina, bomo zanj zagnali še štiri Memcached strežnike.

Poizvedbe za vse teste bomo izvajali vzporedno iz štirih niti, vsaka poizvedba pa bo poslana na naključno izbrani strežnik. Čas branja podatka oziroma izvajanja poizvedbe bomo merili neposredno na aplikacijskih strežnikih, da izločimo morebitne zakasnitve do niti v testu, ki je zahtevek poslala.

Vsi testi bodo izvedeni nad različnimi seti enako velikih množic podatkov z različnim številom zahtevkov (tabela 3.1).

Test	Velikosti testnih množic	Število zahtevkov
`Branje po ključu`	50 000, 100 000, 500 000	10 000, 100 000, 1 000 000, 2 000 000
`Branje po ključu z izseljevanjem`	300 000	1 000 000
`Poizvedba po atributu`	50 000, 100 000, 500 000	2 000
`Poizvedba po atributu in agregacija`	50 000, 100 000, 500 000	2 000

Tabela 3.1: Testni parametri.

Poglavje 4

Implementacija

Za potrebe prikaza integracije in prikaza delovanja vmesnikov porazdeljenega predpomnjenja smo razvili preprosto spletno aplikacijo za hranjenje vozil. Omogoča nam dodajanje vozil, poizvedovanje po ključu ter vnaprej definirane poizvedbe in agregacije po nekaterih atributih.

4.1 Razvojno okolje in tehnologije

Aplikacijo smo razvili v programskem jeziku **Java** (verzija 8) [27], programsko kodo pa smo pisali v integriranem razvojnem okolju **IntelliJ IDEA Ultimate** [28] na operacijskem sistemu **Linux** (distribucija Ubuntu 16.10) [29]. Za osnovni vir podatkov smo izbrali zbirko vozil [38], jo razširili in shranili v podatkovno bazo **MySQL** [32].

Programski jezik Java in podatkovno bazo MySQL smo izbrali, ker je ta par pogosto izbran pri načrtovanju poslovnih aplikacij.

Za aplikacijski strežnik, na katerem bo tekla naša aplikacija, smo izbrali odprtokodno rešitev **Apache Tomcat** [30], za pospešen razvoj pa smo uporabili programsko ogrodje **Spring Boot** [31]. Spring Boot s svojimi vtičniki močno olajša predvsem prve korake za postavitev osnovne aplikacijske arhitekture. S programskim ogrodjem Spring smo rešili tudi komunikacijo s podatkovo bazo MySQL in vstavljanje odvisnosti med storitvami v izvorni

kodi (angl. dependency injection).

Za izgradnjo aplikacije in upravljanje z odvisnostmi od tretjih knjižnic smo uporabili programsko ogrodje **Apache Maven** [33]. Ta poskrbi, da imamo odvisnosti do tretjih knjižnic urejene na enem mestu. Uporabljamo ga tudi za izgradnjo aplikacije v arhiv spletne aplikacije (angl. Web application ARchive – WAR), primeren za postavitev na aplikacijski strežnik.

Za nadzor izvorne kode (angl. source control) smo uporabili **Git** [34], pri testiranju pa smo si pomagali s knjižnico **JUnit** [35].

4.2 Podatkovni model

Osnovni podatek za našo analizo je objekt vozilo z naslednjimi atributi:

`id` – enolični identifikator (ključ) vozila

`year` – leto izdelave vozila

`make` – znamka vozila

`model` – model vozila

`mileage` – prevoženi kilometri

Povprečna velikost entitete je približno 90 **bajtov**.

Njegova deklaracija v programskem jeziku Java je prikazana v izvorni kodi 4.1.

Izvorna koda 4.1: Vehicle

```
1  // Vehicle.java
2  public class Vehicle {
3
4  private long id;
5  private int year;
6  private String make;
7  private String model;
```

```
8     private long mileage;
9
10    ...
11  }
```

4.3 Metode za upravljanje s predpomnilnikom

Za upravljanje s predpomnilnikom želimo z vsemi tremi vmesniki podpreti naslednje metode:

- `get(key)`, ki vrne vozilo s ključem `key`.
- `put(key, vehicle)`, ki shrani vozilo `vehicle` v predpomnilnik pod ključem `key`.
- `findByMake(make)`, ki vrne seznam vseh vozil znamke `make`.
- `getMileageByMake(year)`, ki za vsako znamko vrne število prevoženih kilometrov za vozila od vključno letnika `year` dalje. Vrnjen seznam vsebuje objekte tipa `MileageByMake` ki vsebuje model in seštevek prevoženih kilometrov.

4.4 Memcached

4.4.1 Namesitev strežnika

Memcached smo testirali v načinu strežnik/odjemalec, zato smo se najprej posvetili namestitvi in zagonu strežnikov Memcached na našem sistemu. Namestitev strežnika je bila opravljena v nekaj sekundah z zagonom programskega ukaza:

```
$ apt-get install memcached
```

Zagon štirih strežnikov pa smo opravili s štirikratnim klicem naslednjega ukaza:

```
$ memcached -m 512 -p port -d
```

kjer smo `port` napolnili z vrednostnimi 11211, 11212, 11213 in 11214.

4.4.2 Integracija odjemalca

Za komunikacijo s strežniki smo uporabili javansko knjižnico odjemalca **SpyMemcached** [40]. V dokument `pom.xml`, kjer je shranjena konfiguracija projekta, ki jo uporablja ogrodje Maven smo dodali odvisnost (izvorna koda 4.2) na knjižnico, ki je dostopna na javnem repozitoriju projekta Maven.

Izvorna koda 4.2: SpyMemcached

```
<!-- SPYMEMCACHED -->
<dependency>
  <groupId>net.spy</groupId>
  <artifactId>spymemcached</artifactId>
  <version>2.12.3</version>
</dependency>
```

Vzpostavitev osnovnega delovanja smo dosegli že s programsko kodo prikazano v bloku SpyMemcached integracija (izvorna koda 4.3). Dodatno smo implemetirali le našo lastno metodo `getAddresses()`, ki vrača seznam objektov `java.net.InetSocketAddress` z naslovi naših Memcached strežnikov.

Izvorna koda 4.3: SpyMemcached integracija

```
MemcachedClient cache = new MemcachedClient(getAddresses());
Vehicle vehicle = new Vehicle();
// shrani vozilo
cache.add("key", 60, vehicle);
// preberi vozilo
Vehicle cachedVehicle = (Vehicle) cache.get("key");
```

4.4.3 Izvajanje poizvedb in agregacij

Ker Memcached ne podpira izvajanja poizvedb in agregacij neposredno nad podatki v predpomnilniku smo se odločili za drugačen pristop testiranja te funkcionalnosti. Poizvedbe smo opravili z uporabo knjižnice **Spring Data JPA** [36]. Razred `Vehicle.java` smo opremili z anotacijami

```
@Entity
@Table(name = "vehicle")
```

atribut `id` pa z anotacijo

```
@Id
```

iz programskega paketa `javax.persistence`. Za objekt Vehicle se je z razširitvijo vmesnika `JpaRepository` iz programske knjižnice Spring Data JPA pripravil vmesnik za poizvedbo nad vozili v podatkovni bazi MySQL (izvorna koda 4.4).

Izvorna koda 4.4: VehicleRepository

```
1  // VehicleRepository.java
2  public interface VehicleRepository extends JpaRepository<Vehicle,
       Long> {
3
4    List<Vehicle> findByMake(String make);
5
6    @Query(value = "select new com.tomazic.thesis.dto.MileageByMake(v
         .make, sum(v.mileage)) from Vehicle v where v.year >= ?1
         group by v.make")
7    List<MileageByMake> getMileageByMake(int year);
8  }
9
10 // MileageByMake.java
11 public class MileageByMake {
12   private String make;
13   private Long mileage;
```

```
14
15    ...
16  }
```

Novo uvedeni vmesnik `VehicleRepository` nam omogoča, da nad tabelo `vehicle` v podatkovni bazi izvajamo SQL poizvebe, ki jih sicer želimo izvajati nad podatki v predpomnilniku.

V storitev za dostopanje do predpomnilnika pa smo dodali še metode prikazane v izvorni kodi 4.5. Te nam omogočajo dostop do novega vmesnika. Rezultat pod enoličnim ključem poizvedbe shranimo v Memcached. Za enolični ključ pa skrbi metoda `buildQueryKey(String queryKey, String... arguments)`.
S tem smo dosegli, da bomo, v primeru ponovne poizvedbe z enakimi argumenti, rezultat pridobili iz predpomnilnika in ponovna poizvedba na MySQL ne bo potrebna.

Izvorna koda 4.5: Memcached

```
1   public void put(Long key, Vehicle value) {
2     cache.put(key, value);
3   }
4
5   public Vehicle get(Long key) {
6     Vehicle vehicle = (Vehicle) cache.get(String.valueOf(key));
7     if (vehicle == null) {
8       vehicle = repository.getOne(key);
9       if (vehicle != null)
10        add(key, vehicle);
11    }
12    return vehicle;
13  }
14
15  public List<Vehicle> searchByMake(String make) {
16    String key = buildQueryKey("searchByMake", make);
```

```
17    List<Vehicle> vehicles = (List<Vehicle>) cache.get(key);
18    if (vehicles == null) {
19      vehicles = vehicleRepository.findByMake(make);
20      cache.add(key, EXPIRATION_SECONDS, vehicles).get();
21    }
22    return vehicles;
23  }
24
25  public Map<String, Long> sumMileageByMakeFromYear(int year) {
26    String key = buildQueryKey("sumMileageByMakeFromYear", String.
          valueOf(year));
27    Map<String, Long> mileageByMake = (Map<String, Long>) cache.get(
          key);
28    if (mileageByMake == null) {
29      mileageByMake = vehicleRepository.getMileageByMake(year).stream
            ().collect(Collectors.toMap(MileageByMake::getMake,
            MileageByMake::getMileage));
30      cache.add(key, EXPIRATION_SECONDS, mileageByMake).get();
31    }
32    return mileageByMake;
33  }
```

4.5 Infinispan

4.5.1 Vzpostavljanje vgrajenega predpomnilnika

Vmesnik Infinispan smo implementirali v vgrajenem načinu, saj je ta način navadno prva izbira, hkrati pa potrebuje manj konfiguracije za vzpostavitev osnovnega delovanja.

Za delovanje se moramo najprej povezati v gručo, za kar smo poskrbeli z upraviteljem vgrajenega predpomnilnika ECM (angl. Embedded Cache Manager). Prikazano v izvorni kodi 4.6.

Izvorna koda 4.6: Upravitelj vgrajenega predpomnilnika

```
1  EmbeddedCacheManager cacheManager = new DefaultCacheManager(
2    GlobalConfigurationBuilder.defaultClusteredBuilder()
3      .transport().nodeName(UUID.randomUUID().toString())
4      .addProperty("configurationFile", "udp.xml")
5      .addProperty("statistics-available", "true")
6      .globalJmxStatistics().enable()
7      .build()
8  );
```

Za konfiguracijo gruče s programskim ogrodjem JGroups smo izbrali privzete nastavitve za nepovezovalni protokol UDP (angl. User Data Protocol). Za potrebe testa nam ta protokol ustreza, v produkcijskem okolju pa bi si želeli uporabiti transportni sloj TCP (angl. Transmission Control Protocol). To smo zagotovili z nastavitvijo
`addProperty("configurationFile", "udp.xml")`.
Dokument `udp.xml` z osnovnimi nastavitvami je že vključen v knjižnico JGroups in ga ni potrebno ročno dodajati, če ga ne želimo spreminjati.

Vozlišča JGroups gruče smo poimenovali naključno z `java.util.UUID` naključnim generatorjem. Z zadnjima dvema nastavitvama smo vklopili še statistiko nad podatki. Ta je privzeto izključena, saj vpliva na zmogljivost celotnega sistema. Statistiko želimo imeti vklopljeno v začetni fazi razvoja aplikacije, da si z rezultati pomagamo k boljšim nastavitvam gruče in porazdeljenega predpomnilnika. Ko smo z nastavitvami zadovoljni, lahko statistiko izklopimo, da pridobimo še nekaj na zmogljivosti.
Sledi nastavitev predpomnilnika:

Izvorna koda 4.7: Nastavitev predpomnilnika

```
1  cacheManager.defineConfiguration("vehicleCache",
2    new ConfigurationBuilder()
3      .memory().storageType(StorageType.BINARY)
4        .evictionType(EvictionType.MEMORY).size(2048)
5      .expiration().lifespan(-1)
```

```
6       .indexing()
7         .index(Index.PRIMARY_OWNER)
8         .addIndexedEntity(Vehicle.class)
9         .setProperty("default.directory_provider", "ram")
10        .setProperty("default.indexmanager",
11          "org.infinispan.query.indexmanager.InfinispanIndexManager")
12        .addProperty("lucene_version", "LUCENE_71")
13        .addProperty(
14          "hibernate.search.default.indexwriter.ram_buffer_size",
15          "512")
16      .jmxStatistics().enable()
17      .clustering().cacheMode(CacheMode.DIST_ASYNC)
18        .hash().numOwners(1)
19      .build()
20  );
```

Opis nastavitev:

- `"vehicleCache"`: prvi argument nastavitve `defineConfiguration` predstavlja ime predpomnilnika.
- `memory()`: nastavimo tip hranjenja (binarni) in da podatke začnemo izseljevati iz predpomnilnika, ko ta doseže velikost 2048MB. Tu bi lahko alternativno nastavili tudi število elementov namesto velikosti predpomnilnika.
- `expiration().lifespan(-1)`: nastavimo lastnost, da podatki nikoli ne zastarajo in jih hranimo v predpomnilniku, dokler niso odstranjeni iz drugega razloga (npr. pomanjkanje prostora).
- `indexing()`: nastavljamo različne parametre indeksiranja kot so: katere podatke indeksiramo, kje hranimo indekse, kako jih upravljamo, ipd.
- `jmxStatistics().enable()`: vklopimo statistiko.

- `clustering().cacheMode(CacheMode.DIST_ASYNC)`: nastavimo delovanje predpomnilnika v porazdeljenem načinu.
- `hash().numOwners(1)`: nastavimo število kopij vsakega podatka v gruči.

Za pravilno indeksiranje moramo z anotacijo opremiti atribute razreda `Vehicle.java`, po katerih želimo indeksirati. Ta je:

```
@Field(store = Store.YES, analyze = Analyze.NO)
```

iz knjižnice `org.hibernate.search.annotations`.

4.5.2 Izvajanje poizvedb in agregacij

Upravljanje s podatki v predpomnilniku s pomočjo vmesnika Infinispan je po uspešni konifiguraciji zelo preprosto. S pomočjo ECM pridobimo željeni predpomnilnik ter iz njega dva objekta za pomoč pri iskanju in sestavljanju poizvedb.

```
1    Cache<Long, Vehicle> cache = cacheManager.getCache("vehicleCache"
        );
2    SearchManager searchManager = Search.getSearchManager(cache);
3    QueryFactory queryFactory = Search.getQueryFactory(cache);
```

S pridobljenimi objekti nato izvajamo zahtevke, poizvedbe in agregacije v predpomnilniku, kot je prikazano v izvorni kodi 4.8.

Izvorna koda 4.8: Infinispan

```
1  public void put(Long key, Vehicle value) {
2    cache.put(key, value);
3  }
4
5  public Vehicle get(Long key) {
6    return cache.get(key);
7  }
8
```

```
9   public List<Vehicle> searchByMake(String make) {
10    org.apache.lucene.search.Query query = searchManager.
          buildQueryBuilderForClass(Vehicle.class).get()
11        .keyword().onField("make")
12        .matching(make)
13        .createQuery();
14    CacheQuery<Vehicle> makeQuery = searchManager.getClusteredQuery(
          query, Vehicle.class);
15    return makeQuery.list();
16  }
17
18  public Map<String, Long> sumMileageByMakeFromYear(int year) {
19    org.infinispan.query.dsl.Query q = queryFactory.from(Vehicle.
          class)
20        .having("year").gte(year)
21        .select(Expression.property("make"), Expression.sum("mileage"
              ))
22        .groupBy("make")
23        .build();
24    List<Object[]> results = q.list();
25    Map<String, Long> mileageByMake = new HashMap<>();
26    results.forEach(o -> mileageByMake.put(String.valueOf(o[0]), Long
          .valueOf(String.valueOf(o[1]))));
27    return mileageByMake;
28  }
```

4.6 Hazelcast

4.6.1 Vzpostavljanje vgrajenega predpomnilnika

Za implementacijo vmesnika Hazelcast smo prav tako izbrali vgrajeni način predpomnilnika. Za osnovno delovanje predpomnilnika smo uporabili nasle-

dnje nastavitve:

Izvorna koda 4.9: Hazelcast nastavitve

```
1  Config cfg = new Config();
2  MapConfig mapConfig = cfg.getMapConfig("vehicleCache");
3  mapConfig.setStatisticsEnabled(true);
4  mapConfig.setEvictionPolicy(EvictionPolicy.LRU);
5  mapConfig.getMaxSizeConfig().setMaxSizePolicy(PER_NODE)
6    .setSize(1500000);
7  mapConfig.addMapIndexConfig(new MapIndexConfig("make", false));
8  mapConfig.addMapIndexConfig(new MapIndexConfig("year", false));
9  HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
```

Vse nastavitve so pri vmesniku Hazelcast zelo jasne in ne potrebujejo dodatnih pojasnil. Iz konfiguracije lahko razberemo, da je predpomnilnik poimenovan `vehicleCache` ter da ima vklopljeno beleženje statistike. Podatke bo izseljeval po algoritmu najdlje neuporabljenega podatka LRU (angl. least recently used), ko bo število podatkov v vozlišču preseglo 1 500 000 objektov. Z zadnjima dvema nastavitvama pa določimo še indeksiranje po atributih `make` in `year`.

Za pridobitev instance predpomnilnika preprosto izvedemo naslednji klic nad Hazelcast instanco:

```
IMap<Long, Vehicle> cache = instance.getMap(VEHICLE_CACHE);
```

4.6.2 Izvajanje poizvedb in agregacij

Način za pridobitev in zapis podatka s ključem se od Infinispana bistveno ne razlikuje, kar vidimo v izvorni kodi 4.10. Opazimo pa, da poizvedba po znamki vozila in agregacija kilometrov po znamki zahtevata dodatne objekte.

Implementacije razredov teh objektov predstavljajo:

MakePredicate: pravilo, ki preverja ujemanje atributa `make` razreda

`Vozilo` z vrednostjo argumenta `make`, podanega v konstruktorju razreda `MakePredicate`.

FromYearPredicate: pravilo, ki preverja ujemanje atributa `year` razreda `Vozilo` z vrednostjo argumenta `year`, podanega v konstruktorju razreda `FromYearPredicate`.

MakeMileageAggregator: agregat, ki sešteva vrednosti atributov `mileage` razreda `Vozilo` in vodi evidenco vsot za vsako različno vrednost atributa `make`.

Za implementacijo teh razredov je bilo sicer potrebne nekoliko več programske kode in nekaj več logike za agregat. Pomagali smo si z navodili iz uradne spletne strani vmesnika Hazelcast in z implementacijo nismo imeli nikakršnih težav. Implementacija je na voljo v izvorni kodi 4.11.

Izvorna koda 4.10: Hazelcast

```
1  public void put(Long key, Vehicle value) {
2    cache.set(key, value);
3  }
4
5  public Vehicle get(Long key) {
6    return cache.get(key);
7  }
8
9  public List<Vehicle> searchByMake(String make) {
10   Collection<Vehicle> values = cache.values(new MakePredicate(make)
       );
11   return new ArrayList<>(values);
12 }
13
14 public Map<String, Long> sumMileageByMakeFromYear(int year) {
15   Predicate<Long, Vehicle> fromYearPredicate = new
       FromYearPredicate(year);
```

```
16    return cache.aggregate(new MakeMileageAggregator(),
          fromYearPredicate);
17  }
```

Izvorna koda 4.11: Hazelcast pravila in agregat

```
1   private static class MakePredicate implements Predicate<Long,
        Vehicle> {
2     private String make;
3     MakePredicate(String make) {
4       this.make = make;
5     }
6     @Override
7     public boolean apply(Map.Entry<Long, Vehicle> mapEntry) {
8       return make.equals(mapEntry.getValue().getMake());
9     }
10  }
11
12  private static class FromYearPredicate implements Predicate<Long,
        Vehicle> {
13    private int year;
14    FromYearPredicate(int year) {
15      this.year = year;
16    }
17    @Override
18    public boolean apply(Map.Entry<Long, Vehicle> mapEntry) {
19      return mapEntry.getValue().getYear() >= year;
20    }
21  }
22
23  public static class MakeMileageAggregator extends Aggregator<Map.
        Entry<Long, Vehicle>, Map<String, Long>> {
24    Map<String, Long> mileageByMake = new HashMap<>();
25    @Override
```

```
26    public void accumulate(Map.Entry<Long, Vehicle> input) {
27      String make = input.getValue().getMake();
28      Long mileage = mileageByMake.get(make);
29      if (mileage == null) {
30        mileage = 0L;
31      }
32      mileage += input.getValue().getMileage();
33      mileageByMake.put(make, mileage);
34    }
35    @Override
36    public void combine(Aggregator aggregator) {
37      MakeMileageAggregator aggr = (MakeMileageAggregator) aggregator
            ;
38      for (Map.Entry<String, Long> toCombine : aggr.mileageByMake.
            entrySet()) {
39        doCombine(toCombine);
40      }
41    }
42    private void doCombine(Map.Entry<String, Long> toCombine) {
43      String make = toCombine.getKey();
44      Long mileage = mileageByMake.get(make);
45      if (mileage == null) {
46        mileage = 0L;
47      }
48      mileage += toCombine.getValue();
49      mileageByMake.put(make, mileage);
50    }
51    @Override
52    public Map<String, Long> aggregate() {
53      return mileageByMake;
54    }
55  }
```

V tem poglavju navedeni izseki izvorne kode ne zadostujejo za pravilno delovanje aplikacije, ampak prikazujejo le bistvene dele, pomembne za uspešno integracijo posameznih vmesnikov.

Poglavje 5

Rezultati

5.1 Branje podatka po naključnem ključu

V testu branja po naključnem ključu smo iz štirih niti vzporedno izvajali zahtevke na aplikacijske strežnike, kjer teče naša aplikacija. Za vsak zahtevek je bil naključno generiran veljaven ključ, pod katerim zagotovo obstaja entiteta vozila v porazdeljenem predpomnilniku (v nadaljevanju predpomnilnik) oziroma podatkovni bazi. Aplikacija poskusi pridobiti entiteto vozila po danem ključu iz predpomnilnika. Če ga ne najde, preusmeri zahtevek na podatkovno bazo MySQL (v nadaljevanju MySQL). Pridobljen podatek shrani v predpomnilnik in vrne entiteto.

5.1.1 Prazen predpomnilnik

Najprej smo teste izvedli na praznem predpomnilniku, kjer so bili vsi zahtevki preusmerjeni na podatkovno bazo. Vsak ponovljeni zahtevek za isti ključ pa je podatek vrnil iz predpomnilnika, če je bil ta že uspešno sinhroniziran med vozlišči predpomnilnika.

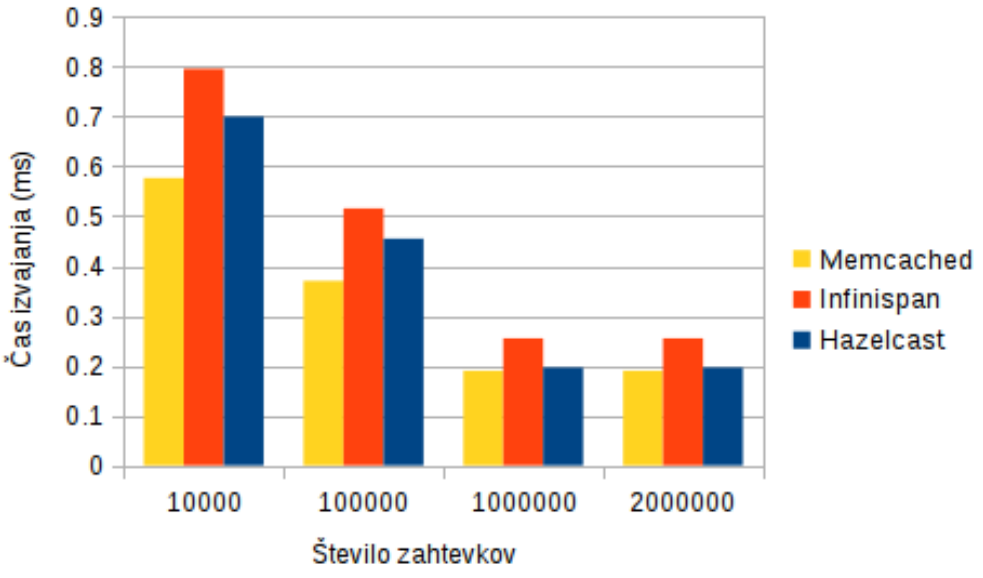
Parametri testa:

- Testna množica: 100 000 entitet
- Število zahtevkov: 10 000, 100 000, 1 000 000, 2 000 000

- Število aplikacijskih vozlišč: 4
- Število predpomnilnih strežnikov (Memcached): 4

Na sliki 5.1 je prikazan povprečen čas, porabljen za pridobitev zahtevka iz predpomnilnika oziroma MySQL. Opazimo lahko, da je čas, potreben za pridobitev podatka z Memcached vmesnikom, za $\approx 20\%$ boljši od preostalih dveh. Povprečje je nižje predvsem na račun testov z manjšim številom zahtevkov (10 000 in 100 000). Do razlike v času pride zaradi načina delovanja Memcached odjemalcev in strežnikov. Pri Memcached namreč teče SpyMemcached odjemalec na vsakem izmed vozlišč naše aplikacije. Vsak odjemalec ve, da so Memcached strežniki natančno štirje in zaradi uporabe dosledne zgoščevalne funkcije ve tudi na katerem izmed strežnikov v predpomnilniku se ta podatek nahaja, če je vanj že bil vstavljen. Odjemalci in strežniki med seboj ne komunicirajo in s sinhronizacijo ne izgubljajo časa.

Ko so vsi podatki v predpomnilniku (število zahtevkov višje od 100 000), pa se povprečen čas branja giba okoli $\approx 0,2$ milisekunde.

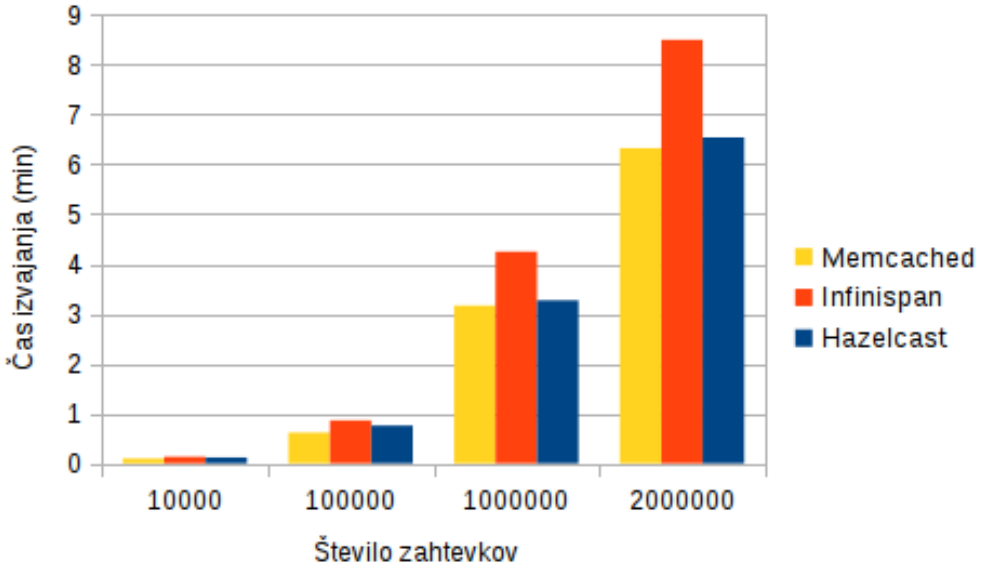


Slika 5.1: Povprečen čas trajanja zahtevka pri praznem predpomnilniku.

Razvidno je tudi, da je razlika najbolj očitna pri majhen številu zahtevkov. Naša testna množica namreč šteje 100 000 entitet. Ko so vse entitete vnešene v predpomnilnik, so časi pri ostalih dveh predpomnilnikih popolnoma konkurenčni. Upoštevati moramo še, da je za vsak povzetek pri Memcached nujno potrebna poizvedba na enega izmed strežnikov zaradi načina odjemalec/strežnik, v katerem deluje. V testu so se namreč vsa vozlišča in strežniki nahajali na istem sistemu in so dostopni časi med njimi skoraj zanemarljivi. Pri ostalih dveh pa uporabljamo vgrajeni način predpomnilnika in je statistično $\frac{1}{4}$ zahtevanih podatkov že na vozlišču, ki obdeluje zahtevek. V primeru počasne povezave med odjemalci, strežniki in vozlišči ali pa v primeru velikega števila poizvedb v primerjavi s številom podatkov z različnimi ključi, bi se ostala dva vmesnika odrezala bistveno bolje.

Pridobljeni rezultat torej še ne pomeni, da je Memcached najboljši izmed treh. Pridobljeni čas ima tudi svojo ceno in je lahko celo zanemarljiv v primeru večjega števila poizvedb v primerjavi s številom podatkov v testni množici. V primeru, da nam začne primanjkovati predpomnilnika, bomo s horizontalno razširitvijo predpomnilnika – dodajanjem novih Memcached strežnikov – prisiljeni ponovno nastaviti vse odjemalce, kar pomeni ponovni zagon vseh vozlišč naše aplikacije. Tu sta ponovno v ogromni prednosti ostala dva vmesnika, ki ponujata dinamično horizontalno razširljivost.

Infinispan in Hazelcast sta se odrezala zelo primerljivo. Hazelcast je nekoliko hitrejši že od samega začetka, razlika pa se z večanjem števila zahtevkov manjša. To nakazuje, da Infinispan potrebuje nekoliko več časa za izvedbo začetnih poizvedb.



Slika 5.2: Skupen čas izvajanja programa pri praznem predpomnilniku.

Na sliki 5.2 si lahko ogledamo vpliv trajanja povprečne poizvedbe na celoten čas, ki je bil potreben za izvedbo testa. Če aplikacijo uporablja mnogo različnih uporabnikov, razlika nekaj milisekund v dostopnem času ne predstavlja bistvene prednosti ali težave. Če pa moramo prebrati ogromno podatkov, pa lahko majhna razlika pomeni veliko.

Iz pridobljenih rezultatov lahko povzamemo naslednje ugotovitve:

- Nizek povprečen čas pri majhnem številu zahtevkov nakazuje, da Memcached porabi najmanj časa, da shrani podatek, ki je takoj za tem na voljo vsem odjemalcem.
- Prednost vmesnika Memcached v hitrosti je zanemarljiva pri majhnem številu podatkov, ki bodo velikokrat ponovno brani.
- Infinispan in Hazelcast ob zapisu podatka ustvarita več prometa v omrežju, pri branju pa manj. Pri zapisu morata o podatku obvestiti preostala vozlišča, pri branju pa ne zahtevata dodatne poti do predpomnilnih strežnikov, saj delujeta v vgrajenem načinu.

Test z izseljevanjem

V tem testu smo testirali, kako se obnesejo vmesniki, ko imajo opravka z velikim številom vpisov novih podatkov v predpomnilnik, ko je ta že poln.

Parametri testa:

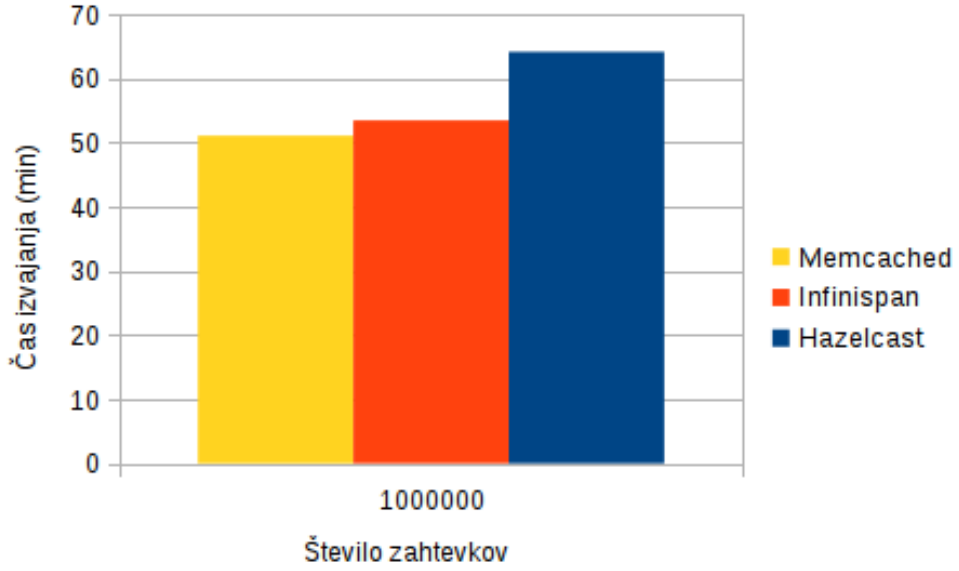
- Testna množica: 300 000 entitet
- Prostor v predpomnilniku: za $\approx$ 150 000 entitet
- Število zahtevkov: 1 000 000
- Število aplikacijskih vozlišč: 4
- Število predpomnilnih strežnikov (Memcached): 4

Kadar predpomnilnik služi za zmanjšanje prometa na podatkovni bazi, ima ta navadno manj prostora kot podatkovna baza. V takšnih scenarijih je pogosto veliko zahtevkov po istem podatku, le malokrat pa se v krajšem časovnem obdobju potrebujejo vsi podatki, ki so v podatkovni bazi.

Predstavljajmo si, da imamo spletno aplikacijo, katera ima registriranih milijon uporabnikov. Ko se uporabnik prijavi, njegovo entiteto v podatkovni bazi preberemo in jo zapišemo v predpomnilnik. Načeloma bo tam ostala vsaj do konca njegove seje in kakršne koli spremembe ali dostopi do podatkov na njej bodo hitri. Ker je zelo majhna verjetnost, da bo vseh milijon registriranih uporabnikov sočasno prijavljenih v aplikacijo, je dovolj, da ima predpomnilnik prostora za nekaj več entitet, kot je pričakovano največje število sočasno prijavljenih uporabnikov.

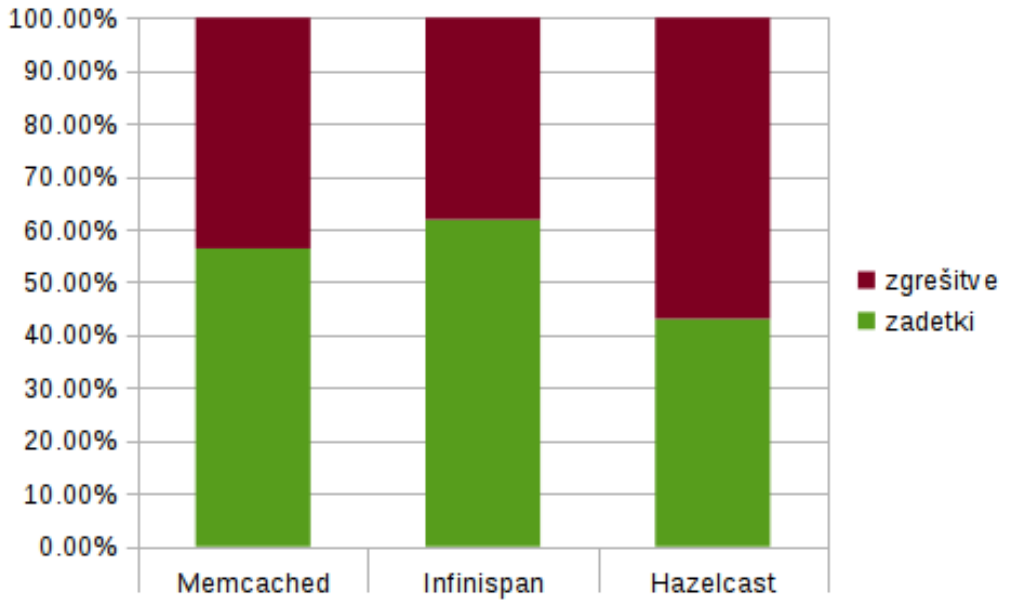
V tem testu smo konfiguracijo namenoma nastavili neoptimalno. Prostora v predpomnilniku imamo le za polovico naših entitet, zahtevki pa bodo prihajali naključno za vse entitete. S tem bomo dosegli, da bo ogromno entitet izseljenih iz predpomnilnika in je tako problem množičnega izseljevanja bolj očiten. Povprečen čas zahtevka je v povprečju trajal $\approx 1,125$ milisekunde pri vseh vmesnikih, kar je skoraj deset-kratnik povprečja iz prvega testa. Na sliki 5.3 vidimo, kakšen je učinek takšne zakasnitve. Skupen čas

izvajanja programa je občutno višji. Tako visok čas branja pa že vpliva tudi na uporabniško izkušnjo uporabnika.



Slika 5.3: Čas izvajanja programa pri praznem predpomnilniku.

Na sliki 5.4 je v odstotkih predstavljeno razmerje zadetkov z zgrešitvami v predpomnilniku. Pri vseh treh vmesnikih je bilo zgrešitev v povprečju več kot $\frac{1}{3}$. Podoben rezultat je pričakovan, saj vsi za razvrščanje podatkov uporabljajo algoritem doslednega razvrščanja (razdelek 2.4.2). Nekoliko večje odstopanje je sicer zaznati pri vmesniku Hazelcast, vendar to lahko pripisujemo tudi različnim konfiguracijam. Velikost predpomnilnika je namreč skoraj nemogoče nastaviti popolnoma enako za vse vmesnike.

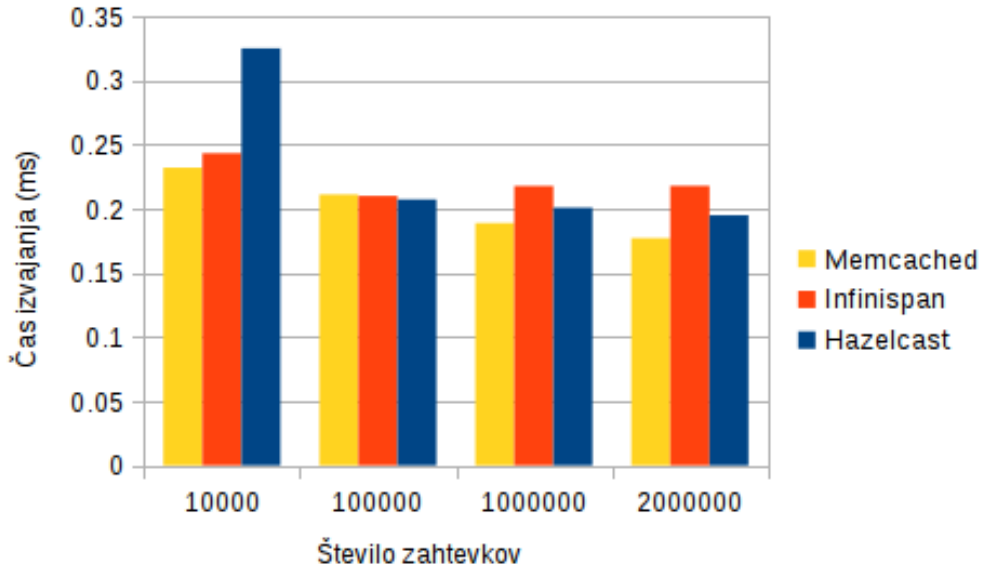


Slika 5.4: Primerjava števila zadetkov z zgrešitvami.

Iz tega testa je jasno, da je izbira velikosti predpomnilnika zelo pomembna ne glede na izbiro vmesnika. Če bi se zgornji scenarij zgodil v vsakodnevno uporabljeni aplikaciji, pa bi si vseeno želeli uporabljati vmesnik Infinispan ali Hazelcast. S tema dvema bi namreč brez večjih težav dodali dodatna vozlišča v gručo in s tem povečali predpomnilnik.

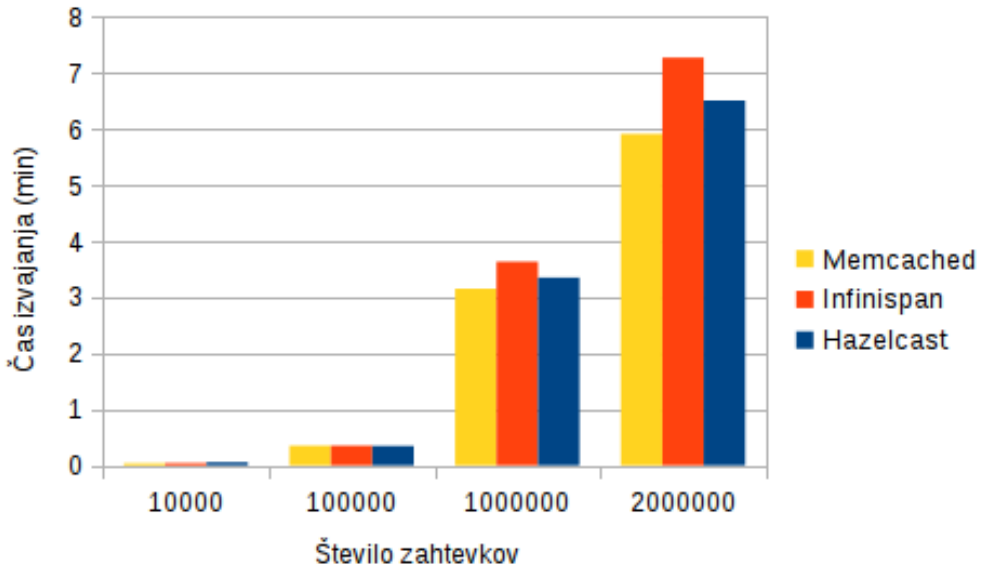
5.1.2 Predhodno napolnjen predpomnilnik

Ponovili smo prvi test branja 5.1.1, le da smo tokrat podatke pred zagonom testa že vstavili v predpomnilnik. S tem smo dosegli, da bo podatek za vsak zahtevan ključ že dostopen v predpomnilniku in branje iz MzSQL ne bo potrebno. Iz slike 5.5 vidimo, da je povprečen čas branja že pri majhni množici podatkov zelo hitrer. Vmesnik Hazelcast je imel nekoliko višji čas pri majhnem številu zahtevkov, z večanjem ponovitev branj pa tega odstopanja ni več opaziti.



Slika 5.5: Povprečen čas zahtevka iz predhodno napolnjenega predpomnilnika.

Tudi na sliki 5.6, kjer je prikazan skupen čas izvajanja programa pri predhodno napolnjenem predpomnilniku vidimo, da večjih razlik ni, se je pa vseeno Memcached odrezal najbolje.



Slika 5.6: Čas izvajanja programa pri predhodno napolnjenem predpomnilniku.

Iz testa lahko povzamemo, da občutnejših razlik med vmesniki pri branju predhodno napolnjenega predpomnilnika ni. Torej je izbira spet odvisna od drugih zahtev, ki jih naša aplikacija potrebuje. Vzemimo za primer aplikacijo, kjer so vsi podatki v predpomnilniku brez hrambe na trajnem pomnilniku. Tu bi si želeli, da podatkov ne izgubimo, če eno izmed vozlišč predpomnilnika odpove. Zato bi v takšnem primeru izbrali Infinispan ali Hazelcast, katera že v osnovi ponujata podvajanje podatkov (angl. replication). S podvajanjem namreč dosežemo, da je vsak podatek hkrati shranjen na več vozliščih in v primeru odpovedi enega izmed njih, podatek ni izgubljen.

5.2 Poizvedba

Porazdeljeni predpomnilniki so v osnovi pomnilniki tipa ključ-vrednost (angl. key-value store). To pomeni, da podatke načeloma pridobivamo na podlagi vnaprej znanega ključa, pod katerim je shranjen. V praksi pa se večkrat

pojavi potreba po izvajanju poizvedb nad podatki. Nekateri vmesniki zato podpirajo tudi poizvedbe. Izmed primerjanih sta to Infinispan in Hazelcast. Memcached smo uporabili kot alternativo, saj bomo poizvedbe izvajali nad podatkovno bazo, rezultat poizvedbe pa pod generiranim ključem shranili v predpomnilnik. V primeru enake poizvedbe bo podatek vrnjen iz predpomnilnika.

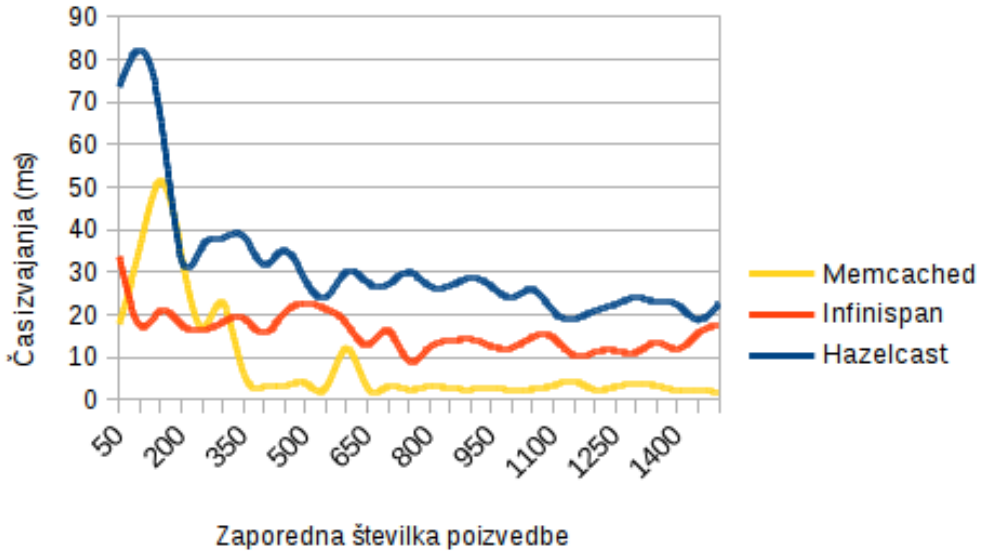
Problem, s katerim se srečamo pri Memcached je tudi omejena največja velikost podatka - $1MB$. Vmesnik dovoljuje, da to omejitev dvignemo, a nas pri tem opozori, da je učinkovitost delovanja lahko zmanjšana. V tem testu smo to omejitev vseeno dvignili na $10MB$ in težav ni bilo opaziti. To je sicer resna omejitev, kajti to pomeni, da bodo določene poizvedbe vedno posredovane na podatkovno bazo. Zelo pomembno je tudi dejstvo, da nimamo avtomatične razveljavitve veljavnosti hranjenega rezultata. Če pričakujemo, da se podatki spreminjajo, bi morali sami poskrbeti za razveljavljanje. Lahko sicer nastavimo krajši čas veljavnosti vrednosti hranjene poizvedbe, vendar se moramo zavedati, da so podatki v predpomnilniku lahko stari, hkrati pa bomo s tem dvignili število poizvedb, preusmerjenih na podatkovno bazo.

Parametri testa:

- Testne množice: 50 000, 100 000, 500 000 entitet
- Prostor v predpomnilniku: dovolj za hranjenje vseh podatkov oziroma rezultatov poizvedb pri Memcached
- Število zahtevkov: 1 500
- Število aplikacijskih vozlišč: 4
- Število predpomnilnih strežnikov (Memcached): 4
- Poizvedba: vrni vsa vozila znamke *znamka*, kjer je *znamka* naključno izbrana izmed 144 znamk vozil. Za vsako znamko se v podatkovni bazi nahaja vsaj eno vozilo.
- Število različnih znamk: 144

Poizvedba nad testno množico 50 000 entitet

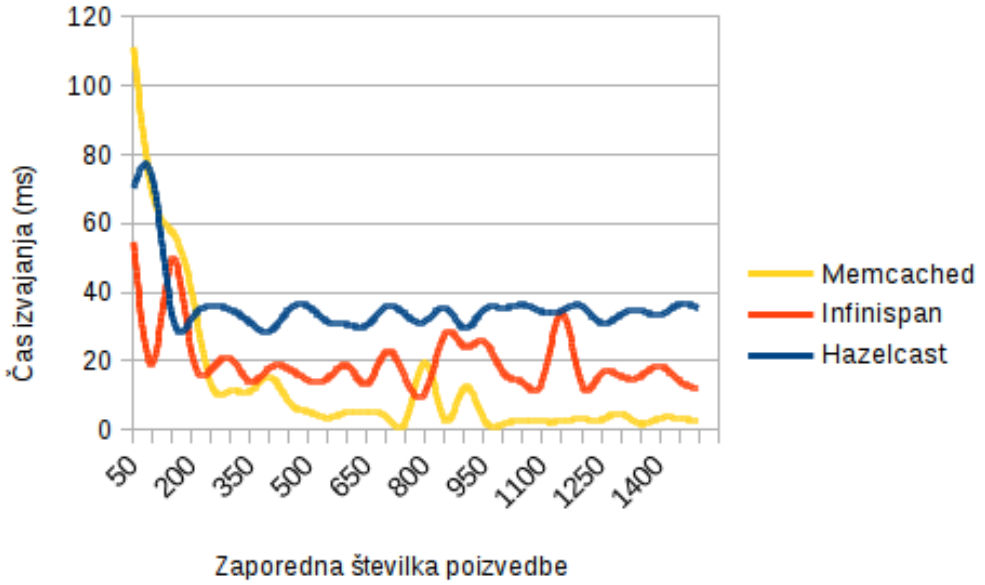
Na sliki 5.7 so prikazani časi izvajanja poizvedb na množici 50 000 vozil. Vsi vmesniki so nove poizvedbe izvajali nekoliko dlje kot ponavljajoče. To je razvidno predvsem pri prvih tristo poizvedbah. Za vmesnik Memcached je bila takšna krivulja pričakovana, saj je bilo potrebno ob prvih novih poizvedbah dostopati do MySQL. Ko pa so se poizvedbe začele ponavljati, je Memcached zelo dobro opravil nalogo. Najslabše rezulate je dosegel Hazelcast, vendar so časi še vedno relativno nizki.



Slika 5.7: Čas izvajanja zaporednih poizvedb (50 000 entitet).

Poizvedba nad testno množico 100 000 entitet

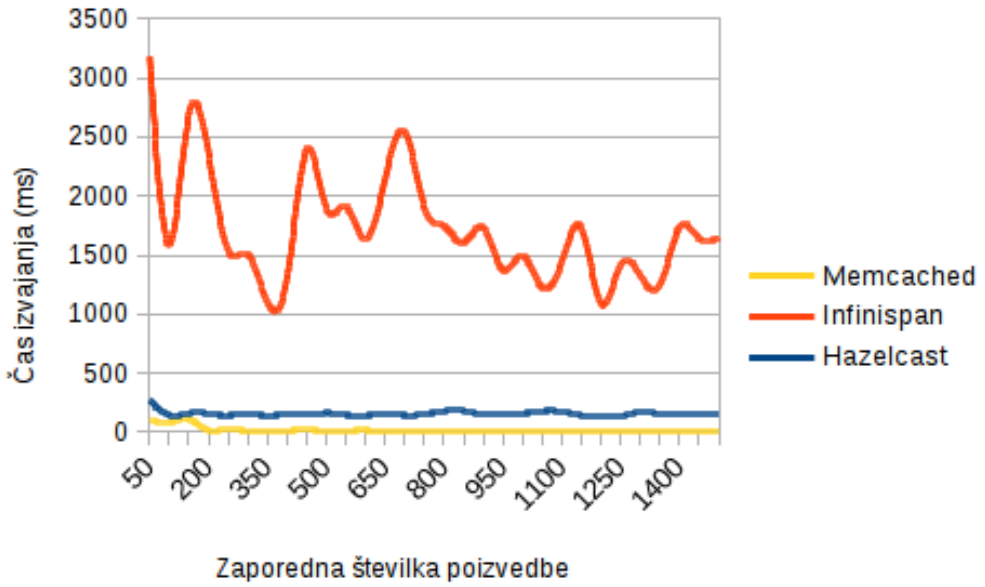
Enak test smo izvedli še na množici 100 000 vozil. Rezultati so prikazani na grafu 5.8. Začetne poizvedbe so trajale nekoliko dlje, rezultati pa se po nekaj poizvedbah niso več bistveno razlikovali od testa s 50 000 entitetami.



Slika 5.8: Čas izvajanja zaporednih poizvedb (100 000 entitet).

Poizvedba nad testno množico 500 000 entitet

Nazadnje smo izvedli še test, kjer smo testno množico povečali na 500 000 vozil. Tu so se z vmesnikom Infinispan pojavile težave z vnosom entitet. Indeksiranje entitet je zahtevalo ogromno komunikacije med vozlišči in bilo nasploh zelo počasno. Pri izvajanju testov se je pokazalo, da je Infinispan, zaradi velikega števila indeksov, postal praktično neuporaben. Ostala dva vmesnika pa sta dosegla le malo slabše rezultate kot pri testih z manjšo testno množico. Dopuščamo možnost, da bi bilo s kakšno nastavitvijo moč težave z vmesnikom Infinispan odpraviti. Sklepamo, da bi se odražala v manjši meri, če bi vsako vozlišče bilo postavljeno na fizično ločen in zmogljivejši strežnik. Sinhronizacija med vozlišči namreč zahteva veliko komunikacije. Ker smo vsa vozlišča gostili na enem fizičnem računalniku, je ta komunikacija potekala skozi navidezni omrežni vmesnik, ki bi lahko predstavljal ozko grlo.



Slika 5.9: Čas izvajanja zaporednih poizvedb (500 000 entitet).

Iz testa lahko povzamemo, da se je Memcached v paru z MySQL odrezal odlično. A se ob tem moramo zavedati, da takšna konfiguracija zahteva veliko večji predpomnilnik in seveda podatkovno bazo. Presenetil nas je Hazelcast, ki je za kofiguracijo zahteval zelo malo vloženega časa, rezultati pa so bili zelo dobri in konstanti tudi z večanjem testne množice.

5.3 Agregacija poizvedbe

Poleg klasične poizvedbe smo opravili še test agregacije podatkov nad poizvedbo. Memcached bo tudi v tem primeru uporabljen v kombinaciji s podatkovno bazo MySQL, enako kot pri testiranju poizvedbe 5.2. Vse naštete omejitve in posledice veljajo tudi za agregacije.

Parametri testa:

- Testne množice: 50 000, 100 000, 500 000 entitet

- Prostor v predpomnilniku: dovolj za hranjenje vseh podatkov oziroma rezultatov agregacij pri Memcached
- Število zahtevkov: 1 500
- Število aplikacijskih vozlišč: 4
- Število predpomnilnih strežnikov (Memcached): 4
- Agregacija: za vsa vozila, izdelana od leta `leto` (vključno), seštej število prevoženih kilometrov in jih združi po znamki. Parameter `leto` je za vsako poizvedbo naključno izbrana letnica med najstarejšim in najnovejšim (vključno) vozilom v množici podatkov.
- Število različnih znamk: 144

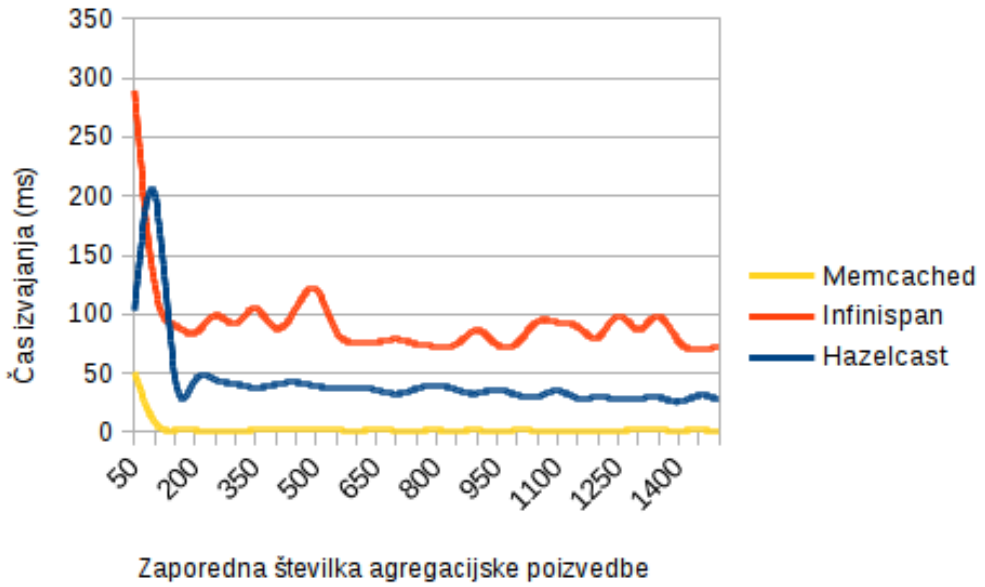
Agregacija nad testno množico 50 000 entitet

Slika 5.10 prikazuje agregacije nad 50 000 entitetami vozil. Opazimo lahko, da se Memcached obnaša približno enako kot pri testiranju poizvedbe 5.2. Večja razlika je opazna med ostalima vmesnikoma. Povprečna agregacija z vmesnikom Hazelcast je trajala za faktor ≈ 1.7 dlje kot pri poizvedbi. Infinispan pa je potreboval kar za faktor ≈ 6.2 več časa za agregacijo kot za poizvedbo iz prejšnjega testa.

V tabeli 5.1 pa se nahajajo povprečni časi agregacij, saj je na grafu čas poizvedb pri Memcached tako nizek, da se natančnejša vrednost iz grafa težko razbere.

vmesnik	povprečen čas agregacije (ms)
`Memcached`	$3,3046666667$
`Infinispan`	$94,3086666667$
`Hazelcast`	$42,9273333333$

Tabela 5.1: Povprečen čas agregacije (50 000 entitet).



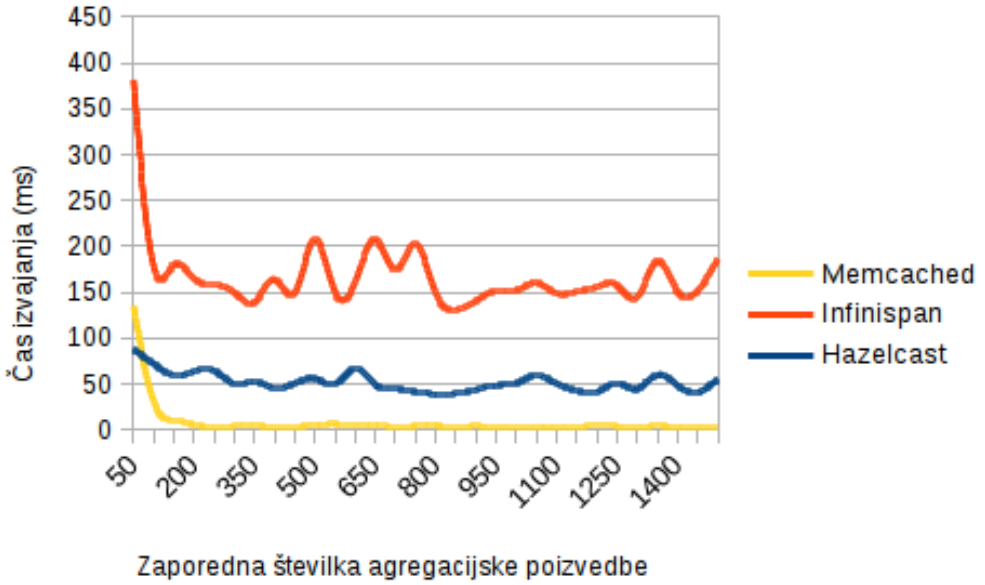
Slika 5.10: Čas izvajanja zaporednih agregacijskih poizvedb (50 000 entitet).

Agregacija nad testno množico 1000 000 entitet

S povečanjem testne množice na 100 000 entitet vozil (slika 5.11) so se pri Memcached vidno povečali časi prvih poizvedb. To je močno vplivalo tudi na skupni povprečni čas (tabela 5.2), ki se je dvignil za faktor $2,86$. Povprečni čas poizvedbe z vmesnikom Infinispan se je dvignil za faktor $\approx 1,8$, Hazelcast pa le za $\approx 1,2$.

vmesnik	povprečen čas agregacije (ms)
`Memcached`	$9,4586666667$
`Infinispan`	$169,5613333333$
`Hazelcast`	$52,602$

Tabela 5.2: Povprečen čas agregacije (100 000 entitet).



Slika 5.11: Čas izvajanja zaporednih agregacijskih poizvedb (100 000 entitet).

Agregacija nad testno množico 500 000 entitet

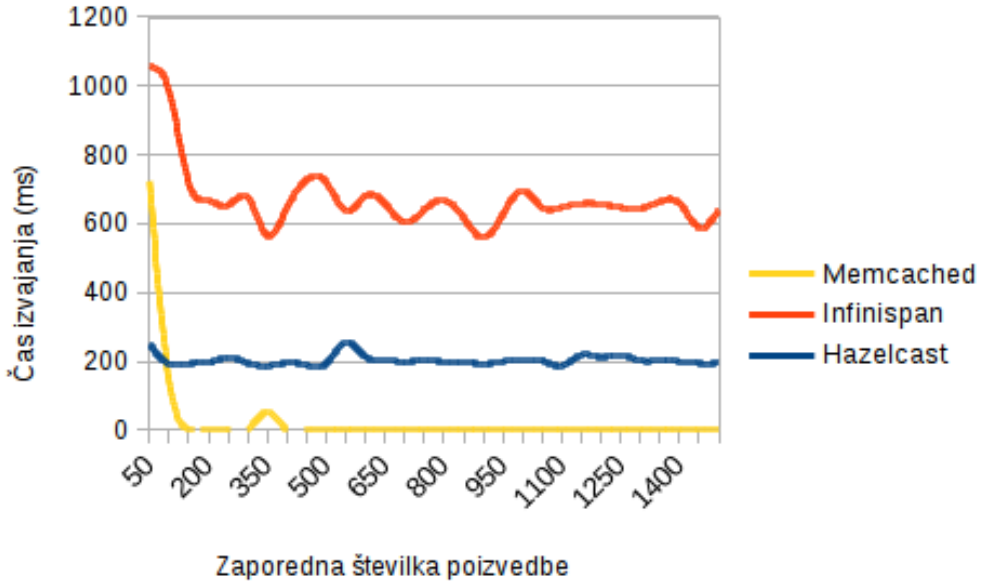
Test smo ponovili še s testno množico 500 000 entitet vozil. Tu so se povprečni časi močno dvignili pri vseh vmesnikih (tabela 5.3). Pri vseh se je povprečni čas poizvedbe povečal za faktor med 3 in 4.

vmesnik	povprečen čas agregacije (ms)
`Memcached`	17, 2766666667
`Infinispan`	1738, 246
`Hazelcast`	157, 6413333333

Tabela 5.3: Povprečen čas agregacije (500 000 entitet.

Na grafični primerjavi 5.12 opazimo občutno povečanje povprečnih časov predvsem pri prvih agregacijah. Ponovno je presenetil vmesnik Hazelcast.

Pri obeh testih agregacije z večjima testnima množicama je povprečni čas v posameznem testu zelo konstanten.



Slika 5.12: Čas izvajanja zaporednih agregacijskih poizvedb (500 000 entitet).

Iz agregacijskih testov lahko povzamemo, da so vmesniki za porazdeljeno pomnjenje uporabni tudi za izvajanje agragacij. Zelo dobre rezultate smo dosegli z vmesnikom Hazelcast. Če primerjamo povprečne čase prvih 200 zahtevkov na vseh treh grafih, opazimo zelo dobre rezultate že pri prvih zahtevkih. Celo boljše kot poizvedbe nad podatkovno bazo MySQL. To nakazuje na to, da se Hazelcast zelo dobre odreže pri agregacijskih poizvedbah, ki se prej nad predpomnilnikom še niso vršile. V primerih, ko se enake agregacijske poizvedbe ne pojavljajo pogosto ali pa se podatki pogosto spreminjajo, je to velika prednost.

Poglavje 6

Sklep

V diplomski nalogi smo se spoznali s predpomnjenjem podatkov v porazdeljenih sistemih. Predstavlili smo arhitekturo porazdeljenih sistemov v obliki mikro storitev in izpostavili potrebo po porazdeljenem predpomnjenju. Izpostavili smo omejitve in izzive predpomnjenja v porazdeljenih sistemih in predstavili rešitev v obliki vmesnikov. Povzeli smo njihovo delovanje in najpogostejše skupne lastnosti ter najpogostejše načine uporabe. Tri izbrane vmesnike, Memcached, Infinispan in Hazelcast, smo podrobneje preučili in predstavili njihovo delovanje in uporabo na praktičnem primeru. S testi smo ustvarili simulacijo delovanja v resničnem svetu in primerjali rezultate ter strnili ugotovitve.

S pridobljenimi rezultati smo prispevali k razumljivosti delovanja porazdeljenega predpomnilnika ter si odgovorili na vprašanje, kdaj ga je možno uporabiti in kakšni so razlogi za njegovo vpeljavo v arhitekturo razvijane aplikacije. Glede na namen uporabe, z rezultati testov pomagamo pri odločitvi izbire vmesnika.

Pri primerjavi izbranih vmesnikov smo ugotovili, da izbira ni očitna in moramo pri tem upoštevati veliko različnih parametrov. Najbolj pomembno je vedeti, kakšne podatke bomo hranili, koliko jih bo, kako se bodo spreminjali, če so kje trajno shranjeni in če si lahko privoščimo njihovo izgubo.

Memcached se je izkazal za učinkovito rešitev kot dodaten sloj med aplikacijo in podatkovno bazo pri pomnjenju podatkov tipa ključ-vrednost. Do določene mere se ga uspe izrabiti tudi za hranjenje poizvedb in agregacij nad podatkovno bazo, za kar pa kot samostojna rešitev ni primeren. Ker ne podpira razširitve brez ponovnega zagona odjemalcev, je bolj primeren v aplikacijah, kjer je velikost predpomnjenih podatkov vnaprej znana.

Če imamo potrebo po predpomnjenju vnaprej predvidene velikosti podatkov dosegljivih pod znanimi ključi, je Memcached vsekakor dobra izbira. Z relativno malo truda lahko močno razbremenimo podatkovno bazo in dosežemo drastične pohitritve pri izvajanju aplikacije.

Infinispan se je v povprečju izkazal najslabše, kar pa še ne pomeni, da ni primerna izbira v nobenem primeru. Dopuščamo možnost, da bi lahko dosegli boljše rezultate z dodatnim testiranjem in spreminjanjem nastavitev. Vsekakor nam je pri integraciji povzročal največ težav. Do osnovnega delovanja smo sicer prišli zelo hitro, pri nastavljanju dodatnih nastavitev pa smo praviloma izgubili preveč časa. Ko smo naleteli na napako, nam je iskanje po spletu le redko pomagalo pri odpravi le te, kar nakazuje na slabo podporo skupnosti. Javanski programski vmesnik (angl. application programming interface – API) pogosto ne vsebuje dokumentacije. Mersko enoto omejitve velikosti predpomnilnika smo našli šele v komentarju izvlečka programske kode v dokumentaciji.

Pri hranjenju podatkov tipa ključ-vmesnik se je odrezal dobro. Prav tako so bili zadovoljivi časi poizvedb in agregacij pri manjšem številu entitet. Opazili pa smo drastično upočasnitev pri vstavljanju podatkov z dodajanjem indeksov iskanja. Vmesnik sicer ponuja podvajanje podatkov, trajno hrambo in dinamično horizontalno razširanje. To so lastnosti, ki si jih želimo, kadar končno število hranjenih podatkov ni znano, izgube pa si ne moremo privoščiti.

Hazelcast se je v vseh pogledih odrezal odlično. Integracija je nadvse enostavna, dokumentacija in primeri uporabe pa zelo uporabni. Vsekakor je dobra izbira v poslovnih aplikacijah, ki upravljajo z velikimi količinami spremenljivih podatkov. Ponuja dinamično horizontalno razširljivost, trajno hrambo, podvajanje podatkov, hkrati pa dosega konstantne rezultate tudi pri večjih množicah podatkov. Za zahtevnejše uporabnike ponuja tudi plačljivo različico za podjetja, ki ponuja dodatno podporo in še nekaj dodatnih funkcionalnosti.

Kljub doseženim ciljem diplomske naloge predlagajmo nekaj izboljšav. Zastavljene cilje bi lahko še izboljšali s primerjanjem razultatov pri hranjenju podatkov različnega tipa in velikosti. Zanimivi bi bili tudi rezultati s spreminjajočimi podatki, ki bi simulacijo še bolj približali k realnim scenarijem.

Literatura

[1] Bonér, Jonas. Reactive microservices architecture: design principles for distributed systems. pages 3–4, 7. O'Reilly Media, 2016.

[2] Newman, Sam. Building microservices: designing fine-grained systems. chapter 1, pages 15–34. O'Reilly Media, Inc., 2015.

[3] Ciurana, Eugene. Developing with google app engine. chapter 7, pages 111–114. Apress, 2009.

[4] Khan, Iqbal. Scale out – distributed caching on the path to scalability. *MSDN magazine*, page 62, 2009.

[5] Namiot, Dmitry and Sneps-Sneppe, Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[6] Roughgarden, Tim and Valiant, Gregory. Cs168: The modern algorithmic toolbox lecture# 1: Introduction and consistent hashing. 2015.

[7] Nielsen, Michael. Consistent hashing. Dosegljivo: `http://michaelnielsen.org/blog/consistent-hashing/`, 2009. [Dostopano: 18. 1. 2018].

[8] Surtani, Manik. A look inside jboss cache. Dosegljivo: `https://dzone.com/articles/a-look-inside-jboss-cache`. [Dostopano: 18. 1. 2018].

[9] Hicks, Gabe. Are monolithic software applications doomed for extinction? Dosegljivo: https://dev9.com/blog-posts/2017/4/are-monolithic-software-applications-doomed-for-extinction. [Dostopano: 18. 1. 2018].

[10] Hámori, Ferenc. How enterprises benefit from microservices architectures. Dosegljivo: https://blog.risingstack.com/how-enterprises-benefit-from-microservices-architectures/. [Dostopano: 18. 1. 2018].

[11] Bozhanov, Bozhidar. Distributed cache – overview. Dosegljivo: https://techblog.bozho.net/distributed-cache-overview/. [Dostopano: 11. 12. 2017].

[12] Cacheonix. Dosegljivo: https://www.cacheonix.org/. [Dostopano: 17. 1. 2018].

[13] Cloud Computing Concepts, Part 1. Dosegljivo: https://www.coursera.org/learn/cloud-computing/lecture/nvMXE/2-2-what-is-a-distributed-system. [Dostopano: 1. 12. 2017].

[14] Infinispan. Dosegljivo: http://infinispan.org/. [Dostopano: 18. 1. 2018].

[15] Hazelcast. Dosegljivo: https://hazelcast.com/. [Dostopano: 18. 1. 2018].

[16] Memcached. Dosegljivo: http://memcached.org/. [Dostopano: 18. 1. 2018].

[17] Redis. Dosegljivo: https://redis.io/. [Dostopano: 17. 1. 2018].

[18] Oracle Coherence. Dosegljivo: https://www.oracle.com/middleware/coherence/index.html. [Dostopano: 17. 1. 2018].

[19] Apache Ignite. Dosegljivo: https://ignite.apache.org/. [Dostopano: 17. 1. 2018].

[20] NCache. Dosegljivo: http://www.alachisoft.com/ncache/. [Dostopano: 17. 1. 2018].

[21] Ehcache. Dosegljivo: https://www.oracle.com/middleware/coherence/index.html. [Dostopano: 17. 1. 2018].

[22] AppFabric. Dosegljivo: https://msdn.microsoft.com/en-us/library/aa139637.aspx. [Dostopano: 17. 1. 2018].

[23] Aerospike. Dosegljivo: https://www.aerospike.com/. [Dostopano: 17. 1. 2018].

[24] GigaSpaces. Dosegljivo: https://www.gigaspaces.com/. [Dostopano: 17. 1. 2018].

[25] Riak KV. Dosegljivo: http://basho.com/products/riak-kv/. [Dostopano: 17. 1. 2018].

[26] JBoss Cache. Dosegljivo: http://jbosscache.jboss.org/. [Dostopano: 17. 1. 2018].

[27] Java 8. Dosegljivo: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html. [Dostopano: 17. 1. 2018].

[28] IntelliJ IDEA. Dosegljivo: https://www.jetbrains.com/idea/. [Dostopano: 17. 1. 2018].

[29] Ubuntu 16.10. Dosegljivo: http://old-releases.ubuntu.com/releases/16.10/. [Dostopano: 17. 1. 2018].

[30] Apache Tomcat. Dosegljivo: http://tomcat.apache.org/. [Dostopano: 17. 1. 2018].

[31] Spring Boot. Dosegljivo: https://projects.spring.io/spring-boot/. [Dostopano: 17. 1. 2018].

[32] MySQL. Dosegljivo: https://www.mysql.com/. [Dostopano: 17. 1. 2018].

[33] Apache Maven. Dosegljivo: https://maven.apache.org/. [Dostopano: 17. 1. 2018].

[34] Git. Dosegljivo: https://git-scm.com/. [Dostopano: 17. 1. 2018].

[35] JUnit. Dosegljivo: http://junit.org/junit5/. [Dostopano: 17. 1. 2018].

[36] Spring Data JPA. Dosegljivo: https://projects.spring.io/spring-data-jpa/. [Dostopano: 17. 1. 2018].

[37] ElastiCache. Dosegljivo: https://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/AutoDiscovery.html. [Dostopano: 10. 12. 2017].

[38] Vehicle Year, Make, and Model data in SQL, NoSQL data format since year 2001. Dosegljivo: https://github.com/arthurkao/vehicle-make-model-data. [Dostopano: 10. 4. 2017].

[39] Hash function. Dosegljivo: https://en.wikipedia.org/wiki/Hash_function. [Dostopano: 09. 12. 2017].

[40] SpyMemcached. Dosegljivo: https://github.com/couchbase/spymemcached. [Dostopano: 09. 12. 2017].