Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Alan Rijavec

Razširljiva arhitektura za agregacijo podatkov iz različnih spletnih virov

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJSKEM PROGRAMU

Mentor: izr. prof. dr. Marko Bajec

Ljubljana 2012

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja. [1]

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

[1] V dogovorju z mentorjem lahko kandidat diplomsko delo s pripadajočo izvorno kodo izda tudi pod katero izmed alternativnih licenc, ki ponuja določen del pravic vsem: npr. Creative Commons, GNU GPL. V tem primeru na to mesto vstavite opis licence, na primer tekst [?]

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01846/2012

Datum: 04.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ALAN RIJAVEC**

Naslov: **RAZŠIRLJIVA ARHITEKTURA ZA AGREGACIJO PODATKOV IZ RAZLIČNIH SPLETNIH VIROV**

SCALABLE ARCHITECTURE FOR DATA AGGREGATION FROM VARIOUS WEB SOURCES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V okviru diplomske naloge preučite obstoječe rešitve za zajem podatkov iz različnih spletnih virov, kot so spletne strani in spletne aplikacije. Na osnovi pregleda obstoječih rešitev izdelajte načrt razširljive arhitekture za agregacijo podatkov iz poljubnih spletnih virov, ki bo omogočala dodajanje vtičnikov za poljuben spletni vir. Načrt uporabite za razvoj prototipne rešitve, s katero demonstrirajte razširljivost arhitekture.

Mentor:

prof. dr. Marko Bajec

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Alan Rijavec, z vpisno številko **63060278**, sem avtor diplomskega dela z naslovom:

Razširljiva arhitektura za agregacijo podatkov iz različnih spletnih virov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Bajca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 26. septembra 2012 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Marku Bajcu za nasvete in vodenje pri izdelavi diplomskega dela.

Zahvale gredo tudi Slavku Žitniku, ki je bil z nasveti vedno v oporo, ko sem se znašel v težavah.

Iskrena hvala gre tudi družini, ki me je podpirala vsa leta študija.

Assist a man in raising a burden; but do not assist him in laying it down.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Sorodne aplikacije **3**

2.1 Prijaznost do strežnikov 4

2.2 Agregatorji 5

2.3 Spletni pajki 7

2.4 Orodja za analizo vsebine 10

3 Platforma aplikacije za semantično zajemanje podatkov iz predefiniranih virov **13**

3.1 Arhitektura aplikacije 14

3.2 Upravljanje s podatki 18

3.3 Iskanje pravega zadetka 20

3.4 Nepopolnost vhodnih in izhodnih podatkov 20

3.5 Časovna potratnost 21

3.6 Preobremenjenost strežnikov 23

4 Semantično zajemanje podatkov **25**

4.1 Osrednji razred 26

4.2 Vtičniki 37

5 Evaluacija **41**

5.1 Predefinirani viri 41

5.2 Primer iskanja 45

6 Zaključek **49**

Povzetek

Programi za programsko zbiranje podatkov iz spletnih virov se uporabljajo že vrsto let. Prvi spletni iskalniki so se pojavili že, ko je internet bil še v povojih.

Aplikacija, razvita v sklopu tega diplomskega dela, je sorodna spletnim iskalnikom, ki so v bistvu spletni pajki. Skupne točke dobimo tudi z agregatorji, ki pa ponavadi zbirajo istovrstne podatke iz različnih virov.

Cilj naloge je razviti program za agregiranje podatkov iz predefiniranih virov. Tipični agregatorji zbirajo le istovrstne podatke, npr. agregatorji novic, kot je Google News, zbirajo novice. Te pa imajo vedno isto podatkovno strukturo. Naš program pridobiva podatke s pomočjo objektov, ki se uporabljajo na podoben način kot vtičniki. 'Vtičniki' so enostavni za implementacijo in je ob spremembah potrebnih malo popravkov. Posebnost naše aplikacije je semantično zajemanje raznovrstnih podatkov, ki jih določajo posamezni 'vtičniki' in ne aplikacija. Gre za aplikacijo, ki jo uporabnik lahko koristi na katerem koli želenem področju pod pogojem, da razvijemo potrebne vtičnike.

Ključne besede: agregator, spletni pajek, semantično zajemanje podatkov

Abstract

Software programs for collecting data from online sources are have been used for a number of years. The first search engines appeared when the Internet was still in its infancy.

The application, developed in the context of this thesis, is akin to web search engines, which are in essence web spiders. We can also find similarities with aggregators, which usually collect the same type of data from different sources.

The aim of this work is to develop a programme to aggregate data from predefined sources. Typical aggregators collect only the information of the same type. News aggregators such as Google News, collect news which always has the same data structure. Our programme obtains data through objects, which are used in a similar way as plug-ins. "Plug-ins" are easy to implement and corrections need little changes. The specialty of our application is semantic capturing of heterogeneous data defined by each 'plug-in' and not by the application. This is an application that the user may benefit from in any desired field provided we develop the necessary plug-ins.

Key words: aggregator, web crawler, semantic data capture

Poglavje 1

Uvod

Splet se z informacijami, ki jih ponuja, vsakodnevno širi. Poleg spleta in količine informacij se večajo tudi potrebe uporabnikov po dostopu do teh informacij, kar zna biti zelo zamudno. Za premostitev tega problema so bili razviti številni spletni pajki in agregatorji.

Google News je kot agergator novic odličen primer praktične uporabnosti teh aplikacij. Uporabnik pregleduje eno spletno mesto, na katerem najde novice iz številnih spletnih medijev širom sveta. Prebiranje dnevnih novic postane tako enostavnejše in hitrejše v primerjavi s pregledovanjem novice po posameznih spletnih straneh založnikov.

Aplikacija, ki je bila razvita v sklopu tega diplomskega dela, je podobna omenjenemu agregatorju. Kljub temu ima naša aplikacija nekoliko drugačne zahteve.

Agregatorji novic zbirajo podatke s točno določeno strukturo. Vsaka novica ima naslov, podnaslov, uvod, glavno vsebino, povezavo itd. Naša aplikacija pa mora znati zbirati različne strukture podatkov, ki jih posredno ali neposredno določajo vtičniki, na enem mestu. Tukaj se pojavi veliko problemov z vstavljanjem izhodnih podatkov in z iskanjem vhodnih podatkov, saj načeloma 'vtičnik', ki skrbi za en spletni vir, ne ve za ostale. Pravilna izbira načina definicije vhodnih in izhodnih struktur podatkov lahko precej olajša implementacijo vtičnikov in ogrodja aplikacije za semantični zajem podatov.

Rešitev mora biti enostavna, saj ne želimo preobremenjevati razvijalcev s tem opravilom, kljub temu pa jim mora omogočati shranjevanje katere koli strukture podatkov, ki jo le-ti potrebujejo.

Zgoraj smo omenili, da razvijamo aplikacijo za semantični zajem podatkov. Kaj pa sploh pomeni semantični zajem podatkov? Za vsak podatek, ki ga pridobimo pri semantičnem zajemu podatkov, poznamo njegov pomen. Poznavanje njegovega pomena omogoča uporabo tega podatka pri drugih 'vtičnikih', od katerih prejete podatke spet uporabimo pri novih poizvedbah. To zaporedno izvajanje poizvedb, pri katerih je izhod ene poizvedbe vhod v drugo, ne bi bilo mogoče brez poznavanja semantike oz. pomena podatkov.

Aplikacija se od tipičnih agregatorjev razlikuje tudi v namenu. Medtem ko so navadni agregatorji načrtovani za določen namen uporabe, je ena od najpomembnejših lastnosti naše aplikacije prilagodljivost na katero koli področje.

V nadaljevanju bomo predstavili razvito aplikacijo za semantični zajem podatkov iz predefiniranih virov, ki upošteva zgornje usmeritve s pozornostjo na prijaznost do strežnikov in enostavnostjo razširitve z dodajanjem vtičnikov.

Poglavje 2

Sorodne aplikacije

Agregatorji in spletni pajki so aplikacije, ki so po delovanju podobne našemu programu. To poglavje služi pregledu teh aplikacij in predstavitvi njihovih najbolj prepoznavnih predstavnikov.

Na začetku se bomo najprej dotaknili teme, ki ni neposredno povezana z naslovom poglavja. V podpoglavju Prijaznost do strežnikov bomo povedali, na kaj moramo biti pozorni, ko uporabljamo aplikacije, kot so spletni pajki ali agregatorji, da ne ogrozimo delovanja strežnikov. Za tem bomo spoznali agregatorje - aplikacije za zbiranje podatkov iz različnih virov. Agregatorji so najboljši približek našemu programu, ki tako kot naš program zbirajo podatke iz več virov. Predstavili bomo tudi Google News, ki je Googlov agregator novic. Seveda obstaja veliko vrst agregatorjev, ki se imenujejo po vsebini, katero zbirajo; med drugimi obstajajo agregatorji socialnih omrežij, video agregatorji, agregatorji recenzij itd. Nadaljevali bomo s spletnimi pajki. Njihova primarna naloga je slediti hiperpovezavam iz že prenesenih spletnih strani. Z njihovo uporabo lahko npr. iščemo podatke na statičnih spletnih straneh, kjer iskalnika ni na voljo. Zaključno podpoglavje bomo namenili orodjem za analizo vsebine, ki lahko pri nestrukturiranih zadetkih naredijo nekakšno ovojnico, iz katere je lažje pridobiti iskane podatke.

2.1 Prijaznost do strežnikov

Spletne strani so primarno namenjene osebam, da jim zagotavljajo informacije, ki jih potrebujejo. Čeprav so strežniki danes že zelo zmogljivi, lahko prevelika količina zahtev spravi strežnik v nedelovanje. Z uporabo programov, kot so spletni pajki (ang. crawler) ali agregatorji, lahko nenamenoma preobremenimo strežnike z zahtevami in prikrajšamo druge uporabnike za informacije. To je ena izmen vrst DoS (denial of service) napadov in lahko pusti pravne posledice, čeprav je bilo dejanje nenamerno.

Razvijalec pajkov ali agregatorjev mora paziti, da do takih neželenih učinkov ne pride. Najenostavnejši način, kako to narediti, je določiti maksimalno frekvenco pošiljanja zahtev na strežnik [1]. Recimo, da uporabnik uporablja agregator novic, katere prebira enkrat dnevno. Agregator torej lahko preverja spletne strani za sveže novice le nekajkrat dnevno. Seveda je pri nastavljanju frekvence potrebno paziti tudi na uporabnost podatkov. Če je frekvenca agregatorja, ki zbira za uporabnika pomembne podatke, prenizka glede na frekvenco spreminjaja podatkov, bodo ti ob pregledovanju zelo verjetno neuporabni.

Zgoraj našteti programi so primerni tudi za zbiranje in repliciranje podatkov, s čimer lahko uporabnik oz. razvijalec takih programov krši avtorske pravice in se tako zakoplje v težave. Preden razvijalec začne programsko dostopati do spletnega mesta, bi se moral prepričati, če ne bi s svojimi dejanji kršil pogojev uporabe spletne strani. Slednji delujejo kot pogodba, ki začne veljati že samo s tem, ko oseba začne uporabljati stran na takšen ali drugačen način [1]. Številne spletne strani niso namenjene pajkom in agregatorjem ampak fizičnim osebam, ki iščejo potrebne informacije. Uporaba teh programov na takih spletnih straneh bo verjetno razlog nezadovoljstva pri administratorjih spletnih strani in lahko povzroči težave. Zato se moramo tudi, če ne želimo replicirati podatkovne baze nekega spletnega mesta, vprašati o etičnosti našega početja.

Boti citebot, ki se pogosto uporabljajo pri spletnih pajkih, so namenjeni izvajanju enostavnih, ponavljajočih in avtomatiziranih nalog na internetu.

Razlog za njihov obstoj sta hitrost in frekvenca, s katero lahko opravljajo naloge, saj sta obe neprimerno večji od zmogljivosti človeka. Boti se lahko uporabljajo za povsem neškodljive naloge, lahko pa služijo tudi manj poštenim uporabnikom za razne napade na spletne strežnike.

Administratorji spletnih strani bodo v primeru, da uporaba botov na njihovih straneh ni zaželjena, poskušali onemogočiti uporabo takih programov. Leta 1994 je Martijn Koster razvil protokol [9], po katerem morajo boti upoštevati navodila datoteke robots.txt. V tej datoteki, ki se mora nahajati v korenski mapi spletnega mesta, je v točno predpisani obliki zapisano, do katerih vsebin lahko posamezni boti dostopajo. Navodila se lahko nanašajo na vse bote ali zgolj na posamezne. Dostop pa se lahko prepoveduje do posameznih datotek ali celotnih map. S parametrom Crawl-delay lahko nastavimo premor med pošiljanjem zahtev istemu strežniku. Spodaj si lahko ogledamo primer take datoteke.

```
User-agent: Googlebot
Disallow: /images/
Disallow: /private/index.html
Crawl-delay: 5

User-agent: *
Disallow: /
Sitemap: http://www.example.com/sitemap.html
```

Slabost tega protokola je, da dejansko ne obstaja mehanizem, ki bi botom preprečeval dostop do prepovedanih vsebin. Odločitev, ali bodo upoštevali navodila, zapisana v datoteki robots.txt, leži v rokah razvijalecev botov.

2.2 Agregatorji

Agregator [1] je spletni robot, ki združuje informacije iz različnih internetnih virov in jih prikazuje v prečiščeni obliki na enem mestu. To je lahko spletna stran ali lokalen dokument, odvisno od tipa agregatorja, ki ga uporabljamo.

Vsebina teh dokumentov je v obliki zloženih objektov, ki so sestavljeni iz večjega števila manjših objektov. Čeprav agregatorji zbirajo informacije iz večjega števila virov, so te informacije podobno sestavljene. To olajšuje shranjevanje podatkov, saj mora agregator poznati le eno podatkovno strukturo ne glede na število virov. Kot primer navajam agregatorje novic. Čeprav lahko agregator shranjuje novice iz poljubnega števila virov, je za vse novice skupno, da so sestavljene iz naslova, teksta, hiperpovezave, datuma objave...

Agregator je lahko sestavljen iz zbirke spletnih robotov ali enega samega robota, imenovanega CatBot (category bot). Navadni roboti ponavadi znajo delati le z določenim virom, za katerega so bili narejeni. To pomeni, da če želimo zbirati podatke iz večjega števila spletnih strani, moramo napisati enako število takih programov. Največja pomanjkljivost navadnih spletnih robotov je ta, da niso odporni na spremembe na spletnih straneh vira. Ti programi se pri iskanju podatkov ponavadi zanašajo na oznake v HTML dokumentu. Če so te oznake spremenjene, lahko to pomeni, da robot ne zmore več obdelovati spletne strani. CatBot je iz tega pogleda zelo prilagodljiv, saj naj bi zmogel obdelovati vse spletne strani znotraj določene kategorije in se ga da enostavno spremeniti v agregator.

2.2.1 Agregatorji novic

Internet postaja vedno večji vir novic. Zaradi interaktivnosti, ki jo spletne strani omogočajo, ter količine in raznolikosti informacij, ki jih ponuja internet, postajajo drugi mediji odvečni. Kot vemo so posamezni viri spletnih novic ponavadi specializirani za določena področja. Znameniti francoski časopis L'Equipe je odličen vir športnih novic, če pa iščemo novice iz sveta financ se bomo raje obrnili na druge vire, kot so npr. The Wall Street Journal ali Finance. Uporabnik, ki ga zanima več področij, bo tako primoran obiskati več spletnih mest. Agregatorji novic poskušajo poenostaviti ta vsakodnevni proces.

Primer takega agregatorja je Google News [7], ki z avtomatičnim agregacijskim algoritmom brska po več kot 25000 virih novic, ki so rangirani po

pomembnosti. Veliko člankov, ki jih ta agregator pridobi, se nanaša na isti dogodek. V ta namen so pri Google News uvedli nov pojem, kopica zgodbe (ang. story cluster). To je skupina novic, ki so osredotočene na določen zorni kot nekega dogodka. Npr. članki, ki se nanašajo na rezultat finalne tekme Evropskega prvenstva 2012 v nogometu spadajo v eno kopico. Članki, ki se nanašajo na incidente v italijanski slačilnici po koncu finalne tekme pa spadajo v novo kopico. Čeprav se vse novice sklicujejo na isti dogodek, je pozornost usmerjena na dva različna zorna kota. Kopice novic se razvrščajo glede na vidnost, ki jim jo določajo založniki. Za vsako zgodbo želijo zvedeti ali je to pomembna zgodba ali ne in na katero geografsko področje se nanaša. Na podlagi teh podatkov lahko sestavijo ponudbo novic za različne jezike in lokacije po svetu.

Znotraj posameznih kopic se članki razvrščajo glede na več parametrov. Kako visoko se bo članek znotraj kopice povzpel je odvisno od lokalnosti založnika, posodabljanja glede na razvoj dokodkov, originalnosti vsebine in drugih parametrov, ki dajejo informacijo o verodostojnosti založnika za določeno novico.

2.3 Spletni pajki

Pajek [1] (ang. web crawler) je spletni robot, ki je specializiran za raziskovanje spletnih strani ali drugih dokumentov. S pomikanjem po hiperpovezavah se pajki sprehajajo od dokumenta do dokumenta dokler ne pregledajo vseh razpoložljivih dokumentov ali presežejo druge pogoje, ki jih lahko določijo uporabniki. Da pajek začne z delom potrebuje vsaj en začetni URL naslov. Med kopico URL naslovov izbere enega in dostopa do dokumenta na tem URL naslovu. Nadaljno delovanje poteka tako, da iz pridobljenega dokumenta izlušči vse hiperpovezave in jih doda v vrsto za pregled. Ko pregleda celoten dokument izbere naslednji URL naslov in nadaljuje delo. Za uporabnike so nekatere vsebine pomembnejše od drugih, zato je pri izbiri naslednje hiperpovezave potrebno upoštevati prioritete uporabnika. Pajek zaključi pro-

ces preiskovanja, ko ni na razpolago hiperpovezave, ki še ni bila uporabljena.

Vsebine oz. dokumenti, do katerih dostopamo enostavno z uporabo hiperpovezav, imenujemo površinski splet (ang. surface web). Nekatere vsebine pa so pajkom skrite. Dostop do teh ponavadi vključuje poizvedovanje po bazi na strežniški strani. Take vsebine imenujemo globoki splet [12] (ang. deep web) in vključujejo nekatere dinamične spletne strani, zasebne spletne strani itd. Za reševanje problema globokega spleta so razvili protokol Sitemaps, ki predvideva datoteko sitemap.xml, v kateri so našteti URL-ji spletne strani. Razvili so tudi druge, bolj strateške metode, kot je praskanje zaslona (ang. screen scraping), pri katerem pajek izvaja avtomatične poizvedbe z izpolnjevanjem spletnih obrazcev. Pri preiskovanju globokega spleta je potrebno razumeti, da se lahko število strani, ki jih je potrebno pregledati, hitro množi. Kot vemo je na spletu ogromno spletnih strani in če bi pajki sledili vsem hiperpovezavam ne bi nikoli zaključili z izvajanjem. Ta problem lahko rešimo z omejevanjem na določeno spletno mesto s podajanjem predpon veljavnih URL naslovov.

Pajki imajo veliko različnih funkcionalnosti. Ena izmed njih je lokalno shranjevanje spletnih mest. HTTrack [10] je aplikacija, ki lahko vzpostavi celotno drevesno strukturo map in lokalno shrani vse datoteke, potrebne za ogled spletne strani, in sicer kadarkoli ter brez potrebe po dostopu do interneta. Podobna orodja se lahko uporabljajo za preverjanje strukture strani, izdelavo seznama datotek, ki jih stran potrebuje in iskanje povezav, ki ne delujejo več. Taka orodja se uprabljajo predvsem v zdrževalne namene. Ena prvih oblik uporabe spletnih pajkov pa so spletni iskalniki. Spletni iskalniki omogočajo uporabniku vnos ključnih besed, po katerih se iskanje izvaja. Starejši spletni iskalniki so dejansko preiskovali splet za vsako poizvedbo in poskušali dobiti zadetke na preiskovanih spletnih straneh ter jih nato ponudili uporabniku. Ta praksa je kmalu postala neuporabna, saj se splet hitro širi in bi tako poizvedovanje v današnjih okoliščinah potekalo preveč časa. Sodobni spletni iskalniki uporabljajo spletne pajke, da preiskujejo splet vnaprej in hranijo podatke o URL-jih in ključnih besedah v ogromnih podatkovnih

bazah, po katerih poizvedujejo ob iskanju zadetkov. Tako poizvedovanje je veliko hitrejše. Kljub temu obstaja določena zakasnitev pri osveževanju podatkovne baze ob spremembah na spletnih straneh.

2.3.1 Apache Nutch

Apache Nutch [4] je spletni iskalnik, napisan v programskem jeziku Java. Ker je odprtokoden je priljubljen pri uporabnikih kot so državne institucije, kjer je pomembna transparentnost rangiranja zadetkov. Kot primer naj omenim Portuguese Web Archive, ki je v lasti portugalske vlade. Prednost iskalnika Apache Nutch je tudi modularna arhitektura, ki omogoča programerjem razvoj lastnih vtičnikov za zbiranje podatkov, poizvedovanja...

Celoten projekt lahko v grobem razdelimo na dva dela, ki sta spletni pajek in iskalnik. Medtem ko spletni pajek pridobiva dokumente in jih pretvarja v indeks, iskalnik iz indeksa bere in odgovarja na uporabniške poizvedbe.

Za temo diplomske naloge je zanimiv predvsem spletni pajek, ki ga uporablja Apache Nutch. Splet si lahko predstavljamo kot graf, kjer so vozlišča strani, povezave pa hiperpovezave. Tako vidi splet tudi Apache Nutchov pajek, ki za shranjevanje spletnih strani in hiperpovezav uporablja graf, imenovan WebDB. Povezava je sestavljena iz dveh URL naslovov, vira in cilja. Stran pa je sestavljena iz URL-ja, izvlečka iz vsebine z zgoščevalno funkcijo MD5, števila izhodnih povezav, pomembnosti strani... Zadnja je določena z raznimi merili, kot je na primer povezanost strani iz drugih spletnih mest. V ta graf se na začatku doda korenske naslove. Pajek nato ustvari seznam strani za pregled, ki jih WebDB po prenosu pregleda ter dopolni z novimi podatki. Nato se znova ustvari seznam strani za pregled. Ta cikel se ponavlja do želene globine, nakar se prenesene strani obdela in izračuna skupen indeks, ki ga iskalnik uporablja pri poizvedbah.

2.4 Orodja za analizo vsebine

Program za semantičen zajem podatkov mora izluščiti podatke iz dokumentov, ki jih prenese iz spleta. Te podatke mora tudi pravilno interpretirati, če jih želimo uspešno uporabiti v nadaljnih poizvedbah. Dandanes so spletne strani dobro razdelane in je iskanje želenih podatkov znotraj dokumenta enostavno. Lahko pa se pojavijo težave pri spletnih straneh, ki imajo podatke razstresene v čistem tekstu brez html oznak. Delo s takimi spletnimi stranmi lahko postane zelo naporno. Za asistenco pri delu s takimi dokumenti si lahko pomagamo z orodji, ki nekoliko olajšajo našo nalogo.

Eno izmed teh orodij je Apache Tika [2, 3]. S tem programom lahko olajšano izvlečemo podatke iz raznovrstnih dokumentov, med katere na primer spadajo HTML dokumenti, PDF-ji, Microsoft Office datoteke in navadne tekstovne datoteke. Poleg samih podatkov nas včasih zanima, kaj ti podatki sploh opisujejo. Kot primer vzemimo aplikacijo, ki iz spletne strani nekega kinocentra pridobiva podatke o filmih, ki se trenutno predvajajo, nato pa s temi podatki poišče na drugi spletni strani kritiko tega filma. S prve spletne strani naj zbira podatke o naslovu filma in glavnih igralcih, na drugi spletni strani pa pridobi kritiko ter ime in priimek avtorja članka. Da bi aplikacija to zmogla narediti, mora iz spletne strani kinocentra znati ločiti, kateri podatek je naslov filma in kateri podatek vsebuje ime in priimek glavnega igralca. Z drugimi besedami, mora aplikacija zaznati strukturo podatkov in jih nato pravilno uporabiti v sledečih poizvedbah. Apache Tika nam lahko pomaga pri pridobivanju takih strukturiranih podatkov tako, da iz vhodnega dokumenta sestavi veljaven XHTML dokument, ki bolje opisuje strukturo teksta. Apache Tika je zelo uporabna predvsem, ko so podatki zapisani v čistem tekstu, čeprav imajo svojo notranjo strukturo. Oglejmo si še delovanja Apache Tika na sledečem primeru:

> **Content analysis**
>
> **Content analysis** or **textual analysis** is a methodology in the social sciences for studying the content of communication.

Uporabili smo tekst, pridobljen iz Wikipedije. Na vrhu je naslov, pod njim pa odstavek, ki vsebuje nekaj krepko tiskanih besed. Podčrtane besede predstavljajo hiperpovezave. Če bi uporabili Apache Tiko na takem tekstu, kot ga vidimo izpisanega na vrhu, bi dobili za rezultat sledeči izhod:

```
<h1>Content analysis</h1>
<p><strong>Content analysis</strong> or
<strong>textual analysis</strong>
is a <a href="..">methodology</a> in the
<a href="...">social sciences</a> for
studying the content of <a href="...">communication</a>.
</p>
```

Opazimo lahko, da je Apache Tika zaznala naslov in odstavek. Pravilno je označila krepko tiskane črke in označila hiperpovezave. Tako strukturirana besedila so nedvomno lažja za procesiranje.

Apache Tika je zelo koristna tako pri slabše strukturiranih HTML dokumentih kot pri ostalih nestrukturiranih besedilih. Služi lahko kot prva ovojnica, ki omogoča lažje zajemanje podatkov.

Poglavje 3

Platforma aplikacije za semantično zajemanje podatkov iz predefiniranih virov

V tem poglavju bodo predstavljene možnosti, ki se nam ponujajo pri načrtovanju aplikacije za semantični zajem podatkov. Najprej si bomo v podpoglavju Arhitektura aplikacije ogledali arhitekturne značilnosti, ki jih aplikacija mora imeti, da bi omogočala vse lastnosti, ki smo jih omenili v uvodu. Zatem bomo primerjali rešitve, ki se ponujajo ob razdeljevanju odgovornosti med različnimi razredi. Preučili bomo različne možne razporeditve odgovornosti. Postavljali si bomo vprašanja, kot so: kateri razred naj išče vhodne podatke za določen vtičnik? Ali naj vtičniki sami skrbijo za kreiranje instance klienta in ali naj uporabljajo zunanje razrede za pomoč?

V podpoglavju Upravljanje s podatki se bomo seznanili s problemi, kot so definicija globalne podatkovne strukture naše aplikacije, iskanje vhodnih podatkov za vtičnike in vstavljanje podatkov. Z besedno zvezo globalna podatkovna struktura označujemo podatkovno strukturo, ki je definirana z uporabo določenega paketa vtičnikov.

Nadaljevali bomo s poglavjem Iskanje pravega zadetka, kjer bomo omenili težave pri določanju rezultatov poizvedb. V naslednjem podpoglavju,

Nepopolnost vhodnih in izhodnih podatkov, bomo opisali različne situacije, ko vhodni ali izhodni podatki niso popolni.

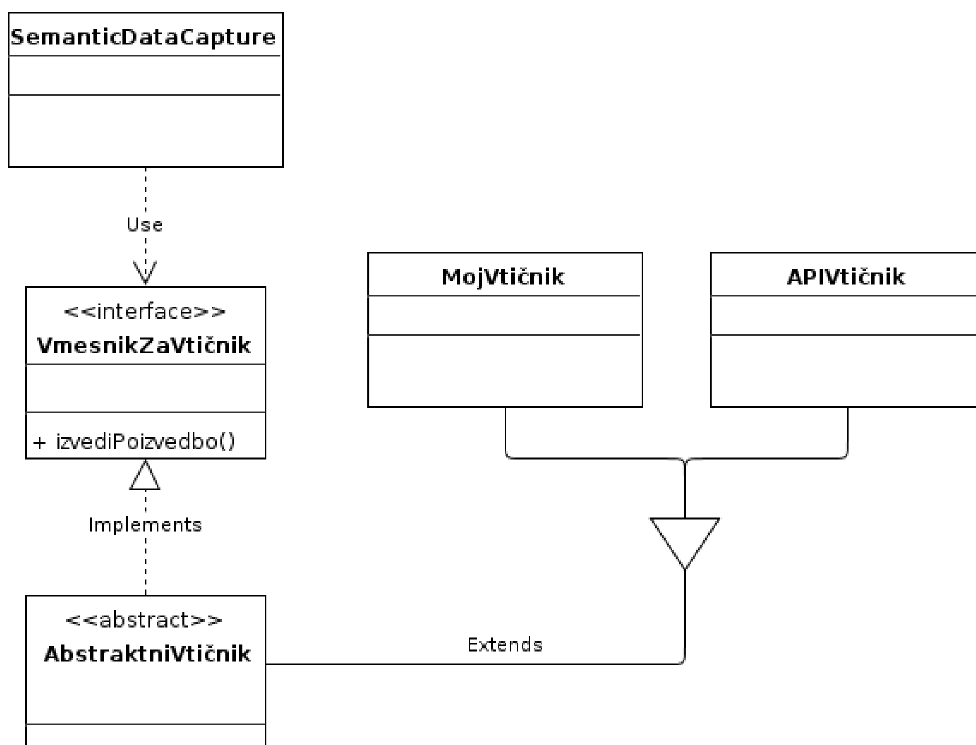
Sledilo bo poglavje Časovna potratnost. V primeru, da so podatki izjemnega pomena in uporabnik pričakuje rezultate v čim krajšem možnem času, postane časovna potratnost aplikacije zelo pomembna. Zaradi tega razloga bomo preverili, kje je največ prostora za pohitritev aplikacije.

Zaključno podpoglavje tega poglavja ima naslov Preobremenjenost strežnikov. Ko pišemo aplikacijo, ki avtomatizira rutinska opravila ljudi, lahko pomotoma preobremenimo vir, ki pričakuje človeške uporabnike. Naša aplikacija je veliko hitrejša v primerjavi z zmožnostmi ljudi, zato je koristno vedeti, na katere težave lahko naletimo ob nepravilni uporabi takšnih aplikacij.

3.1 Arhitektura aplikacije

Kako naj izgleda program za semantično zajemanje podatkov iz vnaprej predvidenih virov? Skrajno neprimerno bi bilo, da bi bila celotna izvorna koda zapisana v enem razredu. Tak program bi bil nečitljiv, predvsem pa težko razširljiv. Poleg omenjenih težav bi bilo potrebno ob vsaki spremembi ponovno prevajati celoten program, kar bi povzročalo določeno količino bolečin pri dodajanju novih virov. Aplikacija bi bila lažje razširljiva in bolj berljiva, če bi bila razdeljena v več razredov. Ko delimo razred v več razredov, ponavadi želimo, da so razredi vsebinsko in funkcionalno čimbolj razmejeni. Zelo hitro lahko odkrijemo dve funkcionalnosti aplikacije za semantični zajem podatkov iz predefiniranih virov. Prva je dostopanje do spletnih strani virov in iskanje podatkov, druga pa usklajevanje poizvedb po virih. Zato se zdi smiselno, da določimo en razred, ki bo usklajeval poizvedbe, drugi razred pa določimo za izvajanje poizvedb in pridobivanje podatkov. Zadnji razred lahko spet razbijemo na več razredov glede na vir, do katerega dostopajo. Zelo priročno bi bilo, če bi lahko te razrede dodajali brez potrebe po spreminjanju osrednjega razreda.

Predlagana rešitev poskuša oponašati arhitekturo vtičnikov. Za vsak nov



Slika 3.1: Uporaba vmesnikov pri izdelavi 'vtičnikov'.

vir napišemo 'vtičnik', ki razširi uporabnost osnovnega programa. Osnovni program pa naj bi znal te 'vtičnike' nadzorovati in usklajevati njihovo delovanje. Z uporabo vmesnikov (ang. interface) za izdelovanje vtičnikov lahko zagotovimo dodajanje novih virov ne da bi spreminjali osrednji razred. Namesto tega mora razred, ki implementira vmesnik, vsebovati metode, naštete v vmesniku. Vmesnik lahko nato ovijemo z abstraktnim razredom, v katerem določimo dodatne abstraktne metode, ki usmerjajo način izdelave vtičnika in tako olajšajo implementacijo le-tega. Primer take implementacije si lahko ogledamo na sliki 3.1. Vmesnik $VmesnikZaVticnik$ vsebuje metodo $izvediPoizvedbo()$. Ta metoda ni implementirana znotraj tega razreda ampak v razredih, ki ta vmesnik implementirajo. Vmesnik omogoča implementacijo različnih 'vtičnikov', ki jih lahko uporabljamo v aplikaciji ravno preko metod, ki jih izpostavlja vmesnik.

3.1.1 Dodeljevanje odgovornosti ogrodju in vtičnikom

Čeprav imamo že začrtane razrede, jim nismo še določili vlog in odgovornosti. Odgovor na vprašanje, kdo bo skrbel za vzdrževanje podatkov in do kakšne mere, je veliko bolj zapleten od pričakovanj. V tem podpoglavju si bomo ogledali naloge, ki se pojavljajo med izvajanjem aplikacije za semantično zbiranje podatkov iz predefiniranih virov, in jih skušali dodeliti ogrodju aplikacije, najsi bo to osrednji razred, ki upravlja z vtičniki ali ogrodjem vtičnika, ali razvijalcem vtičnika. Naloge si bomo ogledali po vrstnem redu pojava pri izvajanju aplikacije.

Pred zagonom vtičnika je potrebno poiskati vhodne podatke, ki naj jih vtičnik prejme. Ker je to potrebno narediti za vsak vtičnik in ker je postopek podoben, bi bilo smiselno, da to odgovornost prejme ogrodje aplikacije. Vendar če podatke zaradi berljivosti zbiramo v strukturirani obliki, kot je na primer XML datoteka, to neposredno zahteva poznavanje strukture vhodnih podatkov za vsak vtičnik. Torej mora biti za vsak vir nekje definirana struktura vhodnih podatkov in, kot bomo videli v nadaljevanju, tudi izhodnih podatkov. Logična izbira bi bila, da so te strukture določene znotraj vsakega

Firma	Skrajšana firma	Sedež
HIT hoteli, igralnice, turizem d.d. Nova Gorica	HIT d.d. Nova Gorica	Delpinova ulica 7A, 5000 Nova Gorica

Slika 3.2: Informacije o cilju hiperpovezave.

vtičnika, saj ima vsak vir svojo strukturo vhodnih in izhodnih podatkov.

S prejetimi podatki lahko vtičnik začne z delom. Tipičen potek dela se začne s kreiranjem HTTP klienta. Nekateri viri zahtevajo tudi prijavo z uporabniškim imenom in geslom. To so, na nivoju programske kode, precej splošne in ponavljajoče skupine ukazov, zato je podpora samega ogrodja vtičnika dobrodošla. V razvitem programu je večina koristnih metod za inicializacijo klienta in prijavo na spletno mesto realizirana v zunanjih razredih, ki ne vplivajo na delovanje vtičnikov, ampak so jim le v oporo.

Vtičnik nato začne proces iskanja rezultatov z uporabo vhodnih parametrov, ki ga lahko enačimo z delovanjem spletnih pajkov. Tukaj lahko poleg že omenjenega Apache Nutch projekta uporabimo razne knjižice, ki nudijo vmesnik (ang. interface) za kreiranje spletnih pajkov in omogočimo njihovo uporabo prek samega ogrodja vtičnika. Tako lahko programerju precej olajšamo delo. Po drugi strani pa lahko minimiziramo število dostopov do strežnika, če je brskanje v celoti prepuščeno programerju. Programer, ki pozna vir, lahko uporabi informacije na spletni strani in z njimi usmerja potek preiskovanja spetnega vira. Informacije, ki jih programer lahko uporabi, so predvsem besedila, ki opisujejo vsebino cilja hiperpovezave. Na sliki 3.2 lahko vidimo primer, kjer poleg same hiperpovezve na stran s podrobnostmi podjetja dobimo še podatke o naslovu in skrajšano ime. Z minimiziranjem števila odprtih strani lahko prihranimo veliko časa, saj je ponavadi med pošiljanjem dveh HTTP zahtevkov potrebno počakati določen čas, medtem ko je samo procesiranje spletne strani veliko hitrejše.

Ko pridemo do strani z rezultati, je potrebno podatke izluščiti iz doku-

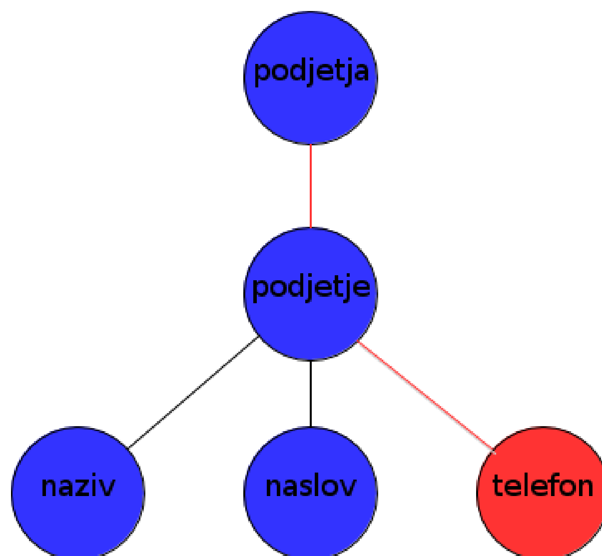
menta. Pri tem si lahko pomagamo z orodji, kot je Apacke Tika, ki pretvori dokument v XHTML dokument, kar olajša zajemanje podatkov. Tudi to funkcionalnost lahko vključimo v ogrodje vtičnika, vendar so današnje spletne strani že dovolj dobro oblikovane, da tudi brez uporabe takih orodij pridobivanje podatkov ne predstavlja pretiranega izziva.

Pridobljene podatke je nato potrebno umestiti med druge. Tudi vprašanje glede shranjevanja podatkov se da rešiti na več načinov. Nalogo bi lahko prevzeli vtičniki sami. Ker pa smo že povedali, da bi bilo smiselno iskanje vhodnih podatkov implementirati znotraj ogrodja aplikacije in ker sta nalogi podobni, je prav tako smiselno tudi to nalogo prepustiti ogrodju aplikacije. Po izvedbi zadnjega koraka smo izčrpali vir s trenutnimi vhodnimi podatki. Korake lahko ponovimo z uporabo naslednjega vira.

3.2 Upravljanje s podatki

Med samim izdelovanjem programa za semantično zajemanje podatkov se je izkazalo upravljanje s podatki kot eno izmed najzahtevnejših opravil. Če bi upravljanje s podatki prepustili vtičnikom oz. razvijalcem vtičnikov, bi nemara precej poenostavili razred, ki nadzoruje in usklajuje delovanje vtičnikov. Tudi iskanje vhodnih in zapisovanje izhodnih podatkov je enostavnejše znotraj posameznega vtičnika, kjer programer pozna strukturo podatkov. Kljub temu je smiselno to nalogo prepustiti ogrodju aplikacije. V prid tej odločitvi prispeva enostavnejše programiranje vtičnikov, ki tako omogoča programerjem vtičnikov, da se osredotočijo na iskanje podatkov.

Pri tem, da ogrodje aplikacije skrbi za upravljanje s podatki, je problematično to, da aplikacija v resnici ne pozna sheme podatkov. Kot smo že povedali, je mogoče dodajati nove vtičnike ne da bi spreminjali ostale razrede. To pomeni, da morajo novi vtičniki nositi informacije o nadgradnji sheme podatkov, tako vhodnih kot izhodnih. Če so podatki shranjeni v hierarhični strukturi, kot je npr. XML datoteka, lahko primer dodajanja dodatnih elementov na obstoječo shemo prikažemo kot je prikazano na sliki 3.3.



Slika 3.3: Dodajanje novih elementov na obstoječo shemo podatkov.

Modri krogi predstavljajo obstoječo shemo, rdeči krogi pa nove elemente. Z rdečo barvo je označena pot do na novo dodanega elementa. Opazimo lahko, da mora vtičnik, ki razširja shemo podatkov, poznati to shemo. Jasno je, da je struktura izhodnih podatkov lahko poljubna, saj dodajamo nove elemente. Potrebno je le paziti, da ne uporabljamo že uporabljenih elementov, ki hranijo po pomenu drugačne podatke. Pri vhodnih podatkih pa se moramo nujno sklicevati na obstoječo shemo, saj so le-ti izhod nekega drugega vtičnika. To pomeni, da če spremenimo izhodno shemo podatkov nekega vtičnika, moramo spremeniti tudi shemo vhodnih podatkov vtičnikov, ki te podatke uporabljajo. Torej, premišljene odločitve pri določanju sheme podatkov olajšajo delo v prihodnosti.

Nezanemarljiv problem, ki nastane, ko zapis izhodnih podatkov izvajamo izven vtičnika, je pripis izhodnih podatkov priadajočim vhodnim podatkom. S slike 3.3 vidimo, da element 'podjetja' lahko hrani podatke o več podjetjih. Zato je potrebno biti pozoren, da pripišemo pravo vrednost telefonske številke na pravo mesto.

3.3 Iskanje pravega zadetka

Med delovanjem vtičnika ponavadi dobimo več rezultatov iskanja. Kako se torej odločimo za pravi zadetek in kdo je nosilec odgovornosti? Očitno rešitev te naloge ni tako splošna, da bi jo vključili v ogrodje aplikacije, torej je potrebno nalogo rešiti za vsak vtičnik posebej. Nalogo si lahko nekoliko olajšamo, če zaupamo viru, da bo pravi zadetek med prvimi na seznamu vseh zadetkov. Nato pa moramo upoštevati vse informacije, ki so nam na voljo, da identificiramo zadetek, ki se najbolje prilega vhodnim podatkom. Pogosto se za pravilen zadetek ne moremo enostavno odločiti zaradi nekonsistentnosti podatkov, npr. za naziv podjetja se uporabljajo akronimi namesto pravih imen podjetij, pojavljajo se napake v črkovanju itd. V nekaterih primerih je nemogoče dobiti zadetek, zato je smiselno po določenem številu neuspešnih poizkusov iskanje zaključiti. Če pa dobimo več smiselnih rezultatov, lahko dodamo vse in prepustimo uporabniku aplikacije, da izbere pravilen rezultat.

3.4 Nepopolnost vhodnih in izhodnih podatkov

Seveda se lahko zgodi, da iz vira ne dobimo vseh podatkov, ki jih pričakujemo. Kaj lahko storimo v takih primerih? Odvisno od vrste podatka. Če podatek uporablja nek drugi vtičnik kot edini parameter, se potem ta ne more izvajati, če pa vtičnik uporablja več parametrov, lahko poizkusimo poiskati zadetek tudi brez popolnih vhodnih parametrov. V določenih primerih se lahko zgodi, da pomanjkanje enega izmed več parametrov ne vrne le enega zadetka. To sicer še ne predstavlja razloga za skrb. Lahko pa se zgodi, da se ta pojav ponovi pri vtičnikih, odvisnih od teh podatkov, kar posledično pomeni da količina podatkov začne naraščati eksponentno. Seveda se lahko zgodi tudi, da kljub temu, da imamo vse parametre podane, dobimo več rezultatov. Take rezultate dobimo, ko pri izvajanju poizvedb uporabljamo podatke, od katerih nobeden ne določa točnega rezultata. Primer takega po-

izvedovanja je iskanje telefonske številke za Janeza Novaka. Zelo verjetno je, da bomo dobili več rezultatov, saj obstaja več Janezov Novakov. Problem eksponentnega naraščanja zadetkov si bomo ogledali v naslednjem podpoglavju.

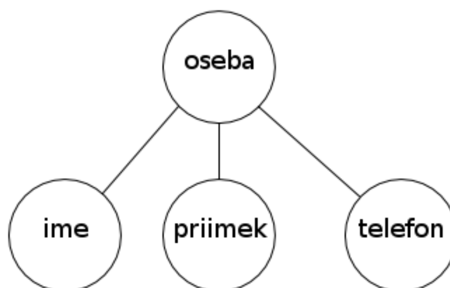
3.5 Časovna potratnost

Poleg kvalitete rezultatov uporabnika zanima tudi hitrost delovanja aplikacij. Uporabniki, ki potrebujejo rezultate v čim krajšem času, bodo zelo nezadovoljni, če bo aplikacija potrebovala veliko časa za iskanje podatkov. Aplikacija mora zagotavljati, da pridemo do dobrih rezultatov v sprejemljivem času. Pohitritev aplikacije se najlažje izvede na tistih mestih, kjer je ta časovno najbolj požrešna.

Izkaže se, da šibka točka aplikacije izvira ravno iz prednosti, ki jo nudijo računalniki - to je hitrost. Namreč, spletni boti, ki so jedro vtičnikov, lahko brskajo veliko hitreje od navadnih ljudi. Posledično se lahko zgodi, da nenamerno izvedemo DoS napad, saj strežniki virov morda pričakujejo le človeške uporabnike. Kot smo že povedali v poglavju 2, spletni pajki, ki spadajo med spletne bote, upoštevajo datoteko *robots.txt*, v kateri se lahko nahaja dodaten parameter, ki določa čas med dvema povezavama na spletno mesto. Ta lahko znaša tudi do sekunde, kar bi pomenilo, da vtičnik z desetimi paketi vhodnih podatkov devet sekund samo čaka, da lahko pošlje zahtevek. To velja v primeru, da za pridobitev rezultatov potrebujemo le en dostop, kar po navadi ne drži.

Iz povedanega sledi, da je potrebno minimizirati število dostopov do spletnih strani vira, saj so ti zelo dragi. V tem poglavju smo že omenili problem eksponentnega naraščanja podatkov, ki ga bomo razložili na primeru.

Poglejmo si shemo podatkov na sliki 3.4. Predstavljajmo si dva vtičnika, ki se izvedeta zaporedno. Prvi naj kot vhodni parameter prejme ime in vrne vse kombinacije tega imena in priimkov, ki jih najde na viru, kot je npr. Facebook. Drugi pa naj išče po telefonskem imeniku pripadajoče telefonske



Slika 3.4: Problem eksponentnega naraščanja zadetkov.

številke. Naj bo začetnemu vhodnemu parametru prvega vtičnika ime 'Janez'. Kot si lahko predstavljamo, bo prvi vtičnik vrnil kar nekaj rezultatov, ki ustrezajo iskalnemu pogoju. Na tem mestu je potrebno poudariti, da se pari z enakim imenom in priimkom ne zapišejo vsakič znova, ampak obstaja le en tovrsten zapis v datoteki. Zatem začne z delovanjem drugi vtičnik. Najverjetneje bo tudi ta vtičnik vrnil za marsikateri par vhodnih podatkov več rezultatov, kot smo že razložili na primeru telefonske številke 'Janeza Novaka'.

Za poenostavitev primera si zamislimo, da pri vsakem iskanju za določeno skupino vhodnih parametrov dobimo deset zadetkov. To število imenujmo razvejanost. Ker smo klicali dva vtičnika zaporedoma in začeli le z enim vhodnim parametrom, lahko izračunamo število zadetkov na sledeči način.

$$1 * 10 * 10 = 10^2 = 100 \tag{3.1}$$

Iz računa je razvidno, da je končno število zapisov sto. Vendar lahko uporabljamo poljubno število vtičnikov, prav tako tudi razvejanosti ne moremo določiti s konstanto. Če enačbo posplošimo z upoštevanjem zgornjih ugovorov pridemo do enačbe za povprečno število zapisov (označimo ga z Z), kjer je X povprečna razvejanost in n število vtičnikov.

$$Z = X^n \tag{3.2}$$

Torej, prehitro naraščanje zapisov lahko povzroči dolgotrajno delovanje

aplikacije. Zato je pri programiranju vtičnikov potrebno paziti na količino izhodnih podatkov, ki jih bo vtičnik vračal.

3.6 Preobremenjenost strežnikov

Večkrat smo že omenili problem strežnikov, ki ne predvidevajo tako hitrega brskanja, kot ga lahko izvedejo spletni pajki. Lahko se zgodi, da strežnik podleže prevelikemu številu zahtevkov in postane neodziven. Zaradi čiste vljudnosti lahko spletni pajki upoštevajo datoteko *robots.txt*, ki določa parametre z upoštevanjem katerih naj bi preprečili težave strežnikov. Kljub temu se administratorji spletnih strežnikov ne odločajo za uporabo te datoteke, prav tako ni obvezno upoštevanje teh navodil s strani spletnih pajkov.

Kljub temu lahko administratorji spletnih mest uporabijo določene protiukrepe, ki zmotijo delovanje spletnih pajkov. Določene spletne strani uporabljajo ob prepogostih dostopih istega klienta CAPTCHA [11] preverjanje. CAPTCHA je vrsta izziv-odgovor testa, ki naj bi zagotavljal, da ga uspe rešiti le človek.

Ta pa ni edini razlog, zakaj bi morali paziti na obremenjenost strežnika. Uporabnik programa, ki je povzročil neodzivnost strežnika, je lahko pravno preganjan kljub nenamernosti dejanja.

Poglavje 4

Semantično zajemanje podatkov

V poglavju 4 bomo opisali razvito aplikacijo za semantični zajem podatkov iz predefiniranih virov. Opis aplikacije bomo začeli z opisom osrednjega razreda, ki upravlja z vtičniki in XML dokumentom z rezultati. Predstavitev tega razreda bo potekala po dogodkih v časovnem zaporedju, kot se pojavijo med izvajanjem aplikacije. Začeli bomo z opisom zagona aplikacije. Razložili bomo, kako se ustvari seznam vtičnikov in katera opravila je potrebno opraviti pred začetkom poizvedovanja. Sledilo bo podpoglavje, namenjeno razlagi poteka iskanja vhodnih podatkov za vtičnike. Vtičniki prejmejo podatke v seznamu zgoščenih tabel. Vsaka zgoščena tabela vsebuje en zaokrožen paket vhodnih parametrov, ki so potrebni za eno poizvedbo. Nato si bomo ogledali še vstavljanje rezultatov poizvedb v XML dokument, ki je podobno iskanju vhodnih podatkov, kljub temu pa dovolj različno, da si zasluži svoje podpoglavje. Na koncu bomo opisali še pogoje za zaustavljenje aplikacije, s čimer bomo zaključili pregled osrednjega razreda.

Nadaljevali bomo s predstavitvijo vtičnikov. Pozornost bomo usmerili predvsem na osnovno delovanje, ki je skupno vsem oz. večini vtičnikom. Vsi vtičniki morajo implementirati vmesnik, ki zagotavlja komunikacijo z razredom za nadzor vtičnikov. Delovanje vtičnikov lahko dodatno determiniramo z

razvojem abstraktnih razredov, ki implementirajo ta vmesnik. Naloga razvijalcev konkretnih vtičnikov nato postane razširitev teh abstraktnih razredov. V našem primeru imamo le en abstraktni razred, ki razširja vmesnik. Lahko pa bi razvili več takih razredov. Vsak od njih bi se lahko prilegal določenim vrstam virov. Tako bi bilo pisanje vtičnikov precej olajšano.

4.1 Osrednji razred

4.1.1 Zagon aplikacije

Ob zagonu aplikacije se prebere XML datoteka, ki vsebuje podatke o vtičnikih, iz katere ustvarimo instance le-teh. Poglejmo si primer take XML datoteke.

```
<plugins>
        <plugin>
                org.plugin.ZacetniPlugin
        </plugin>
        <plugin>
                org.plugin.PrviPlugin
        </plugin>
        <plugin>
                org.plugin.ajpes.DrugiPlugin
        </plugin>
        <plugin>
                org.plugin.simobil.TretjiPlugin
        </plugin>
</plugins>
```

Vsi vtičniki razširjajo abstraktni razred *ASearchPlugin*, kateri implementira vmesnik *SearchPlugin*. Ta vmesnik omogoča, da ni potrebno spreminjati osrednjega razreda ob dodajanju novega vtičnika. Da bo vtičnik deloval, je potrebno le dodati podatek o lokaciji vtičnika v XML datoteko pod pogojem, da vtičnik implementira vmesnik. Abstraktni razred služi le kot osnova za

razvoj vtičnika, dejansko pa ni potrebno uporabljati tega razreda. Opisano arhitekturo aplikacije za semantični zajem podatkov iz predefiniranih virov si lahko ogledamo še na sliki 4.1.

Preden se začne pravo delo je potrebno še kreirati začetni XML dokument, ki ga bodo vtičniki prepoznali in ga znali uporabljati. Spodaj si oglejmo, kako izgleda prazen dokument.

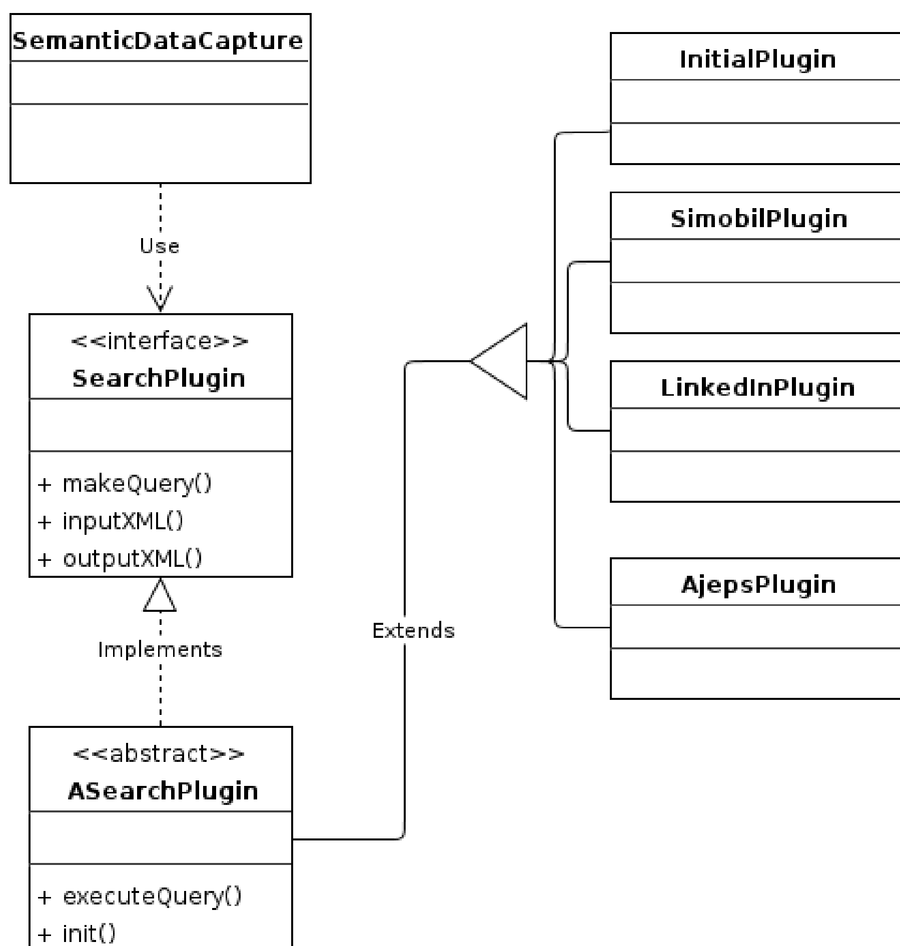
```
<data>
  <dataItem />
</data>
```

Korenski element se imenuje *data*. Ta vsebuje posamezne zapise, ki se imenujejo *dataItemi*. Vsebina samih *dataItemov* je določena porazdeljeno, znotraj neke zaokrožene celote vtičnikov. Vsak vtičnik prispeva delež definicije celotne podatkovne strukture izhodnega dokumenta.

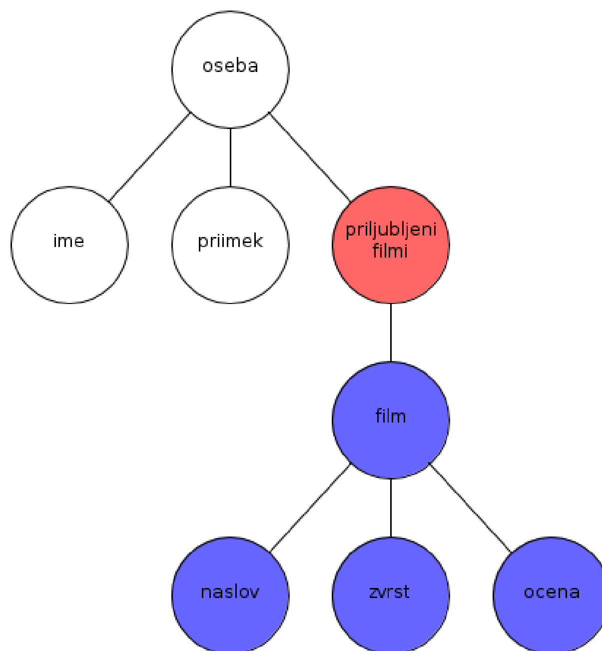
4.1.2 Iskanje začetnih podatkov

Pri naši aplikaciji je odgovornost za pridobivanje vhodnih podatkov dodeljena osrednjemu razredu. Zdi se upravičeno, da želimo vtičnikom posredovati podatke v čimbolj enostavni obliki. Tako bodo programerji vtičnikov imeli manj dela z upravljanjem vhodnih podatkov. Razvita aplikacija hrani podatke v XML dokumentu, ki ga ni priročno uporabljati kot parameter. Veliko priročneje je pošiljanje podatkov v obliki tabele Stringov ali zgoščenih tabel.

Kot je že bilo povedano, je shema podatkov definirana porazdeljeno, tako da vsak vtičnik prispeva del celotne sheme. Jasno je, da vsak vtičnik potrebuje definicijo vsaj dveh vrst podatkov: vhodnih in izhodnih. Vtičniki razvite aplikacije morajo vsebovati metodi, ki vračata shemi teh podatkov. Za zgradbo celotne sheme podatkov potrebujemo le shemo izhodnih podatkov pri vsakem vtičniku, saj se vhodni podatki opirajo na podatke, pridobljene iz ostalih vtičnikov. Če se pojavi potreba po tem, da se pri določenem vtičniku spremeni shemo izhodnih podatkov, to posledično pomeni, da je po-



Slika 4.1: Arhitektura aplikacije za semantični zajem podatkov iz predefiniranih virov.



Slika 4.2: Primer sheme podatkov.

trebno spremeniti vse vhodne sheme vtičnikov, ki uporabljajo podatke vsaj kakšnega premaknjenega elementa sheme. Zato je pri načrtovanju podatkovnih shem potrebno razmišljati vnaprej, da bo v prihodnosti potrebnih čim manj popravkov.

Algoritem za iskanje vhodnih podatkov mora znati uporabljati vsako veljavno shemo podatkov, ki jo dobi. Poleg tega je iz performančnega vidika pomembno, da izloči vhodne podatke, ki so že bili uporabljeni.

Poglejmo si od bliže, kako se izvaja iskanje vhodnih podatkov pri naši aplikaciji za semantični zajem podatkov na primeru iz slike 4.2.

Na sliki 4.2 je predstavljena shema podatkov, ki se rabi za zbiranje informaciji o osebah in filmih, ki so jim najljubši. Na raznih spletnih straneh se je mogoče prijaviti kot uporabnik in dodajati priljubljene filme. Naša aplikacija bi z uporabo vtičnikov najprej zbrala imena in priimke določenih oseb, nato pa bi z uporabo teh podatkov na eni izmed zgoraj omenjenih strani zbrala še podatke o filmih.

Preden nadaljujemo je potrebno razložiti vlogo elementa, ki je na sliki 4.2 označen z rdečo barvo. Ta element namreč služi kot zaboj, v katerem zbiramo v tem primeru podatke o filmih. Te elemente je v definiciji vhodnih in izhodnih shem potrebno označiti z atributom *container*, postavljenim na *true*, saj brez teh oznak aplikacija ne deluje pravilno. Primer vsebine veljavnega XML dokumenta, ki je enakovredna zgornji strukturi podatkov je:

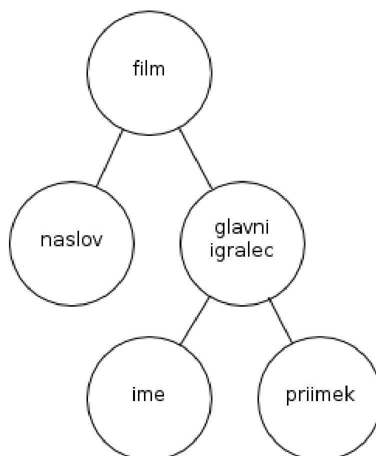
```
<data>
  <dataItem>
    <type>OSEBA</type>
    <ime />
    <priimek />
    <priljubljeniFilmi container=”true”>
      <film>
        <naslov />
        <zvrst />
        <ocena />
      </film>
    </priljubljeniFilmi>
  </ dataItem>
</ data>
```

Drevo na sliki 4.2 je prilagojeno bralcu, zato sta bila elementa *data* in *dataItem* odstranjena iz njega. XML dokument pa je primer dokumenta, ki bi lahko bil rezultat delovanja naše aplikacije. Kot vidimo ima element *priljubljeniFilmi* postavljen atribut *container*. Kot bomo kasneje razložili, to omogoča brskanje vseh kombinacij paketov vhodnih podatkov.

Poglejmo si od bliže kako poteka iskanje vhodnih podatkov. Na sliki 4.2 lahko vidimo vhodno in izhodno podatkovno strukturo vtičnika, ki zbira podatke o priljubljenih filmih oseb. Beli elementi pripadajo vhodni strukturi podatkov. Obarvani elementi predstavljajo podatke, ki jih vtičnik za zbiranje informacij o filmih vrne. Metoda za iskanje paketov vhodnih podatkov deluje kot rekurzivno pregledovanje v globino. Iskanje se začne v korenskem

elementu in se nadaljuje le po elementih, ki so vsebovani v vhodni ali izhodni shemi vtičnika. To optimizira delovanje metode. Implementacija iskanja vhodnih podatkov zagotavlja, da bo prvi list, ki ga metoda sreča izhodni, če ta obstaja. V našem primeru so to modri listi na sliki 4.2. Če aplikacija pride do takega elementa, prekine iskanje trenutnega paketa vhodnih podatkov in nadaljuje z naslednjim. Za pravilno delovanje aplikacije morata shemi podatkov vtičnika zadoščati določenim pogojem. Iz sheme podatkov mora biti razvidno, da enemu paketu izhodnih podatkov lahko pripišemo natanko en paket vhodnih podatkov. V nasprotnem primeru bi prisotnost enega izhodnega paketa podatkov izključila vse pakete vhodnih podatkov. Vtičnik v tem primeru ne bi izvajal poizvedb. Iz slike 4.2 vidimo, da so v našem primeru vsi izhodni paketi podatkov odvisni od istega paketa vhodnih podatkov, ker smo vse podatke o filmih dobili z istimi vhodnimi podatki. Za pravilno delovanje metode mora veljati tudi, da morajo biti elementi z atributom *container*, če jih je več, v shemi vhodnih podatkov vgnezdeni, saj sicer ne moremo dobiti vseh kombinacij vhodnih podatkov. Veja, v kateri je tak element ali gnezdo takih elementov, mora biti preiskana na koncu. Tako bodo preverjene vse veljavne kombinacije vhodnih podatkov.

Iskanje vhodnih podatkov na primeru iz slike 4.2 bi potekalo od desne proti levi. Metoda bi potovala prek elementa *priljubljeni filmi* do elementa *film*, če ta dva obstajata v XML dukumentu z rezultati. Do tega trenutka se metoda ne zaveda, da se nahaja v elementih, ki pripadajo izhodni shemi podatkov. To se zgodi šele, ko dostopa do prvega lista izhodne sheme podatkov. Zatem se metoda vrne do prvega elementa z atributom *container*. V našem primeru je to element *priljubljeni filmi*. Ker lahko ta element vsebuje več vej in ker metoda ne ve, da se nahaja v elementih izhodne podatkovne strukture, poskuša preveriti vse veje. Ker bo algoritem vedno naletel na list izhodne podatkovne strukture, bo preiskovanje v vejah elementa *priljubljeni filmi* neuspešno. Zaradi tega se bo algoritem vrnil na prvi naslednji element s postavljenim atributom *container*. V našem primeru pomeni, da bo preskočil na naslednjo osebo oz. bo zaključil z izvajanjem, če



Slika 4.3: Popravek sheme podatkov.

zbiramo podatke le o eni osebi.

V primeru, da preiskovani XML dokument ne vsebuje izhodnih podatkov, bo algoritem dostopil do elementov *priimek* in *ime* ter vsakega dodal v zgoščeno tabelo. Ob vsakem dodajanju se preveri prisotnost vseh potrebnih parametrov. Ko je preverjanje uspešno se zgoščena tabela doda na seznam parametrov.

S takim delovanjem algoritma poskušamo doseči vse kombinacije vhodnih podatkov, ki so veljavne. Shemo iz slike 4.2 popravimo s shemo na sliki 4.3 in dodamo tretji vtičnik, ki z naslovom filma pridobi podatke o glavnem igralcu (ime in priimek). Sedaj lahko vidimo uporabnost atributa *container*. Algoritem se bo v primeru, da so za določen film že pridobljeni podatki o glavnem igralcu, začel vračati do elementa *priljubljeni filmi*, ki ima postavljen atribut *container*. Tako bo film, katerega glavni igralec je že bil najden, izvzet iz seznama vhodnih parametrov. Če pa uporabnik ne bi označeval elementov z atributom *container*, bi bilo za algoritem nemogoče določiti element, pri katerem naj prekine vračanje.

Aplikacija tako detektira vse vhodne parametre brez rezultatov. Skupina podatkov, ki sestavljajo eno kombinacijo, pri kateri je možno izvesti iskanje, je shranjena v zgoščeni tabeli. Vtičniku pa se dejansko posreduje seznam

takih zgoščenih tabel.

4.1.3 Vstavljanje rezultatov iskanja

Ko dobimo vse možne vhodne parametre, lahko te posredujemo vtičniku. Vtičnik te podatke uporabi za pridobitev izhodnih podatkov. Delovanju vtičnikov se bomo posvetili v poglavju 4.2. Sedaj se raje posvetimo vstavljanju podatkov v XML dokument z rezultati.

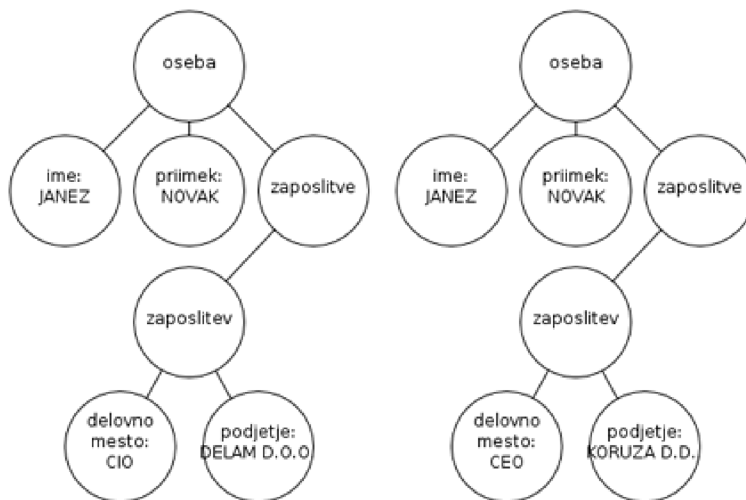
Aplikacija od vtičnika prejme seznam rezultatov v obliki zgoščenih tabel. V vsaki od teh zgoščenih tabel so zapisani tako izhodni kot vhodni podatki. Zadnji so potrebni, da vemo, na katero mesto zapisati rezultat.

Vstavljanje rezultatov se izvaja za vsako zgoščeno tabelo posebej. Najprej z uporabo podatkov v zgoščeni tabeli poskušamo pridobiti vse kandidate, ki se ujemajo z vhodnimi podatki v zgoščeni tabeli. Kandidati so celotni *dataItemi*, dejansko pa se prenašajo le njihovi naslovi in ne celotni objekti.

Čeprav se na prvi pogled zdi protislovno, se lahko zgodi, da ne dobimo nobenega kandidata. Tak primer se pojavi, ko vtičnik spremeni kakšen vhodni parameter. Primer, kjer se to pogosto dogaja, je klicanje vtičnikov s pomanjkljivimi podatki, kjer se poskuša dopolniti vhodne parametre iz rezultatov iskanja.

Samo iskanje kandidatov je podobno iskanju vhodnih podatkov, le da se v tem primeru ne išče le tistih kandidatov, ki jim manjkajo izhodni podatki. Ko so kandidati identificirani, se lahko začne vstavljanje podatkov. V primeru, da je seznam kandidatov prazen, se rezultat doda kot nov *dataItem*. V nasprotnem primeru pa se rezultati dodajajo vsakemu kandidatu posebej.

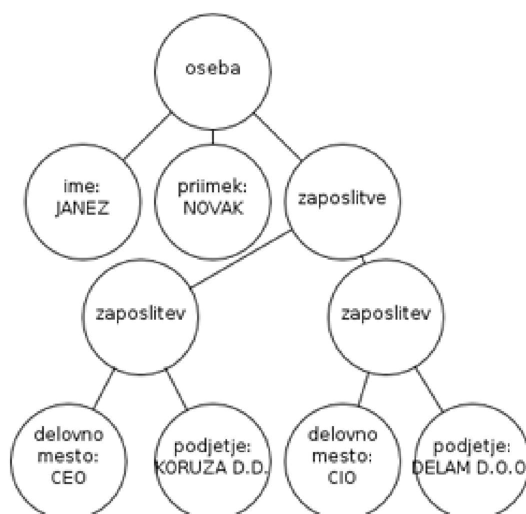
Dodajanje rezultatov je izvedeno kot združevanje dveh XML dokumentov. Pred začetkom združevanja je seveda potrebno iz zgoščene tabele zgraditi XML dokument. Nato se rekurzivno prebijamo po tem XML dokumentu in iščemo elemente v XML dokumentu z rezultati z enakimi vrednostmi. Vzemimo primer na sliki 4.4. Predstavljajmo si, da smo po izvedenem vtičniku dobili prikazane izhodne podatke. Vhodna podatka v vtičniku sta ime in priimek, ki pa ju nismo poznali, torej je prišlo do spremembe vhodnih



Slika 4.4: Primera rezultatov iskanja, ki se dodata v še prazen XML z rezultati.

parametrov med izvajanjem vtičnika, saj je ta prejel prazne vrednosti. XML dokument z rezultati naj ima prazna elementa *ime* in *priimek*. Ker za vhodne podatke $JANEZ\ NOVAK$ ne obstajajo kandidati, se prva *oseba* vstavi v celoti. Pri dodajanju druge *osebe* pa že obstaja kandidat za vhodne podatke $JANEZ\ NOVAK$. Ta je najden, nato pa se začne spajanje izhodnih podatkov z XML dokumentom z rezultati. Rekurzija poteka po elementih izhodnih podatkov. Ta XML dokument je sestavljen tako, da so vhodnim podatkom dodani izhodni. Pri preverjanju podatkov o imenu in priimku se podatki ujemajo, zato se vstavljanje nadaljuje, ampak že ob vstopu v element *delovnomesto* odkrijemo, da se podatki ne ujemajo z izhodnimi, se algoritem začne vračati do prvega elementa s postavljenim atributom *container*, ki je v našem primeru element *zaposlitve*. Metoda nato ugotovi, da v rezultatih ni več elementov *zaposlitev*, združevanje XML dokumentov pa še ni bilo zaključeno. V tem primeru se celotno drevesno strukturo izhodnih podatkov klonira in pripne na element *zaposlitve*. Spremenjeni *dataItem* bo po končanem dodajanju izhodnih podatkov izgledal kot na sliki 4.5

Da metoda pravilno deluje morajo najprej biti preverjeni vsi vhodni po-



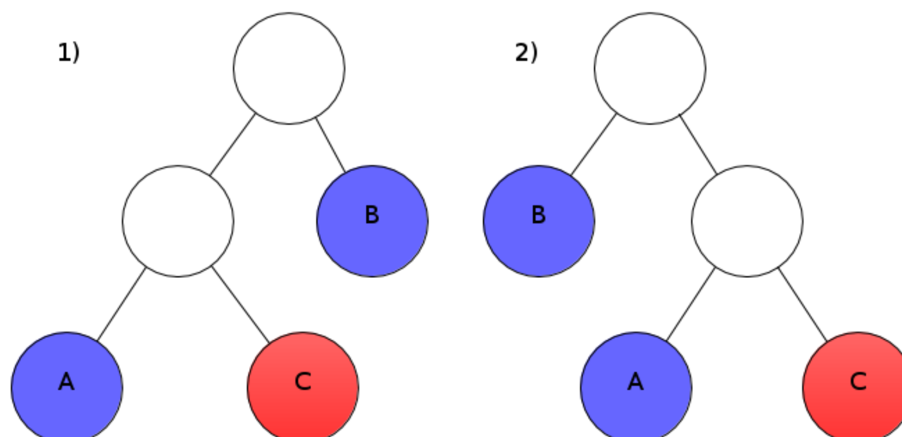
Slika 4.5: Novi *dataItem* po uporabi podatkov iz slike 4.4.

datki in šele nato lahko začnemo dodajati izhodne. Metoda zagotavlja le, da bo prvi pregledan list vhodni. Če sta strukturi vhodnih in izhodnih podatkov pravilno oblikovani pa bodo najprej pregledani vsi vhodni podatki. Tako podatkovno strukturo zagotovimo s pravilno strukturo in pravilnim vrstnim redom vstavljanja elementov.

Poglejmo si primer s slike 4.6. Obe strukturi sta si, kar se vsebine tiče, enakovredni. Modra lista A in B predstavljata vhodne podatke, rdeči list C pa predstavlja izhodni podatek. Algoritem dodajanja rezultatov deluje po algoritmu pregledovanja v globino. Branje poteka od leve proti desni. Pri prvi podatkovni strkuturi bo algoritem za vstavljanje podatkov naletel najprej na element A, nato pa vstavil element C. Zadnji bo na vrsto prišel element B, kjer lahko metoda ugotovi, da vstavljeni podatki ne pripadajo vhodnim. Vstavljeni element C bo tako zapisan na napačnem mestu.

V drugem primeru bo algoritem najprej preveril elementa A in B. V primeru, da oba sovpadata izhodnim podatkom, bo vstavljen element C. Če kateri od njiju nima pravilne vrednosti se vstavljanje elementa C ne bo izvedlo.

Pomembno je tudi, da se ob spremembi vhodnih podatkov starejši podatki



Slika 4.6: Neveljavna in veljavna podatkovna struktura.

ne izgubijo. V primeru, da temu ne bi bilo tako, bi verjetno ob različnih vrstnih redih zagona vtičnikov dobili različne rezultate. Kot primer vzemimo dva vtičnika, ki oba kot vhodne podatke potrebujeta ime in priimek osebe. Ob primeru, ko podamo le *priimek* z vrednostjo NOVAK, bosta vtičnika poleg izhodnih podatkov dopolnila tudi vhodni podatek - *ime*. Prvi naj dobi podatke JANEZA in GREGORJA NOVAKA, drugi pa FILIPA in JANEZA NOVAKA. Na začetku XML dokument zgleda takole:

```
<data>
  <dataItem>
    <ime />
    <priimek>NOVAK</ priimek>
  </ dataItem>
</ data>
```

Če se sedaj izvedeta vtičnika prvi za drugim in se po klicu prvega vtičnika začetni podatki izgubijo bodo med rezultati samo dva zadetka (*JANEZ NOVAK* in *GREGOR NOVAK*). Če pa le zamenjamo vrstni red zagona vtičnikov dobimo druga dva zadetka, torej *JANEZA NOVAKA* in *FILIPA NOVAKA*.

V obeh primerih so zadetki odvisni od vrstnega reda zagona vtičnikov,

kar seveda ni priročno, saj se nam lahko iskani podatki na tak način izmuznejo. Da se to ne bi dogajalo, je potrebno ob vsakem spreminjanju vhodnih podatkov narediti nov *dataItem* in ne spreminjati že vpisanih podatkov.

4.1.4 Zaustavljanje aplikacije

Zgoraj smo opisali delujočo aplikacijo za semantično zbiranje podatkov iz predefiniranih virov. Česar še nismo povedali je, kako lahko aplikacija zazna, da lahko zaključi z delom.

Če naredimo povzetek delovanja aplikacije, lahko ugotovimo, da se vtičniki kličejo v zanki. Eden za drugim se po zaporedju, v katerem so našteti, izvedejo, in nato vrnejo rezultate. Aplikacija poskuša vstaviti prejete podatke, vendar ne hrani nobenega podatka o spreminjanju dokumenta. Zato se po vstavljanju podatkov izvede še preverjanje XML dokumenta pred vstavljanjem in po vstavljanju. Ker se pri vstavljanju podatkov nikoli ne briše, je dovolj, da preverimo, če so vsi elementi v novem dokumentu tudi v starem. Če temu ni tako, pomeni, da je bil dokument spremenjen. Posledično obstaja možnost, da bo kateri od vtičnikov uporabil novi podatek, zaradi česar se bo po klicu vseh vtičnikov do konca seznama vtičnikov ponovil klic vseh vtičnikov od prvega do zadnjega.

4.2 Vtičniki

4.2.1 Osnovno delovanje

Delovanje vtičnikov je v določeni meri vnaprej določeno na različnih ravneh. Na najosnovnejši ravni je določena metoda, ki mora biti implementirana, da bo vtičnik veljaven. Ta pogoj je vsiljen s tem, da morajo vsi vtičniki implementirati vmesnik *SearchPlugin*. Poleg samega predpisovanja izgleda vtičnikov omogoča tudi dinamično kreiranje instanc vtičnikov, kar je posebej priročno pri dodajanju novih vtičnikov. Ta razred predvideva implementacijo treh metod, ki so:

- public abstract ArrayList<HashMap<String,String>>makeQuery (ArrayList<HashMap<String,String>>dataList);
- public abstract Element inputXML ();
- public abstract Element outputXML();

Zadnji dve metodi vračata vhodno in izhodno strukturo podatkov, ki ju potrebuje aplikacija pri delovanju. XML strukturo je potrebno zgraditi od elementa *dataItem* naprej. Elemente, ki lahko vsebujejo več istovrstnih podatkov, je potrebno označiti z atributom *container*, postavljenim na vrednost "true". Aplikacija predvideva, da so le elementi v listih nosilci podatkov. Dodaten element, ki se v strukturi še predvideva, je *type*. Ta element se uporablja le za ločevanje različnih *dataItemov*, npr. če bi obstajale dve vrsti *dataItemov* v našem XML dokumentu. Element *type* je potrebno dodati le v izhodni strukturi, lahko pa je podan v obeh, če npr. hočemo specificirati, da naj aplikacija išče vhodne podatke le po določenih tipih *dataItemov*. XML strukture in vsi XML dokumenti so zgrajeni z objekti *Element* iz knjižnice, ki jo ponuja JDOM [13].

Prva metoda na zgornjem seznamu je osrčje vtičnikov. Ta metoda se kliče iz aplikacije in mora poskrbeti, da vrne vse dobljene rezultate. Kot lahko opazimo, potrebuje za parameter seznam zgoščenih tabel, kjer so ključi in vrednosti Stringi. Vsaka zgoščena tabela predstavlja en paket vhodnih podatkov za izvajanje poizvedb po viru. Metoda kasneje vrne seznam zgoščenih tabel z dodanimi izhodnimi podatki, ki jih aplikacija uporabi pri nadaljnih poizvedbah.

Vtičnike lahko izdelujemo z uporabo vmesnika *SearchPlugin*. Če pa hočemo več podpore ogrodja aplikacije, lahko razširimo abstrakten razred *ASearchPlugin*, ki predstavlja naslednjo raven v definiciji vtičnikov. Ta razred vsebuje sledeče abstraktne metode:

- public abstract void init();
- public abstract ArrayList<HashMap<String,String>>executeQuery (ArrayList<HashMap<String,String>>dataList);

Prva metoda je lahko tudi prazna. Tukaj ima programer vtičnika možnost postoriti določene stvari takoj po kreiranju instance razreda. Določeni viri, na primer, zahtevajo registracijo z uporabniškim imenom in geslom. Zaradi občutljivosti samih podatkov aplikacija omogoča, da se uporabniška imena in gesla prebere iz zunanje XML datoteke, ki izgleda tako kot je prikazano spodaj.

```
<login-data>
        <ime-spletnega-vira>
                <uporabnik>
                        tinezvon
                </uporabnik>
                <geslo>
                        geslo123
                </geslo>
        </ime-spletnega-vira>
</login-data>
```

Edini obvezni element je $login-data$. Ostali so lahko poimenovani poljubno, čeprav veljajo neka pravila v strukturi tega dokumenta. Najprej, v korenskem elementu se nahajajo elementi, ki se navezujejo na posamezen vir, pri čemer ima lahko vsak vir poljubno število takih elementov. V teh elementih se lahko nahajajo le listi XML dokumenta, kjer so podane vse vrednosti v poljubno poimenovanih elementih. Načeloma lahko uporabnik shrani kateri koli podatek, ne le uporabniških imen in gesel.

Naslednja tipična stvar, ki jo lahko opravimo v metodi *init*, je prevajanje imen parametrov. Namreč, zelo verjetno se bo zgodilo, da se ime parametra v notranjem XML dokumentu ne ujema z imenom parametra, ki ga uporabljamo pri GET ali POST metodah. Abstraktni razred vsebuje dve tabeli Stringov, ki sta:

- protected String[] inputParameters;
- protected String[] translatedParameters;

V tabelo *inputParameters* lahko zapišemo imena spremenljivk v notranjem XML dokumentu, medtem ko v tabeli *translateParameters* vpišemo prevedena imena spremenljivk tako, da imena parametrov in njihovi prevodi ležijo na istih pozicijah v tabelah. Ko to storimo, potem lahko kličemo metodo *getParametersInString*, ki prejme kot parameter zgoščeno tabelo s podatki in kodiranje, v kataerem želimo, da se parametri zapišejo.

Zadnja metoda na seznamu se imenuje *executeQuery*, ki že po izgledu spominja na metodo *makeQuery* iz razreda *SearchPlugin*. Ne samo, da ta metoda prejme in vrača skoraj enake podatkovne strukture, tudi njena vloga je podobna. V razredu *ASearchPlugin* je metoda *makeQuery* že implementirana. Znotraj nje se kliče metoda *executeQuery*, ki kot parameter prejme le en paket vhodnih podatkov. Seveda lahko kljub temu vrne več paketov izhodnih podatkov, kot smo že omenili v prejšnjih poglavjih. Z uporabo abstraktnega razreda *ASearchPlugin* si torej lahko še dodatno olajšamo programiranje vtičnikov.

Naslednji korak v tej smeri lahko dosežemo z uporabo zunanjih knjižnic oz. razredov za pomoč. Za potrebe aplikacije je bil izdelan razred *LoginHelp*, ki je v podporo pri inicializaciji klientov in registraciji na strežnikovi strani.

Veliko informacij je trenutno dostopnih na raznih socialnih omrežjih, kjer uporabniki sami dopolnjujejo informacije o sebi. Zradi tega se v aplikaciji uporablja Scribe [5], ki je OAuth [6] knjižnica za Java programski jezik. Razvil jo je Pablo Fernandez in naj bi delovala z vsemi APIji. Aplikacija trenutno podpira povezovanje na Facebook, LinkedIn, Google, Twitter in Yahoo APIje.

Poglavje 5

Evaluacija

V poglavju 5 bomo prikazali delovanje razvite aplikacije za semantični zajem podatkov iz predefiniranih virov na resničnem primeru. Najprej bodo predstavljeni realizirani vtičniki. Preučili bomo pogoje uporabe, ki so dostopni na spletni strani vira, povedali bomo, katere podatke vtičniki sprejmejo in katere oddajo, ter opozorili na posebnosti vtičnika, če jih ta ima.

Drugi del poglavja je namenjen predstavitvi rezultatov poganjanja naše aplikacije. Predvsem nas bo zanimala performančna plat delovanja aplikacije. S to usmeritvijo bomo merili različne čase. Nato bomo te čase analizirali in poskušali priti do zaključkov o možnih razlogih za pridobljene rezultate.

5.1 Predefinirani viri

Spodaj bomo preleteli vse štiri vtičnike, ki so bili razviti v sklopu tega diplomskega dela. Vtičnike bomo preučili od enostavnejših proti zahtevnejšim in pri vsakem opozorili na kaj velja biti pozoren, ko razvijate podoben vtičnik. Vsi opisani vtičniki razširjajo razred *ASearchPlugin*. Da bo predstava, kaj vtičniki sploh delajo jasnejša, si oglejmo diagram sheme podatkov na sliki 5.1. Shema se nadaljuje od elementa *dataItem* naprej, torej je izvzet koren XML dokumenta *data*. Opazimo lahko, da zbiramo splošne podatke o osebah in podjetjih, kjer so te osebe zaposlene. Elementi s postavljenim *container*

atributom so obarvani rdeče.

5.1.1 Začetni vtičnik

Pri zagonu aplikacije moramo imeti podane začetne podatke, katere lahko vtičniki uporabljajo. Zaradi ponovne uporabe obstoječe arhitekture je zvenelo logično razviti vtičnik za zajem začetnih podatkov. Sam vtičnik je lahko postavljen na katero koli mesto, saj bodo zaradi delovanja aplikacije predhodni vtičniki brez vhodnih podatkov in se zato ne bodo izvedli.

Branje vhodnih podatkov z vtičniki omogoča, da imamo lahko poljubno število takih vtičnikov, ki se na primer zaženejo, ko se pojavi potreba po tem. Poleg tega je tak način odpornejši na napake, kot če bi recimo uporabniki sami vpisovali začetne podatke v XML datoteko.

Vhodna shema našega začetega vtičnika je sledeča:

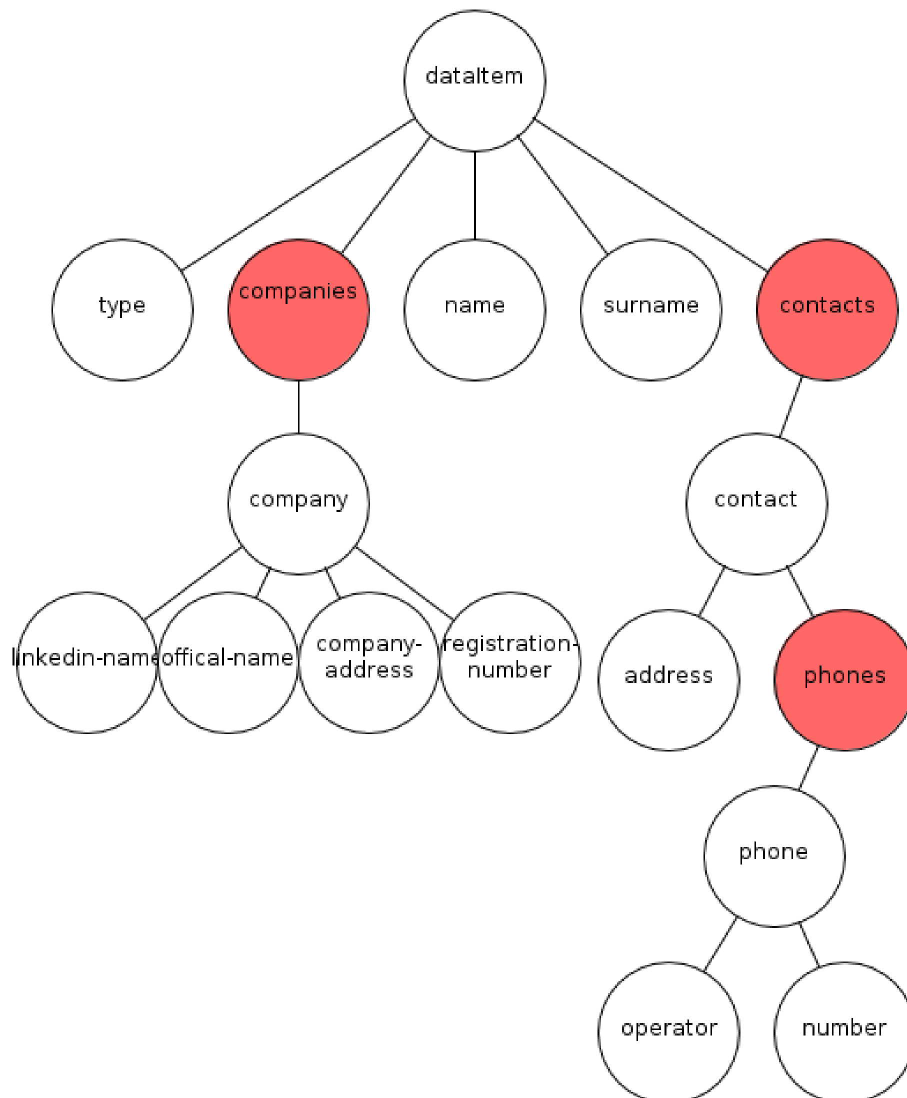
```
<data>
        <dataItem />
</data>
```

To sovpada z začetnim stanjem XML dokumenta, kreiranim ob zagonu aplikacije. Torej aplikacija kliče vtičnik samo, ko obstaja *dataItem*, podoben zgornjemu, in ki nima podanih izhodnih podatkov tega vtičnika. Vtičnik nas nato vpraša po *imenu* in *priimku*, *tip* dataItema pa dopolni sam in vse skupaj vrne kot izhodne podatke.

Delovanje tega vtičnika je precej enostavno, zato si poglejmo ostale, resnejše vtičnike.

5.1.2 Si.mobil

Nadaljujemo z vtičnikom za Si.mobilov [14] telefonski imenik. Ta prejme za vhodne parametre ime in priimek ter vrne celoten kontakt, ki vključuje naslov, telefonsko številko in operaterja. Preden začnemo delati z virom se moramo prepričati o legalnosti početja, kar lahko storimo na Si.mobilovi spletni strani. Med posebnimi pogoji uporabe spletne strani ni prepovedi



Slika 5.1: Celotna shema podatkov razvite aplikacije.

uporabe spletnih pajkov dokler ne zlorabljamo podatkov, pridobljenih na tak način. Ker teh podatkov ne shranjujemo na lokalnem disku in ker jih ne posredujemo naprej je torej dejanje v skladu s pogoji uporabe.

Vtičnik je sam po sebi zelo enostaven. Vir ne zahteva registracije, obrazec za iskanje pa je zelo preprost. Parametra ime in priimek združimo s presledkom in nato posredujemo skupaj z drugimi predpisanimi parametri strežniku, ki odgovori s seznamom zadetkov. Iz tega seznama lahko že takoj izluščimo rezultate. Če število zadetkov presega omejitev na stran, moramo pregledati še ostale zadetke, tako da postopoma večamo parameter, ki določa številko strani.

Težavni so zadetki, ki vsebujejo več priimkov ali imen, saj je programsko težko ločiti eno od drugega, zato so pri tem delu možne napake in nepričakovani rezultati. Program predvideva, da je prva beseda v nazivu ime, ostalo pa je del priimka.

5.1.3 LinkedIn

Vtičnik za LinkedIn [15] uporablja API. Vhodna parametra za vtičnik sta ime in priimek, izhod vtičnika pa je naziv podjetja, pri katerem je oseba z določenim imenom in priimkom zaposlena. Na sliki 5.1 je izhodni element označen z imenom $linkedin - name$. Seveda se pojavi problem, če je oseba zaposlena na dveh ali več podjetjih. Podatki so shranjeni kot nizi v zgoščeni tabeli in predvidevajo le eno vrednost na parameter. V takem primeru se ustvari več paketov izhodnih podatkov, vsak paket za eno podjetje.

V povezavi s prejšnjim vtičnikom je jasno, da nikakor ne moremo neposredno povezati osebe, ki je bila najdena v Si.mobilovem telefonskem imeniku z osebo, ki je bila najdena z LinkedInovim APIjem. Kljub temu bodo rezultati zapisani v istem *dataItemu*. Odločitev o povezavi med podatki je prepuščena uporabniku aplikacije.

Uporaba APIja precej olajša programiranje vtičnika, saj se da preko URL zahtevka določiti strukturo podatkov, ki bo vrnjena. Podatki so posredovani v XML obliki in se vračajo v skupinah po 25 zadetkov, zato je možno, da je

potrebno izvesti več klicev APIja.

5.1.4 Ajpes

Vtičnik za Ajpes [16] je precej zahtevnejši od prejšnjih. Ob inicializaciji vtičnika se je potrebno prijaviti na strežniku z uporabniškim imenom in geslom. Pri tem opravilu najdemo pomoč v razredu *LoginHelp*, ki uporablja razrede spletnega pajka Apache Nutch za vzpostavitev varne povezave.

Ob klicu vtičnika mu posredujemo imena podjetij, katerih informacije želimo pridobiti. Po izvedeni poizvedbi lahko dobimo večje število zadetkov. Če bomo vrnili vse zadetke, bo ostal veljaven le zadnji, saj bodo ostali prepisani, zato ker vhodni podatki ostanejo vedno enaki. V iskanju pravilnega zadetka pa naletimo na določene težave, ker se imena podjetij na LinkedIn omrežju redko kdaj ujemajo z imeni podjetij na Ajpesovi spletni strani. Vtičnik za Ajpes zato vrne prvi zadetek, ki zadošča določenim pogojem. Na Ajpesovi spletni strani sta dostopna dva naziva podjetja, celotno ime in kratko ime. Torej se mora vhodni parameter vtičnika ujemati z vsaj enim od nazivov. Predhodno so vsem nazivom odstranjene vse oznake vrste družbe, ki si podjetje lasti (npr. *s.p.*). Ko dobimo zadetek, preberemo naziv podjetja, naslov in matično številko podjetja. Ti podatki se nato vrnejo aplikaciji kot rezultat iskanja.

5.2 Primer iskanja

Delovanje aplikacije za semantični zajem podatkov iz predefiniranih virov bomo prikazali na primeru, ki bo izveden kar na avtorju diplomskega dela. Torej začetni podatki bodo *ALAN RIJAVEC*. Zaporedje vtičnikov po katerem bodo klicani je sledeče:

1. začetni vtičnik,

2. LinkedIn,

3. Ajpes,
4. Si.mobil.

Vsak vtičnik bo ob prvem zagonu dobil vse potrebne podatke, tako da bodo po prvi iteraciji pridobljeni že vsi podatki. Kljub temu se bo izvedla druga iteracija, v kateri pa se ne bo izvedla nobena poizvedba zaradi pomanjkanja novih vhodnih podatkov. Vtičnik za vpis vhodnih podatkov bo spremenjen tako, da bo avtomatsko odgovoril s pravilnimi podatki. V nasprotnem primeru bi v izvajalni čas všteli tudi čas, ki ga uporabnik potrebuje za vnos iskalnih podatkov.

Za analizo delovanja aplikacije bomo izmerili tudi čase, kjer se program največ zadržuje. Časi, ki jih bomo merili so:

- čas izvajanja celotnega programa,
- skupen čas izvajanja vtičnikov,
- čas, ko vtičniki čakajo,
- čas za prenašanje dokumenta h klientu.

Razlika med časom izvajanja celotnega programa in skupnim časom izvajanja vtičnikov je večinoma čas, ki je potreben za iskanje vhodnih parametrov in dodajanje rezultatov v XML dokument, medtem ko je razlika med skupnim časom izvajanja vtičnikov in časom, ko vtičniki stojijo, čas, ki je potreben za prejemanje in obdelovanje prejetega dokumenta, ki ga je strežnik vira poslal. Ker je čas prejemanja dokumenta verjetno eden izmed daljših bomo izmerili tudi tega. Čas prejemanja dokumenta lahko nekoliko zmanjšamo z večjo pasovno širino povezave. Ne moremo pa ga znižati pod določeno mejo in ne moremo ga predvideti, saj je za zamudo lahko kriva obremenjenost strežnika z drugimi zahtevami.

S temi podatki lahko dobro ocenimo, kje se program največ časa zadržuje in kje lahko najbolj pohitrimo izvajanje celotne aplikacije. Preden pogledamo v rezultate iskanja pa poskusimo predvideti razmerja časov. Zaradi

razmeroma velikih premorov med pošiljanjem zahtev na isti strežnik je vsak dodaten klic izjemno drag, zato lahko upravičeno pričakujemo, da bo aplikacija večinoma čakala na pošiljanje nove zahteve. Ostali časi bi morali biti proti temu neznatni, če odmislimo nalaganje strani, ki je vedno odvisno tudi od obremenjenosti strežnika.

Poizkus je bil izveden desetkrat. Za vsakega izmed poizkusov so bili izmerjeni vsi časi in aplikacija se je izvedla po pričakovanjih. Končen XML dokument izgleda takole:


```
<data>
  <dataItem>
    <name>ALAN</name>
    <surname>RIJAVEC</surname>
    <type>PERSON</type>
    <companies container="true">
      <company>
        <linkedin-name>ISKRA AVTOELEKTRIKA</linkedin-name>
        <offical-name>ISKRA AVTOELEKTRIKA D.D.</offical-name>
        <company-address>POLJE 15, 5290 SEMPETER PRI GORICI</company-addr
        <registration-number>XXXXXXXXXX</registration-number>
      </company>
    </companies>
    <contacts container="true">
      <contact>
        <address>OSEVLJEK 20, 5292 RENCE</address>
        <phones container="true">
          <phone>
            <operator>SIMOBIL</operator>
            <number>040XXXXXX</number>
          </phone>
        </phones>
      </contact>
```

	Povprečen čas (v sekundah)
Čas izvajanja celotnega programa	8,48
Skupen čas izvajanja vtičnikov	6,8
Čas čakanja	2,0
Čas prenašanja spletnih strani	3,8

```
    </contacts>
  </dataItem>
</data>
```

Nekateri podatki v XML dokumentu so zaradi občutljivosti skriti. Vsak vtičnik je uporabil natanko en paket vhodnih podatkov in našel natanko en zadetek, zato je njihov vpliv na končne čase enak. Rezultati, ki sledijo, so povprečne vrednosti desetih meritev.

Čas izvajanja celotnega programa znaša 8,48 sekunde. Skupen čas izvajanja vtičnikov pa 6,8 sekunde. Hitro lahko opazimo, da večino časa pri izvajanju aplikacije porabijo vtičniki za pridobivanje podatkov. Ta čas znaša 3,8 sekunde. Čas prenašanja dokumentov je skoraj dvakrat daljši od časa čakanja, oba skupaj pa zasegata veliko večino porabljenega časa, tj. skoraj 70Na tak način bi lahko delovali drugi vtičniki, medtem ko bi bil določen vtičnik v stanju mirovanja. Za tako delovanje bi morali globlje poseči v delovanje aplikacije, saj bi morali vtičniki delovati kot niti.

Če od časa delovanja aplikacije odštejemo seštevek časa izvajanja vseh vtičnikov, dobimo 1,68 sekunde. V ta čas je všteto inicializiranje vseh vtičnikov. Ne smemo pozabiti, da se med kreacijo instanc vtičnikov inicializirajo tudi klienti vtičnikov in se prijavijo na strežnike, kjer je potrebno. V našem primeru se je potrebno prijaviti le na Ajpesov strežnik. Določeno količino časa lahko torej pripišemo tem opravilom.

Poglavje 6

Zaključek

V diplomskem delu smo si najprej ogledali sorodne aplikacije in na kaj moramo biti pazljivi pri njihovi vestni uporabi. Tukaj smo se izmed sorodnih aplikacij posvetili agregatorjem in spletnim pajkom ter predstavili delovanje nekaterih bolj znanih predstavnikov vsakega.

V nadaljevanju smo preleteli težave oz. izzive pri izdelovanju aplikacije za semantični zajem podatkov iz predefiniranih virov. Kot zelo pomembno točko oblikovanja naše aplikacije, smo izpostavili izbiro arhitekture. Uporabljena je arhitektura vtičnikov, ki omogočajo hitrejšo in enostavnejšo razširitev aplikacije. Instance vtičnikov so kreirane dinamično. To omogoča, da bi bilo z manjšimi spremembami v ogrodju aplikacije možno dodajati vtičnike med njenim delovanjem. Ta lastnost je zelo priročna, ko je vsako vstavljanje aplikacije izjemno drago. Vsi razviti vtičniki razširjajo isti abstraktni razred. Naslednji korak pri izboljšanju aplikacije bi lahko bil razvoj raznih abstraktnih razredov. Ti naj še dodatno olajšajo razvoj vtičnikov. Abstraktni razredi so lahko optimizirani za posamezne vrste virov, kot so navadne spletne strani ali APIji, ali pa so bolj splošno naravnani. Zelo verjetno je, da bodo vtičniki za specifične vrste virov lažji za implementacijo, zato je mogoče to boljša izbira, čeprav zahteva več dela pri razvoju ogrodja aplikacije. V primeru, da se odločimo za bolj splošne rešitve, lahko olajšamo implementacijo z razredi, ki ponujajo podporo pri posameznih opravilih, kot je npr. prijavljanje na

stran.

V jedru diplomskega dela smo si podrobneje ogledali delovanje izdelane aplikacije. V povezavi s problemi z obremenjenostjo strežnikov ponujamo način za pohitritev delovanja aplikacije. Ta bi delovala hitreje, če bi vtičniki oddajali pridobljene podatke sproti za vsak odgovor strežnika in bi lahko bilo pognanih več vtičnikov hkrati. Tako bi bolje izkoristili pasovno širino povezave in zmanjšali vpliv čakanja med dvemi klici na isti strežnik.

Za zaključek smo še pogledali primer delovanja razvite aplikacije, ki je bil pognan za avtorja tega diplomskega dela. S pridobljenimi podatki smo ugotovili, da je največ možnosti za izboljšavo aplikacije ravno pri čakanju vtičnikov na pošiljanje nove zahteve. Preprosto znižanje časa čakanja nas ne privede daleč, saj lahko povzročimo težave strežnikom. Rešitve gre iskati v vzporednem izvajanju vtičnikov in večanjem procesiranja prejetih dokumentov na račun manjšega števila potrebnih klicev strežnika.

Literatura

[1] J. Heaton, “Programming Spiders, Bots and Aggregators in Java”, Sybex, 2002

[2] C. A. Mattman, J. l. Zitting, “Tika in action”, Manning, 2011

[3] Apache Tika projekt. Dostopno na:
http://tika.apache.org/

[4] Apache Nutch projekt. Dostopno na:
http://nutch.apache.org/

[5] Scribe - knjižnica za delo z APIji. Dostopno na:
https://github.com/fernandezpablo85/scribe-java

[6] OAuth - standard za avtorizacijo. Dostopno na:
http://oauth.net/

[7] Under The Hood: Google News & Ranking Stories. Dostopno na:
http://searchengineland.com/google-news-ranking-stories-30424

[8] Internet bot (Wikipedia) (2012). Dostopno na:
http://en.wikipedia.org/wiki/Internet_bot

[9] Robots exclusion standard - protokol (Wikipedia) (2012). Dostopno na:
http://en.wikipedia.org/wiki/Robots_exclusion_standard

[10] HTTrack aplikacija. Dostopno na:
http://www.httrack.com/

[11] CAPTCHA test (Wikipedia) (2012). Dostopno na:
http://en.wikipedia.org/wiki/CAPTCHA

[12] Globoki splet - definicija (Wikipedia) (2012). Dostopno na:
http://en.wikipedia.org/wiki/Deep_Web

[13] JDOM knjižnica za delo z XML dokumenti. Dostopno na:
http://www.jdom.org/

[14] Si.mobil - vir podatov. Dostopno na:
http://www.simobil.si/

[15] LinkedIn - vir podatkov. Dostopno na:
http://www.linkedin.com/

[16] Ajpes - vir podatkov. Dostopno na:
http://www.ajpes.si/