Kontekstno ispravljanje pravopisnih pogrešaka primjenom n-gramskog sustava

Matošević, Teo

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1323

KONTEKSTNO ISPRAVLJANJE PRAVOPISNIH POGREŠAKA PRIMJENOM N-GRAMSKOG SUSTAVA

Teo Matošević

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1323

KONTEKSTNO ISPRAVLJANJE PRAVOPISNIH POGREŠAKA PRIMJENOM N-GRAMSKOG SUSTAVA

Teo Matošević

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1323

Pristupnik:	**Teo Matošević (0036542778)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Gordan Gledec

Zadatak: **Kontekstno ispravljanje pravopisnih pogrešaka primjenom n-gramskog sustava**

Opis zadatka:

Usluga strojne provjere pravopisa "ispravi.me - hrvatski akademski spelling checker" kontinuirano se unapređuje funkcionalnostima koje poboljšavaju iskustvo upotrebe. U sklopu usluge razvija se i n-gramski sustav hrvatskoga jezika. N-gramski se sustavi koriste u modeliranju jezika za analizu teksta, a u obzir uzimaju ograničenu povijest ili ovisnost n riječi. U sklopu rada usluge ispravi.me kao nužan element kvalitetnog ispravljanja pravopisnih i gramatičkih pogrešaka pokazao se upravo n-gramski sustav. Cilj je završnog rada proučiti dosadašnji razvoj usluge strojne provjere pravopisa i korištenjem n-grama omogućiti kontekstno ispravljanje pravopisnih pogrešaka, posebno u situacijama kad pogrešno napisana riječ rezultira drugom ispravnom riječi. Radu treba priložiti izvorni i izvršni kod razvijenog sustava te potrebnu dokumentaciju.

Rok za predaju rada: 14. lipnja 2024.

ZAHVALA

Želim se zahvaliti dragom mentoru prof. dr. sc. Gordanu Gledecu na vođenju, usmjeravanju i strpljenju tijekom pisanja završnog rada, čime je omogućio da kvalitetno obradim cjelinu, te organizaciji Srce na pristupu njihovom serveru.

Teo Matošević

Uvod

Ispravljanje pravopisnih pogrešaka u tekstualnim zapisima proces je koji se mora podijeliti na više koraka. Razlog za to leži u raznolikosti problema koji se rješavaju u svakom koraku, pa se prema tome izrađuje programska potpora prilagođena specifičnoj vrsti problema. Broj koraka, njihov redoslijed i točno što se ispravlja u svakom koraku ovise o jeziku.

Pravopisne pogreške se mogu kategorizirati u dvije skupine, „*non-word*“ i „*real-word*“ pogreške. „*Non-word*“ pravopisne pogreške su riječi koje ne postoje u sklopu nekog jezika. Unutar rečenice „*On od mefe zahtijeva da dolazim na nastavu.*“, treća riječ po redu („*mefe*“) je primjer „*non-word*“ pogreške zato što ta riječ ne postoji u hrvatskom jeziku. S druge strane spektra pravopisnih greški su riječi koje postoje unutar određenog jezika, ali u kontekstu unutar kojeg se nalaze nisu validne. Te pogreške se zovu „*real-word*“ pogreške. U rečenici „*On od mene zahtjeva da dolazim na nastavu.*“, riječ „*zahtjeva*“ postoji u hrvatskom jeziku, ali u kontekstu priložene rečenice nije ispravna.

Hascheck [1] je programska potpora koja uspješno obavlja zadatak pronalaska i ispravljanja „*non-word*“ skupine pogrešaka. S druge strane, pronalazak i ispravljanje riječi iz „*real-word*“ skupine pogrešaka, Hascheck (usluga dostupna na https://ispravi.me/) odrađuje u samo određenim slučajevima. Cilj ovog rada jest izgradnja sustava koji će moći pronaći i ispraviti pogreške iz „*real-word*“ skupine u sklopu hrvatskog jezika.

Programska potpora koja će omogućavati pronalazak i ispravljanje takvih pogreški, analizirat će kontekst oko riječi koja se provjerava i donosit će zaključke ovisno o frekvencijama n-grama unutar konteksta.

Nastavak dokumenta puža pregled dostupnih resursa i relevantne literature. Sljedeće poglavlje definira predviđeno rješenje i metode izračuna predikcija. U trećem poglavlju definirana je arhitektura sustava s obzirom na predviđeno rješenje iz prethodnog poglavlja, a četvrto poglavlje sadrži implementaciju programske potpore. Rezultati razvijenog sustava prikazani su u petom poglavlju.

1. Usluga strojne provjere pravopisa ispravi.me

Hascheck, poznat i kao Ispravi.me, programska je podrška za računalnu provjeru pravopisa teksta na hrvatskom jeziku. Hascheck je jedna od najstarijih internetskih usluga u Hrvatskoj, koja od 1994. godine besplatno nudi pravopisne provjere. U početku je funkcionirala putem elektroničke pošte, a od 2003. godine dostupna je kao web-aplikacija, pružajući jednostavniji i brži pristup korisnicima. Od 2016. godine, korisnički dio usluge preimenovan je u Ispravi.me, dok Hašek ostaje pogonski mehanizam u pozadini.

Hascheckova baza riječi podijeljena je u tri cjeline: hrvatski općejezični fond, hrvatski posebnojezični fond i engleski općejezični fond. Baza se kontinuirano nadograđuje učenjem novih riječi iz pristiglih tekstova. Sustav je također u mogućnosti naučiti nove riječi iz tekstova pristiglih na obradu. Učenje sustava je nadgledano i zahtjeva ljudski input, čime se osigurava točnost i relevantnost nadogradnji baze.

Kao što je navedeno u uvodu rada, Hascheck je učinkovit u prepoznavanju „*non-word*“ pogreški poput zamijenjenih slova. Primjerice, ako korisnik napiše „knijga“ umjesto „knjiga“, Hascheck će prepoznati da „knijga“ nije riječ u hrvatskom jeziku i predložiti ispravak. Velika prednost Haschecka je mogućnost ispravka stilskih pogrešaka (upotrebu neprikladnih ili redundantnih riječi i izraza) i pleonazma (suvišne riječi koje nepotrebno ponavljaju značenje). Međutim, kontekstualne pogreške iz kojih nastaju riječi koje postoje u sklopu hrvatskog jezika, Hascheck rijetko uspijeva ispraviti. Sustav svoje znanje temelji na obrađenim leksikografskim djelima i statističkom modelu hrvatskog jezika.

Hascheck je iz korisničkih upita 2007. godine započeo prikupljati n-grame [2]. Najmanji n-gram koji se prikuplja je unigram, a najveći je pentagram. N-gramski sustav je podijeljen po duljini n-grama te uz svaki n-gram stoji njegova frekvencija. Sustav se uz svaki korisnički upit nadograđuje, čime se osigurava stalno ažuriranje i poboljšanje baze podataka.

Pokušaj ispravljanja „*real-word*“ skupine pogrešaka problem je kojeg je pokušalo riješiti mnogo ljudi na mnogo različitih načina. Fossati [3] je koristio mješovite trigrame koji uključuju riječi i njihove oznake vrste riječi. Taj pristup omogućava integraciju leksičkih i sintaksnih informacija radi poboljšane kontekstne provjere pravopisa. Ključni elementi metode uključuju uporabu *confusion setova* za određivanje sličnosti riječi. Svaki promatrani trigram sadrži samo jednu riječ, dok su ostale dvije riječi predstavljene oznakom vrste riječi. Na taj način se smanjuje rijetkost podataka.

Confusion set predstavlja skup riječi koje su međusobno slične po izgovoru, pisanju ili značenju, a koje korisnici često zamjenjuju. Primjer jednog *confusion seta* bile bi riječi „*zahtijeva*“ i „*zahtjeva*“. Te dvije riječi se razlikuju u samo jednom slovu, a obje riječi postoje u sklopu hrvatskog jezika.

Wilcox-O'Hearn [4] je u svom radu koristio metodu koju su razvili Mays, Damerau i Mercer 1991. godine [5]. Metoda koristi vjerojatnosti trigramskog modela za procjenu koja rečenica u tekstu je vjerojatno ispravna, s obzirom na vjerojatnosti trigramskih nizova riječi. Prema njihovom modelu, rečenica koja ima nižu vjerojatnost prema trigramskom modelu smatra se potencijalnom pogreškom, pri čemu se alternativne verzije riječi iz trigramskog modela koriste za predlaganje ispravaka.

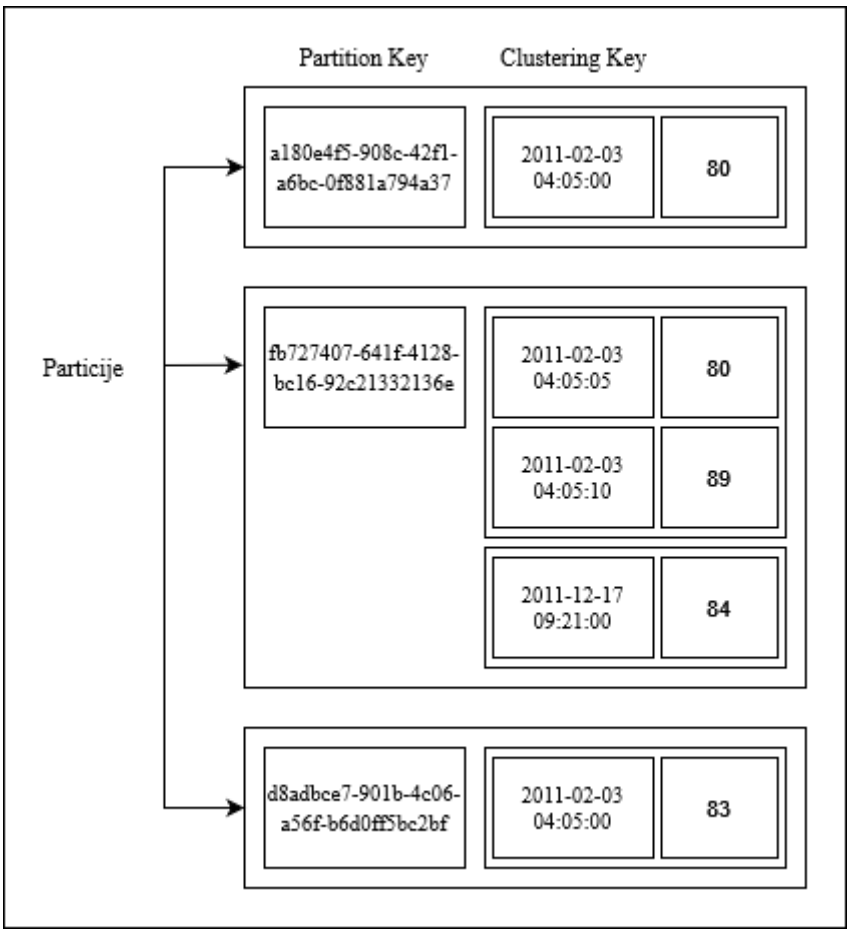
Razvoj sustava koji omogućuje pronalazak i ispravljanje pravopisnih pogrešaka, na standardnoj računalnoj opremi nije moguć. Radi toga, razvoj sustava voditi će se na serveru „*Supek*“ koji nudi organizacija Srce. U sklopu prethodnog projekta, na tom serveru postoji ScyllaDB [6] baza podataka sa unesenim n-gramima te njihovim frekvencijama. U sklopu ovog rada moguć je pristup toj bazi podataka. Uneseni n-grami su preuzeti iz Haschecckovog općejezičkog fonda riječi. ScyllaDB je poznata po svojoj brzini i skalabilnosti, što je idealno za rad s velikim količinama podataka kao što su n-grami.

N-gramskih zapisa potrebnih za korištenje sustava ima jako puno, a način na koji su oni spremljeni u ScyllaDB je detaljno opisano u nastavku. Radi primjera, različitih unigrama postoji preko 3,2 milijuna. S povećanjem duljine n-grama količina različitih n-grama znatno raste, pa tako različitih bigrama ima preko 440 milijuna, a trigrama skoro 1750 milijuna.

ScyllaDB je napisana u programskom jeziku C++ i koristi asinkroni model za postizanje visokih performansi. Zapisi unutar iste mogu, ali ne moraju biti zapisani unutar više particija. ScyllaDB nije relacijska baza podataka, ali način pisanja upita izgleda jako slično jeziku SQL, stoga je jednostavna za korištenje ljudima koji se nisu prije susreli s istom.

Posebnost ScyllaDB je u načinu strukturiranja modela podataka te na načinu pristupa. ScyllaDB je u odnosu na klasične relacijske baze stroža kod definiranja modela podatkovnih struktura koje će se pohranjivati. To svojstvo može stvarati probleme ali, može biti i odlika. U službenoj dokumentaciji piše: „*In ScyllaDB, as opposed to relational databases, the data model is based around the queries and not just around the domain entities. When creating the data model, we take into account both the conceptual data model and the application workflow: which queries will be performed by which users and how often.*“.

Način stvaranja tablica i tipovi podataka su vrlo slični relacijskim bazama, ali se svrha primarnog ključa razlikuje. Primarni ključ sastoji se od *Partition Key* i *Clustering Key*.



Sl. 1 Odnos između *Partition Key* i *Clustering Key*

Partition Key u ScyllaDB određuje u koju će se particiju spremiti taj podatak. Moguće je imati *Partition Key* koji se sastoji od više stupaca tablice. Unutar svake particije se može smjestiti velika količina podataka, pa zbog toga postoji *Clustering Key* koji sortira redove unutar svake particije. Osim što je moguće imati više od jednog stupca tablice kao *Clustering Key*, također je moguće imati samo *Partition Key* nad nekom tablicom i u potpunosti zanemariti *Clustering Key*.

U trenutku kada je potrebno dohvatiti podatke iz neke tablice, u svakom se zahtjevu moraju kao parametri navesti svi stupci koji su dio *Partition Key*. Zbog toga je u dokumentaciji navedeno kako je prvo potrebno analizirati koji će se upiti vršiti nad tablicom. Tek nakon analize upita se implementiraju modeli podataka u obliku tablice, slično kao u SQL standardu. Važno je pametno definirati *Partition Key* i *Clustering Key* kako bi se iskoristila prednost unutarnje arhitekture ScyllaDB. Particije unutar baze ne smiju postati jako velike (pojam velike particije se ne može izraziti nekim brojem nego postoji alat koji traži velike

particije), ali isto tako nije dobro imati samo jedan zapis po svakoj particiji. Uz sva ta ograničenja dobiva se baza podataka čiji su podatci pravilno raspoređeni po internoj strukturi, a pretraga je ekstremno brza i s mnogo zapisa.

2. Predviđeno rješenje programske potpore

U prethodnom poglavlju navedena je mogućnost pristupu podatcima o n-gramima iz baze podataka. Predviđana programska potpora trebala bi moći pretraživati navedene podatke velikom brzinom uz konstantnu vremensku složenost, što znači da vrijeme potrebno za pretragu ne ovisi o količini podataka. Dobiveni podatci će sadržavati frekvencije unigrama svih riječi iz određenog *confusion seta*, te bigrame i trigrame za svaku riječ iz *confusion seta* koji na svojim rubovima sadrže promatranu riječ.

Nakon prikupljanja podataka treba biti omogućena analiza dobivenih podataka. Algoritam koji će analizirati dobivene podatke treba biti fleksibilan i modularan. Njegova izmjena ne smije rezultirati potrebom za mijenjanjem ostatka sustava. Potrebno je omogućiti i dodavanje novih algoritama i migraciju trenutno korištenog algoritma na neki novi algoritam uz minimalnu intervenciju. Takvo rješenje je vrlo važno budući da ne postoji najbolji algoritam koji će uvijek moći pronaći pravopisne pogreške iz strukturiranih podataka u n-gramskom zapisu, ali može se pokušati doći što je bliže moguće.

Ako je skup n-grama definiran sa $W = \{w_1, w_2 \dots\}$, za svaku riječ iz *confusion seta* definirat će se skup n-grama $W_{\dot{u}}$, takav da $\dot{u}$ označava pojedinu riječ iz *confusion seta*, a n-grami unutar skupa su bigrami i trigrami koji na rubovima sadrže riječ $\dot{u}$. Maksimalni broj elemenata tako definiranog skupa $W_{\dot{u}}$ je četiri.

Indeks sigurnosti za pojedini n-gram iz skupa $W_{\dot{u}}$, uz pomoć kojeg će se računati sigurnost pri donošenju odluke, dan je izrazom (1). Što je manji indeks sigurnosti, to je sustav sigurniji u odluku da je riječ $\dot{u}$ prikladna kontekstu.

$$S_{\dot{u}}(w_i) = \left(\frac{f_{\dot{u}}}{N_1}\right) \times \left(\frac{f_{w_i} + 1}{N_{L(w_i)} + V_{L(w_i)}}\right) \tag{1}$$

Ovdje, $\dot{u}$ označava unigram iz *confusion seta*, $f_{\dot{u}}$ označava frekvenciju tog unigrama. N_n označava sumu frekvencija svih n-grama, gdje n označava duljinu n-grama, a V_n označava količinu n-grama, gdje n označava duljinu n-grama. Frekvenciju pojedinog n-grama označava f_{w_i}, a $L(w_i)$ je definirana u nastavku (2). Razlog pribrajanju broja jedan frekvenciji pojedinog n-grama i pribrajanju V_n u N_n jest mogućnost postojanja n-grama sa frekvencijom nula. Radi toga se frekvencije moraju izgladiti. U sklopu ovo rada frekvencije su se izglađivale postupkom koji se zove *Laplace smoothing*.

$$L(w_i) = |\{\text{riječi u } w_i\}| \tag{2}$$

Prethodno je navedena modularnost sustava i algoritma koji će obrađivati dobivene podatke. U sklopu ovog rada razvijena su tri različita algoritma koji će donositi odluke, ali i dalje postoji mogućnost razvijanja novih ili promjena prethodno razvijenih algoritama. Odluke su također definirane kao indeks sigurnosti, s time da se indeks sigurnosti nad nekim skupom računa kombinirajući indekse sigurnosti pojedenih n-grama u tom skupu. Svaki će algoritam na drugačiji način definirat kriterij kombinacije indeksa sigurnosti pojedinih n-grama. Kao i za indekse sigurnosti pojedinih n-grama, tako i za indekse sigurnosti skupa n-grama $W_{\dot{u}}$, što je indeks sigurnosti manji, to je sustav sigurniji u odluku da je riječ $\dot{u}$ prikladna kontekstu. Prvi će algoritam tražiti maksimalni indeks sigurnosti u skupu $W_{\dot{u}}$ i njega vratiti kao rezultat. Taj algoritam dan je izrazom (3).

$$SM(W_{\dot{u}}) = -\log_{10}\left(\max_i\left(S_{\dot{u}}(w_i)\right)\right) \tag{3}$$

Drugi algoritam će kao povratnu vrijednost vratiti sumu svih zasebnih indeksa sigurnosti (4).

$$SS(W_{\dot{u}}) = -\log_{10}\left(\sum_i S_{\dot{u}}(w_i)\right) \tag{4}$$

A treći će algoritam također sumirati zasebne indekse sigurnosti ali će pri odluci prednost dati relativnim frekvencija bigrama i trigrama (6). Taj algoritam će računati pojedine indekse sigurnosti na drugačiji, ali vrlo sličan način (5).

$$S'_{\dot{u}}(w_i, x) = \left(\frac{f_{\dot{u}}}{N_1}\right) \times \left(\frac{f_{w_i} + 1}{N_{L(w_i)} + V_{L(w_i)}}\right)^x \tag{5}$$

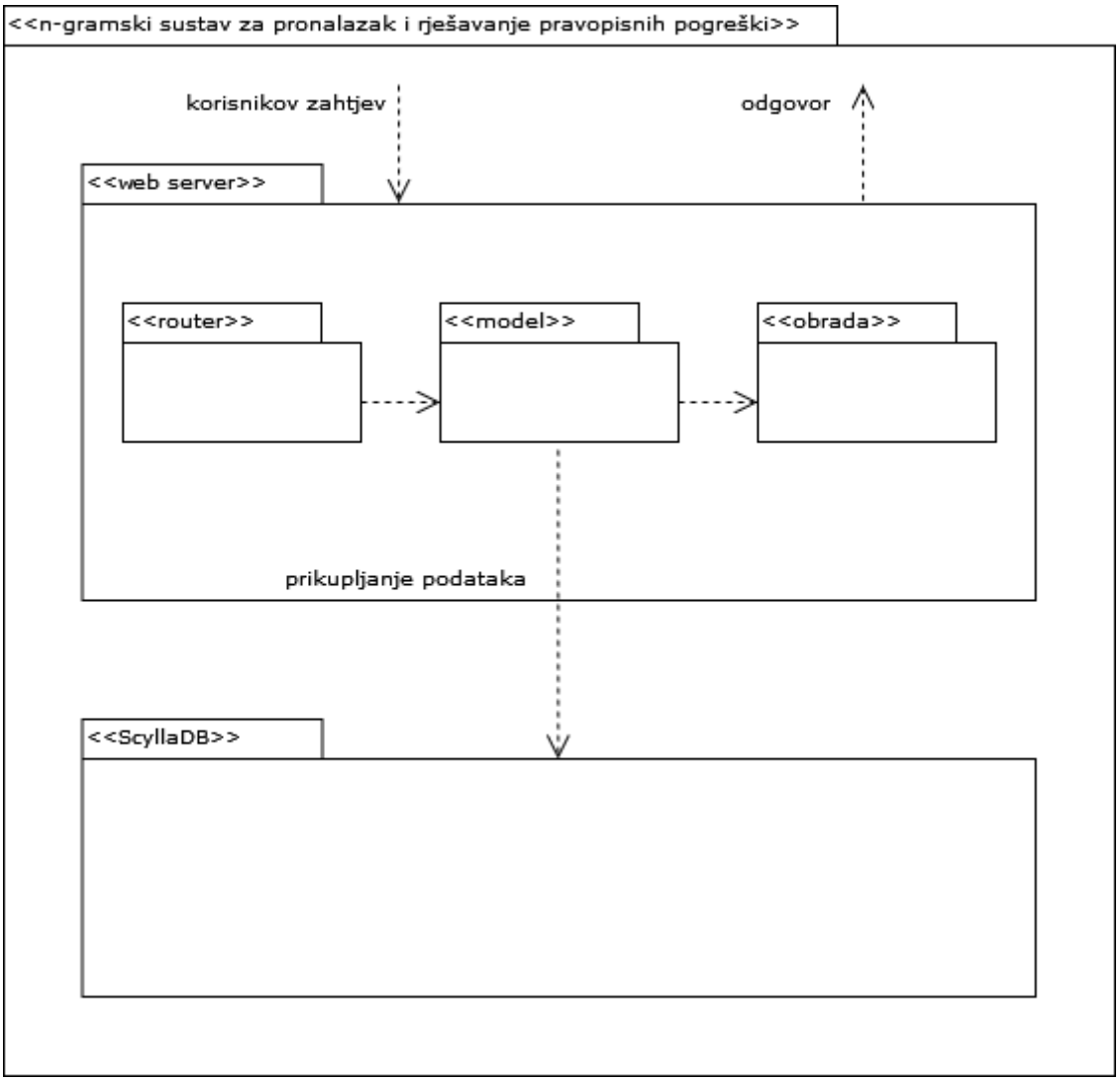
Jedina razlika između definiranih načina računanja indeksa sigurnosti jest to da funkcija u izrazu (5) relativnu frekvenciju promatranog bigrama ili trigrama podiže na neku potenciju x. U slučaju da je $S'_{\dot{u}}$ potrebno svesti na $S_{\dot{u}}$, potrebno je x postaviti na jedan.

$$SP(W_{\dot{u}}) = -\log_{10}\left(\sum_i S'_{\dot{u}}\left(w_i, \frac{1}{2}\right)\right) \tag{6}$$

Funkcija definirana u izrazu (6) daje veću prednost relativnim frekvencijama bigrama i trigrama zato što je predani eksponent manji od 1.

Servis koji će prikupljati podatke te donositi zaključak na kraju treba biti implementiran planski uz mogućnost nadogradnje. Potrebno je da servis bude jednostavan za tumačenje ostalim ljudima i skalabilan, što znači da može podnijeti povećanje opsega posla bez smanjenja performansi.

Programska potpora za pronalazak i ispravljanje pravopisnih pogrešaka uz pomoć konteksta je primarno namijenjena krajnjim korisnicima, stoga je važno omogućiti korisnicima jednostavno i intuitivno korištenje sustava. Rješenje za ovaj problem jest izgradnja dodatnog aplikativnog sloja iznad našeg sustava koji će omogućiti korisnicima spajanje putem HTTP protokola. Aplikativni sloj djelovat će kao posrednik između korisnika i sustava, olakšavajući komunikaciju i korištenje usluga koje sustav nudi.



Sl. 2 Okvirna arhitektura programskog rješenja

U priloženom dijagramu (Sl. 2) prikazana je arhitektura programske potpore kakva bi ona u teoriji izgledala uz navedene specifikacije i zahtjeve sustava. Dijagram uključuje sve ključne komponente i procese koji su potrebni za postizanje optimalne funkcionalnosti sustava.

Važno je napomenuti kako je akcija prikupljanja podataka vremenski najzahtjevnija i najskuplja akcija unutar cijelog sustava. Zbog toga je optimizacija te akcije ključna za uspjeh i učinkovitost cijelog sustava.

U slučaju da proces prikupljanja podataka nije dovoljno optimiziran i odvija se sporije od vremenske tolerancije krajnjeg korisnika, cijeli sustav gubi svoju prednost i korisnost.

Cilj je stvoriti sustav koji ne samo da ispravlja pravopisne pogreške, već to čini brzo i precizno, pružajući korisnicima kvalitetnu uslugu u stvarnom vremenu.

3. Arhitektura programskog rješenja

Prije pisanja programskog kôda potrebno je detaljno proučiti tehnologije čije se vrline mogu iskoristiti u implementaciji našeg specifičnog rješenja.

Prethodno naveden oblik unutarnje arhitekture ScyllaDB pogodan je za programsko rješenje koje se razvija u sklopu ovog rada zbog toga što će se pretraživanje u većini slučajeva odvijati u konstantnoj vremenskoj složenosti, $O(1)$, a u najgorem slučaju u logaritamskoj vremenskoj složenosti, $O(\log N)$, gdje N označava broj zapisa unutar particije.

Potrebni upiti koje baza treba podržavati:

- Pretraga nad tablicom unigrama tako da se kao parameter preda riječ, a kao rezultat se dobije riječ te njena frekvencija.
- Pretraga nad tablicom bigrama tako da se kao parametri predaju prva i druga riječ, a kao rezultat se dobiju obje riječi te frekvencija bigrama.
- Pretraga nad tablicom bigrama tako da se preda samo jedna riječ (prva ili druga riječ bigrama). Kao rezultat se dobiju svi zapisi koji na zadanoj poziciji sadrže predanu riječ.
- Pretraga nad tablicom trigrama tako da se kao parametri predaju prva, druga i treća riječ, a kao rezultat se dobiju sve tri riječi te frekvencija trigrama.
- Pretraga nad tablicom trigrama tako da se u upitu predaju dvije riječi od mogućih tri (jedna od tri mogućih kombinacija, prva i druga, prva i treća ili druga i treća riječ). Kao rezultat se dobiju svi zapisi koji na zadanim pozicijama sadrže predane riječi.
- Upis u sve tablice.
- Promjena frekvencije nad n-gramima u svim tablicama.

Izgled tablice unigrama u ScyllaDB bazi prikazana je slijedećom naredbom (naredba jako sliči naredbi za kreiranje tablica u SQL standardu):

```
CREATE TABLE n_grams.one_grams (
    word text PRIMARY KEY,
    freq int
);
```

Kôd 1. Naredba za kreiranje tablice unigrama

Tablica unigrama sadrži riječ i njenu frekvenciju, a stupac „*word*“ je primarni ključ. U ovom specifičnom slučaju „*word*“ je *Partition Key*, a *Clustering Key* ne postoji.

Definiran je uvjet da svaki upit mora sadržavati sve stupce iz *Partition Key*. Navedeni upit, „*Pretraga nad tablicom bigrama tako da se preda samo jedna riječ…*“, zahtijeva više različitih mogućih kombinacija parametra nad istim skupom podataka. U slučaju da pokušamo definirati tablicu bigrama na sličan način kao tablicu unigrama, stvara se problem oko definicije primarnog ključa. U slučaju da su obje riječi dio *Partition Key*, a niti jedna riječ nije u *Clustering Key*, kada kao parametar predamo samo jednu riječ, upit se neće moći izvršiti. Isto tako, ako je samo prva riječ dio *Partition Key*, a druga riječ dio *Clustering Key*, prilikom slanja druge riječi kao parametar ScyllaDB će također očekivati i prvu riječ u listi parametra.

Radi toga postoje dvije tablica bigrama. U prvoj tablici je prva riječ *Partition Key*, a druga riječ *Clustering Key*, a u drugoj je tablici prva riječ *Clustering Key*, a druga riječ *Partition Key*.

```
CREATE TABLE n_grams.two_grams_1_pk (
    word_1 text,
    word_2 text,
    freq int,
    PRIMARY KEY (word_1, word_2)
) WITH CLUSTERING ORDER BY (word_2 ASC)
```

Kôd 2. Naredba za kreiranje tablice bigrama u kojoj je prva riječ *Partition Key*, a druga riječ *Clustering Key*

```
CREATE TABLE n_grams.two_grams_2_pk (
    word_2 text,
    word_1 text,
    freq int,
    PRIMARY KEY (word_2, word_1)
) WITH CLUSTERING ORDER BY (word_1 ASC)
```

Kôd 3. Naredba za kreiranje tablice bigrama u kojoj je druga riječ *Partition Key*, a prva riječ *Clustering Key*

Iz prethodna dva isječka kôda, „*word_1*“ i „*word_2*“ su unutar definicije primarnog ključa zadane različitim redoslijedom. Pretpostavljeno ponašanje u ovom slučaju postavit će prvi stupac koji je naveden unutar zagrada kao *Partition Key*, a sve ostale stupce će postaviti kao *Clustering Key*.

Tako su zadovoljeni zahtjevi upita nad bigramima, ali je udvostručena količina redaka u bazi. Taj pristup bi se mogao smatrati lošim sa stajališta klasičnih SQL baza podataka, ali

budući da to nije slučaj, takva arhitektura je prihvatljiva. Zauzeće memorijskog prostora od strane bigrama se udvostručilo, a prednost toga je ekstremno kratko vrijeme potrebno za izvršavanje upita na tako velikom skupu podataka. U službenoj dokumentaciji ispod naslova „*Things we should NOT focus on:*“, navedeno je „*Avoiding dana duplication: To get efficient reads, we sometimes have to duplicate data...*“.

Iz prethodnih zahtjeva, ako je potrebno predati obje riječi kao parametre, taj upit se može napraviti nad objema tablicama, ali ako se žele dobiti svi redovi takvi da je prva riječ „*zahtijeva*“, taj upit će se morati provesti nad prvom tablicom (Kôd 2). Analogno vrijedi za drugu riječ i drugu tablicu (Kôd 3).

Sličan pristup vrijedi kod tablica za trigrame jer postoje tri tablice sa istim podatcima. U prvoj tablici prva i druga riječ čini *Partition Key*, dok je treća riječ *Clustering Key*. Analogno u drugoj tablici, prva i treća riječ čine *Partition Key*, a druga riječ *Clustering Key* te u trećoj tablici, druga i treća riječ čine *Partition Key*, a prva riječ *Clustering Key*.

```
CREATE TABLE n_grams.three_grams_1_2_pk (
    word_1 text,
    word_2 text,
    word_3 text,
    freq int,
    PRIMARY KEY ((word_1, word_2), word_3)
) WITH CLUSTERING ORDER BY (word_3 ASC)
```

Kôd 4. Naredba za kreiranje tablice trigrama u kojoj prva i druga riječ čine *Partition Key,* a treća je riječ *Clustering Key*

```
CREATE TABLE n_grams.three_grams_1_3_pk (
    word_1 text,
    word_3 text,
    word_2 text,
    freq int,
    PRIMARY KEY ((word_1, word_3), word_2)
) WITH CLUSTERING ORDER BY (word_2 ASC)
```

Kôd 5. Naredba za kreiranje tablice trigrama u kojoj prva i treća riječ čine *Partition Key,* a druga je riječ *Clustering Key*

```
CREATE TABLE n_grams.three_grams_2_3_pk (
    word_2 text,
    word_3 text,
    word_1 text,
    freq int,
    PRIMARY KEY ((word_2, word_3), word_1)
) WITH CLUSTERING ORDER BY (word_1 ASC)
```

Kôd 6. Naredba za kreiranje tablice trigrama u kojoj druga i treća riječ čine *Partition Key,* a prva je riječ *Clustering Key*

Bitno je napomenuti razliku određivanja primarnog ključa od tablica bigrama. U ovom slučaju unutarnje zagrade određuju koji stupci će određivati *Partition Key*. Kada unutarnjih zagrada ne bi bilo, primijenilo bi se pretpostavljeno ponašanje. Prva riječ unutar vanjskih (u tom slučaju jedinih) zagrada postala bi *Partition Key*, a ostale bi sačinjavale *Clustering Key*.

Mogućnost slanja upita sa svim kombinacijama ulaznih parametra prikazana je na slici u nastavku (Sl. 3). Sintaksa upita je, kao što je moguće vidjeti, vrlo slična SQL standardu. U prvom upitu „*word_1*“ i „*word_2*“ se zadaju kao parametri. Dodan je i limit kako bi ispis bio kratak. Vraćena su dva retka iz tablice, a kako je baza riječi izgrađena od tekstova koje krajnji korisnici unose, mogu se očekivati i redci koji ne sadrže smislene vrijednosti poput redaka prikazanih na slici (Sl. 3). Drugi i treći upiti su slični prvom. Razlika je samo u kombinaciji parametra.

Zadnji upit prikazuje slučaj kada se kao parametri predaju sve tri riječi. U tom se slučaju, kao što je već navedeno, može koristiti bilo koja tablica.

Upiti nad tablicama koje sadrže bigrame se grade analogno, dok se upiti nad tablicom koja sadrži unigrame mogu napisati na samo jedan način.

```
cqlsh> SELECT * FROM n_grams.three_grams_1_2_pk
   ... WHERE word_1 = 'od' AND word_2 = 'mene' LIMIT 2;

 word_1 | word_2 | word_3 | freq
--------+--------+--------+------
     od |   mene |      0 |   26
     od |   mene |   0006 |    1

(2 rows)
cqlsh> SELECT * FROM n_grams.three_grams_1_3_pk
   ... WHERE word_1 = 'od' AND word_3 = 'zahtijeva' LIMIT 2;

 word_1 | word_3    | word_2 | freq
--------+-----------+--------+------
     od | zahtijeva |     10 |   10
     od | zahtijeva |     12 |    1

(2 rows)
cqlsh> SELECT * FROM n_grams.three_grams_2_3_pk
   ... WHERE word_2 = 'mene' AND word_3 = 'zahtijeva' LIMIT 2;

 word_2 | word_3    | word_1 | freq
--------+-----------+--------+------
   mene | zahtijeva |    Dio |    1
   mene | zahtijeva |   Naći |    2

(2 rows)
cqlsh> SELECT * FROM n_grams.three_grams_1_2_pk
   ... WHERE word_1 = 'od' AND word_2 = 'mene' AND word_3 = 'zahtijeva';

 word_1 | word_2 | word_3    | freq
--------+--------+-----------+------
     od |   mene | zahtijeva |  563

(1 rows)
```

Sl. 3 Upiti nad tablicama koje sadrže trigrame

Programski jezik u kojem će biti napisana programska potpora koja će voditi komunikaciju s bazom podataka i manipulirati tim podatcima, mora zadovoljavati nekoliko ključnih stavki:

- Jezik mora biti brz zbog prirode programske potpore koja se razvija. Brzinu analiziramo s više gledišta. Primjerice brzina alociranja memorije na stogu i gomili ili brzina izvođenja petlji.
  - U ovom aspektu prednost će imati programski jezici na sistemskoj razini budući da su takvi jezici u pravilu mnogo brži od nekih drugih jezika koji sadrže dodatne apstrakcije.
- Memorijska sigurnost je ekstremno važna kako se ne bi dogodilo da u razvijenom programskom rješenju imamo curenje memorije.
  - U ovom slučaju, prednost će imati jezici koji ili imaju nativno riješen problem automatskog upravljanja memorijom uz pomoć, primjerice skupljanja smeća, ili sadrže funkcije za jednostavnim upravljanjem memorijom unutar standardne biblioteke.

  - Samo statički tipizirani jezici dolaze u obzir.
- Jednostavno izvršavanje.
  - U obzir dolaze samo kompajlirani jezici. Prednosti takvih su između ostalog brže izvršavanje, efikasnije korištenje resursa, ali u sklopu ovog rada najbitnija je provjera tipova i pronalazak greški za vrijeme kompajliranja, a ne za vrijeme izvršavanja programa.
- Jednostavna integracija sa ScyllaDB.
  - Postoje već napravljeni upravljački programi koji omogućuju integraciju nekih programskih jezika sa ScyllaDB. Jezici za koje ne postoje razvijeni upravljački programi ne dolaze u obzir pri odabiru.
- Moćni alati unutar standardne biblioteke jezika koji omogućuju paralelno izvođenje programa
  - ScyllaDB je napravljena tako da može prihvatiti mnogo konkurentnih zahtjeva u isto vrijeme bez gubitka performansi.

Savršen jezik uz sve prethodne uvjete nije moguće pronaći. Osim svih tih uvjeta, jezici oko kojih ne postoji velika zajednica ili su jako zahtjevni za koristiti, nisu zanimljivi. Programski jezici između kojih je bio izbor uz prethodne uvjete:

- C++
- Golang
- Rust

Programska potpora razvijana u sklopu ovog rada koristit će programski jezik rust. Rust je moderan programski jezik dizajniran za sigurnost, paralelnost i učinkovitost. Njegov jedinstveni sustav vlasništva osigurava sigurnost memorije bez potrebe za skupljanjem smeća, sprječavajući greške poput dereferenciranja pokazivača na nevaljanu adresu i curenja memorije. Sadrži puno stroža pravila koja se tiču količine pokazivača s pravom da pokazuju na neku memorijsku lokaciju i vrstu tih pokazivača (smiju li mijenjati podatak na memorijskoj lokaciji ili smiju samo čitati). Stroža pravila omogućuju pisanje sigurnijeg kôda budući da se većina grešaka hvata tijekom kompajliranja.

Paralelizam u rustu je lagan za korištenje i u sklopu standardne biblioteke nudi razne korisne alate za komunikaciju paralelnih dretvi i slanje podataka između njih. Jedan takav alat su kanali. Osim mogućnosti koje nudi standardna biblioteka, najpoznatija asinkrona izvršna knjižica tokio olakšava korištenje asinkronih procesa unutar samog jezika i unapređuje korisnikovo iskustvo pri pisanju asinkronog rusta.

Performanse rusta usporedive su s C i C++ zahvaljujući direktnom kompajliranju u strojni kôd. Alat Cargo pojednostavljuje upravljanje paketima i gradnju projekata, dok bogata

standardna biblioteka i sve veći ekosustav podržavaju razne operacije i integraciju s postojećim C/C++ kôdom.

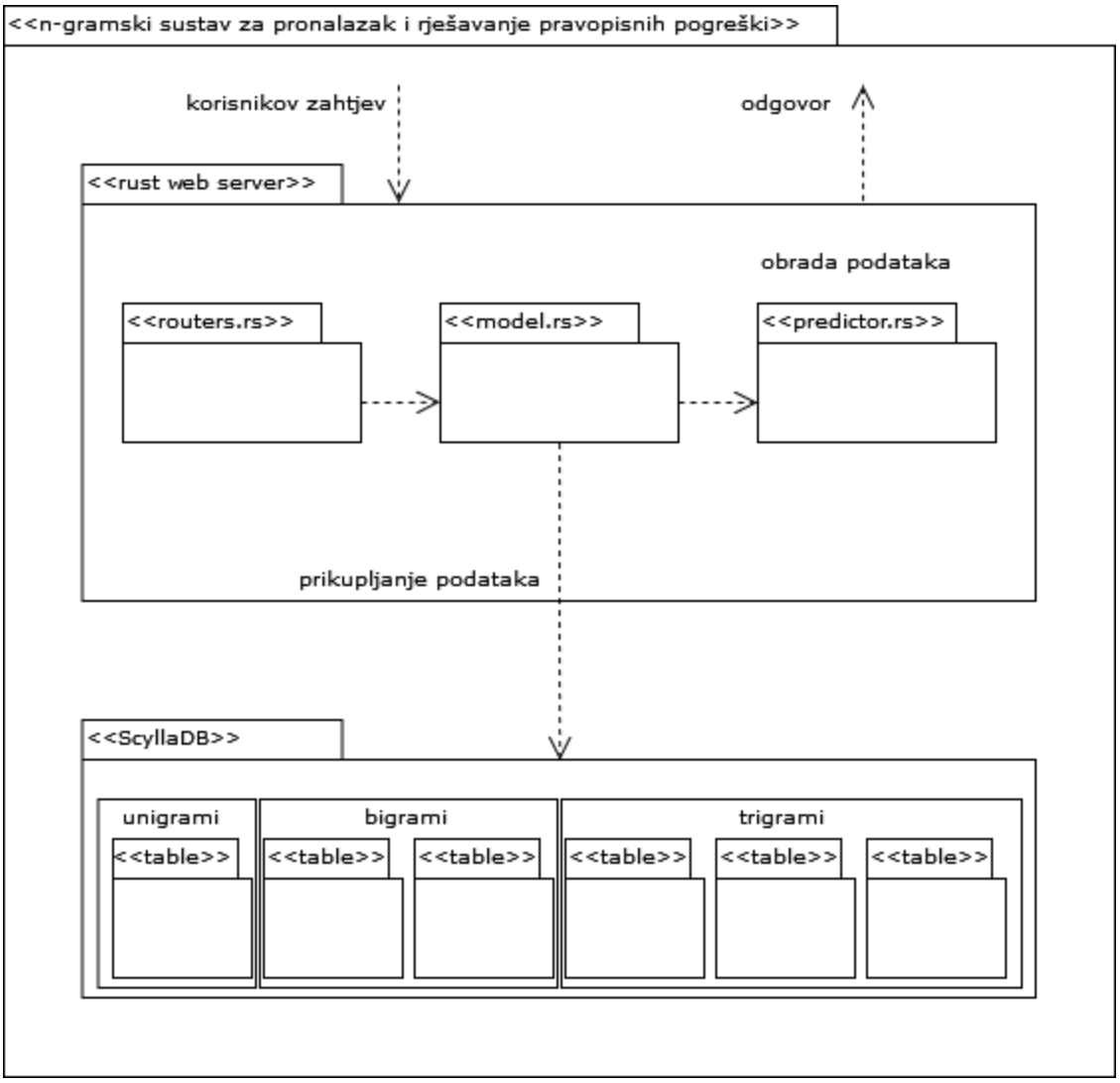
Rustov sustav za upravljanje memorijom omogućuje efikasno i predvidljivo korištenje resursa, a njegova statička analiza smanjuju greške tijekom izvođenja.

O tome kako izgleda programska podrška koja u sklopu ovog rada obavlja svoj posao, bit će riječi u nastavku.

Pristup podatcima iz baze podataka potrebno je omogućiti krajnjim korisnicima putem web poslužitelja. Taj će servis nakon pristupa podatcima po potrebi promijeniti njihovu strukturu i vratiti rezultate krajnjem korisniku.

Uz rust i tokio, programska potpora će također koristit actix (radni okvir koji olakšava kreiranje i upravljanje web poslužiteljem).

Na slici u nastavku (Sl. 4) prikazana je arhitektura cijelog sustava.



Sl. 4 Stvarna arhitektura programskog rješenja

4. Implementacija programskog rješenja

U sklopu ovog poglavlja, pri analizi programskog kôda fokus neće biti na sintaksi programskog jezika rust, pa će u primjerima gdje je moguće biti prikazan pseudokôd. U sklopu ovog poglavlja prvo će se obrađivati komunikacija web poslužitelja s bazom podataka, optimizacija istih upita te onda protok podataka od dohvate do korištenja i tumačenja dohvaćenih podataka.

Više informacija o jeziku može se pronaći u službenoj dokumentaciji [7].

4.1. Komunikacija poslužitelja i podatkovnog sloja

```
Funkcija init() vraca Rezultat(Arc<Session>, String):

    uzmi uri iz okruženja sa ključem "SCYLLA_URI"
    ako ne postoji, koristi zadani uri "127.0.0.1:9042"

    pokušaj napraviti sesiju sa SessionBuilder koristeći
        poznati uri
    čekaj rezultat pokušaja izgradnje sesije

    ako je izgradnja sesije uspješna:
        omotaj sesiju u Arc
        vrati sesiju

    ako izgradnja sesije nije uspješna:
        vrati grešku "Failed to connect to ScyllaDB"
```

Pseudokôd 1. Spajanje na ScyllaDB

Prethodni isječak pseudokôda je odgovoran za spajanje na ScyllaDB bazu podataka. Važno je istaknuti moćnu podršku paralelizma u rustu uz pomoć Arca. Arc je pametni pokazivač koji omogućuje sigurno korištenje ScyllaDB sesije kroz više dretvi. Kako bi to postigao koristi atomsku operaciju za upravljanje referentnim brojanjem, čime omogućava sigurno dijeljenje podataka bez potrebe za ručnim upravljanjem referencama. Takav pristup kreacije ScyllaDB sesije je nužan zbog toga što je ta operacija vremenski i memorijski skupa. Tako se samo pri inicijalizaciji programa kreira sesija omotana pametnim pokazivačem koji vodi računa o sigurnom korištenju njene memorijske lokacije.

Već spomenut rustov sustav vlasništva zahtijeva da svaki resurs ima samo jednog vlasnika (varijabla odgovorna za taj resurs). Kada vlasnička varijabla izađe iz dosega programa, resurs se oslobađa. Zbog tog uvjeta potrebno je prilikom svakog korištenja sesije klonirati istu. Time je omogućeno stvaranje nove reference na isti podatak, bez mijenjanja vlasništva inicijalnog resursa. Ovakav pristup je siguran za korištenje unutar višedretvene programske potpore.

```
Funkcija execute_one(self, session, params) vraca
    Rezultat(RowIterator, QueryError):

    napravi kopiju pripremljene upita iz
        self.prepared_query

    pokušaj izvršiti upit sa session koristeći prethodno
        kreiranu kopiju upita i parametre
    čekaj na rezultat pokušaja izvršavanja upita

    ako je izvršavanje upita uspješno:
        vrati pokazivač na početak iteratora(rows_stream)

    ako izvršavanje upita nije uspješno:
        vrati grešku QueryError::ScyllaError
```

Pseudokôd 2. Upit prema bazi podataka

Prikazani isječak pseudokôda služi za izvršavanje upita prema ScyllaDB bazi podataka. Nakon kreiranja sesije (Pseudokôd 1), referenca sesije se klonira bez mijenjanja vlasništva (Arc). Klonirana sesija se šalje prikazanom potprogramu (Pseudokôd 2).

Taj potprogram je metoda nad strukturom „*QueryFactory*“ koja sadrži prethodno pripremljeni upit u radnoj memoriji. Više o prethodno pripremljenom upitu i njegovim prednostima bit će rečeno u nastavku.

```
struktura QueryFactory {
    prepared_query: instanca strukture PreparedStatement,
}
```

Pseudokôd 3. Struktura „*QueryFactory*“

Metode nad strukturama u rustu, između ostalih parametara, sadrže referencu na instancu te strukture (*self*). Prethodno pripremljeni upit dohvaćamo uz pomoć te reference.

Potprogram (Pseudokôd 2) također prima parametre koje će predati pripremljenom upitu. Nakon izvršavanja upita funkcija će vratiti pokazivač koji omogućuje iterativan prolazak po dohvaćenim redcima tablice.

4.2. Optimizacija upita

ScyllaDB omogućuje optimiziranje upita na nekoliko načina. U sklopu ovog rada, implementirana je većina postupka optimiranja upita.

4.2.1. Prethodno pripremljeni upiti

Upravljački program za integraciju rusta i ScyllaDB nudi mogućnost pripremanja upita prije samog izvođenja. Osim tog načina izvršavanja upita moguće je upit izvršiti bez prethodnog pripremanja. Korištenje prethodno pripremljenog upita u razvijanom programskom rješenju znatno poboljšava performanse sustava.

U slučaju da je potrebno dohvatiti unigram iz baze, upit bi izgledao ovako:

```
SELECT word, freq FROM n_grams.one_grams WHERE word = ?
```

Ovaj upit pretražuje tablicu unigrama i prima riječ kao parametar te vraća tu riječ i njenu frekvenciju.

Struktura „*QueryFactory*“ (Pseudokôd 3) sadrži metodu koja vraća instancu te iste strukture.

```
Funkcija build(session, query, consistency) vraca
    Rezultat(Self, String):

    pokušaj pripremiti upit koristeći session i query
    čekaj na rezultat pokušaja pripreme upita

    ako je priprema upita uspješna:
        postavi konzistentnost (consistency) na pripremljeni
            upit (prepared_query)
        vrati novi QueryFactory objekt sa pripremljenim
            upitom (prepared_query)

    ako priprema upita nije uspješna:
        vrati grešku "Failed to prepare query"
```

Pseudokôd 4. Kreacija „*QueryFactory*“ instance

Navedeni potprogram (Pseudokôd 4) kreira instancu *„QueryFactory*“ strukture. Potprogram prima sesiju, upit i tip konzistencije upita. Nad primljenom sesijom pokušava se pripremiti upit kojeg će se kasnije moći izvršavati.

Sve akcije nad sesijom su asinkrone, pa tako i kreiranje pripremljenog upita. U sklopu ovog rada uvijek se koriste upiti kreirani na prikazan način.

Pri izvršavanju prethodno pripremljenog upita, parametri za upit se naknadno predaju. ScyllaDB te parametre postavi svugdje gdje se nalazi znak upitnik. Količina parametara treba biti jednaka količini upitnika u upitu.

4.2.2. Paginiran upit

Prilikom izvršavanja upita, upravljački program za integraciju rusta i ScyllaDB nudi dva načina dohvate podataka. Jednostavan način jest dohvata svih podataka odjednom. Taj način je jednostavniji za implementirat, ali je sporiji u slučaju dohvate velike količine podataka. Takav upit traje duže zbog toga što je teret na radnu memoriju ScyllaDB i rust programa mnogo veći.

Paginirani upiti omogućuju dohvat podataka tako da se, umjesto dohvaćanja svih podataka odjednom, vrati pokazivač koji omogućuje iterativan prolazak po dohvaćenim redcima tablice. Navedeni potprogram (Pseudokôd 2) vraća takav pokazivač. ScyllaDB u tom slučaju iterativno dodaje podatke za iterativni prolaz. Prilikom primanja tog pokazivača, umjesto da

se spreme svi podaci u neku varijablu, iterativnim prolazom punit će se neka struktura podataka (lista, vektor ili nešto drugo).

```
pokušaj izvršiti upit koristeći factory.execute_one
    (s, values) i čekaj na rezultat

    ako je izvršavanje upita uspješno:
        pretvori rezultat (rows) u tipizirani tok redova
            (row_stream) sa tipom (String, i32)

    ako izvršavanje upita nije uspješno:
        vrati grešku
```

Pseudokôd 5. Dohvat pokazivača na objekt koji omogućuje iterativni prolaz

Izvršavanje potprograma (Pseudokôd 2) koji vraća pokazivač na objekt s kojim je moguće iterativno prolaziti po dohvaćenim redcima prikazano je u prethodnom isječku pseudokôda (Pseudokôd 5). Varijabla „*row_stream*“ je dohvaćeni pokazivač.

Trajanje izvođenja tog isječka pseudokôda neće ovisiti o količini podataka koja se dobiva budući da će ScyllaDB vratiti pokazivač te naknadno dodavati podatke za iterativni prolaz. U ovom slučaju redci koji budu dobiveni spremat će se strukturirano u obliku:

```
struktura QueryResult {
    input: String,
    frequency: int32,
    length: int32,
}
```

„*QueryResult*“ struktura sadrži n-gram u tekstualnom obliku („*input*“), frekvenciju tog n-grama i duljinu n-grama.

```
inicijaliziraj prazni vektor words_received

dok ima dostupnih redova u row_stream:
    čekaj da dobiješ sljedeći red (row)

    raspakiraj red u varijable word i freq
    ako je raspakiranje uspješno:
        dodaj word u words_received

        pošalji tuple (key, QueryResult)
            preko kanala tx, gdje se QueryResult sastoji od:
                n-gram
                freq
                length
    ako raspakiranje nije uspješno:
        vrati grešku
```

Pseudokôd 6. Iterativni prolaz po dobivenim redcima

Priloženi isječak pseudokôda prikazuje prolaz po svim dohvaćenim redcima. U svakoj iteraciji, podatci se obrađuju na način na koji će biti objašnjen u nastavku rada.

Pokazalo se kada su dohvaćeni podatci veliki (stotine tisuća redaka), brzina prikupljanja podataka se smanji za više od pet puta sa korištenjem paginiranih upita.

4.2.3. Paralelno izvršavanje upita

Programska potpora razvijana u sklopu ovog rada će u većini slučajeva morati izvršiti više od jednog upita. Navedeno je da je ScyllaDB dobro optimirana za paralelne upite. Razvijani sustav će u trenutcima kada je potrebno izvršiti više upita svaki upit odraditi paralelno. U takvim situacijama koristit će se kanali. Kanali su komunikacijski mehanizam prijenosa podataka među dretvama. Dio su standardne biblioteke, a ime modula putem kojeg se koriste unutar programa je „*mpsc*“. Ime modula „*mpsc*“ je akronim za „*multiple producer, single consumer*“ (više proizvođača, jedan potrošač). To znači da više dretvi može slati podatke (proizvođači) u kanal, dok jedna dretva prima podatke (potrošač) iz kanala.

Prije kreiranja dretvi stvara se kanal te se inicijalizira prazni vektor rukovatelja dretvi:

```
stvori tx i rx iz mpsc::channel()
stvor prazni vektor handlers
```

Povratna vrijednost inicijalizacije kanala su dvije varijable, „*tx*“ („*transmitter*“) i „*rx*“ („*receiver*“).

```
za svaki element v u value.queries:
    kloniraj referencu na sesiju (s)

    kloniraj kanal za slanje (tx_clone)

    kreiraj asinkroni zadatak koristeći tokio::spawn:
        u asinkronom zadatku:
            pozovi funkciju process sa argumentima:
                ključ (key)
                sesija (s)
                upit (query)
                parametri
                kanal za slanje (tx_clone)
            čekaj na izvršenje funkcije process

    dodaj handle kreiranog zadatka u listu handlers
```

Pseudokôd 7. Pokretanje paralelnog izvršavanja

Radi rustovog sustava vlasništva potrebno je klonirati varijablu „*tx*“ budući da se ona šalje drugom potprogramu. Izvršna knjižnica tokio služi za kreaciju novog asinkronog procesa unutar kojeg će se odvijati posao svake dretve. Na kraju se kreirani rukovoditelj dretve dodaje u vektor rukovoditelja.

Nakon kreacije i pokretanja procesa potrebno je pričekati da svi procesi završe prije nego se kôd može nastaviti izvršavati. Programski kôd će zaglaviti u izvršavanju ovog bloka programskog kôda prikazanog pseudokôdom:

```
za svaki handle iz handlers:
    čekaj handle
```

sve dok svaki proces ne završi. To je nužno kako bi se naknadno podatci obradili bez grešaka.

```
oslobodi kanal za slanje (tx)

za svaki rezultat u kanalu za primanje (rx):
    za svaki sentence_result u listi sentence_results:
        ako sentence_result.sentence odgovara result.0:
            dodaj result.1 u listu rezultata
            prekini unutarnju petlju
```

Pseudokôd 8. Prikupljanje podataka iz kanala

Kada svi procesi završe, iz kanala putem varijable „*rx*“ izvlače se dobiveni podatci. Petlja u prikazanom pseudokôdu će se izvršavati sve dok ne završe svi proizvođači u kanalu („*transmitter*“). Pri kreaciji kanala povratna vrijednost sadrži jednog potrošača. Za svaki paralelni proces kloniran je inicijalni proizvođač. Zbog toga u opsegu u kojem je kreiran kanal postoji jedan više proizvođač nego što se procesa pozvalo. U slučaju da se na početku ne oslobodi inicijalni „tx“, program bi zaglavio u petlji. Slanje podataka u kanal prikazano je isječkom pseudokôda unutar potprograma „*process*“ (Pseudokôd 6).

Budući da je razvijana programska potpora vrlo resursno zahtjevna za računalo, bez ovih optimizacija sustav ne bi mogao odrađivati posao u skladu sa zahtjevima krajnjeg korisnika.

4.3. Protok podataka

U sklopu ovog rada potrebno je bilo napraviti dvije HTTP krajnje točke:

- Krajnja točka za dohvat podataka o određenom n-gramu
- Krajnja točka za pravopisnu provjeru nekog tekstualnog zapisa uz pomoć konteksta

Krajnja točka za dohvat podataka o određenom n-gramu putem URL-a prima nekoliko obaveznih i opcionalnih parametara:

- word1
  - Prva riječ po redu u sklopu nekog n-grama.
- word2
  - Druga riječ po redu u sklopu nekog n-grama.
- word3
  - Treća riječ po redu u sklopu nekog n-grama.
  - Opcionalni parametar budući da je krajnja točka implementirana za bigrame i trigrame.
  - U slučaju da je parametar unesen, a analizira se bigram, parametar će se zanemariti.
  - U slučaju da parametar nije unesen, a analizira se trigram, programska potpora će javiti pogrešku.
- amount
  - Opcionalni parametar koji određuje koliko će se parova riječi te njene frekvencije vratiti krajnjem korisniku. U nastavku će biti pojašnjeni parovi riječi i njihovih frekvencija.
  - U slučaju da parametar nije zadan, 50 je pretpostavljena vrijednost.
- n
  - Specifični n-gram.
  - Podržani n-grami su bigrami i trigrami, pa taj parametar može poprimiti samo vrijednosti 2 ili 3.
- vary
  - Indeksi koji će varirati.
  - Parametar oblikovan kao lista (brojevi odvojeni zarezom)
  - U slučaju da se radi o bigramu, podržane vrijednosti su 1 i 2. Radi li se o trigramu, 3 je također dio podržanih vrijednosti.

Zamisao krajnje točke je da primi konkretni n-gram („od mene zahtijeva“) i indekse koje je potrebno varirati. U slučaju da je primljen samo indeks 1, razvijana programska potpora treba potražiti unutar baze podataka sve trigrame gdje je druga riječ „*mene*“, a treća riječ „*zahtijeva*“. Ako količina („*amount*“) nije zadana, programska potpora treba vratiti 50

(pretpostavljena vrijednost) riječi koje se u trigramu nalaze na prvom mjestu, tako da ostatak trigrama izgleda „*mene zahtijeva*“. Treba biti dohvaćena i frekvencija tih trigrama, a ispis će biti poredan po frekvenciji silazno. Ako je razvijanoj programskoj potpori poslan zahtjev čiji URL parametri izgledaju ovako:

```
?n=3&word1=od&word2=mene&word3=zahtijeva&vary=1,3&amount=2
```

web poslužitelj će vratiti podatke u obliku prikazanom na slici (Sl. 5).


```
{
    "time_taken": "42 ms",
    "n_gram_length": 3,
    "provided_n_gram": "od mene zahtijeva",
    "provided_n_gram_frequency": 563,
    "varying_indexes": [
        1,
        3
    ],
    "vary": [
        {
            "index": 1,
            "word": "od",
            "solutions": [
                {
                    "word": "od",
                    "frequency": 563
                },
                {
                    "word": "Od",
                    "frequency": 6
                }
            ]
        },
        {
            "index": 3,
            "word": "zahtijeva",
            "solutions": [
                {
                    "word": "i",
                    "frequency": 21606
                },
                {
                    "word": "da",
                    "frequency": 15837
                }
            ]
        }
    ]
}
```


Sl. 5 Prikaz rezultata krajnje točke za dohvat podataka o određenom n-gramu u .JSON formatu

Iz dohvaćenih podataka važno je napomenuti brzinu izvođenja. Tako kratko vrijeme izvođenja omogućile su optimizacijske metode navedene u prethodnim poglavljima.

Krajnja točka za pravopisnu provjeru nekog tekstualnog zapisa uz pomoć konteksta prima samo jedan parametar kroz formu:

- text
  - Tekstualni zapis kojeg se želi pravopisno provjeriti.

Svi *confusion setovi* koji će se uzimati u obzir pri izvršavanju programa definirani su u radnoj memoriji programske potpore.

Predani tekstualni zapis se prvo dijeli na rečenice. U slučaju pojave zareza unutar neke rečenice, ona će se razlomiti oko zareza. U takvim zapisima se poslije traže riječi iz *confusion setova*. Prilikom pronalaska takve riječi, kreće proces izgradnje specifičnih upita za pronađenu riječ.

```
funkcija extract_context(index, words) vraca String:

    inicijaliziraj prazan string context

    za i u rasponu od (index - 2) do (index + 2), uzimajući u
        obzir granice niza words:
            dodaj riječ words[i] u context
            dodaj razmak u context

    ukloni suvišne razmake sa krajeva stringa context

    vrati context
```

Pseudokôd 9. Izdvajanje konteksta iz rečenice

Prikazani pseudokôd prikazuje način izdvajanja konteksta koji će se analizirati pri pravopisnoj provjeri. Potprogram prima indeks riječi koja je pronađena unutar nekog *confusion seta* i riječi iz kojih će se vaditi kontekst. Kontekst će sadržavati riječ iz *confusion seta*, dvije prethodne riječi i dvije sljedeće riječi. U slučaju da ne postoje takve riječi, primjerice da se analizirana riječ nalazi na početku rečenice, kontekst će sadržavati manje riječi. Primjera radi, u rečenici „*On od mene zahtijeva da dolazim na nastavu.*“, izvađeni kontekst je „*od mene zahtijeva da dolazim*“ budući da je riječ „*zahtijeva*“ u nekom od *confusion setova*. U slučaju da se riječ iz nekog *confusion seta* nalazi na drugoj poziciji u rečenici, kao na primjer u rečenici, „*Marko zahtijeva da mu se plaća poveća.*“, izvađeni kontekst će biti „*Marko zahtijeva da mu*“. Razlog takvom izdvajanju konteksta su podržani n-grami u programskoj potpori (bigrami i trigrami).

Radi lakšeg razumijevanja postupka koji slijedi, riječ koja je inicijalno bila pronađena unutar nekog *confusion seta*, bit će referencirana kao aktualna riječ.

Nakon kreiranja pojedinog konteksta, za svaki kontekst kreiraju se sljedeći upiti:

- Dohvaćanje unigrama aktualne riječi i svih ostalih riječi unutar *confusion seta* u kojem se nalazi aktualna riječ.
- Dohvaćanje bigrama koji se sastoji od riječi koja prethodi aktualnoj riječi te same aktualne riječi.
  - U slučaju da se aktualna riječ ne nalazi na prvom mjestu u izvađenom kontekstu.
- Dohvaćanje bigrama koji se sastoji od aktualne riječi te riječi koja slijedi aktualnu riječ.
  - U slučaju da aktualna riječ nije na kraju konteksta.
- Dohvaćanje trigrama koji se sastoji od dvije riječi koje prethode aktualnoj riječi te aktualne riječi.
  - U slučaju da se aktualna riječ ne nalazi na prvom ili drugom mjestu u izvađenom kontekstu.
- Dohvaćanje trigrama koji se sastoji od aktualne riječi te dvije sljedbenice aktualne riječi.
  - U slučaju da se aktualna riječ ne nalazi na predzadnjem mjestu u izvađenom kontekstu.

Za upite koji se odnose na bigrame i trigrame kreiraju se isti upiti, ali umjesto aktualne riječi koriste se sve riječi iz *confusion seta* gdje je pronađena aktualna riječ. U slučaju da je izvađeni kontekst „*od mene zahtijeva da dolazim*“, kreirani upiti za točku pod rednim brojem dva, bili bi:

```
SELECT * FROM n_grams.two_grams_1_pk WHERE
word_1 = 'mene' AND word_2 = 'zahtijeva';

SELECT * FROM n_grams.two_grams_1_pk WHERE
word_1 = 'mene' AND word_2 = 'zahtjeva';
```

zbog toga što se unutar istog *confusion seta* nalaze riječi „*zahtijeva*“ i „*zahtjeva*“.

Uz upite se također spremaju potrebni parametri za pojedini upit. Tako strukturirani podatci se prosljeđuju potprogramu (Pseudokôd 7 i Pseudokôd 8) koji će paralelno izvršiti upite.

Rezultati upita su strukturirani na sljedeći način:

```
struktura SentenceResult {
    sentence: String,
```

```
    word: String,
    results: vektor instanci strukture QueryResult,
}
```

Svaki rezultat sadrži rečenicu u tekstualnom obliku, aktualnu riječ i rezultate. Rezultati su zapisani u obliku vektora, a svaki član vektora je prethodno navedena struktura *QueryResult*.

Rezultati prethodno izvađenog konteksta prikazani su u sljedećem isječku programskog kôda u .JSON formatu. Samo za prethodno definirane upite će se prikazati ispis zbog toga što je cijela povratna vrijednost vrlo velika.

```
{
    "sentence": "od mene zahtijeva da dolazim",
    "word": "zahtijeva",
    "results": [
        // upiti
        {
            "input": "mene zahtijeva",
            "frequency": 588,
            "length": 2
        },
        {
            "input": "mene zahtjeva",
            "frequency": 647,
            "length": 2
        }
        // upiti
    ]
}
```

Kôd 7. Isječak povratne vrijednosti izvršenih upita

4.4. Analiza dohvaćenih podataka

Programska potpora razvijana u sklopu ovog rada treba omogućiti analizu i tumačenje dohvaćenih podataka kako bi krajnji korisnik mogao razumjeti.

Definiran je „*trait*“ (slična funkcionalnost kao sučelje u objektno orijentiranim programskim jezicima) koji će definirati jednu funkciju.

```
definiraj trait Predict:

    definiraj funkciju predict:
        ulazni parametri:
            self - referenca na instancu objekta
            data - objekat tipa TimedSentenceResults
            confusion_set - vektor vektora stringova
            number_of_ngrams – mapa sa sumama frekvencija
                pojedinih n-grama

        izlazni rezultat:
            objekat tipa PredictionResults
```

Pseudokôd 10. Apstraktna definicija funkcije „*predict*“ koju će ostale strukture implementirati

Navedeni potprogram u listi parametra prima rezultate dobivene postupkom navedenom u prethodnom odlomku s vremenom izvođenja tog procesa („*TimedSentenceResults*“). Također prima sve *confusion setove* i sumu svih frekvencija pojedinog n-grama kako bi se mogle odrediti relativne vjerojatnosti.

Razlog ovog pristupa je mogućnost programske potpore da u jednom trenutku ima definirano više različitih algoritama koji će dobivene podatke protumačiti na zaseban način.

```
definiraj funkciju predict sa generičkim tipom T:

    ulazni parametri:
        predictor - objekat tipa T koji implementira
            trait Predict
        data - objekat tipa TimedSentenceResults
        confusion_set - vektor vektora stringova
        number_of_ngrams – mapa sa sumama frekvencija
            pojedinih n-grama

    izlazni rezultat:
        objekat tipa PredictionResults

    pozovi metodu predict na predictor objektu
        sa argumentima:
            data, confusion_set, number_of_ngrams

    vrati rezultat metode predict
```

Pseudokôd 11. Potprogram koji će pozivati drugi potprogram za računanje predikcija

Potrebno je napomenuti kako ove dvije definicije potprograma pod istim nazivom „*predict*“ nisu isti potprogrami. Potprogram na početku poglavlja (Pseudokôd 10) je metoda nad strukturom dok je prethodni (Pseudokôd 11) potprogram funkcija definirana kao javna funkcija dostupna cijelom projektu.

Prethodni (Pseudokôd 11) „*predict*“ potprogram poziva konkretnu implementaciju „*predict*“ metode nad nekom strukturom.

Konkretna implementacija „*predict*“ potprograma mora vratiti povratnu vrijednost u obliku:

```
struktura PredictionResults {
    time_elapsed: String,
    results: vektor instanci strukture PredictionResult,
}
```

a rezultati su strukturirani na ovaj način:

```
struktura PredictionResult {
    context: String,
    word_examined: String,
    results: mapa gdje su ključevi String, a vrijednosti f64,
}
```

Svaki rezultat u vektoru rezultata sadrži izvučeni kontekst koji se analizirao, riječ na kojoj je bio fokus u analizi (aktualna riječ) te rezultate upita gdje je ključ mape pojedina riječ iz *confusion seta*, a vrijednost frekvencija n-grama koji se sastoji od statičkih vrijednosti i ključa kao varijabilne vrijednosti.

5. Rezultati

Programska potpora razvijana u sklopu ovog rada, namijenjena je za korištenje uz veliku količinu *confusion setova*, ali za predstavljanje rezultata sustava korišten je mali podskup *confusion setova*, koji je prikazan u nastavku (Kôd 8). Bitno je napomenuti da je pogrešan pristup korištenja sustava generiranje *Confusion setova* za svaku riječ. Uz to, također je često nepotrebno kreirati *Confusion setove* za veznike i ostale kratke riječi. *Confusion setovi* bi u pravilu trebali biti definirani za pogreške koje korisnici često rade.

```
zahtijeva zahtjeva
djelom dijelom
djelu dijelu
plaća plača placa
sto što
car čar
pisači pisaći
```

Kôd 8. *Confusion setovi* korišten u sklopu ovog testiranja

Navedena je mogućnost programske potpore da implementira više različitih algoritma pri izračunu. U drugom poglavlju spomenuta su tri algoritma uz pomoć kojih je bio testiran sustav. U nastavku će ti algoritmi biti referencirani oznakama SM, za $SM(W_{\dot{u}})$, SS za $SS(W_{\dot{u}})$ i SP za $SP(W_{\dot{u}})$.

Svakom algoritmu dan je sljedeći tekstualni zapis na ulaz:

„*Od njega zahtjeva dodatno vrijeme. Jako mu sliči, a jednim djelom i na njegovog brata. O njegovom dijelu je napisao članak. Svidjela mu se plaća koju je dobio. Dogodilo se baš kao što je rekao. Nogomet je izgubio svoju čar. Prije deset godina pisači su bili marljiviji.*“

Većina postupka je identična za sve algoritme. Sustav će prvo razdvojiti tekst po rečenicama te također zarezima unutar rečenica. Nakon toga se prolazi po svih takvim skupovima riječi. U slučaju da taj skup riječ sadrži riječ koja se nalazi u nekom od *confusion setova*, oko te riječi kreira se kontekst. Kao što je već navedeno ranije u dokumentu, kontekst se sastoji od dvije riječi koje prethode riječ iz *confusion setova*, tu riječ i dvije riječi koje slijede. U slučaju da se riječ iz *confusion seta* nalazi na prva ili zadnja dva mjesta u skupu riječi, kontekst će biti kraći. Taj postupak vađenja konteksta se obavlja za sve riječi koje se pronađu. Nakon toga se za svaki kontekst definiraju n-grami. Definira se unigram riječ iz *confusion seta*, bigrami tako da je ta riječ na početku i na kraju, te analogno trigrami. Za kontekst „*a jednim*

djelom i na“, definirati će se unigram „*djelom*“, bigrami „*jednim djelom*“ i „*djelom i*“, te trigrami „*a jednim djelom*“ i „*djelom i na*“. Također će se na taj način definirati i n-grami za svaku riječ iz *confusion seta* unutar kojega je bila prvotno pronađena riječ. Nakon tog postupka definirati će se upiti za svaki n-gram, te će se ti upiti izvršiti. Nakon izvršavanja upita dobivene su sve frekvencija definiranih n-grama. Nakon toga se nekom od algoritmu provjeru predaju ti podatci. Predani podatci su formatirani na način da se za svaki kontekst šalje riječ pronađena u nekom od *confusion setova*. Također se za svaku riječ iz aktualnog *confusion seta* šalju svi n-grami koji na mjestu pronađene riječi sadrže tu riječ i frekvenciju tako definiranog n-grama. Svaki algoritam će takve podatke drugačije protumačiti, a matematički postupak donošenja odluka prikazan je na početku dokumenta.

Za prethodno navedeni kontekst, podatke koje će pojedini algoritam dobiti na ulaz izgledat će ovako:

- djelom - 165047
- dijelom - 906214
- jednim djelom - 6220
- jednim dijelom - 44099
- djelom i - 9775
- dijelom i - 21038
- a jednim djelom - 187
- a jednim dijelom - 1326
- djelom i na - 113
- dijelom i na – 2206

Iz prethodnih vrijednosti i formula navedenih na početku dokumenta, koristeći za primjer prvi algoritam, indeks sigurnosti za riječ *djelom*, iznosi 11,0297, a za riječ *dijelom* 9,6358. Budući da je na početku dokumenta definirana činjenica da je za manje indekse sigurnosti sustav sigurniji u odluku, sustav će u ovom slučaju zaključiti da je riječ *dijelom* pravopisno ispravna u danom kontekstu.

Postupak je analogan za sve ostale kontekste, a rezultati pojedinih algoritama su u nastavku.

Tablica 1. Rezultati za rečenicu „*Od njega zahtjeva dodatno vrijeme.*“, gdje je izvučeni kontekst „*Od njega zahtjeva dodatno vrijeme*“

	SM	SS	SP
zahtijeva	11.2644	10.9001	8.1259
zahtjeva	10.9154	10.5486	7.739
Točan (DA/NE)	NE	NE	NE

Tablica 2. Rezultati za rečenicu „*Jako mu sliči, a jednim djelom i na njegovog brata.*“, gdje je izvučeni kontekst „*a jednim djelom i na*“

	SM	SS	SP
djelom	11.0297	10.8074	8.7868
dijelom	9.6358	9.4425	7.5346
Točan (DA/NE)	DA	DA	DA

Tablica 3. Rezultati za rečenicu „*O njegovom dijelu je napisao članak.*“, gdje je izvučeni kontekst „*O njegovom dijelu je napisao*“

	SM	SS	SP
dijelu	9.7895	9.7136	7.8244
djelu	10.9641	10.7122	8.4317
Točan (DA/NE)	NE	NE	NE

Tablica 4. Rezultati za rečenicu „*Svidjela mu se plaća koju je dobio.*“, gdje je izvučeni kontekst „*mu se plaća koju je*“

	SM	SS	SP
plaća	9.3621	9.3351	7.6311
plača	12.0365	12.0179	9.9305
placa	12.6439	12.6141	10.4121
Točan (DA/NE)	DA	DA	DA

Tablica 5. Rezultati za rečenicu „*Dogodilo se baš kao što je rekao.*“, gdje je izvučeni kontekst „*baš kao što je rekao*“

	SM	SS	SP
što	5.7558	5.6494	4.4987
sto	9.3363	9.1664	7.4831
Točan (DA/NE)	DA	DA	DA

Tablica 6. Rezultati za rečenicu „*Nogomet je izgubio svoju čar.*“, gdje je izvučeni kontekst „*izgubio svoju čar*“

	SM	SS	SP
car	13.4812	13.4673	10.5193
čar	12.5175	12.4989	10.1611
Točan (DA/NE)	DA	DA	DA

Tablica 7. Rezultati za rečenicu „*Prije deset godina pisači su bili marljiviji.*“, gdje je izvučeni kontekst „*deset godina pisači su bili*“

	SM	SS	SP
pisači	13.6807	13.6383	10.9062
pisaći	14.1298	14.115	11.3855
Točan (DA/NE)	DA	DA	DA

Iz priloženih rezultata, usporedba se ne smije raditi između dva različita algoritma. Usporedba se jedino smije raditi za svaki algoritam zasebno, analizirajući njegove interne rezultate. Iz rezultata se može zaključiti da u sklopu ova tri algoritma ne postoji algoritam koji se ističe. Svi algoritmi donose iste zaključke, ali neki su u svoju odluku sigurniji.

Impresivna brzina pretraživanja podataka te analiza dobivenih podataka se ne smije zanemariti. Redom kao što su poredani u tablicama, trajanje upita bilo je 32, 29 i 30 milisekundi. ScyllaDB optimizira upite koji se ponavljaju više puta, ali uzimajući u obzir količinu pohranjenih redaka, to su iznimno impresivne brzine. Prilikom testiranja rijetko se događalo da vrijeme izvođenja bude veće od 100 milisekundi.

Zaključak

Cilj rada bio je omogućiti provjeru pravopisnih grešaka primjenom n-gramskog sustava. N-gami i njihove frekvencije pohranjene su u bazi podataka. Za određivanje koje će se riječi provjeravati, definirani su *confusion setovi* sa sličnim riječima koje sve postoje u hrvatskom jeziku. Za svaku riječ u ulaznom tekstu, provjerava se postoji li u nekom od *confusion setova*. Ako postoji dohvaćaju se frekvencije svih bigrama i trigrama koji sadrže sve riječi iz tog *confusion seta* na rubovima. Iz tih informacije se informacija donosi zaključak.

Funkcionalnost konkretnog algoritma koji se koristi u testiranju je manje bitna nego mogućnost dodavanja novih i izmjene starih algoritama. U slučaju da se pronađe algoritam koji daje bolje rezultate od trenutno implementiranih algoritama, migriranje sustava na taj algoritam je vrlo jednostavan i kratak proces.

U slučajevima kada je programska potpora dala krivu procjenu, razlog tome bio je to što su korisnici češće pisali pogrešne riječi. U slučaju da krajnji korisnici Haschecka bolje poznaju hrvatski pravopis i češće koriste pravopisno točne riječi, rezultati programske potpore razvijane u sklopu ovog rada sigurno bi bili bolji. Testiranje sustava je donijelo slične rezultate kao testiranje prikazano u prethodnom poglavlju. U otprilike 70 % slučajeva programska potpora razvijana u sklopu ovog rada uspije preporučiti pravopisno pravilnu riječ.

Ispravak n-gramske baze bi sigurno pomogao ovakvom sustavu koji isključivo barata statističkim podatcima. Jedan od načina ispravljanja baze bi mogao biti pronalaženje pogrešnog unigrama u sredini pentagrama, te njegovu postavljanje njegove frekvencije u nulu. Također propagiranje krivog pentagrama prema kraćim n-gramima i ponavljanje postupka postavljanja frekvencije u nulu. Taj postupak bi se morao odvijati za samo najčešće slučajeve budući da je pentagrama, pa tako i pogrešnih pentagrama jako puno.

Literatura

[1] Šandor Dembitz, Gordan Gledec, Hascheck (1993), *Hrvatski akademski spelling checker*. Poveznica: https://ispravi.me/; pristupljeno: ožujak 2024.

[2] Gledec, Gordan & Dembitz, Šandor & Blaskovic, Bruno. (2012). Croatian Language N-Gram System. Frontiers in Artificial Intelligence and Applications. 243. 696. 10.3233/978-1-61499-105-2-696.

[3] Fossati, D., Di Eugenio, B. (2007). A Mixed Trigrams Approach for Context Sensitive Spell Checking. In: Gelbukh, A. (eds) Computational Linguistics and Intelligent Text Processing. CICLing 2007. Lecture Notes in Computer Science, vol 4394. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-70939-8_55

[4] Wilcox-O'Hearn, A., Hirst, G., Budanitsky, A. (2008). Real-Word Spelling Correction with Trigrams: A Reconsideration of the Mays, Damerau, and Mercer Model. In: Gelbukh, A. (eds) Computational Linguistics and Intelligent Text Processing. CICLing 2008. Lecture Notes in Computer Science, vol 4919. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78135-6_52

[5] Eric Mays, Fred J. Damerau, Robert L. Mercer, Context based spelling correction, Information Processing & Management, Volume 27, Issue 5, 1991, Pages 517-522, ISSN 0306-4573, https://doi.org/10.1016/0306-4573(91)90066-U. (https://www.sciencedirect.com/science/article/pii/030645739190066U)

[6] ScyllaDB Inc., ScyllaDB (2015), *ScyllaDB*. Poveznica: https://www.scylladb.com/; pristupljeno: ožujak 2024.

[7] Rust Foundation, Rust (2015), *Rust*. Poveznica: https://www.rust-lang.org/; pristupljeno: ožujak 2024.

Sažetak

Kontekstno ispravljanje pravopisnih pogrešaka primjenom n-gramskog sustava

Programska potpora Hascheck, poznata i kao Ispravi.me, jako dobro obavlja posao pronalaska i ispravljanja „*non-word*“ pravopisnih pogreški, a za „*real-word*“ pravopisne pogreške, Hascheck će rijetko prepoznati te ispraviti pogrešku. Cilj ovog rada bio je izgraditi programsku potporu koja će omogućiti rješavanje „*real-word*“ vrstu pravopisnih pogreški uz pomoć n-gramskog sustava.

Razvijano programsko rješenje je podignuto na superračunalu kojeg održava organizacija Srce. N-gramski sustav se sastoji od ScyllaDB baze podataka koja sadrži n-grame i njihove frekvencije. Aplikativni sloj koji komunicira sa bazom napisan je u rustu, a iznad aplikativnog sloja izgrađen je web poslužitelj kako bi se omogućio pristup krajnjem korisniku.

Sustav ima predefinirane *confusion setove*, a pri upitu prima tekstualni zapis. Iz tog tekstualnog zapisa svugdje gdje pronađe riječ iz nekog *onfusion seta*, izvlači kontekst. Pomoću konteksta se kreiraju svi potrebni n-grami za svaku riječ iz *confusion seta* unutar kojega je prvotno bila pronađena riječ. Iz baze podataka se dobivaju frekvencije izgrađenih n-grama pa se te frekvencije analiziraju. Omogućeno je modularno kreiranje algoritma koji će iz frekvencija n-grama doći do zaključka. Jednostavno je dodavati i mijenjati algoritme.

Sustav implementira tri algoritma za računanje predikcije sa različitim matematičkim načinom izračuna. Programsko rješenje je u mogućnosti da bude operativno sa puno *confusion setova*, ali tijekom testiranja korišten je mali podskup. Svakom od tri definirana algoritma na ulaz je predan isti tekstualni zapis. Rješenja su pokazala slično ponašanje svih triju algoritama.

Rezultati prikazuju da je sustav u prosjeku točan u 70 % slučajeva. Sustav griješi u slučaju kada su korisnici za taj određeni kontekst više puta unijeli pravopisno pogrešnu riječ. Kada bi se iz baze podataka maknuli pravopisno pogrešni n-grami, sustav bi sigurno puno bolje funkcionirao.

***Ključne riječi**: n-gram, confusion set, rust, ScyllaDB, kontekst, Hascheck*

Summary

Contextual spellchecking based on n-gram system

The Hascheck software, also known as Ispravi.me, performs very well in detecting and correcting "*non-word*" spelling errors. However, for "*real-word*" spelling errors, Hascheck rarely identifies and corrects the mistake. The aim of this work was to develop software that would enable the resolution of "*real-word*" spelling errors using n-gram based system.

The developed software solution is deployed on a supercomputer maintained by the organization Srce. The n-gram system consists of a ScyllaDB database that contains n-grams and their frequencies. The application layer that communicates with the database is written in Rust, and a web server is built above the application layer to provide access to the end user.

The system has predefined confusion sets, and upon request, it receives a text input. From this text input, it extracts the context wherever it finds a word from any confusion set. Using the context, all necessary n-grams are created for each word from the confusion set within which the word was originally found. The frequencies of the constructed n-grams are obtained from the database and these frequencies are analyzed. The system allows for modular creation of algorithms that will derive conclusions from the n-gram frequencies. Adding and changing algorithms is straightforward.

The system implements three algorithms for prediction calculation, each using a different mathematical calculation method. The software solution can operate with many confusion sets, but a small subset was used during testing. The same text input was provided to each of the three defined algorithms. The solutions exhibited similar behavior across all three algorithms.

The results show that the system is accurate in 70 % of cases. The system makes errors when users have repeatedly entered a misspelled word for a specific context. If the misspelled n-grams were removed from the database, the system would certainly function much better.

Keywords: *n-gram, confusion set, rust, ScyllaDB, context, Hascheck*

Skraćenice

HTTP	Hypertext Transfer Protocol	protokol za prijenos hiperteksta
URL	Uniform Resource Locator	usklađeni lokator resursa