Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Matek

Delo je pripravljeno v skladu s Pravilnikom o podeljevanju Prešernovih nagrad študentom, pod mentorstvom doc. dr. Dejana Lavbiča

Adaptivni sistem za učenje jezika SQL

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana 2015

Povzetek

Dandanes je na seznamu ključnih kompetenc vsakega razvijalca programske opreme vsekakor moč zaslediti tudi poznavanje jezika SQL. Poglobljeno znanje pisanja poizvedb in obvladovanja osnovnih konceptov je ne le zaželeno, temveč pričakovano. Proces učenja jezika SQL študentom povzroča vrsto preglavic. V diplomskem delu smo se lotili lajšanja učenja pisanja poizvedb z izdelavo adaptivnega sistema za generiranje namigov. Sistem temelji na množici zgodovinskih podatkov preteklih poskusov reševanja tovrstnih nalog. Za inteligentno rabo znanja, skritega v podatkih, je bil uporabljen Markovski odločitveni proces, ki omogoča napovedovanje v negotovih razmerah. Poleg pametnega agenta smo razvili tudi komponento za procesiranje samega jezika SQL ter preprost spletni vmesnik. Rezultati testiranj so pokazali, da je razvit sistem zmožen ponuditi uporabne namige, prilagojene posameznemu študentu, in da obenem predstavlja dobro osnovo za nadaljnji razvoj na tem področju.

Ključne besede: inteligentni sistemi za učenje, učenje jezika SQL, Markovski odločitveni proces, procesiranje jezika, strojno učenje, priporočilni sistemi.

Abstract

Nowadays one can identify SQL proficiency as a key competence of any software developer. Broad knowledge of writing correct and efficient SQL queries is not only desired but also required. The process of SQL learning turns out to be anything but straightforward. With this thesis we have set to reduce the effort needed to learn SQL by developing a robust system for hint generation. The system is based on a set of historical data which represent past attempts at solving SQL related exercises. In order to use the knowledge hidden within such data, we have have used Markov decision processes which enable us to make predictions under uncertain circumstances. Next to the agent we have also developed a way to process SQL language and a simple web-based interface. Evaluation has shown, that the system is capable of offering useful hints which are tailored to individual students. We agree, that the system represents a solid foundation for future work in this field.

Keywords: intelligent tutoring systems, SQL learning, Markov decision process, language recognition, machine learning, recommendation systems.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CGI	Common Gateway Interface	skupni prehodni vmesnik
CRUD	Create, Read, Update, Delete	ustvari, beri, posodobi, briši
DFA	Deterministic Finite State Automaton	deterministično končen avtomat
ERD	Entity Relation Diagram	entitetno relacijski diagram
ITS	Intelligent Tutoring Systems	inteligentni sistemi za učenje
JAX-RS	Java API for RESTful services	Java API za spletne storitve REST
JSON	Javascript Object Notation	notacija objektov Javascript
MDP	Markov Decision Process	Markovski odločitveni proces
MVC	Model-view-controller	model, pogled, krmilnik
NFA	Nondeterministic Finite State Automaton	nedeterministično končen avtomat
REST	Representational State Transfer	reprezentacijski prenos stanja
SOAP	Simple Object Access Protokol	protokol za dostop do enostavnih objektov
XML	Extensible Markup Language	razširljiv označevalni jezik

Poglavje 1

Uvod

Jezik SQL (angl. Structured Query Language) je skozi leta postal "de-facto" standard za poizvedovanje po relacijskih podatkovnih bazah ter manipulacijo s podatki. K splošnemu sprejetju jezika je prispevala tudi njegova prva standardizacija leta 1986 s strani ANSI (angl. American National Standards Institute) ter leta 1987 s strani ISO (angl. International Organization for Standardization). Dandanes znanje jezika SQL predstavlja eno izmed poglavitnih kompetenc vsakega razvijalca programske opreme. Zato je za učitelje še posebej pomembno, da predajo znanje pisanja poizvedb v jeziku SQL na učinkovit in preprost način. Kljub temu da je jezik SQL preprost in strukturiran, se študentje srečujejo z različnimi težavami pri učenju.

V preteklem desetletju je razvoj računalništva omogočil izvedbo učenja s pomočjo računalniško podprtih orodij. Slednja se uspešno uporabljajo za učenje različnih ved, kot je npr. algebra, logično dokazovanje itd. Tudi na področju podatkovnih baz se je razvila vrsta orodij z namenom hitrejšega in učinkovitejšega učenja jezika SQL. Sicer so obstoječi sistemi za učenje uspešni, vendar nas razvoj področij, kot je umetna inteligenca, vodi v stalno izboljševanje obstoječih procesov učenja. Na podlagi slednjega je nastalo diplomsko delo, ki vpelje nov sistem za učenje jezika SQL na osnovi zgodovinskih podatkov reševanja nalog iz poizvedb SQL.

V začetnih poglavjih dela je predstavljeno področje inteligentnih sistemov za učenje (angl. Intelligent tutoring systems) skupaj s trenutnim stanjem na področju računalniško podprtega učenja. V nadaljnjih poglavjih je opisana splošna arhitektura izdelanega sistema za učenje, kateri sledi podroben opis vseh komponent sistema. V zaključnem delu se nahajajo evalvacija izdelanega sistema ter sklepne ugotovitve.

Poglavje 2

Motivacija

Učenje jezika SQL primarno poteka na visokošolskih tehnoloških ustanovah, in sicer v okviru učenja podatkovnih baz ter sistemov za upravljanje s podatki. Glavni cilj učenja jezika SQL je usposobiti kandidata, da je ta zmožen manipulirati s podatki ter iz podatkov pridobiti uporabne informacije. Glede na [17] reševanje nalog ter ocenjevanje poteka v okolju podobnem profesionalnemu okolju. Omenjena raziskava navaja kot osrednji razlog dejstvo, da študent privzame način pisanja poizvedb, kot ga bo kasneje uporabljal v delovnem okolju.

Večina poizvedb, napisanih v jeziku SQL, je preprostih in kratkih [17]. Torej bi pričakovali, da bo učenje jezika SQL prav tako hitro in učinkovito. Vendar se izkaže, da temu ni tako, saj imajo študentje mnoge težave pri učenju [17]. Raziskava poudarja, da mora študent spisati poizvedbo, ki jo sistem za upravljanje s podatki pred izvedbo dodatno pretvori. Tako se težavnost znatno poveča, saj študent nima takojšnje povratne informacije, kakšne rezultate daje njegova poizvedba. Druga raziskava [14] omenja, da težavo predstavlja tudi pomnjenje podatkovnih shem, saj študentje preprosto pozabijo imena atributov ter tabel. Napačno razumevanje določenih konceptov, kot so agregacija podatkov, omejena agregacija podatkov, združevanje tabel (angl. join), je zelo pogosto [14].

Današnji sistemi za učenje delno ublažijo težave študentov pri učenju. Večina jih omogoča testiranje napisanih poizvedb, tako da ima študent povratno informacijo o rezultatu podane rešitve [17]. Takšen pristop, glede na [17], poveča učinkovitost učenja ter izboljša motivacijo študentov za poizkušanje pisanja pravilnih rešitev. Za lažje pomnjenje shem se besedilu naloge nemalokrat doda slika izseka logičnega podatkovnega modela z imeni atributov in tabel.

Res, da testiranje poizvedb pred oddajo poveča hitrost učenja ter omogoči študentu, da prilagaja poizvedbo za izpolnitev cilja, a obstajajo situacije, kjer je učenje še vedno neučinkovito. Preprost primer je scenarij, kjer študent izbere napačen pristop k reševanju naloge. Kljub temu da mu sistem omogoča testiranje poizvedbe, obstaja majhna verjetnost, da bo študent zmožen preiti iz napačnega pristopa reševanja k pravilnemu. Drug primer je scenarij, kjer študent ne dojame, kaj od njega zahteva naloga. Takrat bi bilo konstruktivno študentu ponuditi idejo, kako se lotiti reševanja naloge. Vse omenjene situacije opisujejo inteligentne sisteme za učenje pri katerih študent dobi pomoč v obliki namigov. S pomočjo namigov je učenje do določene mere hitrejše in učinkovitejše.

2.1 Inteligentni sistemi za učenje

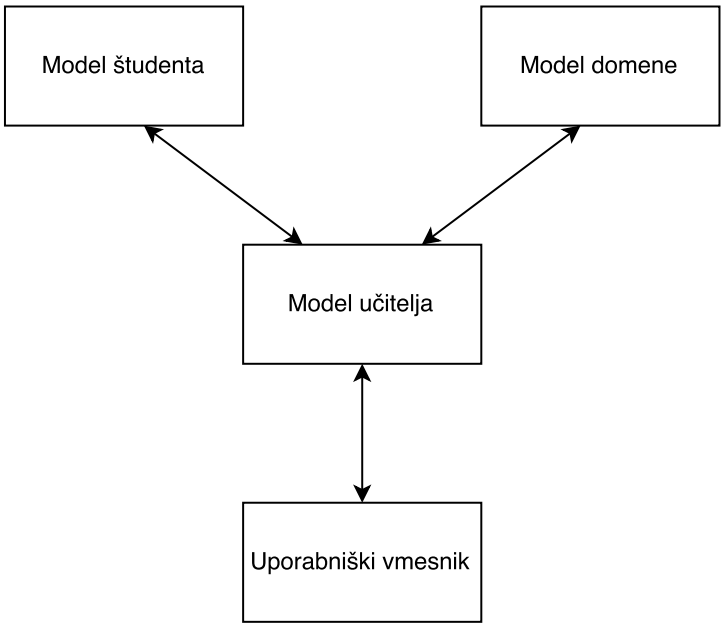
Inteligentni sistemi (ITS) se od običajnih, računalniško podprtih sistemov za učenje razlikujejo v prilagajanju uporabniškim zahtevam po načinu učenja. Medtem ko so namigi in vsebina pri običajnih sistemih statični, se pri sistemih ITS vsebina za vsakega študenta določi dinamično (sistem je adaptiven). Sistem za učenje je inteligenten, v kolikor izpolnjuje tri zahteve [15]:

1. Sistem dovolj dobro pozna domeno, v kateri deluje, da lahko samostojno rešuje probleme v tej domeni.

2. Sistem je zmožen razpoznati, do kolikšne mere je študent usvojil znanje, ki ga želimo podati.

3. Sistem je sposoben prilagajati težavnost nalog, da zmanjša razkorak med znanjem eksperta domene in znanjem študenta.

Glede na zgornje zahteve lahko v vsakem inteligentnem sistemu identificiramo tri ključne komponente: model domene (angl. domain model), model študenta (angl. student model) in model učitelja (angl. instructor model).

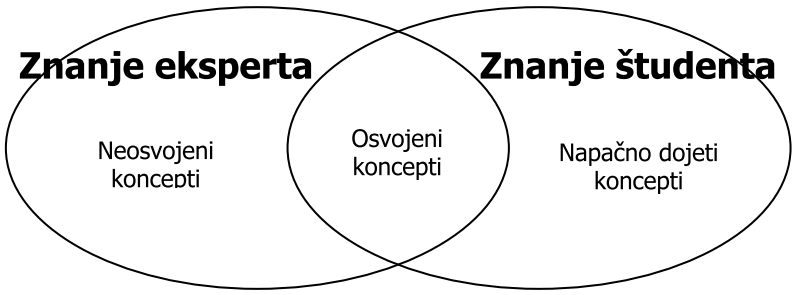


Slika 2.1: Osnovna arhitektura ITS sistemov

Model domene predstavlja obsežno zbirko znanja iz domene, ki jo sistem lahko uporabi za evalvacijo in reševanje nalog. Način pridobivanja zbirke znanja je odvisen od posameznega sistema. Običajno se znanje vnese ročno s pomočjo ekspertov domene. Vneseno znanje je lahko tudi dinamično. Zbirka znanja je skupna vsem študentom, medtem ko je model študenta različen od posameznika do posameznika, odvisno od pridobljenega znanja ter količine rešenih nalog [15, 13]. Model študenta lahko definiramo tudi kot okno v model domene, saj študent v določenem trenutku obvlada podmnožico celotne domene. Običajno takšen model vsebuje informacije, kot so, katere naloge je

študent rešil, katera poglavja je obiskal in v idealnih razmerah, katere koncepte je usvojil in katerih ne. Preprosti sistemi predpostavljajo, da je študent usvojil koncept, če ga je uspešno apliciral n-krat. Bolj kompleksni sistemi uporabljajo Bayesovo formulo za določanje verjetnosti, da je študent usvojil določen koncept.

Slika 2.1 prikazuje, kako omenjene komponente med seboj sodelujejo v procesu učenja. Model učitelja uporablja informacije o posameznem študentu, ki jih pridobi iz modela študenta (adaptivni del sistema). Skupaj z informacijo o domeni oblikuje namige ter izbira prihajajoče naloge. Pomemben dodatek v nekaterih sistemih predstavlja model odstopanj (angl. perturbation model) [13]. Kot je razvidno iz slike 2.2, lahko študent določene koncepte napačno dojame. Posledično mnogi sistemi vsebujejo modele za določanje, katere koncepte mora študent ponovno usvojiti. Model odstopanj je običajno predstavljen kot zbirka pogostih napak, h katerim so študentje nagnjeni med reševanjem. Na ta način lahko sistem zazna, katere napake je študent storil.



Slika 2.2: Model odstopanj v ITS sistemih [13]

2.2 Obstoječi pristopi k razvoju sistemov ITS

Skozi razvoj računalniško podprtega učenja se je razvilo več inteligentnih sistemov za učenje. Najbolj uveljavljeni med njimi so kognitivni tutorji (angl. cognitive tutors), novejši pristopi uporabljajo koncept omejitev ali zgradijo model študenta s pomočjo strojnega učenja. V nadaljevanju so opisani omenjeni pristopi k izgradnji sistemov ITS.

2.2.1 Kognitivni tutorji

Dolga leta so bili kognitivni tutorji najaktualnejša rešitev na področju sistemov ITS. Kljub temu da so danes v razvoju nove metode, kognitivni tutorji še vedno predstavljajo pomemben gradnik v računalniško podprtem učenju.

Leta 1982 je bila dokončana teorija ACT* (angl. Adaptive control of thought), na podlagi katere je nastala večina današnjih inteligentnih sistemov za učenje [1]. Glavni prispevek omenjene teorije je ta, da lahko procese človeškega mišljenja delimo na deklarativne in proceduralne. Razlaga za delitev je dokaj preprosta. V kolikor želimo opraviti določeno nalogo, potrebujemo določena proceduralna znanja. Človek mora, preden pridobi ustrezna proceduralna znanja, usvojiti ustrezno deklarativno znanje. Bistveno za učenje je, da deklarativno znanje usvojimo vsaj enkrat. Tudi če kasneje ne poznamo potrebnega deklarativnega znanja, lahko še vedno opravimo nalogo, če smo le usvojili ustrezno proceduralno znanje.

Preprost primer ponazarja učenje izreka o trikotniški neenakosti. Sprva moramo poznati izrek, ki trdi, da je vsota dolžin poljubnih dveh stranic v trikotniku večja od dolžine tretje stranice. Znanje izreka predstavlja deklarativno znanje. Glede na teorijo ACT*, kognitivno znanje temelji na pretvorbi priučenega deklarativnega znanja v proceduralno znanje [1]. V omenjenem primeru bi proceduralno znanje usvojili z uporabo naučenega izreka za npr. dokazovanje drugih izrekov ali pa računanje vrednosti v trikotniku.

Za doseganje kompetence v določeni domeni bi, glede na teorijo, bilo dovolj, da usvojimo celotno deklarativno znanje. A hkrati bi interpretacija deklarativnega znanja brez proceduralnih pravil povzročila preveliko obremenitev delovnega spomina posameznika. Velja tudi nasprotno; za doseganje kompetence v domeni bi lahko celotno znanje usvojili v proceduralni obliki [13]. Zaradi prevelikega števila proceduralnih pravil tudi takšna alternativa ni sprejemljiva. Torej za uspešno učenje potrebujemo zadostno količino tako deklarativnega kot tudi proceduralnega znanja. Pridobivanje deklarativnega znanja ni težavno, saj nam ga podajajo učitelji v obliki definicij in izrekov. Izdatnejšo težavo predstavlja pridobitev proceduralnega znanja, saj mora vsak posameznik skozi reševanje nalog usvojiti ustrezne postopke. Poudarek kognitivnih tutorjev je zato predvsem na pridobivanju proceduralnih znanj.

Kognitivni tutorji so sprva bili razviti v namene potrjevanja teorije ACT*. Zaradi osredotočenosti na proceduralno znanje predpostavljajo, da študentje že imajo potrebno deklarativno znanje. Njihov model domene predstavlja nabor proceduralnih pravil. Cilj učenja je, spodbuditi študenta, da se obnaša kot določajo proceduralna pravila v modelu domene. Učenje poteka po metodi sledenja modelu (angl. model-tracing) [13]. Med reševanjem študent oddaja nepopolne rešitve. Sistem sledi študentu med reševanjem in mu v primerih, ko zaide s pravilne poti, ponudi namig. V kolikor sistem ne prepozna akcije študenta, ga obvesti o splošni napaki. Zaradi kombinatorične zahtevnosti sledenja študentovi rešitvi skozi celoten model domene, se velikokrat študenta prisili, da se vrne na pravilno pot reševanja. Za izgradnjo modela študenta se uporabi Bayesova verjetnost. Izračuna se verjetnost, da je študent usvojil proceduralno pravilo. Izračun poteka vsakič, ko sistem ugotovi, ali je rešitev pravilna ali napačna.

Eno izmed prvih orodij na področju kognitivnih tutorjev je bilo orodje *LISP tutor* [1] za učenje istoimenskega programskega jezika. Tutor je deloval tako, da je študentu ponudil predlogo programske kode, ki jo je bilo potrebno

dopolniti. V primeru, da je študent zašel s pravilne poti, je vskočil program in zamenjal napačno funkcijo s pravilno. Ker orodje uporablja tehniko sledenja modelu, mora ves čas imeti podatek o tem, kako študent namerava rešiti naslednji korak naloge. V ta namen je orodje ob nejasnih situacijah študenta vprašalo po njegovi nameri. Orodje se je izkazalo za zelo uspešno in je bilo preizkušeno v univerzitetnem okolju.

2.2.2 Modeliranje na podlagi omejitev

Osnovna težava kognitivnih tutorjev je njihova omejenost. Kognitivni tutorji usmerjajo študenta preko natančno določene poti, ki je bila predvidena ob vnosu proceduralnih pravil. Študent tako postane odvisen od pomoči tutorja, da ga ta vodi do rešitve. Za proceduralne domene, kot je npr. aritmetika, je takšno delovanje povsem ustrezno. V kompleksnejših, deklarativnih domenah, kot je jezik SQL, postane učenje težavno, saj študent ne poskuša sam z reševanjem nalog, temveč se zanaša na orodje, da ga vodi k rešitvi. Jedro težave predstavlja dejstvo, da je metoda sledenja modelu preveč restriktivna, saj ne upošteva, da imajo lahko naloge več rešitev [12, 13]. Neupoštevanje večih rešitev je posledica vnašanja proceduralnih pravil, saj je za vsako nalogo vnesena le ena pravilna pot do rešitve. Težave se ne da preprosto odpraviti tako, da bi vnesli več proceduralnih pravil za določeno nalogo, saj obstaja preveč kombinacij.

Kot odgovor na omenjene pomanjkljivosti se je razvil nov pristop k modeliranju sistemov ITS, katerega osnova predstavljajo omejitve (angl. constraints). Leta 1994 je Ohlsson pripravil novo teorijo človeškega učenja, ki se močno razlikuje od obstoječe ACT* teorije [12, 13]. Osrednje vodilo teorije je učenje iz napak med reševanjem nalog. Za razliko od teorije ACT* je Ohlsson trdil, da se naučimo proceduralnih pravil, ko ugotovimo, da smo med reševanjem napravili napako. Ohlssonova teorija trdi tudi, da je pojav napak med reševanjem pogost in običajen pojav, saj je naš delovni spomin preobremenjen. V danem trenutku sicer imamo potrebno deklarativno znanje,

vendar obstaja preveliko število kombinacij, da bi se lahko pravilno odločili. Ko enkrat usvojimo potrebno proceduralno znanje, se lažje odločamo, katere dele deklarativnega znanja je potrebno uporabiti.

Ohlsson uporablja omejitve za opisovanje preslikav med delčki deklarativnega znanja in trenutno situacijo. Omejitve imajo splošno obliko:

if *relevanceCondition* true **then**
    $satisfactionCondition$ true/false
end if

Pogoj relevantnosti pove, ali je delček deklarativnega znanja relevanten. Pogoj zadoščenosti podaja, ali je bil, v primeru da je pogoj relevantnosti izpolnjen, delček deklarativnega znanja pravilno uporabljen.

Bistvena razlika pri učenju na podlagi teorije ACT* in Ohlssonove teorije je v količini podrobnosti. Modeli ACT* usmerjajo študenta po točno določeni poti, tako da predpišejo idealni postopek za dosego cilja. Modeli, ki temeljijo na omejitvah, ne predpisujejo poti reševanja naloge, temveč le preverjajo poznavanje ključnih delov deklarativnega znanja. Tako je model domene predstavljen kot zbirka omejitev, ki jim mora študent zadostiti [12, 13]. V določenem trenutku se celoten nabor omejitev preveri nad rešitvijo študenta. Če je določena omejitev relevantna (pogoj relevantnosti je izpolnjen), se preveri pogoj zadoščenosti. V primeru, da je tudi pogoj zadoščenosti izpolnjen, lahko predpostavimo, da je študent usvojil koncept, ki ga modelira omejitev. V nasprotnem primeru študent ni izpolnil omejitve, omejitev postane del modela odstopanj. Sistem lahko uporabi zbirko neizpolnjenih omejitev, da poskuša ponovno priučiti študenta konceptov, ki jih ni dojel. Model študenta ravno tako temelji na omejitvah, in sicer zajema omejitve, ki jih je študent usvojil.

V sklopu raziskave [13] je bilo razvito orodje *SQL-Tutor* [5], ki uporablja modeliranje z omejitvami za učenje jezika SQL. Omejitve v sistemu bodisi preverjajo rešitev študenta z idealno rešitvijo bodisi preverjajo sintaktično pravilnost rešitve. Omejitve so zapisane v programskem jeziku LISP. Za preverjanje pogoja zadoščenosti in pogoja relevantnosti se uporablja ujemanje vzorcev (angl. pattern-matching). Za potrebe delovanja orodja je bila pripravljena obsežna zbirka omejitev. Orodje se je izkazalo za dokaj uspešno. Raziskava navaja, da se znanje študentov izboljša že po dveh urah uporabe.

2.2.3 Metode s področja umetne inteligence

Razvoj področja umetne inteligence je prinesel s seboj tudi pojav novih področij, kot sta strojno učenje in rudarjenje podatkov (angl. data mining). Omenjeni metodi sta zelo obetajoči in uporabni tudi v inteligentnih sistemih za učenje. Gradnja sistemov ITS je namreč časovno in stroškovno potratna operacija. Razlog se nahaja v naravi določanja modela domene pri kognitivnih tutorjih in sistemih, ki temeljijo na omejitvah. Da sistem ITS doseže določeno mero uporabnosti, je potrebno vložiti več 100 ur vnašanja pravil in omejitev, ki predstavljajo zbirko znanja (model domene). Metode iz umetne inteligence so nam v pomoč, saj do določene mere omogočijo avtomatsko generiranje zbirke znanja iz zunanjih podatkov. Seveda to velja le pod predpostavko, da imamo za posamezno domeno na voljo dovolj podatkov, iz katerih se lahko učimo.

Eden izmed prvih poskusov avtomatizacije generiranja modela domene je dokumentiran v [8]. Temeljni cilj razvitega sistema je omogočiti ekspertom domen, neveščim v programiranju, da izdelajo vsebino za učenje v sistemih ITS. Eksperti domen bi vnašali pravila po t. i. programiranju po zgledu (angl. programming by demonstration), kjer bi sistem iz opazovanja učiteljev med reševanjem sam zgradil ustrezne programske konstrukte. Avtorji sistema so uporabili strojno učenje za avtomatizacijo generiranja produkcijskih pravil, ki se uporabljajo pri učenju in podajanju namigov. Natančneje, uporabili

so prilagojeno različico iterativnega poglabljanja, pri čemer so za mejo poglabljanja vzeli verjetnost pojavitve določene funkcije. Sistem se je izkazal za zelo uspešnega. Testiranje je bilo opravljeno na primerih učenja seštevanja ulomkov, večstolpičnega seštevanja ter igre tri v vrsto. Sistem je zmožen za skoraj vse primere izdelati natančen predpis proceduralnih pravil, ki se lahko uporabijo v procesu učenja z metodo sledenja modelu.

Vse večji pomen imajo tudi zgodovinski podatki, ki jih lahko beležimo med potekom učenja. Dovolj velika količina podatkov oziroma dovolj kvalitetni podatki nam omogočajo, da iz njih izluščimo koristne informacije. Tako lahko iz podatkov o učenju razberemo, kje študentje napravijo največ napak, katere naloge rešijo zadovoljivo itd. Prednost zgodovinskih podatkov so pričeli izkoriščati tudi sistemi za učenje. Iz shranjenih rešitev študentov je mogoče zgraditi bazo znanja, ki se lahko uporabi kot model domene. Eden izmed takšnih sistemov je t. i. *Hint factory* [21]. Avtorji sistema so uporabili pretekle podatke o reševanju študentov za izgradnjo Markovskih odločitvenih procesov (angl. Markov Decision Process). Posamezna rešitev študenta se sprva pretvori v omenjeni MDP, nato pa se vsi dobljeni grafi MDP združijo v en sam, celovit graf. Združen graf predstavlja celotno zbirko rešitev študentov in vse različne poti reševanja. Vsak posamezen graf je sestavljen iz zaporedja stanj, vsako stanje opisuje rešitev študenta do določenega koraka. Za vsako stanje v združenem grafu se nato izračuna ocena, ki pove, kako ustrezno je stanje z vidika študentove rešitve. Računanje ocen stanj poteka po iterativni enačbi, dokler se ocene stanj ne ustalijo. Generiranje namigov poteka tako, da se študentovo stanje preslika v eno izmed stanj v združenem grafu MDP. Nato se poišče naslednik ujemajočega stanja z najvišjo oceno. Naslednik hkrati predstavlja tudi namig. Avtorji so opisani sistem preizkusili v praktičnem okolju in ugotovili, da je v kar 90 % primerov sistem zmožen ponuditi namig.

2.3 Primerjava pristopov k razvoju sistemov ITS

V preteklih poglavjih so bili predstavljeni pomembnejši pristopi k razvoju sistemov ITS. Kljub različnim metodam, ki so uporabljene v opisanih pristopih, je smiselna primerjava. Gradnja sistemov ITS je bila dolga leta drag in dolgotrajen proces, pri katerem je morala sodelovati vrsta strokovnjakov. Želimo si najti kompromis med kompleksnostjo izgradnje takšnega sistema, vloženim časom in ceno. Prav tako si želimo, da sistem deluje uspešno in da je učenje učinkovito.

Analizo pričnemo s kognitivnimi tutorji, saj so ti najbolj uveljavljeni na področju inteligentnih sistemov. Osnovno vodilo kognitivnih tutorjev je, da študentje usvojijo koncept, ko pravilno rešijo podano nalogo. Sistem zato nenehno usmerja študenta nazaj na pravilno pot reševanja. Z vidika študenta je sistem vsiljiv, saj stalno popravlja vnose in nas opozarja, če zaidemo s poti. Ker je model domene zgrajen iz množice proceduralnih pravil, je ta pravila potrebno vnesti ročno. Potrebujemo vrsto strokovnjakov domen, ki poleg znanja iz domene obvladajo tudi programiranje. Količina potrebnih pravil nam ni v pomoč, saj potrebujemo ogromno pravil, da zadostimo vsem variacijam. Več kot ima naloga različnih načinov reševanja, več pravil je potrebnih, več časa je potrebno vložiti za vnos teh pravil in hkrati se povečajo stroški. Glede na raziskavo [13] je dodatna slabost kognitivnih tutorjev ta, da niso primerni za vse domene. Raziskava navaja, da so kognitivni tutorji primerni predvsem za proceduralne domene, kjer je postopek reševanja dokaj natančno določen. Deklarativne domene, kjer je možnih več poti reševanja, sistem le stežka rešuje. Razlog je seveda v tem, da je potrebno v primeru alternativnih rešitev dodati ustrezna proceduralna pravila v model domene. Večina kognitivnih tutorjev ravno iz tega razloga za vsako nalogo pripravi le eno pot reševanja. Posledice vnosa manjšega števila pravil občuti študent, katerega učenje ni več tako učinkovito, saj sistem podpira le en način reševanja.

Za kvaliteten proces učenja je nujno, da študentu omogočimo svobodo pri reševanju, saj le na tak način pridobi ustrezna proceduralna znanja.

Na modeliranje z omejitvami lahko gledamo kot na nekoliko olajšano različico kognitivnih tutorjev. Namesto striktnega sledenja določeni poti reševanja, dovolimo poljuben pristop k reševanju, kjer preverjamo zadoščenost nabora splošnih omejitev. V kolikor želimo bolj podroben nadzor, lahko uvedemo specifične omejitve. Zaradi omenjene splošnosti omejitev lahko nastanejo težave pri povratni informaciji, ki jo vrnemo študentu [13]. Raziskava navaja, da lahko nastopi situacija, kjer je povratna informacija zavajujoča. Težavo povzročajo omejitve, ki za testiranje pogojev uporabljajo idealno rešitev. Modeliranje z omejitvami za razliko od kognitivnih tutorjev ne vsebuje podatkov o korakih reševanja, temveč preverja le splošne omejitve. Sistem zaradi pomanjkanja informacije o postopku reševanja študentu ponudi zavajujočo povratno informacijo. Povratna informacija zavaja študenta, ker se nanaša na idealno rešitev shranjeno v sistemu za učenje in ne na rešitev študenta. Večja težava, ki izhaja iz omenjene nevšečnosti, je nezmožnost sistema, da oceni, kaj bi študent moral storiti v naslednjem koraku [13]. Avtorji sistemov za modeliranje z omejitvami so zato bili primorani vključiti algoritme za reševanje nalog ter razširitve za omejitve, ki vključujejo popravljalno funkcijo v primeru kršene omejitve. Vse razširitve povečujejo ceno sistema, čas izdelave ter kompleksnost.

Dodatna težava sistemov z omejitvami je omejenost nabora nalog, ki preverjajo specifične omejitve [13]. V primeru, ko študent ne dojame koncepta modeliranega z določenimi omejitvami, mu sistem ponudi naloge, ki preverjajo iste omejitve. Ker pa je število nalog majhno, študent hitro izčrpa nabor nalog. Poleg že omenjenih težav je potrebno, tako kot pri kognitivnih tutorjih, ročno vnašanje omejitev v sistem. Situacija je še dodatno otežena, saj omejitve uporabljajo ujemanje vzorcev. Tako mora oseba, ki vnaša omejitve, biti spretna tudi na tem področju. Vendar za razliko od kognitivnih tutor-

jev pomanjkanje katere od omejitev ne predstavlja velike pomanjkljivosti za sistem.

Tako modeliranje z omejitvami kot kognitivni tutorji gradijo model na statičen način. V praksi to pomeni, da ITS, ki temelji na modeliranju z omejitvami ali na kognitivnih tutorjih, lahko zgradi le visoko usposobljena oseba, ki razume domeno ter kako sistem uporablja domeno za učenje [13]. Slednjo zahtevo lahko eliminiramo, v kolikor uporabimo metode na podlagi umetne inteligence. Vse bolj se uveljavlja dinamično grajenje modela domene, kjer lahko pri izgradnji sistema ITS sodelujejo nevešče osebe ali pa sploh ni potrebe po človeški prisotnosti. Z dinamičnim grajenjem zbirke znanja odstranimo potrebo po strokovnjakih ter obenem omogočimo, da učitelji sodelujejo pri izgradnji inteligentnih sistemov, tudi če ne poznajo delovanja takšnih sistemov.

Statični sistemi ITS prav tako redko omogočajo napovedovanje naslednjega koraka v sklopu študentove rešitve. Razlog je v tem, da takšni sistemi ne vsebujejo podatkov, ki bi jim omogočili sklepanje o študentovi rešitvi, temveč se zanašajo le na svojo zbirko znanja. Težava je očitna predvsem, ko študent odda prazno rešitev in prosi za namig. Takrat mu sistem ni zmožen ponuditi uporabnega namiga, saj nima rešitve, na katero bi apliciral svoje omejitve ali svojo pot reševanja, samostojnega reševanja naloge pa sistem ni sposoben. V kolikor uporabimo model domene, ki temelji na zgodovinskih podatkih, dobimo možnost preslikave študentove rešitve na rešitve njegovih sovrstnikov. Tako omenjene težave vsaj do določene mere rešimo. Kljub mnogim prednostim pristopov z umetno inteligenco, je kompleksnost takšnih rešitev visoka. Poleg visoke kompleksnosti velikokrat potrebujemo dovolj veliko količino zgodovinskih podatkov, prav tako pa morajo biti podatki kvalitetni. Generične rešitve v takšnem primeru ni, saj je zgrajen sistem ITS odvisen od narave uporabljenih podatkov.

Primerjava analiziranih pristopov nam je ponudila vpogled v stanje inteligentnih sistemov ter njihovega procesa izgradnje. Opazimo, da si opazovani sistemi glede na prednosti sledijo v sosledju. Kognitivni tutorji so eden izmed prvih poskusov inteligentnih sistemov in so bili dolgo uveljavljeni kot glavni standard na tem področju. Modeliranje z omejitvami ubere nov pristop in tako izboljša obstoječe kognitivne tutorje. Metode iz umetne inteligence dosežejo še večjo samostojnost sistemov ITS, saj vpeljejo dinamično komponento v grajenje modelov domen in študentov.

Poglavje 3

Priporočilni sistem

V okviru diplomske naloge je bil izdelan sistem za lajšanje učenja jezika SQL. Glavna motivacija pri izdelavi inteligentnega sistema so bili zgodovinski podatki reševanja nalog iz domene jezika SQL. Na voljo so nam bili podatki reševanja nalog iz let 2014 in 2015 v okviru obveznega predmeta Osnove podatkovnih baz. Predmet se predava v 1. letniku dodiplomskega študija na Fakulteti za računalništvo in informatiko in predstavlja uvod v sisteme za upravljanje s podatki. Podatki so med drugim vsebovali čas oddaje rešitve, identifikator naloge in sheme ter dejansko poizvedbo, ki predstavlja rešitev študenta. Vseh zabeleženih rešitev je bilo približno 32000. Želeli smo uporabiti znanje, skrito v naših podatkih za izdelavo sistema, ki bo v pomoč študentom pri reševanju tovrstnih nalog. Prednosti sistema so v priporočilnem modulu in njegovem generiranju namigov. Poleg namigov sistem omogoča tudi ostale, že uveljavljene prednosti, kot so testiranje poizvedb med reševanjem ter prilaganje izsekov slik logičnega podatkovnega modela za lažjo vizualizacijo.

Na osnovi zgodovinskih podatkov smo opravili analizo uspešnosti reševanja nalog s strani študentov. Za potrebe vrednotenja poizvedb smo uporabili obstoječe orodje *SQLer*, ki se pri predmetu Osnove podatkovnih baz uporablja za točkovanje in ocenjevanje nalog ter je bilo razvito v okviru Laboratorija za

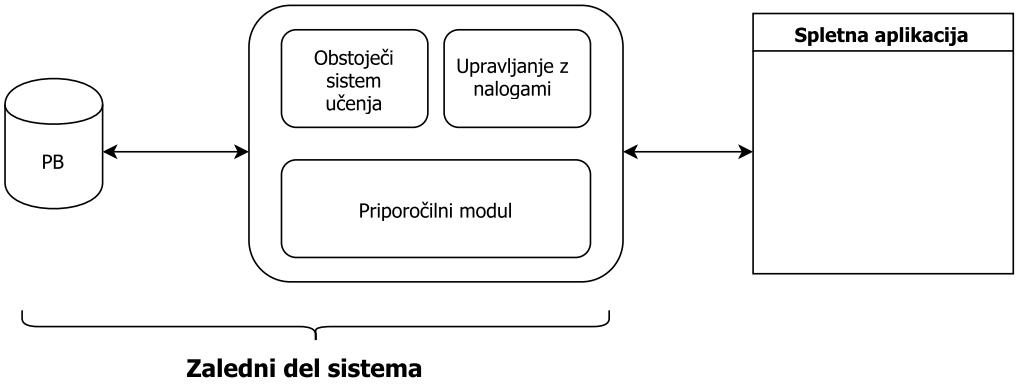
podatkovne tehnologije. Orodje je bolj podrobno opisano v okviru poglavja 4.6. Poleg orodja za ocenjevanje poizvedb smo imeli na voljo tudi seznam nalog, ki se ocenjujejo v okviru predmeta, ter za vsako nalogo podatek o težavnostni stopnji. Uporabili smo le podatke, katerih rešitev je bila sintaktično pravilna poizvedba SQL. Rezultati analize so predstavljeni v tabeli 3.1. Prvi stolpec tabele vsebuje podatke o težavnostni stopnji posamezne naloge, pri čemer število 1 predstavlja najlažje naloge, število 5 pa najtežje. Drugi stolpec tabele opisuje glavne koncepte, ki jih posamezna naloga skuša preveriti. Tretji stolpec predstavlja povprečen odstotek doseženih točk pri nalogi izmed vseh možnih točk za to nalogo. Rezultati v tabeli kažejo na to, da imajo študentje presenetljivo težave že pri osnovnih nalogah, saj znaša pri najlažji nalogi uspešnost "le"49 %. Tudi pri težjih nalogah uspešnost le malce naraste z maksimalno vrednostjo pri 64 %, medtem ko pri določenih nalogah uspešnost pade pod 40 %. Rezultati seveda izpostavljajo dejstvo, ki smo ga že opisali v poglavju 2. Študentje imajo težave pri razumevanju določenih konceptov, kot je npr. uporaba konstrukta `EXISTS`, ali pa jih napačno razumejo in uporabljajo. Za boljše učenje smo se odločili izdelati priporočilni modul in z njim razširiti obstoječi sistem za učenje. Razširitev poleg testiranja poizvedb omogoča tudi zahtevke po namigih, pri čemer posamezen namig vrne izboljšano poizvedbo, ki je bližje pravilnemu stanju kot prvotna poizvedba študenta. Cilj sistema je, izboljšati uspešnost študentov pri reševanju nalog ter nadgraditi sam proces učenja. Da je učenje zares učinkovito, potrebujemo tudi mehanizme, ki prilagajajo število namigov, ki jih ima študent na voljo. S tem se onemogoči pasivnost študenta pri učenju in prepreči, da bi se študent zanašal le na namige.

Kot napovedujejo shranjeni podatki, smo izbrali metode umetne inteligence za izgradnjo sistema. Odločili smo se za uporabo strojnega učenja nad podatki ter napovedovanje naslednjega koraka iz trenutnega stanja študentove rešitve. Natančneje, naš sistem temelji na metodi *Hint factory* iz [21], ki uporablja Markovske odločitvene procese za izgradnjo modela domene ter vrednostno iteracijo (angl. value iteration) za določanje ocen stanj. Omenjena

Težavnost naloge	Opis konceptov	Povprečna uspešnost
1 od 5	Preprosta `SELECT` poizvedba brez filtriranja in stikov.	49 %
2 od 5	Poizvedba s filtriranjem v `WHERE` stavku. Uporaba rezerviranih besed `IS NULL` za preverjanje pogoja.	37 %
3 od 5	Uporaba agregacijske funkcije `COUNT` ter grupiranja po enojnem atributu. Dodatna uporaba stika dveh tabel.	47 %
3 od 5	Združevanje rezultatov dveh poizvedb z uporabo konstrukta `UNION`.	60 %
4 od 5	Uporaba vgnezdene poizvedbe, uporaba agregacijske funkcije `MAX` znotraj vgnezdene poizvedbe.	64 %
4 od 5	Uporaba konstrukta `EXISTS` skupaj z negacijo `NOT` za preverjanje neobstoja množice podatkov v `WHERE` sklopu.	35 %
5 od 5	Uporaba dvojno vgnezdene poizvedbe za določanje druge največje vrednosti atributa.	51 %

Tabela 3.1: Rezultati analize uspešnosti reševanja nalog iz poizvedb SQL

metoda do sedaj še ni bila uporabljena nad domeno učenja jezika SQL, zato je zanimivo opazovati učinkovitost metode na tej domeni. Odločitev o uporabi omenjene metode temelji na, poleg uporabe zgodovinskih podatkov, tudi sami naravi metode. Metoda je namreč prilagojena za učenje iz zgodovin-



Slika 3.1: Osnovna arhitektura sistema za učenje

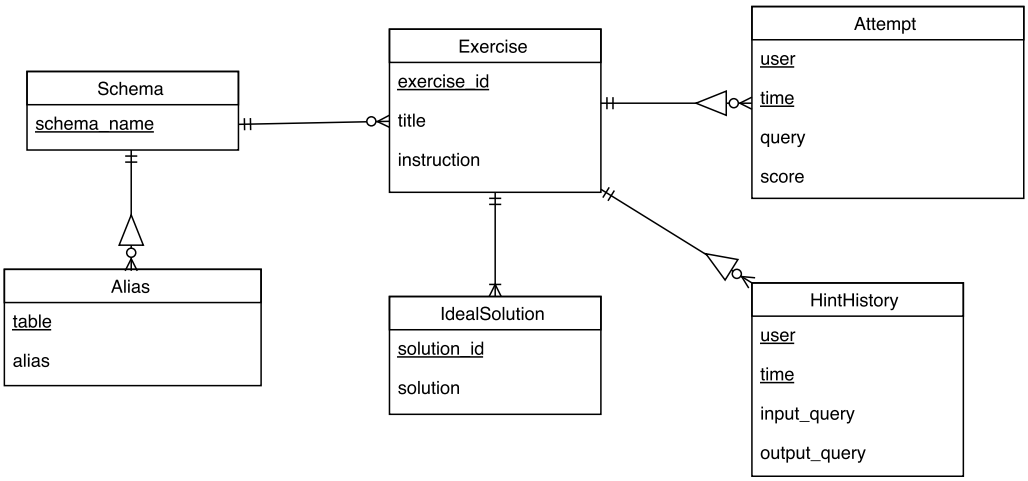
skih podatkov ter napovedovanje naslednjih korakov reševanja iz obstoječih podatkov. Poleg strojnega učenja je pomembnejša komponenta sistema tudi komponenta za analizo in procesiranje samega jezika SQL, ki je sestavni del priporočilnega modula na sliki 3.1. V procesu preslikave študentove rešitve na model domene je pomembna primerjava dveh poizvedb SQL, zato je takšna komponenta nujno potrebna. Dodatna zahteva pri zasnovi sistema je bila ta, da namigi ne razkrijejo celotne rešitve, temveč ponudijo študentu le naslednje stanje, naslednji korak na poti do rešitve. Tako postanejo namigi dejansko uporabni.

Poleg priporočilnega modula sistem vsebuje tudi modul za upravljanje z nalogami. Učitelji in strokovnjaki domene lahko na preprost način preko spletne aplikacije vnašajo nove naloge, urejajo obstoječe, določajo idealne rešitve, ki pomagajo v procesu generiranja namigov, itd. Spletna aplikacija poleg upravljanja nalog omogoča tudi simulacijo reševanja nalog ter prikaz namigov. Dodatna funkcionalnost je beleženje aktivnosti študentov pri reševanju, vključno z izhodnimi namigi, ki jih sistem generira. Tako lahko v prihodnosti napovedujemo uspešnost sistema ter analiziramo kakovost namigov. Slika 3.1 prikazuje najosnovnejšo arhitekturo izdelanega sistema. Sistem vse-

buje tipične komponente, kot so trajna shramba (podatkovna baza), zaledni sistem, ki teče na aplikacijskem strežniku ter uporabniški vmesnik v obliki spletne aplikacije.

3.1 Podatkovni model sistema

Za podporo delovanja zalednega sistema je bil izdelan konceptualni model podatkovne baze. Na sliki 3.2 je prikazan entitetno relacijski diagram (ERD) podatkovnega modela. Atributi v entitetah, ki so del primarnega ključa, so podčrtani. Povezave, ki vsebujejo trikotne figure, predstavljajo razmerje med močnim in šibkim entitetnim tipom.



Slika 3.2: ERD diagram podatkovnega modela sistema

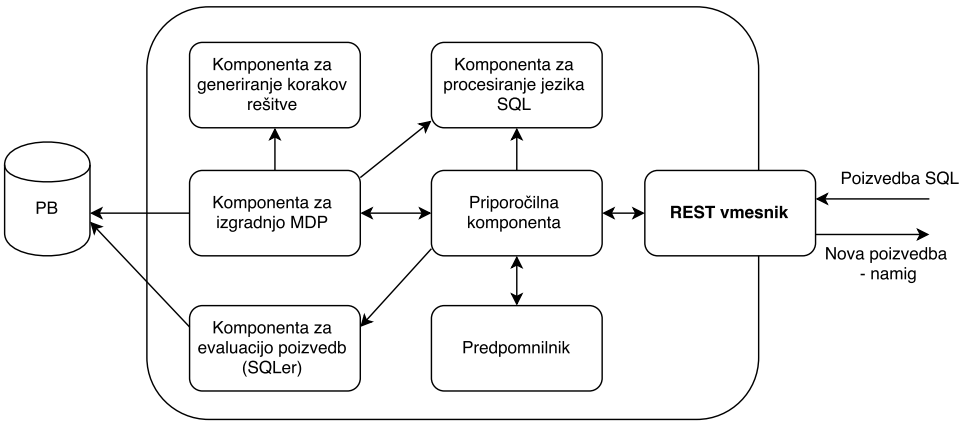
Podatkovni model podpira določanje podatkovnih shem, nad katerimi se izvajajo naloge. Podatki o shemi se beležijo v okviru entitete *Schema*. Entiteta *Alias* omogoča uporabniku prijazno poimenovanje referenc tabel v poizvedbi SQL, ki se vrne skupaj z namigom. Kot je opisano v prihajajočih poglavjih, lahko skrbniki sistema sami določijo imena referenc tabel. Entiteta *Exercise* beleži podatke o samih nalogah ter predstavlja osrednjo entiteto sis-

tema. Za vsako nalogo se zabeleži naziv naloge, shema, nad katero se bodo izvajale poizvedbe v okviru naloge, ter navodilo naloge, ki lahko vsebuje tudi povezave na slike izseka logičnega podatkovnega modela ciljne sheme. Vsaka naloga ima lahko več idealnih rešitev. Idealne rešitve za posamezno nalogo se beležijo v sklopu entitete *IdealSolution*. Skrbniki sistema lahko s pomočjo strokovnjakov domene vnesejo nabor idealnih rešitev, ki pohitrijo generiranje namigov v primerih, ko še ni dovolj zgodovinskih podatkov. Postopek je bolj podrobno opisan v poglavju 4.3.4.

Prvotne zgodovinske podatke smo migrirali v naš lastni podatkovni model. Za pohitritev generiranja namigov smo pred migracijo za vsak zgodovinski zapis izračunali oceno poizvedbe s pomočjo orodja *SQLer*. Rezultat smo skupaj z ostalimi podatki zabeležili v entiteto *Attempt*. Entiteta tako hrani podatke o času poskusa študenta, o imenu študenta, samo poizvedbo ter oceno poizvedbe. Entiteta ne vsebuje le statične migrirane podatke, temveč se dinamično dopolnjuje z novimi poskusi med reševanjem nalog. Tako ima sistem adaptivno zbirko znanja, ki se stalno nadgrajuje. Za potrebe beleženja izdanih namigov smo predvideli entiteto *HintHistory*. Entiteta beleži vse zahtevke po namigih s strani študentov. Poleg časovne komponente in identifikatorja študenta se zabeleži tudi vhodna poizvedba, ki jo je napisal študent, ter izhodna poizvedba v okviru namiga.

3.2 Komponente zalednega sistema

Slika 3.3 prikazuje arhitekturo priporočilnega dela zalednega sistema. Proces generiranja namigov poteka na sledeč način. Komunikacija spletne aplikacije z zalednim delom sistema poteka preko spletnih storitev, in sicer spletne storitve REST. Koncept storitev REST je opisan v poglavju 3.4.1. Kot vhod sistem prejme poizvedbo SQL, za katero študent želi namig oziroma dopolnitev. Sama spletna storitev v nadaljnjih korakih komunicira s priporočilnim modulom. Priporočilni modul skrbi za posodobitve modela domene ter za



Slika 3.3: Shema arhitekture priporočilnega modula

samo generiranje namigov. Model domene je, glede na metodo *Hint factory*, predstavljen kot kombinirana množica zgodovine rešitev, predstavljenih z grafom MDP. Za kreiranje objekta MDP iz zgodovinskih podatkov priporočilni modul uporablja komponentno za izgradnjo MDP. Slednja skrbi za več procesov. Sprva pretvori poizvedbo SQL iz zgodovinskih podatkov v drevesno strukturo s pomočjo komponente za procesiranje jezika SQL. Nad dobljeno drevesno strukturo nato opravi še dodatno transformacijo s pomočjo komponente za generiranje korakov rešitve. Rezultat transformacije je množica dreves (gozd), ki si sledijo v sosledju in predstavljajo korake na poti reševanja določene poizvedbe SQL. Komponenta nato združi vse korake reševanja v objekt MDP in postopek ponovi za vse zgodovinske zapise v podatkovni bazi. Kot rezultat dobimo velik, skupen objekt MDP, ki vsebuje vse poti reševanja izbrane s strani študentov pri reševanju nalog. Nastali MDP se vrne priporočilni komponenti, ta pa ga vstavi v predpomnilnik. Ob naslednjen dostopu je tako MDP predpomnjen in ga lahko priporočilna komponenta pridobi neposredno brez uporabe drugih komponent. Dodatna prednost predpomnjenja je ta, da se izognemo časovno potratnim operacijam nad podatkovno bazo. Priporočilna komponenta po pridobitvi objekta MDP izvede ujemanje najboljšega stanja glede na ocene stanj in trenutno stanje, v

katerem se nahaja študent. Kot namig se vrne poizvedba v iskanem stanju. Psevdokoda opisanega postopka je prikazana v okviru algoritma 1. Samo ujemanje najboljšega stanja je podrobno opisano v poglavju 4.4.

Algoritem 1 Generiranje namigov

```
1:  procedure GENERATEHINT(input_query, schema, exerciseId)
2:      if CACHE(schema, exerciseId) ≠ null then
3:          mdp ← CACHE(schema, exerciseId)
4:      else
5:          mdp ← CREATEMDP(schema, exerciseId)
6:          CACHE(schema, exerciseId) ← mdp
7:      end if
8:      return MATCHBESTSTATE(mdp, input_query)
9:  end procedure
10:
11: procedure CREATEMDP(schema, exerciseId)
12:     mdp ← empty
13:     data ← GETHISTORICDATA()
14:     for i ← 1 to | data | do
15:         tree ← PROCESSSQLQUERY(data(i))
16:         forest ← GENERATESOLUTIONSTEPS(tree)
17:         mdp ← mdp ∪ forest
18:     end for
19:     return mdp
20: end procedure
```

Ker se sistem neprestano prilagaja in dopolnjuje svojo zbirko znanja, priporočilna komponenta dostopa tudi do komponente za evalvacijo poizvedb. Tako je sistem zmožen za vsako vhodno poizvedbo določiti oceno ter jo vstaviti v podatkovno bazo kot poskus. Ker je objekt MDP nespremenljiv (angl. immutable), bo dodan poskus upoštevan ob naslednji ponovni gradnji MDP. Sistem zato ob določenih časovnih intervalih izprazni predpomnilnik, da za-

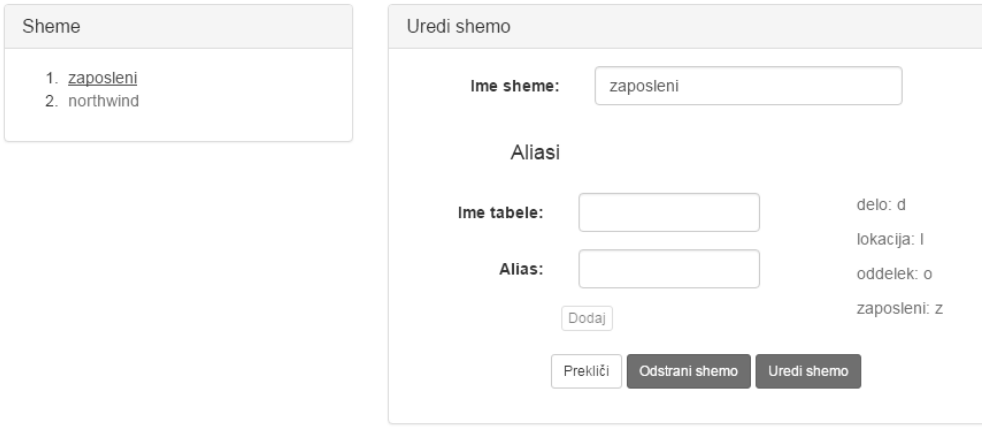
gotovi ponovno gradnjo MDP. Omeniti je potrebno, da je sistem zmožen ponuditi namig le, če je vhodna poizvedba sintaktično pravilna ter se pravilno izvede nad podatkovno bazo. Ob generiranju namigov se postopek zabeleži v podatkovno bazo tako, da imamo pregled nad izdanimi namigi. Poleg opisanih komponent se uporablja še komponenta za upravljanje z dostopom do podatkovne baze, ki na sliki 3.3 ni prikazana. Komponenta skrbi za komunikacijo med zalednim sistemom in podatkovno zbirko ter upravljanje s transakcijami.

Ob priporočilnem modulu se na strežniku nahaja tudi modul za upravljanje z nalogami. Primarna naloga modula je, zagotavljanje podpore upravljanju z nalogami preko spletne aplikacije. Implementirane so osnovne CRUD operacije za spreminjanje podatkov o nalogah in shemah. Pri tem se uporablja obstoječa komponenta za upravljanje z dostopom do podatkovne baze.

3.3 Komponente spletne aplikacije

Spletna aplikacija je sestavljena iz treh oddelkov: upravljanje s shemami, upravljanje z nalogami ter simulacija reševanja nalog. Slika 3.4 prikazuje obrazec za ustvarjanje, urejanje in brisanje shem. Skrbnik sistema lahko upravlja z naborom shem, ki so na voljo za poizvedovanje v okviru reševanja nalog. Za vsako shemo lahko urednik določi imena tabel in pripadajoča imena referenc (*alias*). Določanje imen referenc se uporablja za boljšo uporabniško izkušnjo, in sicer tako, da so imena referenc tabel v izhodnih namigih ustrezno preimenovana.

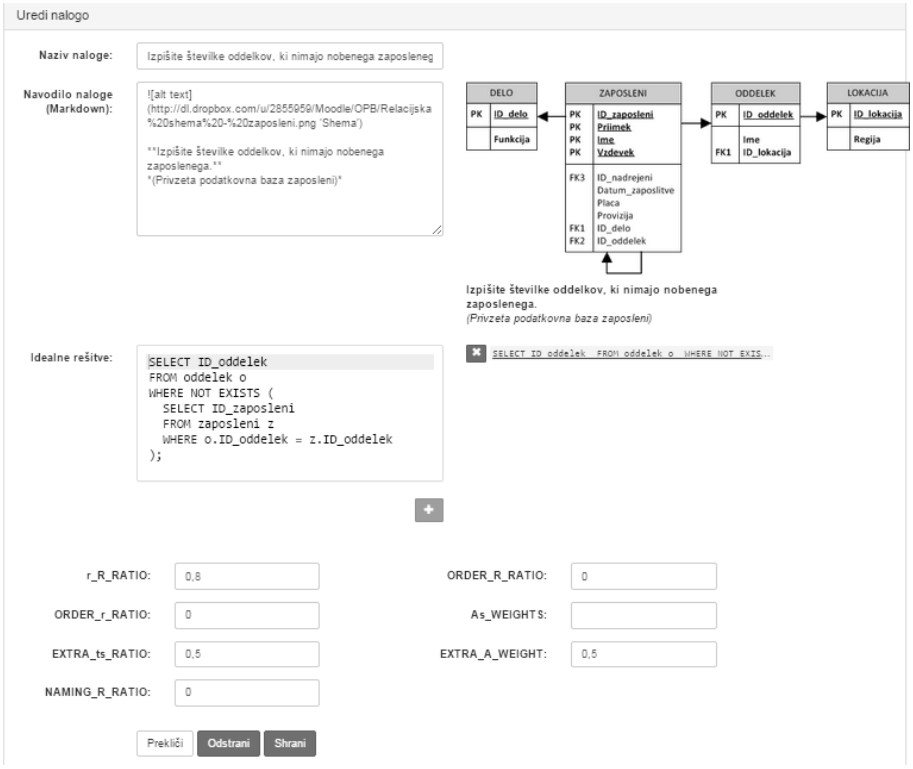
Razen urejanja shem lahko skrbnik sistema za vsako shemo pregleduje tudi seznam nalog, dodaja nove naloge in ureja/briše obstoječe naloge. Pri vsaki nalogi ima skrbnik omogočen vnos navodila naloge v jeziku *Markdown*, opisanem v poglavju 3.4.7. Tako lahko za vsako nalogo dodamo sliko izseka logičnega podatkovnega modela, ki razbremeni študenta pri reševanju nalog.



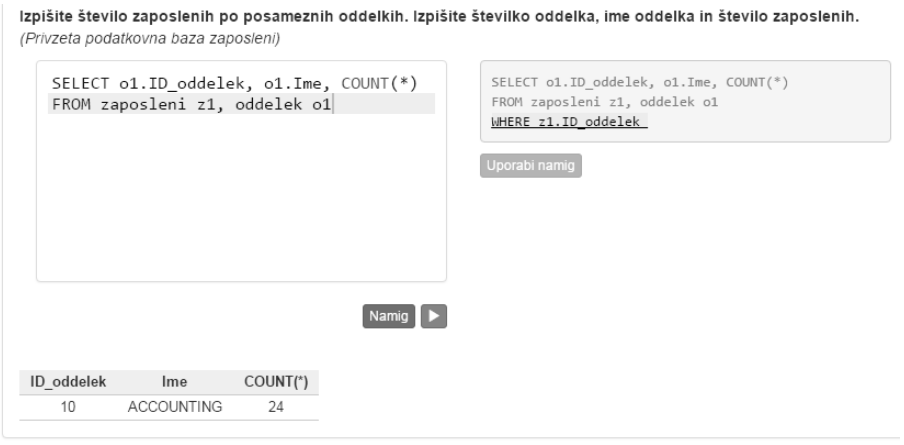
Slika 3.4: Oddelek za urejanje podatkovnih shem

Skrbnik mora določiti vsaj eno pravilno rešitev naloge. Pravilnim rešitvam pravimo tudi idealne rešitve, saj se uporabijo v procesu izgradnje MDP struktur in tako omogočajo še boljše generiranje namigov. Opisan obrazec je prikazan na sliki 3.5. Dodatno lahko skrbniki sistema urejajo koeficiente, ki jih uporablja komponenta za evalvacijo poizvedb v okviru vrednotenja.

Zadnji oddelek omogoča simulacijo reševanja nalog na podoben način, kot ga izkusi študent. Pri vsaki nalogi je poleg navodila naloge na voljo gumb za testiranje poizvedbe. Za testiranje poizvedb se uporablja vmesni strežnik (angl. proxy), ki je že bil izdelan v okviru obstoječega sistema za reševanje nalog pri predmetu Osnove podatkovnih baz. Testiranje poizvedb je časovno omejeno, zato da se zagotovi pravična raba sistemskih sredstev. Študent lahko vsakih 5 sekund testira poizvedbo. Vmesni strežnik že vrne rezultate poizvedbe v obliki HTML, ki se prikažejo pod nalogo, kot je razvidno iz slike 3.6. Poleg gumba za testiranje je na voljo tudi gumb za namig. Ob kliku na gumb se po končani zahtevi na strežnik prikaže pogled *Diff* za primerjanje študentove poizvedbe in poizvedbe v okviru namiga. Študent lahko z gumbom 'Uporabi namig' zamenja svojo poizvedbo s tisto iz namiga. Tako je zagotovljeno, da namigi niso vsiljivi, temveč so le v pomoč.



Slika 3.5: Urejanje posamezne naloge



Slika 3.6: Simulacija reševanja naloge

3.4 Uporabljene tehnologije in knjižnice

Pri gradnji sistema je bilo uporabljenih več standardov in tehnologij ter knjižnic. Namen tega poglavja je, podati kratek pregled konceptov uporabljenih tehnologij.

3.4.1 Java Enterprise Edition

JavaEE je platforma za gradnjo skalabilnih, visoko zanesljivih in varnih strežniških aplikacij. Že iz imena platforme je razvidno, da se srečujemo s tehnologijo, namenjeno velikim podjetjem, ki imajo takšne zahteve. Za dosego omenjenih zahtev platforma uvede dodatno delitev aplikacij na vmesnem sloju med odjemalcem in podatki [9]. Uvede se sloj, namenjen poslovni logiki, ter spletni sloj za povezovanje odjemalcev in poslovne logike. JavaEE teče na strežniku, ki implementira sam standard in ki ponuja storitve platforme. Takšni strežniki so aplikacijski strežniki. Za razvoj aplikacij platforma uporablja jezik Java z dodanimi API komponentami ter anotacijami. Z uvedbo anotacij se močno poenostavi konfiguracija, saj strežnik sam pripravi vse vire, označene v anotacijah.

Za podporo obeh slojev JavaEE definira dve vrsti vsebnikov (angl. container): spletni vsebnik in aplikacijski vsebnik. Spletni vsebnik vsebuje vse komponente, ki omogočajo, da se aplikacijski strežnik obnaša kot regularni spletni strežnik. Aplikacijski vsebnik vsebuje komponente poslovne logike, spletne storitve, komponente za povezovanje s podatki itd. Različni aplikacijski strežniki se med seboj razlikujejo glede na implementirane API komponente, ki jih definira standard. Za naš sistem smo uporabili Apache Tomcat 8 aplikacijski strežnik, ki privzeto ponuja implementacijo za *Java Servlet API* [18]. Dodatno smo uporabili še zunanje knjižnice za implementacijo *CDI API* in *JAX-RS API*.

Java Servlet API

Java Servlet API je pomembnejši del specifikacije JavaEE. Prvotno so se servleti razvili z namenom podpiranja dinamičnih spletnih vsebin. Omogočajo komunikacijo tipa zahteva–odgovor. Največkrat ta komunikacija poteka preko protokola HTTP. Servleti so preprosti javanski razredi, s katerimi upravlja spletni vsebnik. Ob vsaki zahtevi vsebnik naloži ustrezen javanski razred, ustvari novo instanco tega razreda in pokliče ustrezno inicializacijsko metodo znotraj razreda. Dodatno servleti omogočajo hranjenje podatkov med sejami, tako da lahko posameznega uporabnika ob zaporednih zahtevkih identificiramo. Seje se po določenem času odstranijo avtomatično.

Spletne storitve in JAX-RS

Spletne storitve omogočajo standarden način medsebojne komunikacije med različnimi komponentami sistema [9]. Prednost spletnih storitev je ta, da zagotavljajo šibko sklopljenost komponent sistema, kar pa omogoča prenosljivost in fleksibilnost. Danes obstajata dva večja razreda spletnih storitev. Spletne storitve SOAP delujejo po istoimenskem standardu in za izmenjavo sporočil uporabljajo format XML. Prenos sporočil SOAP največkrat poteka preko protokola HTTP. Drugi tip storitev so spletne storitve REST, ki se prav tako zanašajo na protokol HTTP za prenos sporočil, vendar pa uporabljajo dodatne formate za sporočila, kot je JSON. Spletne storitve SOAP omogočajo vrsto razširitev za varen in zanesljiv prenos podatkov, medtem ko storitve REST tega ne zagotavljajo, vendar so slednje veliko bolj enostavne za izdelavo in uporabo.

Vmesniki za dostop do virov v storitvah REST so predstavljeni z naslovi URL. Za določanje tipa operacije nad posameznim vmesnikom se uporabi ustrezna metoda HTTP (GET za pridobivanje vira, PUT za posodabljanje, POST za ustvarjanje ali DELETE za brisanje). Poleg definicije naslovov URL in metod HTTP, se za vsak vmesnik odločimo za ustrezno poimenovanje, tako da je že iz naslova razvidno, do katerega vira dostopamo. Defi-

nicija takšnih spletnih storitev je predvidena s strani standarda JAX-RS, ki določa množico anotacij za definicijo vmesnikov in je del JavaEE platforme. Obstaja več implementacij standarda JAX-RS. Naš sistem uporablja orodje *Jersey* [2], ki poleg implementacije standarda JAX-RS uvede dodatne funkcionalnosti za razvoj spletnih storitev v jeziku Java. V okviru aplikacijskega strežnika se za potrebe orodja Jersey ustvari privzet servlet, katerega namen je zagotavljati komunikacijo zahteva-odgovor preko protokola HTTP. Tako je spletna storitev REST izpostavljena potencialnim odjemalcem.

Contexts and Dependency Injection

Namen standarda je, povezovanje komponent spletnega vsebnika s komponentami aplikacijskega vsebnika, pri tem pa doseči šibko sklopljenost med spletnim delom in poslovno logiko [9]. CDI (angl. Contexts and Dependency Injection) opravlja dve nalogi: upravljanje s konteksti ter vstavljanje odvisnosti (angl. dependency injection). Kontekst predstavlja življenjsko dobo posamezne komponente. Razvijalcu se ni potrebno ukvarjati z ustvarjanjem ter odstranjevanjem komponent. Upravljanje konteksta je namreč nadzorovano s strani standarda, ki se odziva na različna stanja komponent in zagotovi razvijalcu stalen dostop do zahtevanih modulov. Vstavljanje odvisnosti poskrbi za šibko sklopljenost strežniškega dela in odjemalcev. Zagotavlja nam, da lahko v ozadju spremenimo implementacijo komponent brez potrebe po spreminjanju odjemalca. Tudi CDI API temelji na anotacijah za preprosto konfiguracijo. Naš sistem uporablja implementacijo *Weld* [7].

3.4.2 Java Database Connectivity

Za potrebe dostopa do podatkovne baze se na strežniku uporablja JDBC API, ki omogoča izvajanje poizvedb SQL preko jezika Java. Dodatna prednost komponente je uporaba predpripravljenih poizvedb (angl. prepared statement), ki preprečujejo klasične napade z vstavljanjem zlonamerne kode SQL (angl. SQL injection). Poleg preprečevanja napadov takšne poizvedbe omogočajo tudi učinkovito izvajanje v primeru zaporednih poizvedb.

3.4.3 Jackson API

Spletne storitve REST omogočajo prenos podatkov v večjem številu formatov. Toda strežnik poganja objektno usmerjen jezik Java, ki deluje na podlagi razredov, zato je potrebna dodatna preslikava pred prenosom podatkov. Postopek preslikave iz javanskih objektov v format za prenos preko omrežnih povezav imenujemo serializacija. Obraten proces se imenuje deserializacija. Standard JAX-RS omogoča uporabo različnih implementacij za potrebe serializacije. V okviru našega sistema smo uporabili implementacijo *Jackson* [10], ki ponuja serializacijo objektov v format JSON. Takšen format je najbolj ustrezen, saj spletna aplikacija temelji na jeziku Javascript.

Ker standard omogoča rekurzivno serializacijo objektov t. j. serializacija referenc objektov znotraj drugega objekta, so pogoste težave s serializacijskimi cikli. Cikel se pojavi, če med objektoma obstaja dvosmerna povezava. Takrat se program izvede nepravilno, pojavi se neskončna zanka. Za reševanje ciklov je potrebno atribute objekta opisati z ustreznimi pripisi, kar povzroči, da orodje omenjene atribute med serializacijo ne upošteva.

3.4.4 Google Guava

Google Guava [3] je javanska knjižnica, ki omogoča predpomnjenje objektov v pomnilniku strežnika. Prednost knjižnice je, da omogoča vrsto politik, po katerih se vnosi v pomnilniku odstranjujejo. Naš sistem uporablja politiko maksimalne velikosti, kar pomeni, da se najstarejši vnosi avtomatsko odstranijo iz pomnilnika, ko ta zasede določeno maksimalno velikost. Obstaja vrsta drugih politik, med zanimivejšimi je časovna politika, kjer se objekt po določenem času neaktivnosti (ni branj ali pisanj na omenjen objekt) avtomatsko odstrani. Lahko se odstrani tudi po določenem času, ko je bil objekt prvič dodan v pomnilnik.

Knjižnica za predpomnjenje uporablja podatkovno strukturo podobno

preslikavi, ki temelji na unikatnih ključih (angl. hash map). Vsak objekt se v pomnilnik vstavi pod določenim ključem, s katerim kasneje dostopamo do objekta. Dodatna zahteva podatkovne strukture je, da zagotavlja varne operacije pri dostopu večih niti (angl. threads). Tako se izognemo dvoumnim situacijam, ko dve ali več niti hkrati dostopajo do istega ključa v pomnilniku.

3.4.5 ANother Tool for Language Recognition

ANnother Tool for Language Recognition (ANTLR) omogoča gradnjo razčlenjevalnikov jezika (angl. parser). Celotna teorija analize jezika ter gradnje prevajalnikov je preobsežna v sklopu te diplomske naloge, zato bodo opisani le osnovni koncepti.

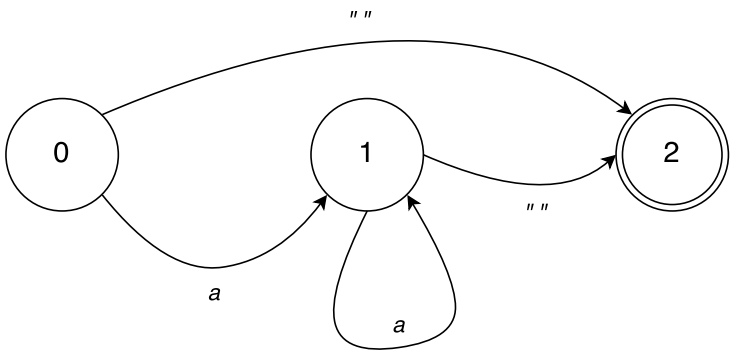
Razčlenjevanje naravnih jezikov ali programskih jezikov se prične z leksikalno analizo [16]. Program analizira vsak znak vhodnega niza. Namen leksikalne analize je, zagotoviti podporo višjim plastem tako, da lahko v vsakem trenutku iz niza vrnemo ustrezno zaporedje znakov oziroma besedo. Poleg grupiranja znakov v besede, program, ki izvaja leksikalno analizo, vsaki izmed možnih besed jezika (predpostavka je, da je število možnih besed končno) dodeli unikaten žeton (angl. token). Žeton se uporablja za referenciranje besed in varčevanje s pomnilnikom. Če jezik ne razlikuje med velikimi in malimi črkami, lahko posamezen žeton dodelimo večim besedam. Znake, ki so redundantni (presledki, komentarji v programski kodi), lahko program preprosto izpusti.

Za grupiranje znakov v besede se uporabljajo regularni izrazi. Regularni izraz nad naborom znakov je definiran kot prazen niz (ujemanje s praznim nizom), katerikoli znak iz nabora dovoljenih znakov, z operatorjem *ali* (bodisi prvi znak bodisi drugi znak), kot konkatenacija dveh znakov (prvi znak, ki mu sledi drugi znak) ali pa kot *Kleeneovo pokritje* (nič ali več ponovitev znaka) [16]. Primeri regularnih izrazov:

- “” (prazen niz)

- a (znak ‘a‘)
- $a|b$ (znak ‘a‘ ali znak ‘b‘)
- ab (znak ‘a‘, ki mu sledi znak ‘b‘)
- $a*$ (niz ““, “a“, “aa“ ...)

Regularne izraze lahko predstavimo kot nedeterministične končne avtomate (NFA). NFA je definiran kot trojček (*vhodna abeceda*, *množica stanj*, *množica prehodov med stanji*) [16]. Primer avtomata za regularni izraz $a*$ je predstavljen na sliki 3.7. Kot je razvidno iz slike, je v začetnem stanju, označenem z 0, možen prehod bodisi v končno stanje bodisi v stanje 1, glede na naslednji znak iz niza. V stanju 1 je možno poljubno število ponovitev znaka *a*. *Kleeneov teorem* trdi, da obstaja povezava med NFA in regularnimi izrazi, in sicer je povezava obojestranska. Vsak regularni izraz lahko pretvorimo v NFA in obratno.



Slika 3.7: Nedeterministični končni avtomat za regularni izraz $a*$ [16]

Za praktično implementacijo NFA niso primerni, saj so nedeterministični. Posledica nedeterminizma je ta, da se mora program v vsakem stanju odločiti za povezavo, ki jo bo obiskal. V primeru napačne odločitve se mora program

vračati v prejšnje stanje. Namesto NFA se zato uporabljajo deterministični končni avtomati (DFA). DFA je razširitev NFA, kjer niso dovoljene povezave, ki predstavljajo prazen niz in kjer nobeno stanje nima dveh izhodnih povezav z enakim znakom [16]. Izkaže se, da je vsak NFA možno pretvoriti v DFA, in sicer z algoritmom *Subset construction* [16]. V kolikor želimo, da je program sestavljen iz DFA učinkovit, potrebujemo minimizacijske postopke, ki bodo zmanjšali število stanj v DFA. Minimizacijski algoritmi delujejo na predpostavki, da so vsa stanja v DFA enaka. Med preverjanjem nato dodajo samo tista stanja, ki se dejansko razlikujejo.

Druga komponenta razčlenjevalnikov jezika so slovarji (angl. grammar). Slovar je sestavljen iz množice pravil v obliki $A \longrightarrow \alpha$, kar pomeni, da lahko vsak znak A zamenjamo z α [16]. Pravila, ki se ne pojavijo na levi strani enačbe, so elementarni znaki abecede (angl. terminal symbols), medtem ko so ostala pravila sintaktične spremenljivke (angl. non-terminal symbols), ki se pretvorijo v osnovne znake abecede. Dodatni primeri pravil v slovarju lahko zajemajo tudi relacijo *ali* ($A \longrightarrow \alpha \mid \beta$) ali pa so na desni strani tudi druge sintaktične spremenljivke ($A \longleftrightarrow B\alpha$). Slovar deluje tako, da aplicira vsa pravila, dokler ni več sintaktičnih spremenljivk, temveč le elementarni simboli abecede. Tako kot za NFA tudi za slovarje velja, da lahko vsak regularni izraz pretvorimo v pripadajoč slovar. Obratna relacija ne velja. Slovar lahko pretvorimo v regularni izraz le, če imamo opravka z *regularnim slovarjem*. Regularni slovarji so tisti slovarji, za katere so vsa pravila podana v obliki $A \longrightarrow aB$ ali $A \longrightarrow a$.

Pomembno spoznanje je dejstvo, da regularni slovarji v resnici modelirajo regularni jezik, ki je podmnožica vseh ostalih jezikov. Noam Chomsky je opisal tudi jezike, ki niso regularni, t. j. jezike z neskončno abecedo, ter jih razdelil v hierarhijo, prikazano v tabeli 3.2 [16]. Jeziki so razdeljeni v hierarhijo tako, da jezik na višjem nivoju vsebuje vse jezike spodnjega nivoja. Regularni slovarji se izkažejo za preveč omejene, da bi lahko z njimi proce-

sirali jezike, kot je SQL. A hkrati so slovarji nižjih nivojev kompleksnejši. Optimalno razmerje med kompleksnostjo in izrazno močjo predstavljajo slovarji brez konteksta. Med njih spadajo že omenjena pravila oblike $A \longrightarrow \alpha$, ki jih DFA in NFA ne morejo procesirati, saj nimajo mehanizmom za pomnjenje. Slovarji brez konteksta uporabljajo avtomate s skladom, ki omogočajo pomnjenje.

Razred jezika	Slovar	Avtomati
3	Regularni	NFA ali DFA
2	Brez konteksta (angl. Context-Free)	Avtomati s skladom (angl. Push down automata)
1	Slovarji s kontekstom (angl. Context-Sensitive)	Linearno omejeni avtomati (angl. Linear bounded automata)
0	Neomejeni	Turingov stroj

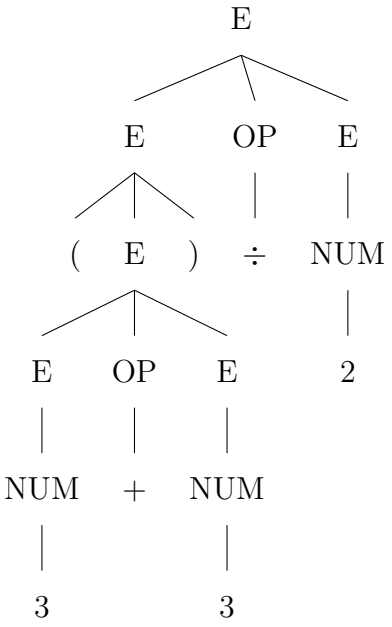
Tabela 3.2: Chomskyeva hierarhija jezikov

Razčlenjevanje jezika (angl. parsing) je proces, kjer preverjamo ali sekvenca besed ustreza določenemu slovarju, ki ima definirano množico pravil. Rezultat procesa je drevo, imenovano *parse tree*, ki prikazuje uporabljena pravila iz slovarja. Primer slovarja in gradnje drevesa za razčlenjevanje aritmetičnih izrazov (povzeto iz [16]):

Slovar:

$$E \longrightarrow E\ OP\ E \mid (E) \mid NUM$$
$$OP \longrightarrow + \mid - \mid \div \mid \times$$
$$NUM \longrightarrow (0\ ..\ 9)*$$

Pripadajoče drevo pri razčlenjevanju niza "$(3 + 3) \div 2$":



Pri gradnji takšnih dreves sta na voljo dva pristopa:

- Pristop *zgoraj-navzdol* začne s prvim simbolom iz niza ter poskuša aplicirati pravila iz slovarja, dokler niso vse leve strani pravil zamenjane z desnimi stranmi.
- Pristop *spodaj-navzgor* začne od konca niza proti začetku, pri čemer zamenjuje desne strani pravil z levimi stranmi, dokler ne doseže prvega simbola v nizu.

Poleg izbire načina gradnje drevesa se lahko v vsakem koraku odločimo za poljubno sintaktično spremenljivko, katero bomo zamenjali s pripadajočim izrazom pravila iz slovarja. Če procesiramo niz od leve proti desni, vedno izberemo najbolj levo sintaktično pravilo. V kolikor procesiramo niz od desne proti levo, izberemo najbolj desno spremenljivko.

Najenostavnejši algoritem za razčlenjevanje jezika po pristopu zgoraj-navzdol je algoritem *Recursive descent parsing* [16]. Omenjen algoritem

privzame, da je vsaka sintaktična spremenljivka na levi strani pravila definicija funkcije, sintaktična spremenljivka na desni strani pravila klic funkcije, elementarni znaki abecede pa predstavljajo ujemanje z znaki iz niza. Razčlenjevanje uporablja sklad. Za vsako ujemajočo levo stran pravila se na sklad potisne desna stran pravila. Nato se s sklada vzame naslednjo sintaktično spremenljivko. Postopek se ponavlja, dokler se na skladu ne nahajajo le elementarni znaki.

Poglavitna težava algoritmov za razčlenjevanje je določanje katera spremenljivka na desni strani naj bo uporabljena v primeru večih ustrezajočih desnih strani pravila (zaradi relacije *ali*). Postopek vpogleda naprej (angl. lookahead) analizira, katera spremenljivka se bo najverjetneje ujemala z naslednjim znakom iz niza [16]. To stori tako, da napove prvi elementarni simbol, ki ga določa sintaktična spremenljivka na desni strani pravila. Družina razčlenjevalnikov *LL(k)* uporablja postopek vpogleda naprej, pri čemer število k predstavlja minimalno število znakov za določanje natanko ene ustrezajoče sintaktične spremenljivke. Orodje ANTLR temelji na omenjeni družini razčlenjevalnikov z dodatnimi izboljšavami. Poleg razčlenjevanja jezika orodje ponuja tudi sintakso za zapis pravil v okviru slovarjev ter zapis regularnih izrazov za leksikalno analizo. Orodje definicijo slovarja pretvori v javanske razrede, ki se lahko uporabijo za razčlenjevanje jezika.

3.4.6 AngularJS

Razvoj spletnih strani poteka vse od prve pojavitve jezika HTML leta 1991. Od takrat se je proces razvoja stalno nadgrajeval. V samih začetkih spleta so bile spletne strani v celoti statične. Kasneje, z uvedbo okolja CGI, so postale vsebine spletnih strani dinamične. Dinamika spletnih strani se je povečala z ostalimi nadgradnjami, kot je jezik PHP, ASP.NET itd. Razvoj je sčasoma dosegel fazo visoko interaktivnih in dinamičnih strani, ki prezentirajo podatke v realnem času. Centralni gradnik v interaktivnosti predstavlja jezik Javascript, ki teče na odjemalčevi strani. Dandanes večina podjetij želi

minimizirati stroške vzdrževanja spletnih strežnikov in hkrati poenostaviti razvoj interaktivnih, a kompleksnih spletnih aplikacij. V ta namen je bilo s strani podjetja Google leta 2009 razvito ogrodje *AngularJS* [4], ki temelji na jeziku Javascript. V nadaljevanju so omenjene le najbolj atraktivne prednosti ogrodja, obstaja še mnogo drugih.

Eden pomembnejših konceptov ogrodja je gradnja t. i. spletnih aplikacij z eno samo stranjo (angl. single page application). Tradicionalne spletne aplikacije z večimi stranmi dinamično vsebino generirajo na strežniku, pri čemer združijo dinamične podatke s statično vsebino HTML. Rezultat nato dostavijo odjemalcu. Postopek sicer zagotavlja gradnjo dinamičnih strani, vendar onemogoča učinkovito predpomnjenje vsebine HTML. Za zmanjšanje porabe pasovne širine je dandanes, zaradi vse večje količine prometa na svetovnem spletu, nujno potrebno učinkovito predpomnjenje statičnih virov. AngularJS uvede združevanje podatkov in statične vsebine na odjemalcu. Naloga strežnika tako postane dostava statičnih vsebin HTML in podatkov, in sicer ločeno. Podatki se običajno dostavijo preko spletne storitve. Tako odjemalec samo prvič prenese statične vsebine, brskalnik pa jih nato predpomni. V kolikor je statična vsebina kombinirana z dinamično, predpomnjenje ni učinkovito, saj je dinamična vsebina ob vsakem dostopu različna.

AngularJS kot eden izmed prvih uvede že poznan koncept MVC v spletne aplikacije. MVC temelji na delitvi aplikacij na podatke (angl. model), poslovno logiko (angl. controller) in predstavitev (angl. view). Pri AngularJS so podatki shranjeni v Javascript objektih, poslovna logika je shranjena kot Javascript koda, pogled pa predstavljajo elementi strukture DOM. Takšna delitev omogoča učinkovito organizacijo programske kode.

Interaktivnost aplikacij AngularJS je zagotovljena preko *Data Binding API*. Omenjena komponenta omogoča dinamično spreminjanje pogleda ob posodobitvah modela ter posodobitve modela ob spremembah pogleda. Tako

nam ni potrebno, npr. ob osvežitvi podatkov s spletne storitve, ponovno izrisati pogleda, saj se ta posodobi samodejno. Prav tako nam ni potrebno dostopati do vsebine pogleda po interakciji uporabnika s pogledom, saj je le-ta na voljo avtomatsko.

3.4.7 Bootstrap

Ogrodje Bootstrap [11] se je razvilo zaradi vse večjega števila naprav. Z večjim številom naprav se je okrepilo tudi število različnih zaslonov in konfiguracij. Obstoječe spletne strani so prilagojene le za določene zaslone kot je npr. zaslon stacionarnega računalnika. V kolikor smo želeli prilagoditi spletno aplikacijo tudi za mobilne naprave ali tablične računalnike, je bilo potrebno vložiti veliko dela. Ogrodje Bootstrap močno poenostavi konfiguracijo za različne zaslone z uvedbo preddefiniranih stilov CSS. Prav tako zagotavlja interoperabilnost med različnimi brskalniki.

3.4.8 Označevalni jezik Markdown

Markdown [6] je jezik za podajanje obogatenega teksta na enostaven in hiter način. Omogoča zapis značk, ki se pred predstavitvijo prevorijo v jezik HTML. Na voljo nam je dodajanje slik, seznamov, spletnih povezav itd.

Poglavje 4

Podroben opis komponent

V prejšnjem poglavju je sistem bil opisan z vidika medsebojnega delovanja komponent. Sledeče poglavje je namenjeno podrobnemu opisu vsake izmed komponent priporočilnega modula zalednega sistema.

4.1 Komponenta za procesiranje jezika SQL

Za potrebe procesa analize poizvedb SQL je nujno potrebna komponenta, ki je zmožna pretvoriti poljubno pravilno obliko poizvedbe SQL v strukturo, ki je bolj primerna za procesiranje z vidika računalniškega sistema. Komponenta za procesiranje jezika SQL skrbi za ustrezno pretvorbo poizvedb SQL v drevesno strukturo, ki se shrani v pomnilniku. Za svoje delovanje komponenta uporablja razčlenjevalnik jezikov ANTLR, pri čemer je bil v okviru diplomske naloge izdelan slovar za jezik SQL. Poleg orodja ANTLR bi za procesiranje jezika lahko uporabili regularne izraze, a kot je že bilo omenjeno v poglavju 3.4.5, regularni jeziki nimajo dovolj velike izrazne moči za procesiranje kompleksnih jezikov, kot je SQL. Orodje ANTLR za razčlenjevanje potrebuje vsaj dva gradnika. Prvi je seznam definicij besed, ki se v procesu leksikalne analize izluščijo iz vhodnega niza. Drugi je seznam sintaktičnih pravil (slovar), ki se v procesu razčlenjevanja pretvorijo v elementarne znake abecede.

V procesu izgradnje slovarja za razčlenjevanje jezika SQL je bil pripravljen seznam besed in osnovnih konstruktov, ki jih uporablja jezik SQL. Za ujemanje besed iz elementarnih znakov ANTLR podpira tudi regularne izraze.

```
STRING
:
        QUOTATION_MARK WORD
        (
                ' ' WORD
        )* QUOTATION_MARK
;

WORD
:
        (
                'a' .. 'z'
                | 'A' .. 'Z'
                | NUMBER
                | UNDERSCORE
                | PERCENT
        )+
;
```

Slika 4.1: Definicija niza iz jezika SQL

Slika 4.1 prikazuje pravilo za ujemanje niza v okviru poizvedbe SQL. Definicije konstruktov, ki so samoumevni (npr. `PERCENT`), niso prikazane. Niz se praviloma prične in konča z narekovajem, bodisi enojnim bodisi dvojnim. Znotraj narekovajev se nahaja beseda, kateri sledi nič ali več besed ločenih s presledki. Beseda ima v jeziku SQL širši pomen, saj lahko vsebuje tudi znak za odstotek ali podčrtaj za podporo izrazu `LIKE`. Običajno je beseda sestavljena iz enega ali več znakov abecede (vključno s števili). Poleg nizov in besed se v jeziku SQL uporabljajo tudi osnovne primerjalne operacije in aliasi.

Za vsako izmed rezerviranih besed jezika SQL je bil definiran ustrezen re-

gularni izraz. Upoštevali smo tudi variacije malih in velikih začetnic, saj orodje ANTLR privzeto ne omogoča ujemanja brez upoštevanja razlik v začetnicah besed. Orodje v procesu leksikalne analize grupira ustrezne znake v besede, glede na pravila, ki smo jih zapisali. Dobljene besede nato uporabi pri razčlenjevanju jezika. Definiciji ključnih besed jezika sledi definicija slovarja. Slovar določa množico pravil, po katerih razčlenjevalnik združuje ključne besede.

```
basicSqlQueryExpr
:
        selectExpr fromExpr* joinExpr* whereExpr* groupByExpr* havingExpr*
        orderByExpr* limitExpr* ';'*
;

sqlQueryExpr
:
        basicSqlQueryExpr
        (
                UNION
                (
                        ALL
                        | DISTINCT
                ) basicSqlQueryExpr
        )*
;
```

Slika 4.2: Definicija sintaktičnih pravil za ujemanje poizvedbe SQL

Slika 4.2 prikazuje definicijo pravil za ujemanje poizvedb SQL. Poizvedba SQL je lahko kompleksna, v smislu, da lahko vsebuje rezultate večih poizvedb, združenih s konstruktom `UNION`. Osnovna poizvedba mora, glede na standard SQL, vsebovati vsaj `SELECT` stavek in `FROM` stavek. Pravilo smo zapisali na način, da je sklop `FROM` opcijski. Razlog leži v tem, da želimo biti zmožni ponuditi namige tudi za primere, ko študent še ni napisal sklopa `FROM`, temveč le osnovni `SELECT` stavek.

Pravilo za `SELECT` sklop, glede na sliko 4.3, v namene projekcije poi-

```
attributeExpr
:
        ALIAS
        | WORD
;

asExpr
:
        AS
        (
                STRING
                | WORD
        )
;

selectAttributeExpr
:
        (
                attributeExpr
                | comparisonExpr
                | functionExpr
        ) asExpr*
;

selectExpr
:
        SELECT DISTINCT*
        (
                selectAttributeExpr
                (
                        COMMA selectAttributeExpr
                )*
                | STAR
        )
;
```

Slika 4.3: Definicija pravila za `SELECT` stavek

zvedbe podpira uporabo atributov, primerjalnih izrazov, funkcijskih izrazov ali projekcijo vseh atributov preko znaka $*$. Atribut je lahko beseda ali alias. Podprta je tudi uporaba rezervirane besede `DISTINCT` in preimenovanje izbranih atributov z rezervirano besedico `AS`. Funkcijski izrazi lahko vsebujejo splošno funkcijo, ki se izvede nad izbranim atributom. Uporabljajo se tako

v SELECT sklopu kot tudi v HAVING sklopu.

```
functionExpr
:
        WORD LPAREN
        (
                functionParamExpr
                (
                        COMMA functionParamExpr
                )*
        ) RPAREN
;

functionParamExpr
:
        STAR
        | DISTINCT* attributeExpr
        | functionExpr
;
```

Slika 4.4: Definicija pravila za ujemanje funkcij

Iz slike 4.4 je razvidno, da lahko funkcije tudi gnezdimo, eno znotraj druge. Jezik SQL podpira kompleksne pogoje v okviru selekcijskega sklopa WHERE. Med drugim so podprte tudi vgnezdene poizvedbe in uporaba naprednejših konstruktov kot so EXISTS, ANY, ALL ... Dodatno lahko uporabnik poljubno vgnezdi logične pogoje OR in AND z uporabo oklepajev. Pravilo, ki podpira omenjene zahteve, je prikazano na sliki 4.5.

Rezultat opredeljenega slovarja in ključnih besed je vrsta javanskih razredov, ki jih orodje ANTLR ustvari ob grajenju ostale izvorne kode. Javanski razredi omogočajo procesiranje vhodnih nizov, ki predstavljajo poizvedbe SQL. V primeru sintaktične napake v vhodnem nizu se proži izjema, katero lahko obravnavamo. Komponenta za procesiranje jezika SQL poleg razčlenjevanja poizvedb SQL opravlja tudi pretvorbo v drevesne strukture. Rezultat razčlenjevanja poizvedbe je že omenjeni *parse-tree*, katerega dodatno pretvorimo v enostavnejšo drevesno strukturo. Pretvorba poteka tako,

```
logicalExpr
:
        logicalExpr AND logicalExpr
        | logicalExpr OR logicalExpr
        | logicalExpr IS NOT* logicalExpr
        | comparisonExpr
        | likeExpr
        | inExpr
        | someExpr
        | anyExpr
        | allExpr
        | existsExpr
        | betweenExpr
        | LPAREN logicalExpr RPAREN
        | logicalEntity
;

logicalEntity
:
        attributeExpr
        | NUMBER
        | BOOL
        | NULL
        | STRING
        | LPAREN sqlQueryExpr RPAREN
;
```

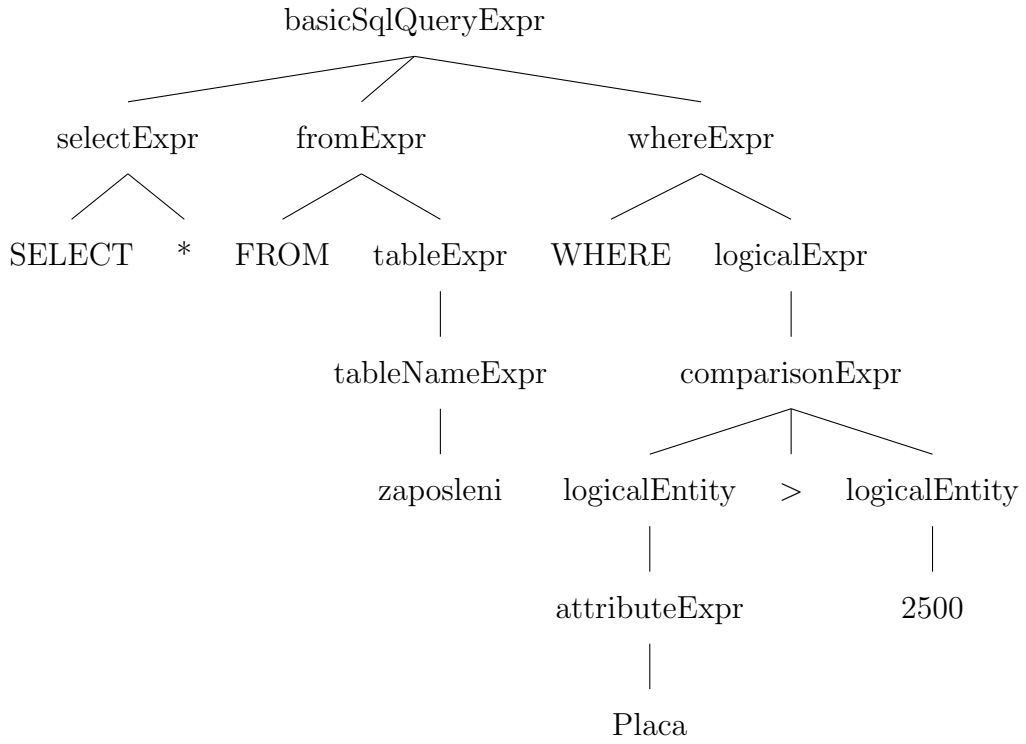
Slika 4.5: Definicija pravila za ujemanje logičnih izrazov

da se sprehodimo po drevesu, ki predstavlja rezultat razčlenjevanja, in sicer opravimo sprehod v globino (angl. depth-first). Pri tem kopiramo vsako vozlišče in ga dodamo v našo lastno drevesno strukturo. Za poizvedbo

```
SELECT *
FROM zaposleni
WHERE Placa > 2500;
```

tako dobimo drevesno strukturo, prikazano na sliki 4.6.

Komponenta za procesiranje jezika SQL opravlja tudi nasprotno transformacijo, in sicer je zmožna iz drevesne strukture pripraviti niz, ki predstavlja poizvedbo. Tudi za takšno transformacijo se uporablja obhod drevesa



Slika 4.6: Drevesna struktura za poizvedbo s preprostim selekcijskim pogojem

v globino, pri čemer se pred vgnezdenimi poizvedbami doda ustrezno število oklepajev. Ostali elementi so že del drevesne strukture in jih le izpišemo. Izpis je ustrezno formatiran v smislu dodanih presledkov med atributi.

4.2 Komponenta za generiranje korakov rešitve

Komponenta za procesiranje jezika SQL in zgodovinski podatki sami po sebi ne predstavljajo zadostnega pogoja za uspešno generiranje namigov. Če želimo zadostiti zahtevi, da naj namig ne razkrije celotne rešitve, temveč le naslednji korak na poti do končne rešitve, potem potrebujemo posamezne korake reševanja. Koraki reševanja nam iz zgodovinskih podatkov niso na voljo, saj je zabeležena le celotna poizvedba SQL. Za podporo korakov reševanja bi

morali sproti, med reševanjem naloge s strani študenta, na strežnik pošiljati parcialne rešitve. Izkaže se, da lahko ob predpostavki, da študent rešuje nalogo v vrstnem redu, glede na sklope, korake rešitve ustvarimo iz obstoječih poizvedb.

Komponenta za generiranje korakov rešitve skrbi za ustvarjanje korakov reševanja iz shranjenih poizvedb SQL. Temelji na prej omenjeni predpostavki reševanja po sklopih. Kot vhod komponenta prejme drevesno strukturo, ki predstavlja poizvedbo SQL. Kot rezultat vrne gozd, ki predstavlja množico poizvedb, posamezna poizvedba predstavlja korak reševanja. Ker smo predpostavili, da reševanje poteka po sklopih, lahko privzamemo, da se študent pomika po drevesni strukturi od leve proti desni. Sprva napiše `SELECT` sklop, ki je v drevesni strukturi najbolj levo, kot zadnjega napiše morebiten `LIMIT` sklop, ki je najbolj desno. Korake reševanja lahko zato dobimo z obhodom v globino po drevesni strukturi, ki predstavlja poizvedbo. Komponenta izvede omenjeni obhod drevesa, pri čemer se ustavi le pri listih drevesa. Listi drevesa namreč predstavljajo uporabnikov vnos in ne sintaktično pravilo. Ko komponenta doseže list drevesa, naredi kopijo drevesa, ki vsebuje vsa obiskana vozlišča na poti do trenutnega lista drevesa, vključno z listom. Omenjen pristop ni najbolj ekonomičen, saj za vse liste drevesa naredi kopijo drevesa in jih doda v gozd. Število rešitev lahko zmanjšamo tako, da ustvarimo kopijo drevesa le v primeru, da smo v listu, ki je najbolj desni otrok očeta. Takšen pristop ne poslabša generiranja namigov in hkrati zmanjša število dobljenih dreves.

Poleg preverjanja pogoja najbolj desnega lista je pred vstavljanjem v gozd potrebno preveriti, ali kopija drevesa predstavlja sintaktično pravilno poizvedbo SQL. Pravilnost koraka rešitve preverimo z obstoječo komponento za procesiranje jezika SQL. Za drevesno strukturo na sliki 4.6 bi komponenta ustvarila sledeče poizvedbe, ki predstavljajo korake reševanja:

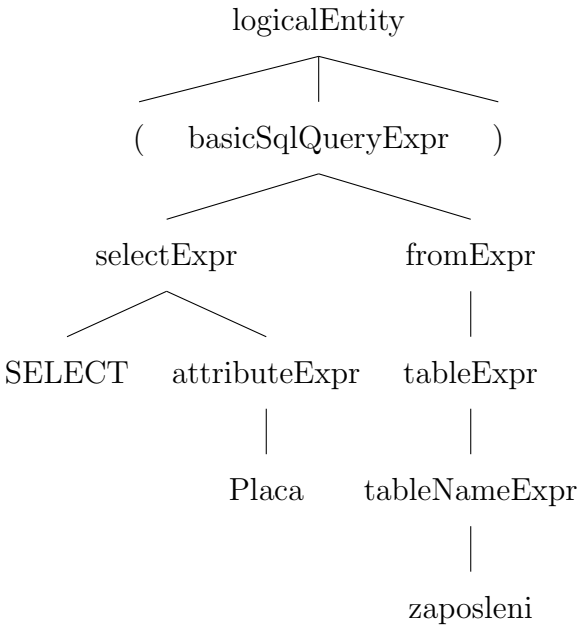
1. `SELECT *`

2. ```
   SELECT *
   FROM zaposleni
   ```

3. ```
   SELECT *
   FROM zaposleni
   WHERE Placa
   ```

4. ```
   SELECT *
   FROM zaposleni
   WHERE Placa > 2500
   ```

Opazimo, da so koraki rešitve predpone končne poizvedbe, končnega koraka. Komponenta sicer deluje dovolj dobro nad preprostimi poizvedbami, a naleti na zaplete pri vgnezdenih poizvedbah. Vgnezdene poizvedbe zahtevajo uporabo oklepajev. Ker komponenta preverja sintaktično pravilnost kopije drevesa, se lahko zgodi, da se vmesni koraki ne generirajo. Razlog je v pomanjkanju zaklepaja pri vgnezdenih poizvedbah. V določenem trenutku komponenta doseže list drevesa znotraj vgnezdene poizvedbe. V zunanji poizvedbi zaradi obhoda v globino še ni ustreznega zaklepaja, zato test sintaktične pravilnosti vrne negativen rezultat. Primer težave je prikazan na sliki 4.7. Ko komponenta doseže vozlišče `Placa`, naredi kopijo drevesa z vsemi do sedaj obiskanimi vozlišči, nato pa preveri, ali dobljeno drevo predstavlja pravilno poizvedbo. Ker obhod še ni obiskal vozlišča `)`, bo test negativen in korak reševanja ne bo dodan v gozd. Komponenta je sposobna reševati omenjene težave tako, da pred testiranjem sintaktične pravilnosti doda ustrezen zaklepaj v kopijo drevesa. Na ta način dobimo ustrezne korake reševanja tudi za vgnezdene poizvedbe.
   ```



Slika 4.7: Primer fiktivne vgnezdene poizvedbe

4.3 Komponenta za izgradnjo MDP

Sistem ima do te stopnje na voljo vse podatke, ki so potrebni za učenje in generiranje namigov. Poleg shranjene zgodovine reševanja nalog ima na voljo tudi komponento, ki je zmožna analize jezika SQL, ter komponento, ki nam vrača korake reševanja. Manjka le še struktura ali algoritem, ki bo povezal pridobljeno znanje in se iz podatkov učil ter napovedoval naslednje pravilno stanje. Iskana struktura je Markovski odločitveni proces (MDP), ki omogoča napovedovanje v negotovih razmerah. Opis struktur MDP in osnovne definicije so povzete po [19] in [20]. Ideja o uporabi struktur MDP je vzeta iz [21].

Kadarkoli imamo opravka z Markovskimi odločitvenimi procesi, želimo optimizirati končni izid nekega dejanja v negotovem, dinamičnem sistemu. Pravimo, da je sistem *stohastičen*, v kolikor se procesi v sistemu spreminjajo glede na določeno verjetnost, negotovost. V takšnih sistemih je pomembno, katere akcije izberemo, saj te niso deterministične. Izbor akcije vpliva na

dogodke v prihodnosti.

Markovski odločitveni proces je peterka,

$$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle \tag{4.1}$$

pri čemer je $\mathcal{S}$ *končna množica stanj,* $\mathcal{A}$ *končna množica akcij,* $\mathcal{P}$ *matrika prehodnih verjetnosti,* $\mathcal{R}$ *funkcija, ki podaja nagrade,* $\gamma \in [0, 1]$ *diskontni faktor*

Posledica zgornje definicije je ta, da je sistem v vsakem trenutku v enem izmed možnih stanj. V vsakem stanju ima agent na voljo končno množico akcij, vsaka akcija z določeno verjetnostjo vodi v ciljno stanje, z določeno verjetnostjo vodi v drugo stanje.

Matrika prehodnih verjetnosti je podana kot

$$\mathcal{P}^a_{ss'} = \mathbb{P}[\mathcal{S}_{t+1} = s' \mid \mathcal{S}_t = s, \mathcal{A}_t = a] \tag{4.2}$$

Matrika prehodnih verjetnosti za vsako akcijo iz množice končnih akcij $\mathcal{A}$ določi prehodno verjetnost, pri prehodu iz stanja s v stanje s' ob trenutku t.

Za vsako stanje s *velja*

$$\sum_a \mathcal{P}^a_{ss'} = 1 \tag{4.3}$$

Vsako stanje ima seveda lahko več akcij, vsota prehodnih verjetnosti za vse akcije je enaka 1. Poleg prehodnih verjetnosti, ki so značilne tudi za Markovske verige, je za MDP pomemben koncept nagrad.

Funkcija nagrad podaja pričakovano nagrado v naslednjem trenutku

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid \mathcal{S}_t = s, \mathcal{A}_t = a] \tag{4.4}$$

Vsaki akciji a iz končne množice $\mathcal{A}$ lahko priredimo vrednost imenovano nagrada akcije. V kolikor agent v stanju s izbere akcijo a, prejme nagrado $\mathcal{R}^a_s$.

Politika π je podana kot

$$\pi(s, a) = \mathbb{P}[\mathcal{A}_t = a \mid \mathcal{S}_t = s] \tag{4.5}$$

Politika π podaja preslikavo med stanji $s \in \mathcal{S}$, akcijami $a \in \mathcal{A}$ in prehodnimi verjetnostmi. Takšna politika natančno določa obnašanje agenta v sistemu. Za vsako stanje s lahko določimo vrednostno funkcijo, ki pove, kako ugodno je stanje za agenta. Stanje je ugodno z vidika nagrad, ki jih lahko pričakujemo v prihodnosti. Nagrade v prihodnosti so odvisne od akcij, ki jih bo agent izbral. Vrednostna funkcija zato upošteva akcije v trenutnem stanju.

Vrednostna funkcija podaja pričakovano nagrado v prihodnosti, ob začetnemu stanju s in sledenju politiki π

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid \mathcal{S}_t = s] \tag{4.6}$$

G_t podaja diskontirano nagrado od časovnega koraka t naprej

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.7}$$

Iz enačbe (4.7) je razvidna vloga diskontnega faktorja γ. Vrednost nagrade po $k+1$ časovnih korakih je $\gamma^k R$, kar pomeni, da večji kot je diskontni faktor, bolj preferiramo dolgoročne nagrade. Manjši kot je diskontni faktor, bolj preferiramo kratkoročne nagrade oziroma krajše rešitve.

Vrednostna funkcija akcij podaja pričakovano nagrado v prihodnosti, ob začetnemu stanju s, izbiri akcije a in nato sledenju politiki π

$$Q^{\pi}(a, s) = \mathbb{E}_{\pi}[G_t \mid \mathcal{S}_t = s, \mathcal{A}_t = a] \tag{4.8}$$

Tako kot za stanja lahko tudi za akcije definiramo vrednostno funkcijo. Dodatno lahko obe vrednostni funkciji razdelimo na neposredno nagrado in diskontirano nagrado naslednjega stanja.

Vrednostno funkcijo $V^{\pi}(s)$ lahko zapišemo kot

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma V^{\pi}(\mathcal{S}_{t+1}) \mid \mathcal{S}_t = s] \tag{4.9}$$

Vrednostno funkcijo akcij $Q^{\pi}(a, s)$ lahko zapišemo kot

$$Q^{\pi}(a, s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma Q^{\pi}(\mathcal{A}_{t+1}, \mathcal{S}_{t+1}) \mid \mathcal{A}_t = a, \mathcal{S}_t = s] \tag{4.10}$$

Obstaja povezava med vrednostno funkcijo stanja ter vrednostno funkcijo akcij. Povezava je podana v enačbi (4.11).

Vrednostno funkcijo $V^{\pi}(s)$ lahko zapišemo kot

$$V^{\pi}(s) = \sum_{a} \pi(a, s) Q^{\pi}(a, s) = \sum_{a} \pi(a, s)(\mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V^{\pi}(s')) \tag{4.11}$$

Cilj reševanja sistemov MDP je poiskati optimalno vrednostno funkcijo, ki maksimizira pričakovano nagrado preko vseh politik π.

Optimalna vrednostna funkcija je podana kot

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{4.12}$$

Optimalna vrednostna funkcija akcij je podana kot

$$Q^*(a, s) = \max_{\pi} Q^{\pi}(a, s) \tag{4.13}$$

Optimalno vrednostno funkcijo dobimo, v kolikor poznamo optimalno politiko $\pi*$. Optimalno politiko lahko izračunamo tako, da maksimiziramo preko vseh vrednosti $Q^*(a, s)$. Enačba, ki določa optimalno vrednostno funkcijo, je podana v (4.14).

Optimalno vrednostno funkcijo lahko zapišemo kot

$$V^*(s) = \max_{a} Q^*(a, s) = \max_{a}(\mathcal{R}_s^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V^*(s')) \tag{4.14}$$

Uporaba struktur MDP je primerna za učenje jezika SQL. Ker imamo na voljo množico zgodovinskih podatkov, iz katerih lahko pridobimo korake rešitev, lahko korake združimo v samostojni odločitveni proces in nato poiščemo optimalno politiko, kateri bo priporočilni modul (agent) sledil. Markovski odločitveni proces je predstavljen kot usmerjen graf, kjer vozlišča predstavljajo stanja, akcije pa povezave med vozlišči. Vsako vozlišče

je objekt, ki vsebuje enega izmed korakov rešitev. Poleg poizvedbe vozlišče vsebuje tudi informacijo o nagradi stanja ter seznam izhodnih ter vhodnih povezav. Povezava je predstavljena z verjetnostjo, izvornim ter ciljnim stanjem. Verjetnost povezave je enaka frekvenci uporabe te povezave izmed vseh povezav, ki izhajajo iz izvornega stanja povezave. Frekvence uporabe povezav izračunamo iz zgodovinskih podatkov. Nagrado stanj lahko izračunamo s pomočjo komponente za evalvacijo poizvedb. Tako izpolnjujemo vse potrebne pogoje za uporabo enačbe (4.14), ki določi pričakovane nagrade za vsako stanje glede na akcije stanja. Agent lahko uporabi izračunane nagrade za odločanje, katero bo naslednje stanje, naslednji korak reševanja. Odločanje poteka v priporočilni komponenti.

4.3.1 Združevanje korakov rešitev

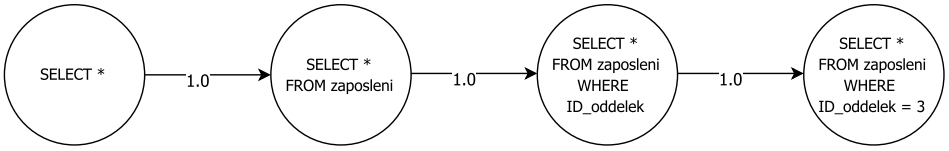
Združevanje korakov rešitev omogoča gradnjo enotnih zbirk znanja iz reševanja posamezne naloge. Rezultat združevanja je usmerjen, nepovezan ciklični graf, kjer enaki koraki rešitev sovpadajo. Proces združevanja sledi zaporedju korakov rešitve in hkrati išče ujemajoča stanja v skupnem MDP. V kolikor obstaja ujemajoče stanje, se vozlišče v skupnemu MDP ne ustvari, temveč se dodajo le nove akcije, ki izhajajo iz tega stanja. Če ujemajočega stanja ni, se dodata novo vozlišče ter akcija, ki vodi v ustvarjeno vozlišče. Ujemanje stanj temelji na primerjanju drevesnih struktur. Algoritem za primerjavo dreves je opisan v poglavju 4.5. Poleg dodajanja vozlišč ter povezav med njimi, se za vsako povezavo posodobi njena verjetnost oziroma frekvenca uporabe. Frekvenca se poveča, v kolikor se ne doda nova povezava, temveč uporabi obstoječa povezava. Primer združevanja korakov rešitev je prikazan za 3 poizvedbe:

```
SELECT *
FROM zaposleni
WHERE ID_oddelek = 3
```

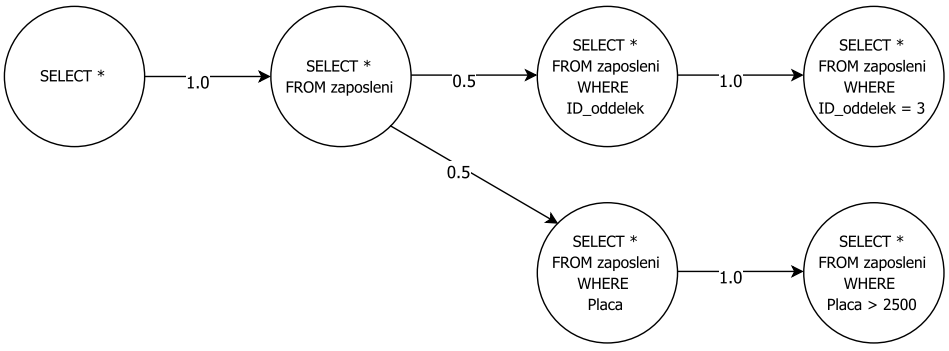
```
SELECT *
FROM zaposleni
WHERE Placa > 2500
```

```
SELECT Ime
FROM zaposleni
```

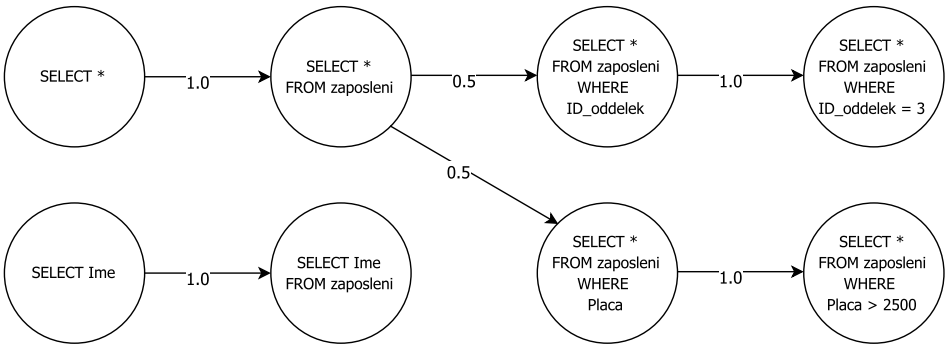
Korake reševanja prve poizvedbe vstavimo kot vozlišča v MDP, saj je ta do tega trenutka prazen. Pri tem dodamo ustrezne povezave in posodobimo verjetnosti:



Pri dodajanju korakov druge poizvedbe ugotovimo, da smo našli ujemajoče stanje `SELECT *`. Vozlišča zato ne dodamo. Ker se nahajamo v prvem koraku poizvedbe, ni potrebno dodati povezav. Tudi v drugem koraku obstaja ujemajoče stanje, zato vozlišča ne dodamo, temveč le sledimo povezavi med prvim in drugim vozliščem v obstoječemu MDP. Ustrezno posodobimo verjetnost, ki še vedno znaša 1.0. Tretji korak zahteva dodajanje novega vozlišča in posodabljanje verjetnosti povezav. Zadnji korak je dodan kot naslednik tretjega koraka:



Koraki zadnje poizvedbe se ne ujemajo z nobenim od obstoječih stanj, zato jih dodamo kot nova vozlišča:



Poleg povezav, ki povezujejo stanja v MDP, se doda še množica povratnih povezav. Namen povratnih povezav je zagotoviti, da bo sistem zmožen preusmeriti študenta iz napačne veje reševanja nazaj na pot, ki vodi do pravilne rešitve. Pomembno vprašanje je, kakšno verjetnost naj ima povratna povezava, saj povratne povezave ne izhajajo iz zgodovinskih podatkov. Eksperimentalne meritve so pokazale, da majhna verjetnost povratnih povezav (5 %) ne preprečuje pravilnih vej rešitev ter hkrati zagotavlja, da se agent lahko vrne iz napačne veje nazaj do ustreznega stanja.

V procesu združevanja se za vsako končno stanje (zadnji korak rešitve) določi ocena tega stanja oziroma nagrada, ki jo agent prejme ob obisku stanja. Ocena temelji na evalvaciji poizvedbe s komponento za evalvacijo poizvedb. V kolikor je izvedba v celoti pravilna, dobi stanje oceno 100. V nasprotnem primeru dobi stanje oceno -100 (stanje je nezaželeno). Vsa vmesna stanja dobijo oceno 0. Ocena vmesnih stanj je dinamična, saj se posodablja ob iskanju optimalne vrednostne funkcije.

4.3.2 Vrednostna iteracija

Rezultat združevanja korakov rešitev je MDP z določenimi ocenami končnih stanj. Preden lahko sistem napoveduje naslednje ugodno stanje, je potrebno za vsa vmesna stanja izračunati ocene. Rešiti je potrebno enačbo (4.14).

Eden izmed načinov reševanja je t. i. vrednostna iteracija. V okviru vrednostne iteracije za vsako stanje računamo optimalno vrednostno funkcijo, dokler ocene vseh stanj ne konvergirajo. Omenjena enačba zahteva, da za vsako izhodno akcijo stanja izračunamo pričakovano nagrado v prihodnosti. Pričakovana nagrada je enaka zmnožku verjetnosti, da nas akcija dejansko pripelje v želeno stanje, ter optimalni vrednostni funkciji ciljnega stanja. Upoštevati je potrebno tudi, da nas akcija z določeno verjetnostjo lahko pripelje v drugo stanje. Kot diskontni faktor γ smo uporabili vrednost 1.0, kar pomeni, da preferiramo dolgoročne nagrade. Z drugimi besedami, želimo, da študent dolgoročno pride do pravilnega končnega stanja.

V sklopu Markovskih procesov je pomemben razmislek, zakaj v našem sistemu niso zadostne običajne metode planiranja oziroma zakaj akcije niso deterministične. Z negotovostjo in vrednostno iteracijo dobijo stanja, kjer je velika verjetnost, da študent zaide v napačno vejo reševanja, slabše ocene. Tako se agent avtomatsko izogiba stanjem, kjer so študentje velikokrat zašli s pravilne poti. Podatke o nagnjenih k zahajanju s pravilne poti dobimo iz zgodovinskih podatkov kot verjetnosti izbire akcije v določenem stanju.

4.3.3 Preimenovanje aliasov

Kljub velikemu številu zgodovinskih podatkov teh ni dovolj, da bi zadostili vsem variacijam poizvedb. Najbolj variabilen del vsake poizvedbe SQL predstavljajo imena referenc tabel ali aliasi. Študentje lahko poljubno poimenujejo aliase, pri čemer mora sistem še vedno zagotavljati namige. Ker primerjava poizvedb temelji na algoritmu za ujemanje drevesnih struktur, ki opravlja natančno primerjavo, zgodovinski podatki niso dovolj. Pojavi se lahko scenarij, kjer se študent odloči za neobičajno ime reference tabele. V takšnem primeru bo sistem bodisi nezmožen ponuditi namig bodisi ponudil namig, ki ni v pretirano pomoč. Da bi povečali ujemanje med zgodovinskimi podatki in poizvedbo študenta tako komponenta za izgradnjo MDP kot priporočilna komponenta vsem poizvedbam preimenujeta aliase na skupno ime.

S tem se zagotovi boljše ujemanje poizvedb. Vsak alias se preimenuje v novo ime, ki je določeno s strani skrbnika sistema. Ker se preimenovani aliasi pojavljajo v izhodnih namigih, generirana imena niso primerna (in človeku prijazna). Preimenovanje zopet poteka z obhodom drevesa v globino in ujemanjem nizov v vozliščih.

4.3.4 Vloga idealnih rešitev

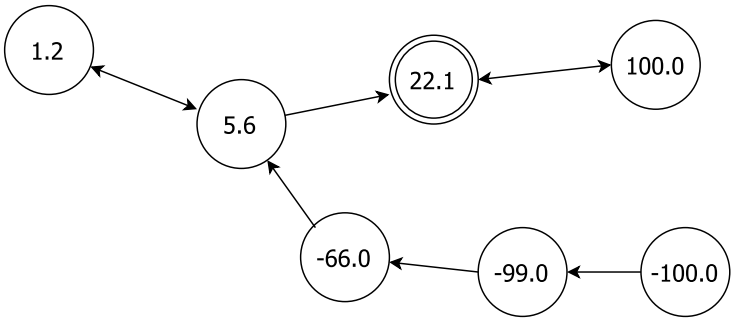
V primerih, ko ni dovolj zgodovinskih podatkov ali je njihova kvaliteta vprašljiva, lahko skrbniki sistema vnesejo eno ali več pravilnih (idealnih) rešitev za posamezno nalogo. Idealne rešitve se tako, poleg ostalih zgodovinskih podatkov, dodajo v MDP med grajenjem le-tega. Idealne rešitve si lahko predstavljamo kot seme, s katerim pospešimo generiranje namigov. Velikokrat se zgodi, da uvedemo nove naloge, za katere še ne obstajajo podatki o reševanju s strani študentov. Idealne rešitve v omenjenih primerih izboljšajo uporabnost sistema.

4.4 Priporočilna komponenta

Priporočilna komponenta uporablja zgrajen MDP in deluje kot agent pri učenju. Kot vhod prejme poizvedbo SQL, za katero študent želi namig. Poizvedbo s pomočjo komponente za procesiranje jezika SQL pretvori v drevesno strukturo. MDP pridobi bodisi iz predpomnilnika bodisi preko komponente za gradnjo MDP. Predpomnilnik temelji na knjižnici *Google Guava*, opisani v poglavju 3.4.4. Ko ima komponenta v lasti objekt MDP, najprej izvede ujemanje najbližjega stanja glede na stanje študenta. Ujemanje poteka tako, da komponenta izvede algoritem za primerjanje drevesnih struktur, opisan v poglavju 4.5, nad vsakim stanjem. Na koncu izbere stanje, ki je kar najbolj podobno stanju študenta, oziroma stanje, katerega drevesna struktura je najbolj enaka drevesni strukturi poizvedbe študenta. V kolikor najde ujemajoče stanje t. j. stanje, ki je popolnoma enako stanju študenta, pregleda sosednja stanja.

Pregled sosednjih stanj temelji na iskanju stanja z višjo oceno od trenutnega stanja. V kolikor takšnega stanja ni, uporabnega namiga ne moremo ponuditi, saj je študent v popolnoma napačni veji reševanja ali pa preprosto nimamo dovolj zgodovinskih podatkov. V takšnih primerih zato kot namig ponudimo poizvedbo enega izmed začetnih stanj z najvišjo oceno. V kolikor obstaja stanje z višjo oceno, ločimo dva primera. Prvi primer predstavlja stanje z višjo oceno, do katerega vodi povratna povezava. Povratna povezava nakazuje, da se študent nahaja v napačni veji ter da lahko študenta usmerimo nazaj na eno izmed pravilnih poti reševanja. Za usmerjanje nazaj na pravilno pot ni dovolj ponuditi namig za naslednje stanje z višjo oceno, saj je takšno stanje še vedno del napačne veje reševanja. Namesto tega se moramo vrniti nazaj do prvega skupnega prednika napačne veje in pravilne veje. Ko enkrat najdemo skupnega prednika napačne in pravilne veje reševanja, pregledamo njegova sosednja stanja. Kot namig ponudimo sosednje stanje z najvišjo oceno. Če bi kot namig ponudili stanje, ki predstavlja skupnega prednika obeh vej, namig ne bi bil uporaben, saj predstavlja predpono poizvedbe, za

katero je študent zahteval namig. Primer vračanja nazaj je prikazan na sliki 4.8. Stanje z oceno 5.6 predstavlja prvega skupnega prednika napačne veje in pravilne veje reševanja. Najboljši naslednik skupnega prednika je stanje z oceno 22.1. Tako bi v primeru, da je študent v stanju z oceno -100.0 dobil namig, ki ga vodi v stanje z oceno 22.1 in ne v stanje z oceno -99.0.



Slika 4.8: Primer vračanja iz napačne veje reševanja.

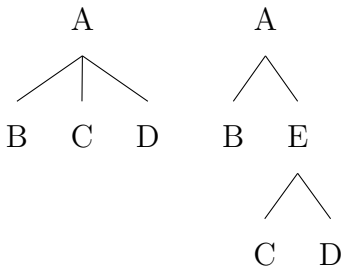
Drugi primer predstavlja stanje z višjo oceno, do katerega vodi običajna povezava. Takrat preprosto ponudimo namig za naslednje stanje. V primeru večih stanj z višjo ocen, izberemo tistega z najvišjo. V kolikor ne najdemo ujemajočega stanja, ponudimo namig za najbližje stanje t. j. stanje, ki je najbolj podobno stanju študenta. Generiranje namigov predstavlja pretvorbo drevesne strukture poizvedbe v niz SQL. Študentu se nato vrne nova poizvedba, ki se v spletni aplikaciji primerja s staro poizvedbo (angl. diff).

4.5 Primerjava drevesnih struktur

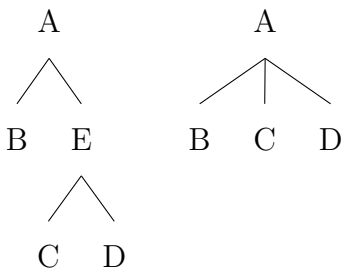
Zmožnost primerjave drevesnih struktur je v sistemu ključnega pomena. V kolikor želimo določiti, ali sta drevesni strukturi enaki ali različni, lahko postopamo podobno kot pri iskanju razlik med dvema nizoma. Običajen pristop za iskanje razlik v nizih se poslužuje računanja razdalje med nizi (angl. Levenshtein distance). Razdalja med dvema nizoma je definirana kot najmanjše

število operacij, ki pretvorijo prvi niz v drugi niz. Operacijo predstavlja brisanje znaka, dodajanje znaka ali spreminjanje obstoječega znaka. Podobno kot pri nizih lahko tudi pri drevesnih strukturah definiramo osnovne operacije [22].

Namesto dodajanja znaka lahko vpeljemo operacijo dodajanja vozlišča. V kolikor dodamo vozlišče kot otroka določenemu očetu, postanejo obstoječi otroci desno od dodanega vozlišča novi otroci dodanega vozlišča. Primer dodajanja vozlišča je prikazan na sliki 4.9. Analogno lahko operaciji brisanja znaka priredimo operacijo brisanja vozlišča. Kadar vozlišče brišemo, povežemo otroke brisanega vozlišča z očetom brisanega vozlišča. Operacija brisanja je tako komplementarna operaciji dodajanja vozlišča. Primer brisanja je prikazan na sliki 4.10.

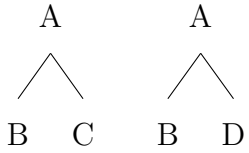


Slika 4.9: Operacija dodajanja vozlišča E v drevo kot otroka vozlišča A na pozicijo 2



Slika 4.10: Operacija brisanja vozlišča E

Definiramo še operacijo preimenovanja vozlišča. Vozlišče preprosto preimenujemo tako, da mu spremenimo oznako. Primer preimenovanja vozlišča je prikazan na sliki 4.11.



Slika 4.11: Preimenovanje vozlišča C v D

Operacije v drevesu lahko predstavimo kot par

$$(a, b) \neq (\Lambda, \Lambda) \tag{4.15}$$

kjer je Λ *prazno vozlišče,* (a, Λ) *operacija brisanja vozlišča,* (Λ, b) *operacija dodajanja vozlišča,* (a, b) *operacija preimenovanja vozlišča*

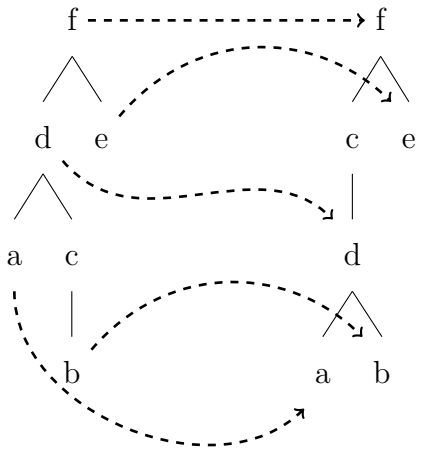
Vsakemu paru iz enačbe 4.15 lahko določimo metriko, ki podaja ceno operacije. Tako lahko posameznim operacijam določimo večjo ceno ali pa uporabimo posebno funkcijo za določanje cene operacije, glede na npr. položaj vozlišč v drevesu. Razdalja med drevesoma je določena kot minimalna cena operacij, potrebnih za preureditev prvega drevesa v drugo drevo.

Naj bo $\gamma(a \to b) \geq 0$ *cena operacije* (a, b) *in* $S = s_1 ... s_k$ *sekvenca operacij. Razdalja med drevesom* T_1 *in* T_2 *je definirana kot*

$$\delta(T_1, T_2) = \min_S \gamma(S) \tag{4.16}$$

pri čemer je $\gamma(S) = \sum_{i=1}^{|S|} \gamma(s_i)$

Ker je γ metrika, je tudi razdalja med drevesoma δ metrika [22]. Sekvenco operacij nad drevesom lahko grafično predstavimo s preslikavo na sliki 4.12.



Slika 4.12: Preslikava med vozlišči prvega in drugega drevesa [22]

Preslikava na sliki 4.12 prikazuje, kako lahko preuredimo prvo drevo v drugo drevo. Vozlišča, ki so povezana s črtkano črto, predstavljajo operacijo preimenovanja vozlišča. Ker se imena vozlišč ujemajo, smo opravili operacijo preimenovanja s ceno 0. Vozlišča v prvem drevesu, ki niso povezana s črtkano črto, se izbrišejo. Vozlišča iz drugega drevesa, ki niso povezana s črtkano črto, se dodajo v prvo drevo. Preslikavo lahko tudi formalno zapišemo.

Naj bo N_1 *število vozlišč v drevesu* T_1*,* N_2 *število vozlišč v drevesu* T_2*,* M *množica parov* (i, j) *ter naj velja* $1 \leq i \leq N_1$ *in* $1 \leq j \leq N_2$*. Trojica* (M, T_1, T_2) *je preslikava med drevesoma* T_1 *in* T_2 *v kolikor za poljuben par* $(i_1, j_1) \in M, (i_2, j_2) \in M$ *velja:*

$$i_1 = i_2 \iff j_1 = j_2 \tag{4.17}$$

$$Left(i_1, i_2) \iff Left(j_1, j_2) \tag{4.18}$$

$$Ancestor(i_1, i_2) \iff Ancestor(j_1, j_2) \tag{4.19}$$

$Left(a, b)$ *podaja relacijo urejenosti otrokov vozlišča t. j. vozlišče a je levo od vozlišča b.* $Ancestor(a, b)$ *podaja relacijo prednika t. j. vozlišče a je prednik vozlišča b.*

Zgornje relacije podajajo osnovne zahteve za obstoj preslikave. Med drugim niso dovoljene preslikave vozlišča iz prvega drevesa v več vozlišč drugega drevesa, prav tako je potrebno ohraniti vrstni red otrok in relacije prednik–potomec. Razdaljo med drevesoma lahko definiramo tudi s pomočjo preslikave.

Razdalja med drevesom T_1 in T_2 je definirana kot

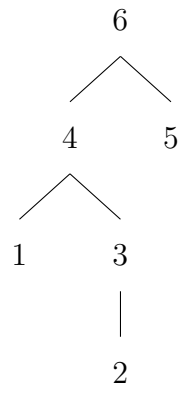
$$\delta(T_1, T_2) = \min_M \gamma(M) \tag{4.20}$$

pri čemer je M veljavna preslikava med drevesoma T_1 in T_2

4.5.1 Algoritem Zhang–Shasha

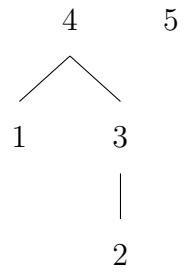
Algoritem Zhang–Shasha [22] je le eden izmed mnogih algoritmov za računanje razdalje med drevesnimi strukturami. Podobno kot mnogo drugih algoritmov deluje po pristopu dinamičnega programiranja za izboljšanje časovne zahtevnosti. Algoritem razširi koncept dreves in namesto razdalje med drevesi računa razdaljo med gozdovi. Drevo je namreč le poseben primer gozda. Za delovanje algoritem uporablja vrsto posebnih vozlišč in notacij, ki so opisane v nadaljevanju.

$T[i]$ predstavlja i-to vozlišče v obratnem obhodu drevesa T (angl. postorder traversal) [22]. Pri obratnem obhodu drevesa najprej obiščemo najbolj levo poddrevo, nato desna poddrevesa, na koncu korensko vozlišče. Primer obratnega obhoda ter indeksov vozlišč je prikazan na sliki 4.13.



Slika 4.13: Obratni obhod drevesa

$l(i)$ označuje najbolj levega potomca vozlišča [22]. Za primer iz slike 4.13 je tako $l(6) = l(4) = l(1) = 1$. $T[i..j]$ označuje urejen gozd, pridobljen z vozlišči i do j drevesa T. Gozd je urejen v smislu naraščajočih oznak korenov dreves. Primer razpada drevesa T iz slike 4.13 na gozd je prikazan na sliki 4.14.



Slika 4.14: Primer gozda za $T[1..5]$ iz slike 4.13

Razdaljo med gozdovoma označujemo z $forestdist$. Za razdaljo med gozdovoma veljajo določene zakonitosti podane v enačbah (4.21)-(4.23).

Za razdaljo med gozdovoma velja

$$forestdist(\varnothing, \varnothing) = 0 \tag{4.21}$$

$$forestdist(T_1[l(i_1)..i], \varnothing) = forestdist(T_1[l(i_1)..i-1], \varnothing) + \gamma(T_1[i] \to \Lambda) \tag{4.22}$$

$$forestdist(\varnothing, T_2[l(j_1)..j]) = forestdist(\varnothing, T_2[l(j_1)..j-1]) + \gamma(\Lambda \to T_2[j]) \tag{4.23}$$

pri čemer je i_1 katerikoli prednik vozlišča i, j_1 katerikoli prednik vozlišča j

Enačba (4.21) je samoumevna, saj potrebujemo natanko 0 operacij za transformacijo praznega gozda v drug prazen gozd. Enačba (4.22) opisuje operacijo brisanja vozlišča iz gozda. Za transformacijo gozda v prazen gozd moramo pobrisati vsa vozlišča. Iz drevesa odstranimo vozlišče z največjim številom in nadaljujemo postopek rekurzivno nad preostalim gozdom, pri tem pa prištejemo ceno operacije brisanja. Postopek ponavljamo, dokler v gozdu ni več vozlišč. Enačba (4.23) opisuje dodajanje vozlišča, pri čemer brišemo vozlišča iz drugega gozda. Cilj je transformirati prazen gozd v ustrezen gozd. V ta namen moramo dodati vsa vozlišča drugega gozda v prvi gozd.

Razdalja med gozdovoma je podana kot

$$forestdist(T_1[l(i_1)..i], T_2[l(j_1)..j]) = \min \begin{cases} (4.25) \\ (4.26) \\ (4.27) \end{cases} \tag{4.24}$$

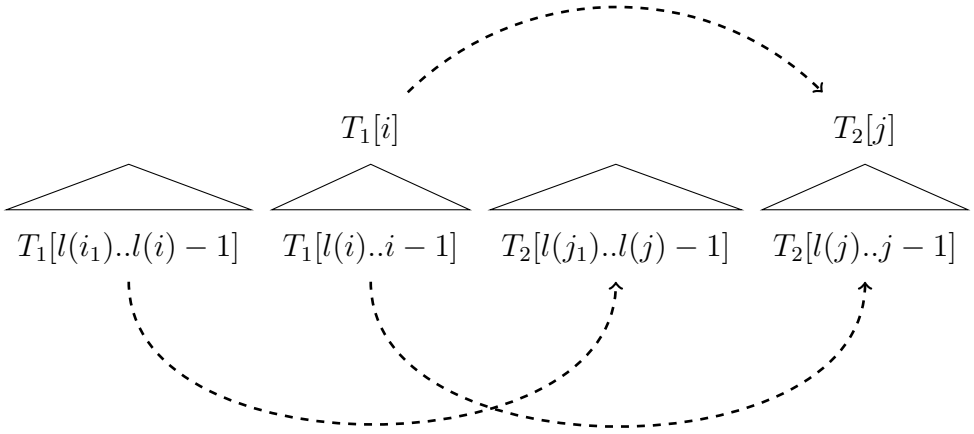
$$forestdist(T_1[l(i_1)..i-1], T_2[l(j_1)..j]) + \gamma(T_1[i] \to \Lambda) \tag{4.25}$$

$$forestdist(T_1[l(i_1)..i], T_2[l(j_1)..j-1]) + \gamma(\Lambda \to T_2[j]) \tag{4.26}$$

$$forestdist(T_1[l(i_1)..l(i)-1], T_2[l(j_1)..l(j)-1]) + forestdist(T_1[l(i)..i-1], T_2[l(j)..j-1]) + \gamma(T_1[i] \to T_2[j]) \tag{4.27}$$

pri čemer je i_1 katerikoli prednik vozlišča i, j_1 katerikoli prednik vozlišča j

Enačbi (4.25) in (4.26) sta podobni kot enačbi (4.22) in (4.23), le da sta razširjeni na primerjavi med nepraznima gozdovoma. Posebna pazljivost je potrebna pri operaciji preimenovanja vozlišča, v enačbi (4.27). V procesu preimenovanja vozlišča lahko namreč drevo razpade na gozd, preimenovanje vozlišč pa mora spoštovati relacijo potomcev glede na enačbo (4.19) [22]. Tako na sliki 4.15 ne bi bila pravilna preslikava med npr. prvim gozdom in drevesom s korenskim vozliščem $T_2[j]$.



Slika 4.15: Pravilne preslikave v okviru operacije preimenovanja vozlišča v gozdu glede na enačbo (4.27) [22]

Opazimo, da je omenjena pazljivost potrebna le v primerih, ko bodisi prvo bodisi drugo drevo razpade na gozd oziroma če velja $l(i) \neq l(i_1)$ ali $l(j) \neq l(j_1)$ [22]. Enačbo (4.27) lahko zato ločimo na dva primera. Prvi primer opisuje operacijo preimenovanja vozlišča, pri čemer drevo ne razpade na gozd. Takrat lahko preprosto nadaljujemo z računanjem razdalje med gozdovoma za obe drevesi, katerim odstranimo vozlišči iz operacije preimenovanja. V kolikor eno izmed dreves ali obe drevesi razpadeta na gozd, nadaljujemo z računanjem razdalje med gozdovi ter izračunamo razdaljo med drevesoma, kjer koren drevesa predstavljata vozlišči iz operacije preimenovanja.

Enačbo (4.27) *lahko zapišemo kot*

$$(4.27) = \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j-1) \\ + \gamma(T_1[i] \to T_2[j]); \qquad l(i) = l(i_1) \wedge l(j) = l(j_1) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ + treedist(i,j); \qquad drugače \end{cases} \tag{4.28}$$

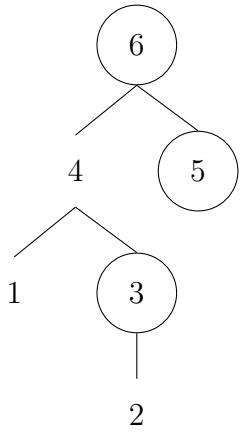
pri čemer je $treedist(i,j)$ *podaja razdaljo med drevesoma s korenom* i *in* j.

Enačba (4.28) nakazuje uporabo dinamičnega programiranja za reševanje enačbe. Dodatno, ker so vozlišča i_1 in j_1 predniki vozlišč i in j, potrebujemo za izračun $treedist(i_1, j_1)$ sprva izračunane $treedist(i, j)$. Omenjena opazka narekuje uporabo pristopa od spodaj navzgor za izračun razdalj. Algoritem ne opravlja izračuna razdalj med vsemi kombinacijami vozlišč v drevesih, temveč uvede poseben tip vozlišč, za katera izračuna razdalje.

Množica vozlišč "keyroot" je definirana kot

$$keyroots(T) = \{k \mid \nexists k' : k' > k \wedge l(k) = l(k')\} \tag{4.29}$$

kjer sta k in k' vozlišči drevesa T



Slika 4.16: Vozlišča "keyroot"

Vozlišča "keyroot" izhajajo iz enačbe (4.28), saj opazimo, da v kolikor se vozlišče i nahaja na poti od vozlišča $l(i_1)$ in vozlišča i_1 in obenem vozlišče j na poti od vozlišča $l(j_1)$ in j_1, nam ni potrebno računati $treedist(i, j)$, temveč lahko dobimo rezultat kot stranski produkt računanja $treedist(i_1, j_1)$ [22]. Primer vozlišč "keyroot" za drevo iz slike 4.13 je prikazan na sliki 4.16.

Algoritem 2 Zhang–Shasha

1: **procedure** ZHANGSHASHA(T_1,T_2)
2: Compute keyroots, left-most leaf descendants of T_1 and T_2
3: **for** $k \leftarrow 1$ to $\mid keyroots(T_1) \mid$ **do**
4: **for** $l \leftarrow 1$ to $\mid keyroots(T_2) \mid$ **do**
5: TREEDIST($keyroots(T_1, k)$, $keyroots(T_2, l)$)
6: **end for**
7: **end for**
8: **end procedure**
9: **procedure** TREEDIST(i, j)
10: $forestdist(\varnothing, \varnothing) \leftarrow 0$
11: **for** $i_1 \leftarrow 1$ to i **do**
12: $forestdist(T_1[l(i)..i_1], \varnothing) \leftarrow$
$forestdist(T_1[l(i)..i_1 - 1], \varnothing) + \gamma(T_1[i_1] \rightarrow \Lambda)$
13: **end for**
14: **for** $j_1 \leftarrow 1$ to j **do**
15: $forestdist(\varnothing, T_2[l(j)..j_1]) \leftarrow$
$forestdist(\varnothing, T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])$
16: **end for**
17: **for** $i_1 \leftarrow l(i)$ to i **do**
18: **for** $j_1 \leftarrow l(j)$ to j **do**
19: $tmp \leftarrow min\{$
$forestdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \rightarrow \Lambda),$
$forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])\}$
20: **if** $l(i_1) = l(i) \wedge l(j_1) = l(j)$ **then**
21: $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) \leftarrow min\{tmp,$
$forestdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1 - 1])$
$+ \gamma(T_1[i_1] \rightarrow T_2[j_1])\}$
22: $treedist(i_1, j_1) = forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1])$
23: **else**
24: $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) \leftarrow min\{tmp,$
$forestdist(T_1[l(i)..l(i_1) - 1], T_2[l(j)..l(j_1) - 1])$
$+ treedist(i_1, j_1)\}$
25: **end if**
26: **end for**
27: **end for**
28: **end procedure**

4.6 Komponenta za evalvacijo poizvedb

Kot komponento za evalvacijo poizvedb se uporablja obstoječe orodje *SQLer*, razvito v okviru predmeta Osnove podatkovnih baz za potrebe vrednotenja poizvedb. Orodje za delovanje potrebuje delujočo povezavo s podatkovno bazo, kjer so shranjene sheme, nad katerimi se preverjajo naloge. Orodje deluje na podlagi primerjave rezultatov poizvedbe študenta ter rezultatov idealne poizvedbe (rešitve naloge). Za vsako nalogo lahko določimo vrsto koeficientov, s katerimi dajemo poudarek na posamezne dele poizvedbe, kot je npr. količina odvečnih atributov v rezultatu, vrstni red atributov, pravilno poimenovanje atributov ... Orodje sprva preveri ujemanje atributov med obema rezultatoma poizvedb. V kolikor obstajajo odvečni atributi v rezultatu ali so napačno poimenovani, se, glede na koeficient, zniža število doseženih točk. Poleg atributov se preverijo tudi rezultirajoče n-terice (angl. tuple), med drugim se preveri pravilni vrstni red in število n-teric. Za primerjavo atributov se uporablja metrika razdalje med nizi, in sicer algoritem Levenshtein.

Poglavje 5

Evalvacija sistema

Evalvacija sistema je potekala na podlagi študije primerov. Primeri so bili izbrani glede na težave, s katerimi se srečujejo študentje, opisane v poglavju 2. Testiranje je potekalo tudi z uporabo obstoječih zgodovinskih podatkov o reševanju nalog, saj nismo imeli na voljo možnosti testiranja v učilnici. Testni podatki zato niso zanesljivi in služijo le kot preprost vpogled v delovanje sistema. Študija primerov je potekala za sledeče scenarije:

1. Študent se loti reševanja naloge na popolnoma napačen način.
2. Študent želi rešiti nalogo zgolj z namigi.
3. Študent delno reši nalogo in želi namig za nadaljno reševanje.

Za vsak omenjen scenarij so bile izbrane reprezentativne naloge s težavnostno stopnjo od najlažje do najtežje naloge. Poleg izbire naloge so bili izbrani tudi ustrezni zgodovinski podatki, ki so služili kot študentov vnos k nalogam.

Scenarij 1: napačen pristop k reševanju

V okviru 1. scenarija smo preverjali, do kolikšne mere je sistem sposoben ponuditi namig, v kolikor študent ubere popolnoma napačen pristop k reševanju naloge. Rezultati testiranja so podani v tabeli 5.1. Za naloge z nizko

težavnostjo testiranja nismo opravili, saj so pri takšnih nalogah študentje skoraj vedno izbrali pravilen pristop.

Idealna rešitev	Rešitev študenta	Prvi in drugi namig

`SELECT COUNT(*) FROM zaposleni, oddelek WHERE zaposleni. ID_oddelek = oddelek.ID_oddelek AND oddelek.Ime = ' SALES';`	`SELECT * FROM ODDELEK;`	`SELECT COUNT(*) FROM zaposleni WHERE ID_oddelek`
		
		`SELECT COUNT(*) FROM zaposleni WHERE ID_oddelek IN ( SELECT ID_oddelek )`

`SELECT * FROM Zaposleni WHERE ID_oddelek IN ( SELECT ID_oddelek FROM Zaposleni GROUP BY ID_oddelek HAVING COUNT(ID_oddelek) >3 AND ID_oddelek =30);`	`SELECT z1.id_zaposleni , z1.priimek, z1. ime, z1.vzdevek, COUNT(z1. ID_zaposleni) FROM zaposleni z1, zaposleni z2 WHERE z1.ID_oddelek = 30`	`SELECT * FROM Zaposleni WHERE ID_oddelek`
		
		`SELECT * FROM zaposleni WHERE ID_oddelek = 30`

`SELECT * FROM Zaposleni WHERE Placa > (SELECT MAX(Placa) FROM Zaposleni, Delo WHERE Zaposleni. ID_delo=Delo. ID_delo AND Funkcija=' Salesperson');`	`SELECT MAX(z.Placa) FROM ZAPOSLENI z, DELO d WHERE z.ID_Delo = d. ID_Delo AND d. Funkcija = ' SALESPERSON'`	`SELECT * FROM ZAPOSLENI z1 WHERE Placa`
		
		`SELECT * FROM ZAPOSLENI z1 WHERE Placa > ALL ( SELECT z2.Placa )`

Tabela 5.1: Študija primerov za scenarij 1

Opazimo, da sistem ponuja namige, ki obsegajo začetne dele poizvedb.

Razlog je v tem, da se študent nahaja v popolnoma napačni veji, iz katere ga ne moremo preusmeriti nazaj do pravilne veje. Zato mu, kot je že bilo omenjeno, ponudimo namig za eno izmed začetnih stanj. Dolžina, do katere preiščemo začetna stanja za iskanje najbolj optimalnega stanja, je eden izmed parametrov sistema. V sklopu testiranja je bil parameter nastavljen na 2, kar pomeni, da preiščemo začetno stanje in njegove naslednike. Iz prve vrstice tabele 5.1 je razvidno, da sistem študenta z namigi ne usmerja po poti idealne rešitve, temveč po poti rešitve drugega študenta, kar je zaželeno, saj tako študent preizkusi tudi nove načine reševanja. Natančen vpogled v 2. vrstico tabele 5.1 prikazuje eno izmed trenutnih slabosti sistema. Nekaj študentov je namreč namesto splošne (in pravilne) poizvedbe SQL, rešilo nalogo tako, da so fiksirali vrednost pogoja (`WHERE ID_oddelek = 30`). Ker se pravilnost poizvedbe preverja le nad rezultatom izvajanja poizvedbe s pomočjo komponente za evalvacijo poizvedb, je tudi takšen način reševanja pravilen. Vendar pa z vidika učenja ni primeren, zato bi sistem moral biti zmožen izločiti poizvedbe s fiksiranimi pogoji. Z vidika uporabnosti namigov so le-ti ustvarjeni tako, da študenta usmerjajo v pravilno smer.

Scenarij 2: reševanje zgolj z namigi

Scenarij št. 2 je preverjal, do kolikšne mere lahko sistem samostojno rešuje naloge. Predpostavili smo, da študent napiše le osnoven `SELECT *` stavek, nato pa nalogo reši zgolj z namigi. Stavek `SELECT *` je potreben, saj sistem ne dovoli generiranja namigov brez vhodne poizvedbe. Rezultati testiranja so prikazani v tabeli 5.2. Prikazani so le končni namigi t. j. namigi, ki vrnejo zadnjo pravilno poizvedbo. Rezultati iz tabele izpostavljajo dejstvo, da je sistem sposoben samostojnega reševanja nalog, saj se zanaša na zbirko znanja, ki so jo zgradili študentje. Kljub temu sistem zaenkrat še ni zmožen izbirati elegantnejših rešitev in tako v določenih primerih ponudi študentu kot namig daljšo, a vseeno pravilno poizvedbo.

Idealna rešitev	Končni namig

`SELECT *`
`FROM zaposleni`
`WHERE Provizija IS NULL;`
	`SELECT *`
	`FROM zaposleni`
	`WHERE ID_zaposleni NOT IN ( SELECT`
	`ID_zaposleni`
	`FROM zaposleni z1`
	`WHERE z1.provizija > 0`
	`)`

`SELECT o.Ime, COUNT(*)`
`FROM zaposleni z, oddelek o`
`WHERE z.ID_oddelek = o.ID_oddelek`
`GROUP BY o.ID_oddelek;`
	`SELECT o1.Ime, COUNT(z1.ID_zaposleni`
	`) AS 'Stevilo_zaposlenih'`
	`FROM oddelek o1, zaposleni z1`
	`WHERE z1.ID_oddelek = o1.ID_oddelek`
	`GROUP BY o1.ID_oddelek`

`SELECT *`
`FROM Zaposleni`
`WHERE ID_zaposleni IN (SELECT`
`ID_zaposleni FROM Zaposleni`
`WHERE ID_zaposleni NOT IN (`
`SELECT ID_nadrejeni FROM`
`Zaposleni));`
	`SELECT *`
	`FROM zaposleni z1`
	`WHERE z1.ID_zaposleni NOT IN (`
	`SELECT DISTINCT z2.ID_nadrejeni`
	`FROM zaposleni z2`
	`)`

Tabela 5.2: Študija primerov za scenarij 2

Scenarij 3: dopolnjevanje rešitve z namigi

Scenarij št. 3 predstavlja situacijo, kjer študent delno reši nalogo, nato pa ne zna nadaljevati z reševanjem. Situacija narekuje generiranje namiga, ki študenta "potisne" v pravo smer reševanja. Rezultati testiranja so prikazani v tabeli 5.3. Prva vrstica prikazuje vračanje iz napačne veje reševanja. Študent je pozabil opraviti stik s tretjo tabelo in se tako znašel v napačni veji. Sistem ga je pravilno opozoril z namigom, ki zahteva uporabo dodatne tabele. Druga vrstica tabele prikazuje preprosto operacijo spremembe

pogoja v `WHERE` sklopu poizvedbe. Sprememba obenem spremeni poizvedbo študenta v pravilno rešitev. Tretja vrstica demonstrira dopolnitev vgnezdene poizvedbe.

Idealna rešitev	Rešitev študenta	Namig

`SELECT COUNT(*)`
`FROM zaposleni z, oddelek o, lokacija l`
`WHERE z.ID_oddelek = o.ID_oddelek AND o.ID_lokacija = l.ID_lokacija AND Regija = 'DALLAS'`
`GROUP BY Regija;`
	`SELECT COUNT(Z.ID_Zaposleni) FROM Zaposleni Z, Lokacija L WHERE Regija= 'DALLAS'`	`SELECT COUNT(z1.ID_zaposleni)`
		`FROM zaposleni z1, lokacija l1, oddelek o1`

`SELECT z.Priimek, n.Priimek`
`FROM zaposleni z, zaposleni n`
`WHERE z.ID_nadrejeni = n.ID_zaposleni;`
	`SELECT z1.priimek,z2.priimek`
	`FROM zaposleni z1, zaposleni z2`
	`WHERE z2.id_zaposleni = z1.id_zaposleni`
		`SELECT z1.priimek, z2.priimek`
		`FROM zaposleni z1, zaposleni z2`
		`WHERE z2.id_zaposleni = z1.id_nadrejeni`

`SELECT *`
`FROM zaposleni`
`WHERE placa = (`
`SELECT MAX(Placa)`
`FROM zaposleni`
`WHERE Placa < (`
`SELECT MAX(Placa)`
`FROM Zaposleni`
`)`
`);`
	`SELECT *`
	`FROM zaposleni`
	`WHERE placa = ( select max(placa) )`
		`SELECT *`
		`FROM zaposleni`
		`WHERE placa = ( SELECT MAX(Placa)`
		`FROM zaposleni`
		`)`

Tabela 5.3: Študija primerov za scenarij 3

Poleg prikazanih rezultatov testiranj iz tabele ne zanikamo obstoja situacij, za katere sistem ni zmožen ponuditi uporabnega namiga. Razlog je

v pomanjkanju zgodovinskih podatkov oziroma variabilnosti le-teh. Tipičen primer je kompleksna rešitev študenta, z mnogimi vgnezdenimi poizvedbami. Obstaja velika verjetnost, da nihče ni reševal naloge na podoben način, zato sistem ni zmožen opraviti ujemanja z boljšim stanjem. Namigi v takšnih primerih vodijo študenta nazaj v eno izmed začetnih stanj in potemtakem niso uporabni. Možnih izboljšav je več. Ena pomembnejših je zamenjava načina ujemanja med stanji. Trenutno ujemanje poteka z uporabo striktnega algoritma za primerjanje drevesnih struktur. Boljša alternativa bi bila določitev ključnih objektov iz vsake poizvedbe in nato opraviti ujemanje med takšnimi objekti. Tako bi se med drugim tudi izognili potrebi po preimenovanju aliasov.

Možnosti izboljšav vidimo tudi v samem ujemanju med drevesnimi strukturami. Trenutno sistem striktno preverja vrstni red atributov v `SELECT` sklopu. Vrstni red atributov ni ključen za proces učenja, zato bi bilo smiselno vpeljati nestriktno preverjanje vrstnega reda. Poleg vrstnega reda atributov je smiselno izboljšati samo obliko namigov. Namigi, ki se vrnejo študentu, vedno vsebujejo nova imena aliasov zaradi preimenovanja imen referenc tabel. Ker je namig predstavljen kot poizvedba drugega študenta, ni prilagojen osebi, ki rešuje nalogo. V prihodnje bi zato cilj sistema moral biti generiranje individualnih namigov.

Morebitna dodatna slabost sistema je v generiranju korakov rešitve oziroma predpostavki, da študentje rešujejo poizvedbo po sklopih. Kot je razvidno iz rezultatov testiranj, so namigi močno usmerjeni v reševanje po pristopu zgoraj–navzdol, zaradi narave korakov rešitev. Zaradi omejenih zgodovinskih podatkov smo bili primorani sprejeti omenjeno predpostavko reševanja po sklopih. V kolikor bi se sistem uporabljal v prihodnosti, bi lahko zamenjali strategijo generiranja korakov rešitev s sprotnim pošiljanjem poizvedb na strežnik. Tako bi dobili bolj realne podatke o postopku reševanja študentov.

Poglavje 6

Zaključek

V okviru diplomskega dela je bil razvit sistem za lajšanje učenja jezika SQL. Sistem uporablja znanje, skrito v preteklih podatkih o reševanju nalog za posredovanje namigov. Temeljne prednosti sistema tako postanejo namigi, prilagojeni trenutni rešitvi študenta. Zgodovinski podatki so pomembna baza znanja, ki razbremenjuje učitelje in omogoča izboljšan proces učenja. Ker so podatki vodilo sistema, je pomembno, da so kvalitetni in številčni. V poglavju 2 so bili predstavljeni inteligentni sistemi za učenje ter pogoji, ki jih morajo izpolnjevati. Na mestu je analiza izpolnjenosti teh zahtev. Prva zahteva predpisuje, da naj bi bil sistem zmožen samostojno reševati naloge iz podane domene. V poglavju 5 smo ugotovili, da je kot posledica obširne baze znanja zahteva izpolnjena. Druga zahteva vztraja pri razpoznavanju, do kolikšne mere je študent usvojil znanje, ki ga želimo podati. Osredotočenost sistema na namige pomeni, da te zahteve ne izpolnjuje oziroma jo izpolnjuje le delno. Tretja zahteva določa, da mora biti sistem zmožen prilagajati težavnost nalog glede na znanje študenta. Ker zahteva logično sledi iz 2. zahteve, ji naš sistem ne zadostuje. Trdimo lahko torej, da smo pripravili adaptiven sistem za lajšanje učenja jezika SQL, ki pa še ni inteligenten v pravem pomenu besede. Snovanje sistemov ITS je dolgotrajen in kompleksen proces, preobsežen za okvire diplomskih nalog. Naš sistem tako predstavlja izhodišče za prihodnje nadgradnje na področju učenja jezika SQL. Pokazali

smo, da je sistem sposoben dajati uporabne namige in usmerjati študente v pravo smer, hkrati pa jih ne omejuje. Tako je proces učenja kompleksnega jezika, kot je SQL, močno olajšan. Seveda ne smemo zanemariti možnosti izboljšav sistema. Predvsem so možne izboljšave na področju individualizacije namigov, tako da bodo le-ti zares prilagojeni posameznemu študentu.

Literatura

[1] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. Cognitive tutors: Lessons learned. *The journal of the learning sciences*, 4(2):167–207, 1995.

[2] Oracle Corporation. Jersey. `https://jersey.java.net/`. [Dostopano 22. 7. 2015].

[3] Charles Fry. Java caching with guava. `https://guava-libraries.googlecode.com/files/JavaCachingwithGuava.pdf`. [Dostopano 29. 7. 2015].

[4] Brad Green and Shyam Seshadri. *AngularJS*. "O'Reilly Media, Inc.", 2013.

[5] Intelligent Computer Tutoring Group. Sql-tutor: an its for sql programming. `http://ictg.canterbury.ac.nz/projects/sql-tutor`. [Dostopano 19. 8. 2015].

[6] John Gruber. Daring fireball: Markdown syntax documentation. `http://daringfireball.net/projects/markdown/syntax`. [Dostopano 19. 8. 2015].

[7] Red Hat. Weld: Home. `http://weld.cdi-spec.org/`. [Dostopano 22. 7. 2015].

[8] Matthew P. Jarvis, Goss Nuzzo-Jones, and Neil T. Heffernan. Applying

machine learning techniques to rule generation in intelligent tutoring systems. In *Intelligent Tutoring Systems*, pages 541–553. Springer, 2004.

[9] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Devika Gollapudi, Kim Haase, William Markito, and Chinmayee Srivathsa. *The Java EE 6 Tutorial: Advanced Topics*. Addison-Wesley, 2013.

[10] Faster XML LLC. JacksonHome - FasterXML Wiki. `http://wiki.fasterxml.com/JacksonHome`. [Dostopano 23. 7. 2015].

[11] Jacob Thornton Mark Otto and Bootstrap contributors. Bootstrap; The world's most popular mobile-first and responsive front-end framework. `http://getbootstrap.com/`. [Dostopano 23. 7. 2015].

[12] Brent Martin. Constraint-based modelling: Representing student knowledge. *New Zealand Journal of Computing*, 7(2):30–38, 1999.

[13] Brent I. Martin. *Intelligent Tutoring Systems: The practical implementation of constraint-based modelling*. PhD thesis, 2002.

[14] Antonija Mitrovic. Learning sql with a computerized tutor. *ACM SIGCSE Bulletin*, 30(1):307–311, 1998.

[15] Martha C. Polson and J. Jeffrey Richardson. *Foundations of intelligent tutoring systems*. Psychology Press, 2013.

[16] James Power. Notes on formal language theory and parsing. *National University of Ireland, Maynooth, Kildare*, 2002.

[17] Julia Coleman Prior and Raymond Lister. The backwash effect on sql skills grading. *ACM SIGCSE Bulletin*, 36(3):32–36, 2004.

[18] Apache Tomcat Project. Apache Tomcat. `http://tomcat.apache.org/`. [Dostopano 22. 7. 2015].

[19] Martin L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[20] David Silver. Lecture 2: Markov decision process. `http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/MDP.pdf`. [Dostopano 5. 8. 2015].

[21] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*, pages 71–78, 2008.

[22] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.