Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Janez Sedeljšak

Razvoj vgrajenega podatkovnega sistema

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2023

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Janez Sedeljšak
Naslov: Razvoj vgrajenega podatkovnega sistema
Vrsta naloge: Diplomska naloga na visokošolskem programu prve stopnje Računalništvo in informatika
Mentor: doc. dr. Boštjan Slivnik

Opis:
Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode naj uporabi, morda bo zapisal tudi ključno literaturo.

Title: Development of an embeded database system

Description:
opis diplome v angleščini

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DBMS	database management system	sistem za upravljanje podatkovnih baz
BCNF	Boyce-Codd normal form	normalna oblika, ki se uporablja pri normalizaciji podatkovne baze
API	application programming interface	aplikacijski programski vmesnik
SQL	structured query language	strukturiran jezik poizvedb
NoSQL	nerelacijske podatkovne baze	nonrelational databases
I/O	input/output operations	vhodno/izhodne operacije
SSD	solid-state drive	negiljivi diski
ER	entitiy relationship (diagram)	(diagram) entitet in relaciji
ORM	object-relational mapping	objektno relacijska preslikava
HTTP	hypertext transfer protocol	protokol za prenos hipertexta
CLI	command line interface	vmesnik za ukazno vrstico
CI/CD	continuous integration, continuous delivery	neprekinjena integracija in postavitev
CSV	comma-separated values	vrednosti, ločene z vejico

Povzetek

Naslov: Razvoj vgrajenega podatkovnega sistema

Avtor: Janez Sedeljšak

V diplomskem delu je predstavljenih trenutno nekaj najbolj uporabljenih sistemov za upravljanje podatkovnih baz (DMBS). V veliki meri so standard podatkovnih baz še vedno relacijske podatkovne baze. V ta namen je tekom dela predstavljen razvoj vgrajenega relacijskega sistema za programski jezik Python.

Sam razvoj namenske knjižnice je pripravljen v programskem jeziku C++, saj gre za nizko nivojski jezik, kjer imamo visoko fleksibilnost pri upravljanju s pomnilniku. Predstavljen je razvoj vseh potrebnih segmentov za dobro delujočo relacijsko podatkovno bazo. Ključnega pomena tekom razvoja je bila uporaba dobrih podatkovnih struktur in algoritmov, ki dobro izkoristijo I/O operacije, ki jih ponuja operacijski sistem in posledično pripeljejo do dobro pripravljenega podatkovnega sistema.

V zadnjem sklopu diplomskega dela smo pripravili analizo uspešnosti implementacije podatkovnega sistema na različnih scenarijih in ob različnih konfiguracijah. Poleg tega je pripravljena tudi analiza z že obstoječimi DMBS – SQLite in MySQL) ob enakovrednih scenarijih testiranja novo pripravljene knjižnice.

Ključne besede: Podatkovne baze, C++, Python, B+ drevesa, Podatkovne strukture.

Abstract

Title: Diploma thesis template

Author: Janez Sedeljšak

The thesis presents several currently used relational systems for working with data (DBMS). Relational databases are still widely used as the standard data storage systems. In this context, the development of an embedded relational database system for the Python programming language is presented.

The development of the dedicated library is implemented in the C++ programming language, which is a low-level language providing high flexibility in memory management. The development of all necessary components for a well-functioning relational database is described. During the development process, a key focus was on utilizing efficient data structures and algorithms that make effective use of I/O operations offered by the operating system, resulting in a well-prepared data system.

In the final part of the thesis, a performance analysis of the implemented data system is conducted under different scenarios and configurations. Additionally, an analysis is performed comparing the newly developed library with existing DBMS (SQLite and MySQL) using equivalent testing scenarios.

Keywords: Databases, C++, Python, B+ trees, Data structures.

Poglavje 1

Uvod

Živimo v dobi, kjer se spopadamo z izzivom obdelave izjemnih količin podatkov, ki jih poznamo kot velike podatke (velepodatki). Ti podatki so razpršeni po različnih platformah in predstavljajo dragocen vir informacij. Ko razmišljamo o dolgoročnem shranjevanju teh podatkov, se osredotočamo na uporabo podatkovnih baz. Na tem področju prepoznavamo dve osnovni skupini – relacijske in nerelacijske podatkovne baze.

1.1 Paradigme v svetu podatkovnih baz

1.1.1 Relacijske podatkovne baze

Trenutno na trgu še vedno prevladujejo relacijske podatkovne baze, ki predstavljajo standardno izbiro. Te baze temeljijo na striktni strukturi entitet, kjer so podatki organizirani v smiselne entitete, ki vključujejo stolpce (atribute) in vrstice (zapiske). Posebej pomembne so logične povezave med posameznimi zapisi, ki omogočajo boljšo organizacijo in interpretacijo podatkov. Te povezave se uresničujejo s pomočjo tujih ključev, kar omogoča vzpostavitev trdnih relacij med različnimi entitetami.

Gre za standard, ki se je prvič pojavil leta 1970, ko ga je razvil IBM [6]. Razvita je bila prva družina relacijskih podatkovnih baz DB2, katero je raz-

vil Edgar F. Codd – matematik izobražen na univerzi Oxford. **TODO**

1.1.2 Nerelacijske podatkovne baze

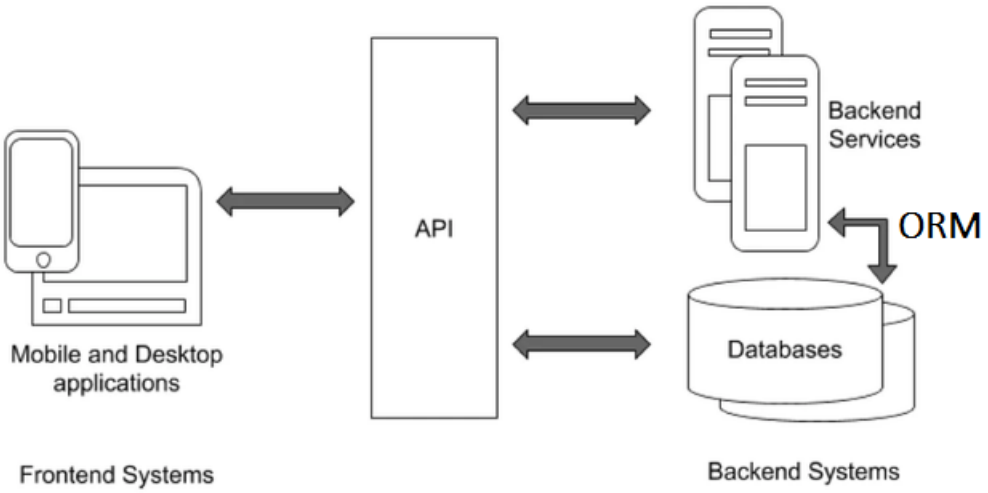
Nerelacijske podatkovne baze predstavljajo novo kategorijo baz, ki temeljijo na bistveno drugačnih osnovah kot tradicionalne relacijske podatkovne baze. Te nove baze so se razvile kot odziv na izzive, s katerimi se srečujejo relacijske podatkovne baze. Kot je poudaril višji predavatelj Aljaž Zrnec, "Podatkovne baze NoSQL niso bile ustvarjene z namenom popolne zamenjave relacijskih baz"[24]. Glavna ovira pri relacijskih podatkovnih bazah izvira iz njihove stroge strukture. V novi kategoriji podatkovnih baz, znanih kot NoSQL, je ključna lastnost prilagodljivost. Sama organizacija podatkov se bistveno razlikuje, pri čemer tradicionalne entitete in relacije med zapisi ustvarjalno nadomeščajo objekti in koncept dedovanja. Ta pristop prinaša tudi eno od glavnih prednosti, ki jih imajo nerelacijske podatkovne baze – sposobnost enkapsulacije posameznih zapisov omogoča učinkovito razporeditev celotne baze na več strežnikov (horizontalno skaliranje podatkov), ki ga tradicionalne relacijske podatkovne baze težje dosežejo.

1.2 Kje se danes uporabljajo relacijske podatkovne baze?

Relacijske podatkovne baze so že desetletja temeljna komponenta informacijskih sistemov in njihova uporaba se je v današnjem sodobnem poslovnem okolju še povečala. Kljub pojavu novejših tehnologij in podatkovnih modelov imajo relacijske podatkovne baze številne aplikacije, kjer izstopajo zaradi svoje strukture, zanesljivosti in možnosti za kompleksno analizo.

1.2.1 Vloga DBMS v informacijskih sistemih

Spodnja slika predstavlja standardno arhitekturno zasnovo za večino modernih spletnih aplikaciji:



Slika 1.1: Struktura sodobnega informacijskega sistema

Na sliki 1.1 je predstavljena struktura odjemalec - strežnik, katero sestavljajo tri ključne komponente. Na eni strani so odjemalci, ki vključujejo mobilne aplikacije, spletne vmesnike ... Ti odjemalci z strežniškim delom aplikacije komunicirajo preko standardiziranih metod na nivoju API komponente.

Večji del informacijskega sistema oz. spletne aplikacije se nahaja v ozadju, kjer je realizirana poslovna logika in upravljanje s podatki – podatkovno bazo. V sodobnih informacijskih sistemih za dodaten nivo varnosti in lažjo manipulacijo podatkov pogosto uporabimo še ORM. Gre za objektno predstavitev posameznih entitet znotraj podatkovne baze in predstavitev zapisov, kot instance objektov. ORM predstavlja dodaten nivo abstrakcije, ki programerju med drugim omogoči tudi avtomatske migracije podatkovne baze, kot tudi lažjo integracijo podatkovne baze znotraj uporabljenega programskega jezika.

1.3 Motivacija za razvoj lastnega DBMS in ORM vmesnika

Glavna motivacija za razvoj izhaja iz želje po boljšem razumevanju notranjega delovanja podatkovnih baz, ter njihovega vpliva na delovanje modernih aplikaciji. Kot programerji se pogosto osredotočamo le na izgradnjo funkcionalnosti, pri čemer podatkovno bazo pogosto spregledamo oz. spregledamo vpliv podatkovnega sloja na delovanje celotne aplikacije.

Podatkovna baza s seboj prinese visoko raven abstrakcije, ki je podprta z obsežno množico algoritmov in konceptov, ki omogočijo zanesljivo, hitro in varno izvajanje operaciji nad podatki. Tekom raziskave je cilj spoznati tudi te koncepte, ki sicer ostajajo skriti v abstrakciji, ki jo prinašata DBMS in jezik SQL.

Pomembnost podatkovnega sloja postane očitna, predvsem ko se aplikacija začne odzivati počasneje. Pogost razlog je obsežna količina podatkov v posamezni entiteti, kar za uporabnika hitro pomeni nesprejemljiv čas odzivnosti celotnega sistema.

Ob iskanju pristopov za optimizacijo podatkovnega sloja, hitro naletimo na koncept indeksiranja podatkov. Ta pristop ni omejen zgolj na relacijske podatkovne baze, temveč se pojavlja širom računalniške znanosti. Osnovna ideja za optimizacijo je ustvarjanje iskalne strukture, ki omogoča izrazito hitrejše iskanje podatkov z izrabo učinkovite podatkovne organizacije.

Dve bolj pogosti podatkovni strukturi za izvedbo indeksiranja predstavljajo zgoščevalne tabele in drevesa. Na področju relacijskih podatkovnih baz se najpogosteje uporablja več nivojsko indeksiranje, ki je realizirano prav z drevesi (v večini primerov gre za B-drevesa). Drevo, je sestavljeno iz vozlišč, kjer ima vsako največ M zapisov in $M + 1$ kazalcev na nadaljnja vozlišča. S pomočjo dodajanja indeksov lahko linearno iskanje – $O(N)$ bistveno pohitrimo, kar z uporabo B-drevesa pomeni časovno zahtevnost $O(log_M(N))$.

Pomembno je poudariti, da cilj implementacije lastne rešitve ne vključuje razvoja kompleksnega in dovršenega DBMS, ki bi se primerjal z naprednimi in vzpostavljenimi rešitvami. Naš cilj je razviti knjižnico, ki bo vsebovala temeljne funkcionalnosti in bo primerna za uporabo v manjših aplikacijah. Osredotočili se bomo predvsem na izgradnjo preprostega in intuitivnega načina za upravljanja s podatki.

Poglavje 2

Sorodna dela

Prve podatkovne baze so se razvile že pred petdesetimi leti in ob tem tudi ogromno različnih pristopov za komunikacijo med programskimi jeziki in DBMS. V računalništvu je izbor orodja ključnega pomena in izbor tehnologij znotraj podatkovnega sloja nikakor ne predstavlja izjeme.

2.1 Izbor DBMS

Pri izbiri relacijske podatkovne baze se soočimo z različnimi možnostmi, med katerimi izstopajo standardi, kot sta DB2, Oracle, ter Microsoft SQL Server (MSSQL), ki predstavljajo največje in najbolj skalabilne DBMS-je. Kljub temu je izbor DBMS-ja, podobno kot pri večini področij v računalništvu, odvisen od zahtev naše aplikacije in narave podatkov, ki jih bomo shranjevali.

Za mnoge primere morda zadostujeta bolj preprosta DBMS-ja, kot sta PostgreSQL ali MySQL, ki ponujata enostavno upravljanje in predstavljata dobro izbiro tudi za shranjevanje obsežnih količin podatkov.

V določenih scenarijih, ko imamo opravka z bolj preprosto in manj obsežno strukturo podatkov (npr. pri razvoju mobilnih aplikaciji), se za najprimernejšo izbiro izkaže SQLite. Gre za minimalističen DBMS, ki za razliko od prej omenjenih izbir za potrebe shranjevanja podatkov uporablja zgolj datoteko, ki jo zlahka enkapsuliramo v izolirano okolje, kot je mobilna naprava.

V področje manjših podatkovnih baz, pa lahko uvrstimo tudi razvit DBMS – Graphenix[16].

2.2 Statističen pregled Python ORM knjižnic

V spodnji tabeli je predstavljena kratka statistična analiza najpogosteje uporabljenih ORM knjižnic, ki so na voljo v programskem jeziku Python. Podatke za število mesečnih in tedenskih prenosov smo pridobili iz spletne platforme PYPI Stats [14], kjer se nahaja sledenje prenosov posameznih paketov znotraj Python ekosistema. Podatki za število GitHub zvezd, pa so pridobljeni direktno iz repositorijev posameznih knjižnic, ki so dostopne na GitHub platformi.

Knjižnica	**GitHub zvezde**	**Mesečni prenosi**	**Tedenski prenosi**
Django [2]	72 **tisoč**	10 milijonov	2.4 milijona
SQLAlchemy [17]	7.5 tisoč	83 **miljonov**	20 **miljonov**
Peewee [10]	10 tisoč	1.1 milijon	281 tisoč
Tortoise ORM [20]	3.7 tisoč	88 tisoč	22 tisoč
SQLObject [18]	133	27 tisoč	6 tisoč

Glede na zgornjo tabelo izluščimo, da v ekosistemu prevladujejo naslednje tri ORM knjižnice:

- **Django ORM [2]** je vgrajen sistem v Django knjižnico za izdelavo spletnih aplikaciji. Omogoča zelo širok nabor operaciji nad podatkovno bazo, poleg tega ima vgrajen tudi svoj CLI. S tem celotno ogrodje bistveno pospeši produktivnost razvijalcev, a hkrati kot ogrodje doda, kar 8.9 MB dodatne teže. Knjižnica podpira pet podatkovnih sistemov, to so PostgreSQL, MariaDB, MySQL, Oracle, SQLite.

- **SQLalchemy [17]** gre za eno najbolj fleksibilnih in uporabljenih knjižnic znotraj programskega jezika. Sama fleksibilnost pomeni, da številne operacije, ki so znotraj Django ORM že avtomatizirane so tukaj prepuščene

implementaciji razvijalca. Fleksibilnost, pa s seboj prinaša tudi bistveno prednost, saj lahko kot razvijalec, ki dobro pozna ozadje delovanja uporabljenega relacijskega podatkovnega sistema uporabimo različne pristope optimizaciji, ki jih znotraj Django ORM ni mogoče realizirati oz. je implementacija bistveno težja. Knjižnica zaradi lahkotne abstrakcije podpira SQLite, MySQL, Oracle, MS-SQL in še ostale DBMS sisteme.

- **Peewee** [**10**] je ena izmed preprostejših knjižnic, ki predstavlja lahkoten nivo abstrakcije. Knjižnica podpira 3 relacijske podatkovne sisteme, to so PostgreSQL, MySQL in SQLite. Gre za dokaj omejen nabor podprtih relacijskih podatkovnih sistemov, vendar zaradi svoje preprostosti knjižnica ima svojo ciljno skupino razvijalcev, ki razvijajo preproste sisteme in ne potrebujejo kompleksnosti, ki jih s seboj prineseta Django ORM in SQLalchemy.

Cilj vseh teh knjižnic je pospešiti proces razvoja aplikacij in zagotoviti dodatno varnost pri delu s podatkovno bazo. Omeniti velja, da te knjižnice omogočajo tudi druge koristne funkcionalnosti, kot je lažja manipulacija s podatki, poenostavljeno vzdrževanje in enostavna migracija strukture. S svojo abstrakcijo in intuitivnimi vmesniki odpravljajo potrebo po neposrednem rokovanju z zapletenimi SQL poizvedbami ter s tem razbremenjujejo razvijalce in omogočajo hitrejši razvoj aplikacij.

2.2.1 Slabost uporabe ORM knjižnic na nivoju poizvedb

Kljub vsem dobrim lastnostim omenjenih knjižnic, pa tudi vsaka izmed njih prinaša dodaten nivo abstrakcije in razvijalcu skrije veliko možnosti za optimizacijo delovanja relacijskega podatkovnega sistema. V algoritmih je pogost kompromis med hitrostjo in porabo prostora, ko pa govorimo o ORM knjižnicah in programskih jezikih, pa višji nivo abstrakcije pomeni kompromis hitrosti delovanja.

Narava relacijskih podatkovnih baz in združevanja tabel je usmerjena v skupen rezultat v obliki ene matrike, kjer imamo kartezičen produkt zapisov različnih entitet, katerega filtriramo glede na povezava, ki so realizirane s pomočjo tujih ključev. Pogosto, pa kot ciljni uporabniki ne želimo tega, temveč želimo drugačno strukturo podatkov, ki pa ni v skladu z osnovno idejo relacijskih baz. V ta namen ORM knjižnice izvedejo več različnih poizvedb in potem znotraj jedra knjižnice združujejo zapise in kreirajo ciljno strukturo ali pa izvedejo eno kompleksnejšo poizvedbo in pridobljene podatke nato pretvorijo v končno strukturo. Primer, take preproste naloge je: Ža vse uporabnike pridobi njihove naloge in zadnjih pet sporočil, ki jih je vsak izmed teh uporabnikov poslal".
Za nalogo je SQLAlchemy jedro pripravilo naslednjo poizvedbo:

```
SELECT * FROM users
LEFT OUTER JOIN tasks AS t ON users.id = t.user_id
LEFT OUTER JOIN messages AS m ON users.id = m.user_id
```

Zgornji pristop pomeni, da za vsakega uporabnika prenesemo $T_i \cdot M_i$ sporočil, kjer je T_i število nalog i-tega uporabnika in M_i število sporočil i-tega uporabnika.

Izrek 2.1 *Skupno število zapisov, ki jih pridobimo je definirano s pomočjo naslednje vsote*

$$\sum_{i=1}^{N} T_i \cdot M_i \tag{2.1}$$

Izrek 2.2 *Skupno število zapisov, ob idealnih pogojih (brez podvajanja podatkov)*

$$N + \sum_{i=1}^{N} T_i + M_i \tag{2.2}$$

Torej najprej potrebujemo N zapisov za vsakega uporabnika, nato pa za vsakega uporabnika pridobimo še T_i nalog in M_i nalog. Če predpostavimo, da

imamo podatkovno bazo, kjer imamo sto uporabnikov in ima vsak izmed njih tisoč sporočil, ter dvesto nalog pomeni, da bomo v primeru prve poizvedbe (2.1) prenesli, kar $100 \cdot 1000 \cdot 200$ oz. 20 000 000 zapisov.

Medtem, ko v primeru druge poizvedbe (2.2) to pomeni prenos bistveno manjše strukture $(100 \cdot (1000 + 200)$ oz. 120 000. Kar predstavlja le 0.6% količine podatkov, ki jih je potrebno prenesti iz podatkovnega sistema do naše ORM implementacije.

Ozko grlo, pa se pojavi še v enem segmentu, namreč pripravljanje končne strukture, ki jo vrne ORM je v vseh treh omenjenih knjižnicah izvedeno na nivoju programskega jezika Python, ki sam po sebi ni optimiziran za filtriranje in združevanje zapisov oz. ne omogoča pristopov, ki jih lahko uporabimo znotraj prevedenih jezikov in tako bistveno pohitrimo kreiranje končne strukture, ki predstavlja glavni rezultat poizvedbe.

Poglavje 3

Razvoj jedra DBMS

V poglavju predstavimo razvoj jedra DBMS s pomočjo programskega jezika C++. V prvi fazi je predstavljena celotna struktura podatkov in način shranjevanja na disk. Za tem predstavimo razvoj B+ drevesne strukture za optimalno indeksiranje podatkov. Nato se spustimo v predstavitev optimizaciji oz. algoritmov, ki omogočajo hitrejše branje podatkov in za zaključek še predstavitev funkcionalnosti, ki jih naš DBMS podpira.

Vsa izvorna koda je dostopna na GitHub repozitoriju [16].

Skozi poglavje bodo vsi primeri dela z DBMS na nivoju programskega jezika Python uporabljali preprost ER model z uporabniki, nalogami in sporočili. Uporabnik lahko ima več nalog in sporočil – pri sporočilih, pa imamo še dodatno vlogo, saj je lahko uporabnik pošiljatelj/prejemnik.

```
class User(Model):
    name = Field.String()
    tasks = Field.VirtualLink("user")
    sent_msgs = Field.VirtualLink("sender")
    recieved_msgs = Field.VirtualLink("reciever")

class Task(Model):
    content = Field.String(size=100)
    user = Field.Link("User").as_index()

class Message(Model):
    content = Field.String(size=50)
    date = Field.DateTime().as_index()
    sender = Field.Link("User").as_index()
    reciever = Field.Link("User").as_index()
```

3.1 Struktura shranjevanja podatkov

3.1.1 Datotečna struktura za shranjevanje podatkov znotraj entitet

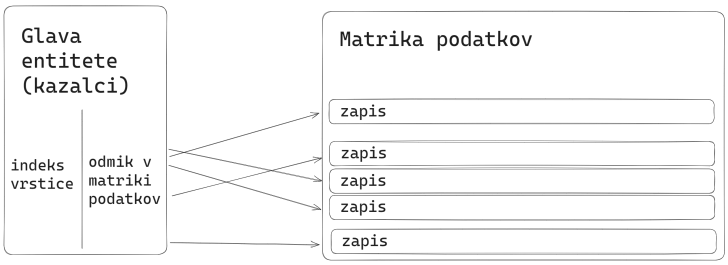
Za vsako posamezno entiteto se privzeto kreirata 2 datoteki $ix\_ < naziv\_entitete > .bin$ in $< naziv\_entitete > .bin$.

1. $< naziv\_entitete > .bin$ (matrika podatkov) – je matrična struktura, ki predstavlja dejanske zapise podatkov. Vsaka vrstica predstavlja en zapis, kjer je dolžina vrstice vsota dolžin vseh atributov, ki se shra-

njujejo v entiteti. Posamezen stolpec matrike pa predstavlja atributov vsakega zaporednega zapisa.

2. $ix\_ < naziv\_entitete > .bin$ (glava entitete s kazalci) – gre za vektorsko strukturo, kjer vsaka zaporedna vrstica vsebuje kazalec na dejanski zapis v matriki podatkov - ta kazalec je realiziran, kot celoštevilski odmik zapisa v matriki. Poleg kazalcev struktura vsebuje tudi kazalec na prvo prosto vrstico v matriki:

   - Matrika je polna – kazalec na konec datoteke
   - V matriki podatkov obstaja fragmentacija – kazalec na prvo prosto vrstico v matriki



Slika 3.1: Struktura podatkovne matrike in glave entitete s kazalci

Popravi sliko - izvedba v tikz Latex

3.1.2 Tipi podatkov

Znotraj našega vgrajenega podatkovnega sistema podpiramo 5 podatkovnih tipov:

1. Cela števila (ang. Int) – v tem primeru gre za vrednost, ki je na nivoju C++ jezika shranjena kot int64_t oziroma predznačeno celo število. Velikost za posamezno vrednost je 8B. Sam razpon vrednosti, ki jih lahko shranimo v posamezen zapis je $[-2^{63}, 2^{63} - 1]$.

2. Realna števila (ang. Float) – zaradi lažje implementacije je velikost realnih števil enaka, kot velikost celih števil – torej 64 bitov. Vrednost je na nivoju podatkov realizirana, kot dvojni "float"oz. "double". Gre za zaporedje bitov, ki so po standardu IEEE 754 [7] ločeni v predznak, eksponent in mantiso.

3. Nizi (ang. String) – Gre za zaporedje znakov, kjer na nivoju posamezne entitete določimo maksimalno dovoljeno dolžino posameznega zapisa. Torej velikost posameznega zapisa je $N * 1B$, kjer je N maksimalna dolžina podatka.

4. Logične vrednosti (ang. Bool) – je najmanjši podatkovni tip, ki ga vsebuje naš DBMS in zavzame vsega 1B; drži pa lahko vrednosti 0/1 oz. True/False, kot to definira Python programski jezik.

5. Datumi (ang. Datetime) – gre za podatkovni tip, ki ponovno shranjen, kot vrednost int64_t in je predstavljen v EPOCH formatu – gre za časovni odmik trenutnega časa od UTC datuma 1. 1. 1970 [4].

6. Povezava (ang. Link) – gre za vgrajen tip relacije, kjer je podatek kazalec na drug zapis v določeni entiteti. V ozadju je to le primarni ključ določenega zapisa, ki se nastavi avtomatsko (ponovno gre za podatek, ki je zapisan kot int64_t vrednost).

7. Virtualna povezava (ang. VirtualLink) – gre za navidezno polje v entiteti, ki ne zavzame nobenega dodatnega prostora za shranjevanje podatkov. Polje je pomembno za kreiranje povezave, kjer na določen zapis vežemo seznam zapisov iz druge tabele.

3.1.3 Realizacija relaciji med entitetami

Sama implementacija relaciji je dokaj preprosta, saj se povezave kreirajo avtomatsko direktno preko primarnega ključa posameznega zapisa. Razvijalcu je tako prepuščena le definicija entitete in atributov, kjer določimo povezave

(oz. relacije) na drugo entiteto. Sama relacija, pa je realizirana s preprostim kazalcem, ki kaže na zapis v glavo entitete, ki jo želimo povezati.

3.2 Indeksiranje z uporabo B+ dreves

3.2.1 B+ drevesa in njihove značilnosti

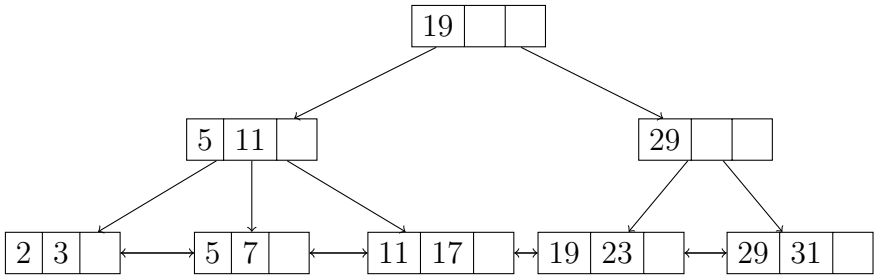
B+ drevesa so podatkovna struktura, uporabljena za učinkovito indeksiranje in iskanje podatkov. Vsako vozišče ima lahko največ M podatkov in $M + 1$ kazalcev na nova vozlišča.

V B+ drevesih so vsi nivoji razen listov drevesa, vmesna vozlišča, ki vsebujejo kazalce na nadaljnja vozlišča znotraj drevesa - služijo le kot povezave ter omogočajo učinkovitejše iskanje in navigacijo po strukturi. Medtem so podatki shranjeni v listih drevesa.

Definicija strukture zahtevna tudi striktno uravnoteženost drevesa, torej da so vsi listi drevesa na istem nivoju.

Kot dodatna optimizacija za iskanje intervalov znotraj drevesa je v naši implementaciji dodana še struktura dvojno povezanega seznama na nivoju listov drevesa. Tako ima vsak list kazalec na predhoden in naslednji list v drevesu. Na primeru podatkovne baze, je to zelo uporabno, ko iščemo npr. prvih petdeset uporabnikov razvrščenih po priimku. V prvi fazi je seveda potrebno imeti kreirano indeksno strukturo za iskan atribut. Zatem pa je iskanje zapisov dokaj trivialno, namreč sprehodimo se le do najbolj levega lista v drevesu, nato pa se sprehodimo po povezanem seznamu od leve proti desni in shranimo kazalce na želene zapise, jih nato le preberemo iz matrike podatkov, ter razvrstimo po ciljnem atributu.

Na spodnji sliki je prikazan preprost primer B+ drevesa realiziran s stopnjo $M = 2$: Dodaj še kazalce iz posameznega zapisa v listu (slika)



Slika 3.2: Primer B+ drevesa, kjer indeksiramo pošiljatelje sporočil

3.2.2 Pravila, ki jih zahteva B+ drevesna struktura

1. Vsako vozlišče ima največ M podatkov in $M + 1$ kazalcev.

2. Listi drevesa so vsi na enakem nivoju.

3. Urejenost - vse vrednosti, ki se nahajajo levo morajo biti manjše ali enake trenutni; vse vrednosti desno, pa marajo biti večje ali enake.

4. Vmesna vozlišča morajo vsebovati $X + 1$ kazalcev, kjer je X število podatkov v vozlišču.

5. Vozlišče je veljavno, kadar ima vsaj $C/2$ zapisov, kjer je C kapaciteta oz. maksimalno število podatkov, ki jih lahko ima vozlišče.

3.2.3 Implementacija iskanja v MySQL in SQLite DMBS

https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html

3.2.4 Implementacija B+ dreves za shranjevanje na disk

Zakaj se na področju podatkovnih baz uporabljajo B drevesa. Ozadje je dokaj preprosto, saj klasične iskalne strukture s pomočjo dreves, ki so realizirane s pomočjo binarnih dreves vsebujejo izrazito ozko grlo. Ozko grlo je pride

do izraza, ko je drevo shranjeno na disku in potrebujemo za vsako vozlišče izvesti eno branje iz diska.

Ko beremo vozlišča pomeni eno branje vozlišča en dostop do diska in v primeru binarnega drevesa to pomeni, da je število dostopov do datoteke enako številu podatkov, ki smo jih prebrali - branje vsekakor predstavlja počasno operacijo in število teh branj želimo čim bolj minimizirati.

B drevesa ta problem rešujejo s samo količino podatkov, ki je shranjena na nivoju enega vozlišča. V klasičnih trdih diskih je velikost enega vozlišča drevesa bila določena, kot velikost sektorja na trdem disku. To je pomenilo, da je branje vsakega vozlišča predstavljalo točno eno branje iz diska.

Klasičen način določanja velikosti posameznih vozlišč se je dokaj spremenil odkar so standard za shranjevanje podatkov postali hitrejši tipi diskov kot so SSD.

3.2.5 Uporaba B+ dreves v bazi podatkov (iskanje po indeksiranih podatkih)

Na nivoju podatkovne baze se B+ drevesa uporabljajo za indeksiranje podatkov oz. zapisov glede na določen atribut npr. datum rojstva, ime, hišno številko itd. Gre za pristop, ki nam omogoča filtriranje posameznih zapisov s pomočjo binarnega iskanja v logaritmičnem času namesto linearnega iskanja skozi celotno matriko.

Določanje indeksov, pa je v klasičnih podatkovnih bazah lahko tudi malo bolj kompleksno, saj poleg iskanja po določenem atributu lahko iščemo tudi po izpeljanih vrednostih npr. po kombinaciji prve črke priimka in imena. V primeru našega podatkovnega sistema to ni realizirano, lahko pa si sami kreiramo dodaten atribut na nivoju entitete in vanj shranjujemo želeno vrednost, ter nato direktno nad atributom kreiramo indeksno strukturo.

3.2.6 Dinamično nalaganje in ohranjanje posameznih segmentov drevesa v pomnilnik

Za dodatno optimizacijo delovanja internih indeksov je dodana tudi implementacija predpomnilnika, kamor lahko shranjujemo posamezne segmente drevesa. Deluje, kot preprost zgoščevalni slovar, kjer je ključ odmik posameznega vozlišča v fizični datoteki vrednost, pa je dejansko vozlišče.

Omenjena implementacija nam omogoča preprosto upravljanje z vozlišči, saj je dostop do vsakega vozlišča omogočen preko skupnega vmesnika, ki sicer pomeni dodatno rabo pomnilnika vendar, pa bistveno dvigne varnost, ko želimo opravljati s posameznim zapisom.

3.2.7 Slabosti uporabe indeksov

Sama raba indeksov je koristna, saj bistveno pohitri postopek iskanja posameznih zapisov glede na določen atribut. Ko govorimo v časovni zahtevnosti to pomeni pohitritev iz $O(N)$ na $O(log_M(N))$.

Na drugi strani pa se skriva kar nekaj lastnosti B+ dreves, ki jih moramo imeti v mislih ob dodajanju nepotrebnih indeksov v našo podatkovno bazo.

- Časovna zahtevnost vnašanja in posodabljanja zapisov v posamezno entiteto se poveča iz $O(1)$ na $O(log_M(N) * X)$, kjer je X število postavljenih indeksov. N pa predstavlja število že obstoječih zapisov v posamezni entiteti.

- Za vsak postavljen indeks se bistveno poveča poraba prostora na disku, saj shranjevanje drevesne strukture s seboj prinese podvajanje atributa, kot tudi dodatek kazalca na glavo entitete.

3.2.8 Ostali pristopi indeksiranja podatkov

Indeksiranje s pomočjo zgoščevalnih tabel ...

3.3 Pomembni gradniki za optimalno izvedbo poizvedb

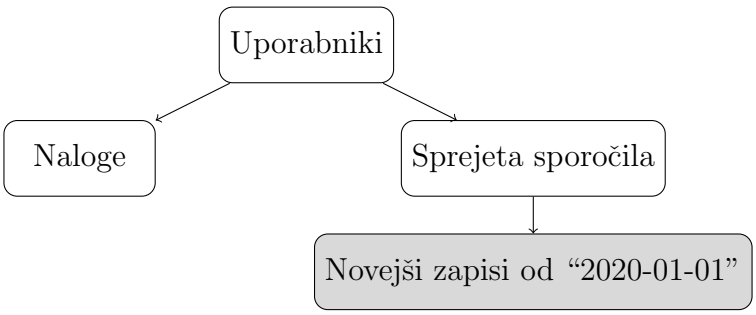
3.3.1 Gradnja dreves za pogojni del poizvedb

Iskanje zapisov glede na strukturo pogoja in kako sestavimo drevsno strukturo

3.3.2 Izvedba poizvedb na več entitetah hkrati

DBMS nima klasičnega načina združevanja entitet, kot nam je to poznano iz ostalih relacijskih podatkovnih sistemov, kjer je to realizirano s pomočjo kartezičnega produkta nad entitetami in potem dodatna povezava s pomočjo tujih ključev.

```
query = User.link(
    tasks=Task,
    recieved_msgs=Message.filter(
        Message.date >= datetime(2020, 1, 1, 0, 0, 0)
    )
)
count, rows = query.all()
```



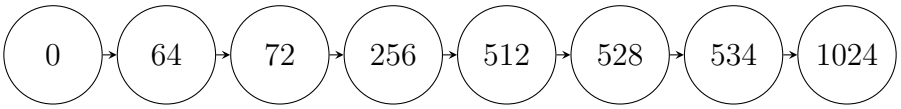
Slika 3.3: Drevesi prikaz poizvedbe, ki se izvaja usmerjeno od vozlišča proti listom

3.3.3 Optimizacija poizvedb z gručenjem (ang. clustering)

Razlaga postopka gručenja

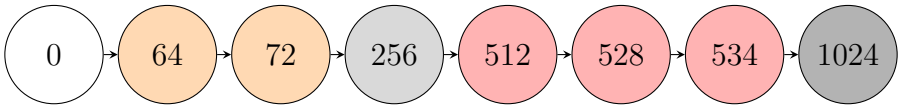
Kot je bilo že omenjeno v prejšnjih poglavjih je največje ozko grlo, s katerim se srečujemo I/O operacije (torej pisanje in branje iz/na disk). V ta namen je tudi v segmentu prenosa podatkov iz datoteke v pomnilnik realizirano s čim manj dostopi do diska. V ta namen je ob prvi fazi nabora dodan algoritem za gručenje podatkov.

Sam problem je zelo preprost imamo urejen vektor celih števil, kjer želimo vrednosti grupirati glede na posamezne odmike in najti skupine oz. gruče in te združiti v eno branje matrične datoteke s podatki. Za primer lahko vzamemo naslednji vektor odmikov v matrični datoteki:



Slika 3.4: Zaporedje odmikov za poizvedbo iz matrične datoteke

Iz odmikov lahko hitro razberemo dve gruči, kateri bi želeli, da naš algoritem odkrije in branje nad odmikoma izvede v enem branju - to sta gruči $[64, 62]$, ter $[512, 528, 540]$. Optimalen pristop branja bi lahko barvno označili po naslednji konfiguraciji:



Slika 3.5: Zaporedje odmikov za poizvedbo iz matrične datoteke z obarvanjem gruč

Linearna izvedba gručenja nad urejenim seznamom

Za izvedbo gručenja obstaja veliko algoritmov, ki nalogo opravijo zelo dobro vendar s seboj prinesejo visoko časovno zahtevnost. V naši izvedbi gručenja smo želeli poiskati algoritem, kjer gručenje opravi v linearnem času, saj ob testiranju ostalih algoritmov v večini primerov samo gručenje traja dlje, kot pa branje posameznih elementov v matriki. V ta namen smo pripravili preprost algoritem, ki se sprehodi skozi seznam in spremlja odmike, ter združuje elemente v kolikor ne presežemo zgornje meje velikosti ene gruče in poleg tega zadovoljimo minimalno velikost za gručo.

Spodaj je predstavljen algoritem za gručanje elementov v urejenem seznamu:

```
fn clusterify(offsets: vector) -> vector of vectors
    clusters := empty vector
    n := size of offsets
    i := 0

    while i < n do
        j := i
        while (not end and not reached cluster threshold) do
            j := j + 1
        end while

        if (reached min cluster size) then
            clusters.add(subvector from index i to j + 1)
        else
            for k := i to j do
                clusters.add(vector containing offsets[k])
            end for
        end if

        i := j + 1
    end while

    return clusters
end function
```

3.3.4 Optimizacija na nivoju urejanja podatkov ob izvedbi nabora nad posamezno entiteto

Ob implementaciji omejevanja število rezultatov s pomočjo ukazov ‘limit‘ in ‘offset‘ moramo poskrbeti za časovno optimalno izvedbo poizvedbe, saj iz

vidika uporabnika relacijskega sistema pričakujemo, da z dodatkom ukaza 'limit' podatkovnemu sistemu povemo, da želimo omejeno število podatkov in posledično pričakujemo hitrejšo poizvedbo.

Tekom razvoja smo testirali nekaj različnih pristopov in podatkovnih struktur. Naš problem lahko definiramo, kot iskanje K najmanjših elementov, kjer je K sestavljen iz dveh komponent L in O, kjer je L maksimalno število elementov oz. limita, O pa predstavlja odmik oz. število zapisov, ki jih želimo preskočiti.

Tekom razvoja smo preizkusili 3 različne pristope za iskanje K najmanjših elementov v seznamu.

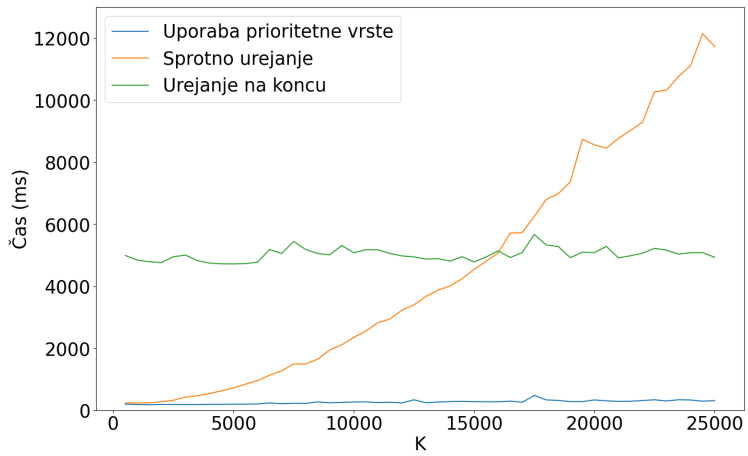
1. Klasično vstavljanje v seznam in ureditev elementov ob koncu vnašanja, ter rezanje elementov od indeksa O, do vključno $O + L$. Časovna zahtevnost tega pristopa je $O(N \cdot log_2(N))$, prostorska zahtevnost, pa je tukaj $O(N)$, saj moramo v pomnilniku držati vse potencialne elemente.

2. Postopno vstavljanje v urejen seznam, v tem primeru uporabimo vgrajeno std::vector [23] strukturo, kjer vstavljanje izvajamo s pomočjo binarnega iskanja. Težava pristopa se skriva v vstavljanju v začetni del seznama, saj moramo potem vse elemente zamikati za eno mesto v desno. Posledično, čeprav je časovna zahtevnost binarnega iskanja $O(log_2(N))$ zaradi premikanja elementov časovna zahtevnost postane $O(N)$. Tako je časovna zahtevnost celotnega postopka vstavljanja $O(N^2)$ v najboljšem primeru pa $O(N \cdot log_2(K))$. Glavna prednost algoritma pred pristopom urejanja na koncu se nahaja v prostorski zahtevnosti. Ta je namreč le $O(K)$, saj v pomnilniku ohranjamo maksimalno K vrstic in jih po potrebi zamenjujemo ob iteraciji skozi vrstice matrike.

3. Za daleč najboljši pristop se je izkazalo vstavljanje v prioritetno vrsto, ki je realizirana, kot kopica. Za podatkovno strukturo uporabimo vgrajeno C++ std::priority_queue [11]. V tem primeru je časovna zahtevnost posameznega vstavljanja $O(log_2(K))$, kar pomeni za N vstavljanj $O(N \cdot log_2(K))$. Prostorska zahtevnost, pa je enako kot v zgornjem

primeru $O(K)$. Problem je zastavljen, tako da K ne more nikoli biti večji od N in je posledično časovna zahtevnost tega pristopa najbolj optimalna. Pomemben segment pri uporabi vgrajene prioritetne vrste je tudi izbor podatkovne strukture, ki jo vrsta uporablja za shranjevanje podatkov. Za najbolj optimalen pristop se je izkazal vgrajeni std::vector [23], ki je na nivoju pomnilnika shranjen v enem kosu, kar omogoči prevajalniku in procesorju, da izrabita še optimizacijo z lokalnostjo.

Za zgornje tri pristope obstaja veliko spremenljivk, ki vplivajo na čas izvajanja samega algoritma za iskanje K najmanjših elementov v seznamu. Največ vpliva ima vrednost N, ki predstavlja število zapisov, ki jih moramo preveriti in primerjati z ostalimi elementi. Je tudi spodnja meja za vse časovne zahtevnosti, vse imajo namreč časovno zahtevnost vsaj $O(N)$. Bistveno bolj zanimiv, pa je vpliv vrednosti K, ki predstavlja tudi razliko v časovnih zahtevnostih med posameznimi algoritmi. Za lažje razumevanje, kako vrednost K vpliva na čas izvajanja smo pripravili tudi grafično analizo, kjer v izoliranem okolju izvedemo vse tri algoritme. Vrednost N je nastavljena na $10\,000\,000$, K pa se spreminja na x osi in je definiran na intervalu $[500, 15000]$.



Slika 3.6: Časovna primerjava pristopov za iskanje K najmanjših elementov v seznamu

Iz zgornjega grafa lahko razberemo, da vrednost K nima bistvenega vpliva na prvi in tretji algoritem. Izjema je drugi algoritem, kateri ima sicer časovno zahtevnost $O(N^2)$, vendar je K zelo pomemben faktor, saj večji kot je K večkrat moramo izvesti vstavljanje v vektor, kar pa postaja sorazmerno počasneje glede na večanje vrednosti K.

Izkaže se, da je algoritem z uporabo prioritetne vrste najbolj optimalen pristop, kar pokažejo tudi perfomančni testi, ki se izvajajo na platformi GitHub, saj z uporabo omenjenega pristopa dobili do 4x pohitritev izvedbe nabora podatkov iz ene entitete.

3.3.5 Uporaba podatkovnih okvirjev ob poizvedovanju podatkov

Akcija za poizvedovanje ob prvi implementaciji deluje po naslednjih korakih:

1. Izgradnja strukture, ki definira vse potrebne parametre za poizvedbo – QueryObject.

2. Izvedba branja na nivoju programskega jezika C++ in pretvorba zapisov v seznam, kjer je posamezen zapis predstavljen kot slovar.

3. Pretvorba posameznega slovarja (vrstice) v instanco objekta.

Ob profiliranju akcije za poizvedovanje podatkov opazimo, da velik delež izvajalnega časa vzame 3. točka – torej pretvorba posameznega zapisa v instanco razreda.

Za hitrejšo izvedbo smo na nivoju programskega jezika Python pripravili podatkovno strukturo – pogled (ang. View). Ta struktura omogoča preprečevanje potrebe po tretjem koraku in tudi spreminja predstavitev posameznih vrstic iz strukture slovarja v terko (ang. Tuple). View, pa predstavlja ovoj, ki deluje na podobnem principu, kot delujejo podatkovni okvirji (ang. DataFrame) v knjižnici za podatkovno analitiko [9]. Struktura smo implementirali na nivoju programskega jezika C++, kar še dodatno pospeši čas dostopa do posameznega zapisa.

Optimizacija v številkah

Za izmeritev učinkovitosti implementacije, smo uporabili scenarij iz performančnih testov celotnega sistema (branje milijona vrstic in shranjevanje celotnega seznama v pomnilnik).

- V prvi implementaciji je povprečni čas izvajanja bil nekje ≈ 7 sekund, kjer je več kot 80% časa predstavljalo instanciranje objektov.
- Po optimizaciji, pa celoten čas branja zahteva povprečno ≈ 1.3 sekunde. Kar pomeni, kar 5x pohitritev izvedbe branja.

3.4 Podprte funkcionalnosti na nivoju vgrajenega ORM sistema

- **Filtriranje podatkov – funkcija "filter" (DBMS – where)**. Parametri te funkcije se drevo, kjer je posamezno vozlišče drevesa predstavljeno z enim ali več pogoji, ki so oviti v AND ali OR pogoj.
- **Omejitev števila zapisov + določanje odmikov – funkciji "limit" in "offset" (ekvivalentno, kot v DBMS)**. Gre za preprosta ukaza, ki sta namenjeni omejevanju števila zapisov, ko iščemo le prvih nekaj zapisov oz. prvih nekaj zapisov z določenim odmikom. Pogost primer uporabe je implementacija strani (ang. paging) na nivoju API-ja. Razlog za implementacijo je število optimizacija prikaza na način, da omejimo število zapisov, ki jih naenkrat prikažemo uporabniku. Optimizira se tako in sloj, pridobivanja podatkov na nivoju programskega jezika Python, pri prenosu na uporabniški vmesnik (najpogosteje gre tukaj za HTTP protokol), ter v zaključni fazi prikaza podatkov.
- **Urejanje zapisov – funkcija "order" (DMBS – order by)**. Gre za malo drugačno izvedbo, kot je to pri klasičnih relacijskih sistemih, saj v našem primeru podatkov ne združujemo v eno matrično strukturo, temveč so podatki združeni v drevesno strukturo, kjer povezave

realiziramo z dedovanjem. Na primer, če bi imeli sledečo poizvedbo na nivoju relacijske podatkovne baze:

```
SELECT * FROM users
INNER JOIN tasks ON tasks.id_user = users.id
ORDER BY users.name, tasks.content
```

Bi podatki bili urejeni po imenu uporabnika in nato še na istem nivoju po vsebini naloge. Ekvivalentna poizvedba na nivoju našega sistema bi izgleda tako:

```
query = User.link(
    tasks=Task.order(Task.content)
).order(User.name)
```

V tem bi dobili za vsakega uporabnika seznam njegovih nalog in bi ta seznam znotraj uporabnika bil urejen glede na vsebino naloge namesto, da ureditev po nazivu naloge vpliva na celotno matriko.

- **Avtomatsko združevanje podatkov iz različnih entitet tekom poizvedb – funkcija "link" (DBMS – različne verzije JOIN stavka).** Predstavimo algoritem združevanja zapisov v skupno strukturo
- **Agregacija oz. združevanje podatkov**
  COUNT, SUM, MIN, MAX
- **Množične akcije**
  bulkcreate, bulkdelete

Poglavje 4

Sodoben pristop razvoja programske opreme

V naslednjem poglavju bomo predstavili izbrane metodologije, ki so ključne za razvoj naše izbrane programske opreme. Jasno se zavedamo, da kakovostna zasnova predstavlja temelj celotnega sistema. Za zagotovitev pravilnega delovanja te zasnove pa se danes pogosto poslužujemo postopka avtomatskega testiranja programske opreme, ki vključuje teste enot in integracijsko testiranje. S skrbno izdelanimi testnimi primeri smo sposobni doseči zanesljivo in pravilno delovanje celotnega sistema.

4.1 Razvoj knjižnice za programski jezik Python

V svetu programiranja je razvoj knjižnic ključnega pomena za širjenje funkcionalnosti posameznega programskega jezika. Python je jezik, ki že sam po sebi vključuje zelo obsežno standardno knjižnico, ki pokriva ogromno funkcionalnosti, ki jih v ostalih programskih jezikih dobimo le z uporabo eksternih modulov/knjižnic.

V namen upravljanja z eksternimi moduli programskega jezika se pogosto srečamo z upravljavci paketov (ang. package manager). Gre za orodje,

ki omogoča enostaven način nameščanja, posodabljanja in odstranjevanja uvoženih knjižnic. Programski jezik Python v svoji dokumentaciji [12] predlaga uporabo PIP upravljalnik paketov za distribucijo knjižnic, ki niso del standardnega modula programskega jezika.

4.1.1 Uporaba Python.h API za razvoj knjižnice

Programski jezik Python je poznan predvsem po uporabi v področjih umetne inteligence, podatkovne analitike, strežniških aplikaciji in preprostih aplikaciji. Gre za interpretiran programski jezik, kar pomeni, da se koda ne prevede do nivoja, kot se to zgodi v primeru C++, kjer se koda prevede do strojnega jezika. Python ima za zagon kode, še dodaten nivo abstrakcije, ki dejansko izvaja kodo in jo sproti prevaja na nivo, ki je poznan računalniku.

Zaradi dodatnega nivoja abstrakcije je jezik sam po sebi bistveno počasnejši od pravih prevedenih jezikov. Poleg tega jezik tudi nima striktnih tipov, kar pomeni, da je za vsako spremenljivko v jeziku najprej potrebno preveriti, za kateri tip gre, kar ponovno prinese časovne zakasnitve.

Področji umetne inteligence in podatkovne analitike, pa temeljita na obdelavi masovnih podatkov. Kar pa posledično pomeni, da Python ni najbolj optimalen jezik za izvedbo teh nalog. Zaradi teh omejitev so nastale knjižnice, ki so v osnovi napisane v jezikih, ki so bistveno hitrejši. Tako se funkcije in strukture, ki jih potem lahko uporabljamo na nivoju programskega jezika Python prevedejo že v strojno kodo in jih potem lahko kličemo direktno iz programskega jezika Python. Med temi knjižnicami so pogosteje uporabljene:

- **Numpy [8]** – odprto kodna knjižnica, ki v Python prinese podatkovni tip polja (ang. array), in veliko vgrajenih funkciji, ki bistveno pospešijo operacije delo s polji, kot tudi matrikami, ki v področju podatkovne analitike in umetne inteligence predstavljajo enega izmed glavnih temeljev. Veliko je tudi knjižnic, ki za delovanje uporabljajo ravno Numpy. Tak primer je npr. Pandas [9] – gre za eno izmed najpogosteje uporabljenih knjižnic za podatkovno analitiko, ki s seboj prinese še nekaj

dodatnih abstrakciji za lažje delo s podatkovnimi okviri (ang. data frame).

- **Tensorflow [19]** – odprto kodna knjižnica za strojno učenje, v osnovi je celoten produkt Tensorflow namenjen širšemu spektru programskih jezikov in ni implementiran le za Python. Gre za splošno namensko knjižnico z že pripravljenimi metodami za kreiranje modelov za strojno učenje, kar bistveno olajša delo programerju, saj je velikosti postopkov avtomatiziranih.

- **UltraJSON [21]** – gre za manj poznano knjižnico, ki dela z JSON strukturami. Glavni funkcionalnosti knjižnice sta pretvorba Python strukture v JSON in obratno. JSON je v zadnjih letih postal glavni standard za prenos podatkov preko HTTP zahtevkov, posledično je vsaka optimizacija na nivoju obdelave JSON strukture zelo dobrodošla.

Vse te knjižnice so v osnovi napisane s pomočjo Python.h API vmesnika. Gre za vmesnik med programskima jezikoma Python in C. Na nivoju jezika C lahko vsak tip predstavimo z vgrajeno PyObject strukturo, katerega lahko neposredno uporabljamo na nivoju programskega jezika C.

4.1.2 Primer uporabe Python.h API vmesnika

Primer preproste funkcije na nivoju programskega jezika C, ki vrne seštevek dveh celih števil:

```c
#include <Python.h>

static PyObject* add_nums(PyObject* self, PyObject* args) {
    int64_t a, b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }

    int64_t sum = a + b;
    return PyLong_FromLong(sum);
}
```

Funkcija je sestavljena tako, da v prvi fazi argumente shrani na naslove spremenljivk, ki so deklarirane na nivoju programskega jezika C (to sta v zgornjem primeru $int64_t a, b;$). Argumenti so poslani kot terka, za katero moramo najprej povedati strukturo, v zgornjem primeru je to ii, kar pomeni dve celi števili. Seveda, lahko kot argumente beremo tudi ostale strukture vendar branje postane bistveno bolj kompleksno, ko želimo brati sestavljene tipe. V tem primeru je v večini primerov lažje preprosto pretvoriti vse v primitivne tipe in na nivoju programskega jezika C operirati direktno s temi.

4.1.3 Naprednejša knjižnica z dodatnimi strukturami za lažji prehod med jezikoma

Izdelava knjižnic s pomočjo Python.h API vmesnika pomeni uporabo programskega jezika C, kar pomeni, da je velik del implementaciji prepuščen razvijalcu, da si tukaj delo malo olajšamo smo se odločili za razvoj knjižnice s pomočjo C++ jezika in knjižnice, ki dela nivo višje, kot Python.h API -

PyBind11 [13]. Gre za knjižnico, ki bistveno olajša pretvorbe podatkovnih struktur med jezikoma npr. kadar želimo, kot parameter poslati strukturo seznama v C++ lahko to preprosto definiramo, kot tip vector in pretvorba iz seznama v vektor bo avtomatska. Knjižnica deluje s pomočjo C++ verzije 11, ki je izšla septembra 2011.

Poleg avtomatskih pretvorb, pa imamo že vseeno možnost uporabe vseh tipov, ki so vključeni v Python.h API vmesnik.

4.1.4 Primer uporabe PyBind11 knjižnice

Spodaj imamo še preprost primer vse potrebne kode na nivoju C++ za izdelavo preproste knjižnice, ki vsebuje funkcijo za seštevek dveh celih števil (main.cpp):

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

int64_t add_nums(int64_t a, int64_t b)
{
    return a + b;
}

PYBIND11_MODULE(my_math_module, m)
{
    m.def("custom_sum", &add_nums, "Add 2 ints");
}
```

Pod pogojem, da je knjižnica pravilno vključena v projekt, lahko iz programskega jezika Python uporabimo funkcijo čustom_sum", kot:

```
import my_math_module
total = my_math_module.custom_sum(5, 10)
print(total) # prints 15
```

4.2 Testno usmerjen razvoj

Da zagotovimo pravilno delovanje nekega bolj kompleksnega sistema je na področju razvoja programske opreme zelo priporočen testno usmerjen razvoj. Gre za način validacije pravilnega delovanja posameznih komponent sistema, kot ločeni segmenti in kot celotna enota. V primeru razvoja izbrane programske opreme smo proces testiranja razdelili na tri segmente – testiranje enot, integracijsko testiranje in performančno testiranje.

4.2.1 Testi enot na nivoju jedra podatkovnega sistema

V prvi fazi želimo, da zagotovimo pravilno delovanje na najnižjem nivoju oz. na jedru celotnega sistema. Gre za testiranje najbolj osnovnih funkciji, ki zagotavljajo pravilno delovanje našega sistema. V tem segmentu je bilo največ pozornosti predane na testiranje vnašanja in branja sistema za indeksiranje, ki je realizirano s pomočjo B+ dreves. Avtomatsko testiranje je na tem nivoju realizirano s pomočjo knjižnice Doctest [3]. Gre za testno ogrodje napisano za programski jezik C++, ki omogoča fleksibilen pristop validacije rezultatov iz posameznih funkciji.

4.2.2 Integracijsko testiranje funkcionalnosti na nivoju končne knjižnice

Naslednja faza testiranja je realizirana s pomočjo integracijskih testov. Gre za pristop testiranja, kjer preizkusimo ali prej posamezno testirane enote delujejo tudi na nivoju integracije ene z drugo [1]. Iz praktičnega vidika, če smo prej testirali ali pravilno delujejo posamezne funkcionalnosti B+ dreves, lahko zdaj testiramo ali se B+ drevo kreira pravilno na nivoju vnašanja podatkov v entiteto, kjer imamo določen atribut definiran kot indeksiran. Sama izvedba testiranja je izvedena na nivoju programskega jezika Python z vgrajeno knjižnico Unittest [22].

4.2.3 Performančno testiranje zahtevnejših akciji

Izvedba tretje faze je sicer zelo podobna fazi integracijskega testiranja z izjemo, da ne posvetimo toliko pozornosti sami pravilnosti rezultatov in ne testiramo toliko različnih scenarijev in robnih pogojev, temveč je pozornost usmerjena predvsem v časovne meritve – torej koliko časa porabi posamezen scenarij za izvedbo.

Pripravljenih imamo 10 performančnih testov, kjer je vsak test ločen scenarij, ki meri čas izvajanja v posameznem scenariju (sak izmed teh testov se izvede petkrat, kjer ob koncu izračunamo povprečni čas izvajanja). Spodaj imamo še primer izpisa izvedbe performančne analize:

```
Avg: 11.315 - Bulk delete 1M codelist records
Avg: 1.373 - Create 100K basic codelist records
Avg: 0.712 - Bulk create 100K basic codelist records
Avg: 0.600 - Create 10K records and read them by IDs
Avg: 1.372 - Read 1M records at once
Avg: 0.010 - Read multilevel query and searilize data
Avg: 0.144 - Read with limit 1500 + order by country from Model
↪  with 1M records
Avg: 0.011 - Read with limit 100 from Model with 1M records
Avg: 0.092 - Read with limit 100 + order by country from Model
↪  with 1M records
Avg: 1.599 - Read with order by country from Model with 1M
↪  records
----------------------------------------------------------------------
Ran 10 tests in 254.074s

OK
[graphenix] Finished tests!
```

4.2.4 Integracija avtomatskega testiranja z GitHub

Platforma GitHub omogoča izvajanje različnih akciji ob različnih dogodkih najpogosteje v praksi to počnemo, kadar izvajamo združenje vej ali pa kar ob vsaki posodobitvi kode. V praksi se poleg avtomatskega testiranja na tem področju lahko izvajajo tudi avtomatske posodobitve strežniške aplikacije ali različne periodične naloge, kot npr. kreiranje rezervnih verziji podatkovne baze iz produkcijskega strežnika.

V našem primeru se ob vsaki posodobitvi kode izvede akcija za preverjanje pravilnega delovanja aplikacije. Akcije se izvedejo za dve verziji programskega jezika Python (v3.10 in v3.11), postavitev pa se izvede na virtualnem Linux Ubuntu okolju.

Ko je okolje pravilno postavljeno se akcije izvedejo v naslednjem zaporedju:

1. Zagon virtualnega okolja za Python programskega okolja (venv [15]).
2. Postavitev potrebnih knjižnic za delovanje jedra (PyBind11 [13]).
3. Namestitev jedra podatkovnega sistema in knjižnice na nivoju programskega jezika Python.
4. Poskus uvoza jedra podatkovnega sistema v CLI, ki je vgrajen v programski jezik Python.
5. Poskus uvoza celotne knjižnice
6. Izvedba nizko nivojskih testov oz. testov enot v programskem jeziku C++
7. Izvedba integracijskih testov na nivoju programskega jezika Python
8. Izvedba performančnih testov

Takoj, ko se katera izmed akciji ne izvede pravilno se ustavi celoten cevovod izvajanja in se javi napaka, ki je vidna na zavihku GitHub akciji.

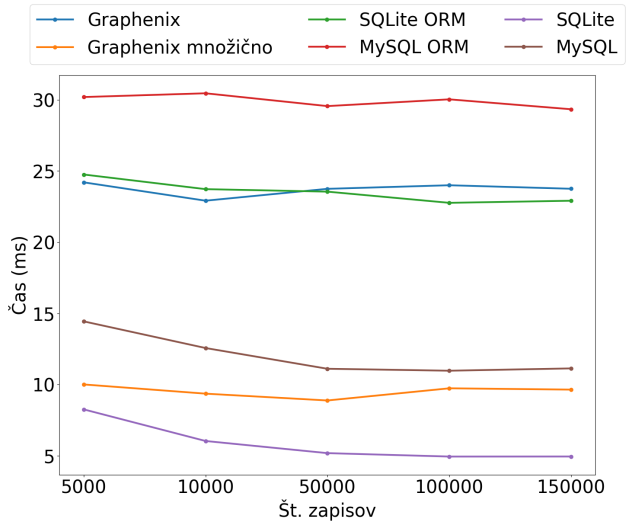
Pristop avtomatskega testiranja in posodabljanja imenujemo tudi CI/CD, ki temelji na hitrem in varnem razvoju, ki razvijalce hitro obvesti o morebitnih napakah, ki se pojavljajo znotraj aplikacije in omogoča, tako avtomatsko testiranje kot tudi avtomatsko posodobitev produkcijskih verziji aplikacije.

Poglavje 5

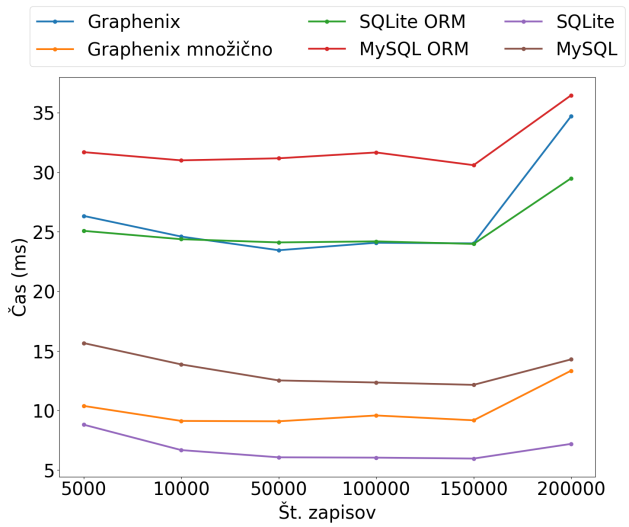
Analiza

V tem poglavju se bomo posvetili temeljni analizi, ki vključuje časovno in prostorsko primerjavo. Osredotočili se bomo na primerjavo med našo razvito knjižnico Graphenix ter obstoječima DBMS-jema SQLite in MySQL. Analizo bomo izvedli na različnih scenarijih, pri čemer se bomo predvsem osredotočili na branje podatkov. Poleg tega bomo proučili vpliv uporabe ORM in izvedli teste tako z uporabo ORM kot tudi brez njega. Na ta način bomo pridobili vpogled v zmogljivost, učinkovitost ter prednosti ter slabosti posameznih rešitev v različnih kontekstih.

5.1 Množično vnašanje podatkov

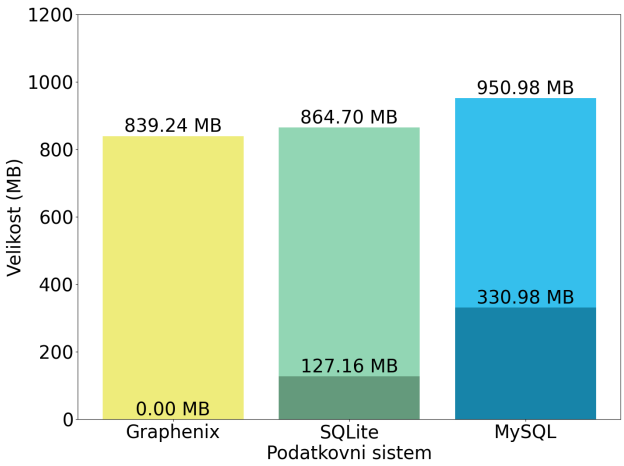


Slika 5.1: Normaliziran čas vnašanja podatkov



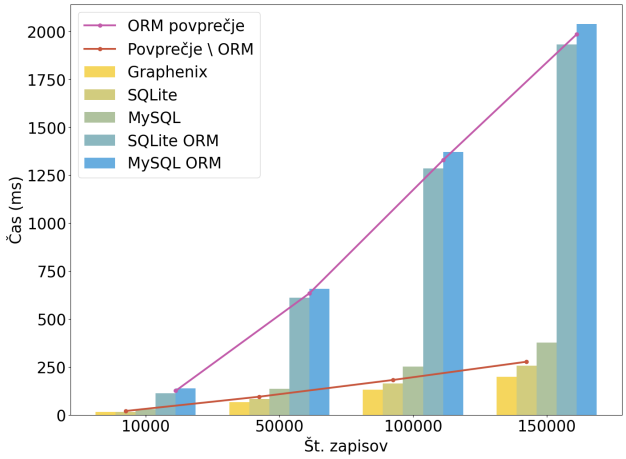
Slika 5.2: Normaliziran čas vnašanja podatkov z dodatnim vnašanjem v B+ strukturo

5.2 Velikosti podatkovnih baz na disku

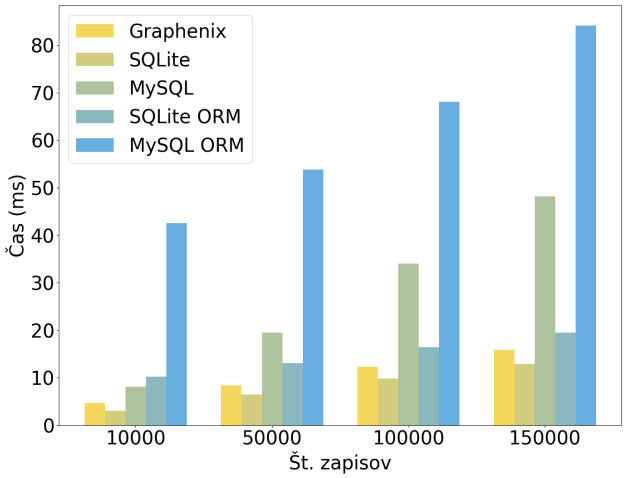


Slika 5.3: Prostorska poraba na nivoju diska posameznega DBMS

5.3 Primerjava poizvedb na eni entiteti



Slika 5.4: Branje brez dodatnih parametrov znotraj poizvedbe

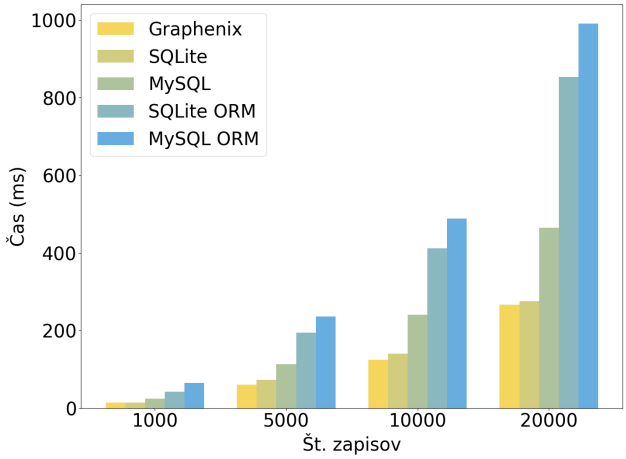


Slika 5.5: Branje z dodanim filtriranjem in urejanjem zapisov

5.4 Izvedba agregacije nad zapisi

5.5 Pohitritev poizvedb s pomočjo indeksiranja

5.6 Primerjava poizvedb z uporabo relaciji



Slika 5.6: Poizvedovanje podatkov iz dveh entitet hkrati – uporabniki in njihove naloge

Poglavje 6

Scenariji oz. primeri uporabe

6.1 Strežniško beleženje podatkov

Pogosta uporaba programskega jezika Python je na strani strežniških aplikaciji. Tukaj je beleženje vseh zapisov zelo pomembno torej vsakega zahtevka, časovne točke in odgovora na zahtevek. Z uporabo pripravljene knjižnice je prilagojeno beleženje podatkov lahko zelo preprosto.

V prvi fazi si lahko pripravimo preprosto strežniško aplikacijo, ki ima vsega eno točko za pridobivanje podatkov in sicer "get-range", kjer lahko sami definiramo 3 argumente, to so "start", "end" in "step", kjer ima vsak od teh argumentov tudi privzete vrednosti.

6.1.1 Aplikacija, ki služi kot osnova za modul beleženja

```
app = Flask(__name__)
@app.route('/get-range')
def get_range(request):
    start = int(request.args.get('start', 0))
    end = int(request.args.get('end', 10))
    step = int(request.args.get('step', 1))
    return jsonify({'Range': list(range(start, end, step))})
app.run()
```

Zelo preprost pristop dodajanja beleženja je z uporabo vzorca dekoratorjev, ki omogočajo dodajanje funkcionalnosti na izbrane vhodne točke naše strežniške aplikacije. V zgornjem primeru že imamo primer dekoratorja in sicer @*app.route*(...), ki funkcijo označi, kot vhodno točko za našo strežniško aplikacijo. Za naš primer bomo na vsako vhodno točko strežniške aplikacije dodali dekorator, kjer se za vsakim odgovorom zapiše HTTP pot, trenutni čas, parametri v zahtevku, odgovor in status le-tega. V prvi fazi rabimo najprej kreirati shemo za shranjevanje podatkov, ter potrebno entiteto, kjer bomo držali podatke o zahtevkih.

6.1.2 Kreiranje sheme za beleženje podatkov

```
class ReqInfo(Model):
    route = Field.String(size=255)
    timestamp = Field.DateTime().as_index()
    route_req = Field.String(size=1024)
    route_res = Field.String(size=1024)
    resp_code = Field.Int()

logging = Schema('logging', models=[ReqInfo])
if not logging.exists():
    logging.create()
```

V zgornjem segmentu pripravimo razred, ki predstavlja entiteto s posameznimi atributi, ki predstavljajo podatke, katere bomo tekom delovanja naše aplikacije beležili. V drugi fazi, pa moramo kreirati celotno shemo, kjer definiramo naziv celotne sheme in pa vse razrede oz. entitete, ki bodo uporabljene znotraj naše sheme.

6.1.3 Dekorator za dinamično dodajanje beleženja

```
def route_with_log(route):
    def decorator(f):
        @wraps(f)
        def wrapper(*args, **kwargs):
            try:
                response = f(request, *args, **kwargs)
            except Exception as err:
                response = None

            ReqInfo(
                route=route,
                timestamp=datetime.now(),
                route_req=json.dumps(dict(request.args)),
                route_res=response.get_data(as_text=True) if response else ''
                resp_code=response.status_code if response else 500
            ).make()

            return response

        app.add_url_rule(route, view_func=wrapper)
        return wrapper
    return decorator
```

Dekorator je v vzorec, kjer funkcija vrača drugo funkcijo in predstavlja nek ovoj funkcije. Tekom izvedbe ovite funkcije lahko argumente in različne segmente izvajanja prestrežemo in jih prilagodimo glede na naše zahteve. Pogosta uporaba vzorca, je tudi ko želimo na različne akcije dodati različne zahteve npr. strežniški vhodni točki želimo dodati preverjanje, če je uporabnik prijavljen z uporabo žetona in v primeru, ko uporabnik ni prijavljen v funkcijo vrinemo funkcionalnost, ki zavrne zahtevo uporabnika in vrne status 401 (ne prijavljen uporabnik). V primeru našega dekoratorja, pa lahko dekora-

tor @*app.route*(...) preprosto zamenjamo z *route_with_log*(...) in funkcija bo ohranila tip vhodne točke, poleg tega pa dobi tudi funkcionalnost beleženja podatkov.

6.1.4 Branje zapisanih podatkov

V končni fazi je glavna prednost uporabe pripravljene knjižnice za beleženje podatkov struktura shranjenih podatkov. Namreč z uporabo DBMS dosežemo fiksno strukturo, ki je v primeru klasičnih datotek za beleženje nimamo. Posledično nam je omogočeno tudi bistveno več načinov pridobivanja statističnih podatkov, saj lahko podatke poljubno filtriramo in razvrščamo. Na nivoju administratorske aplikacije lahko nato prožimo različne poizvedbe nad našo relacijsko podatkovno bazo, ki podatke shranjuje znotraj *ReqInfo* entitete.

Izpis zadnjih treh zahtevkov

```
# izpis zadnjih treh zahtevkov
_, reqs = ReqInfo.order(ReqInfo.timestamp.desc()).limit(3).all()
```

```
ReqInfo(id=3, route=/get-range, timestamp=2023-07-16 03:02:15,
    route_req={}, route_res={"Range":[0,1,2,3,4,5,6,7,8,9]},
    resp_code=200)

ReqInfo(id=2, route=/get-range, timestamp=2023-07-16 03:02:13,
    route_req={"start": "3"}, route_res={"Range":[3,4,5,6,7,8,9]},
    resp_code=200)

ReqInfo(id=1, route=/get-range, timestamp=2023-07-16 03:02:05,
    route_req={"start": "15", "end": "10", "step": "-3"},
    route_res={"Range":[15,12]}, resp_code=200)
```

Statističen pregled zahtevkov

Podatke lahko v pripravljenem DBMS tudi grupiramo, in sicer z uporabo *.agg(...)* metode v poizvedbi, lahko definiramo polje po katerem grupiramo podatke, kot tudi agregacije, ki jih želimo izvesti.

```
route_stats = ReqInfo\
    .agg(by=ReqInfo.route, count=AGG.count())
```

Izpis napak v zadnjem dnevu

Za nabor napak v zadnjem dnevu, lahko preprosto izvedemo poizvedbo, kjer zahtevamo, da je status odgovora >= 400, kar pri HTTP zahtevkih predstavlja razpon napak. Poleg tega, pa dodamo še zahtevo, da mora biti datum napake večji, kot včerajšnji datum ob enaki uri. Za konec zapise uredimo po datumu napake.

```
count, api_errors = ReqInfo\
    .filter(
        ReqInfo.resp_code >= 400,
        ReqInfo.timestamp > datetime.now() - timedelta(days=1)
    )\
    .order(ReqInfo.timestamp.desc())\
    .all()
```

Izvoz vseh podatkov v .csv datoteko

DBMS omogoča tudi izvoz v csv datoteko, ki za nekatere uporabnike lahko pomeni lažjo obdelavo podatkov, poleg tega, pa lahko predstavlja tudi varnostno kopijo podatkovne baze v ločeni datoteki.

Za omenjeno nalogo potrebujemo najprej izvesti nabor vseh podatkov, nato pa s pomočjo privzetega pretvornika podatke izvozi v datoteko s pomočjo metode "dump2csv", ki prejme dva parametra – podatke in naziv izvozne datoteke.

```
_, all_reqs = ReqInfo.all()
ViewSearilizer.default().dump2csv(all_reqs, 'export.csv')
```

6.2 Manjši strežniški API

Strežniške aplikacije, ki zahtevajo fleksibilno in hitro delo s podatkovno bazo postajajo vse bolj pogoste. Python predstavlja enega izmed najpogosteje uporabljenih programskih jezikov na omenjenem področju. Obstaja tudi že ogromno pripravljenih knjižnic za izdelavo API aplikaciji, kot tudi za uporabljanje s podatkovno bazo, ki se uporablja znotraj aplikacije.

Tekom poglavja je predstavljena izdelava strežniške aplikacije, kjer uporabljamo Flask [5] ogrodje. Sama aplikacija, pa je namenjena delu za preprosto uporabljanje sistema profesorjev in laboratorijev.

6.2.1 Podatkovna shema za aplikacijo

V shemi imamo vsega dve tabeli, ki bosta držali podatke o profesorjih in laboratorijih.

```
class Teacher(Model):
    full_name = Field.String(size=256)
    email = Field.String(size=128)
    laboratory = Field.Link()

class Laboratory(Model):
    name = Field.String(size=64)
    room_number = Field.Int()
    teachers = Field.VirtualLink('laboratory')
```

6.2.2 Pretvornik za serializacijo podatkov nad shemo

```
class TeacherSearilizer(ViewSearilizer):
    fields = ('id', 'full_name', 'email')

class LaboratorySearilizer(ViewSearilizer):
    fields = '*'
    teachers = TeacherSearilizer

class BasicLaboratorySearilizer(ViewSearilizer):
    fields = ('id', 'name', 'room_number')

class DetailTeacherSearilizer(ViewSearilizer):
    fields = '*'
    laboratory = BasicLaboratorySearilizer
```

6.2.3 Kreiranje novega zapisa

6.2.4 Nabor podatkov profesorjev po laboratorijih

6.2.5 Nabor podatkov z dinamičnim filtriranjem

Poglavje 7

Sklepne ugotovitve

Splošne ugotovitve tekom izdelave vgrajenega podatkovnega sistema in strnjene ugotovitve glede na analizo

Literatura

[1] Hanmeet Kaur Brar in Puneet Jai Kaur. "Differentiating integration testing and unit testing". V: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE. 2015, str. 796–798.

[2] *Django*. 2023. URL: `https://github.com/django/django` (pridobljeno 20. 7. 2023).

[3] *Doctest*. 2023. URL: `https://github.com/doctest/doctest` (pridobljeno 2. 7. 2023).

[4] *Epoch format*. URL: `https://www.maketecheasier.com/what-is-epoch-time/` (pridobljeno 2. 7. 2023).

[5] *Flask*. 2023. URL: `https://github.com/pallets/flask` (pridobljeno 30. 7. 2023).

[6] *Relational Database*. URL: `https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/` (pridobljeno 29. 6. 2023).

[7] William Kahan. "IEEE standard 754 for binary floating-point arithmetic". V: *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996), str. 11.

[8] *Numpy*. 2023. URL: `https://github.com/numpy/numpy` (pridobljeno 15. 7. 2023).

[9] *Pandas*. 2023. URL: https://github.com/pandas-dev/pandas (pridobljeno 15. 7. 2023).

[10] *Peewee*. 2023. URL: https://github.com/coleifer/peewee (pridobljeno 20. 7. 2023).

[11] *C++ std::priority_queue*. URL: https://en.cppreference.com/w/cpp/container/priority_queue (pridobljeno 22. 7. 2023).

[12] *Python docs - Installing Packages*. URL: https://packaging.python.org/en/latest/tutorials/installing-packages/ (pridobljeno 9. 8. 2023).

[13] *PyBind11*. 2023. URL: https://github.com/pybind/pybind11 (pridobljeno 15. 7. 2023).

[14] *PyPI Stats*. URL: https://pypistats.org/ (pridobljeno 20. 7. 2023).

[15] *Python - venv*. URL: https://docs.python.org/3/library/venv.html (pridobljeno 22. 7. 2023).

[16] Janez Sedeljšak. *Graphenix*. 2023. URL: https://github.com/JanezSedeljsak/graphenix (pridobljeno 2. 7. 2023).

[17] *SQLAlchemy*. 2023. URL: https://github.com/sqlalchemy/sqlalchemy (pridobljeno 20. 7. 2023).

[18] *SQLObject*. 2023. URL: https://github.com/sqlobject/sqlobject (pridobljeno 20. 7. 2023).

[19] *TensorFlow*. 2023. URL: https://github.com/tensorflow/tensorflow (pridobljeno 15. 7. 2023).

[20] *Tortoise ORM*. 2023. URL: https://github.com/tortoise/tortoise-orm (pridobljeno 20. 7. 2023).

[21] *UltraJSON*. 2023. URL: https://github.com/ultrajson/ultrajson (pridobljeno 15. 7. 2023).

[22] *unittest — Unit testing framework*. URL: https://docs.python.org/3/library/unittest.html (pridobljeno 20. 7. 2023).

[23] *C++ std::vector*. URL: https://cplusplus.com/reference/vector/vector/ (pridobljeno 20. 7. 2023).

[24] Aljaž Zrnec in sod. "Podatkovne baze nosql". V: (2011).