

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Janez Sedeljšak

Razvoj SUPB z integracijo v programski jezik Python

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Boštjan Slivnik
SOMENTOR: asist. dr. Marko Poženel

Ljubljana, 2023

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.



Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani <http://www.gnu.org/licenses/>.

Besedilo je oblikovano z urejevalnikom besedil L^AT_EX.

Kandidat: Janez Sedeljšak

Naslov: Razvoj SUPB z integracijo v programski jezik Python

Vrsta naloge: Diplomaska naloga na visokošolskem programu prve stopnje
Računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Somentor: asist. dr. Marko Požnenel

Opis:

Razvijte ultra lahek relacijski SUPB, ki ga je možno uporabljati v enostavnih Python aplikacijah, ki tečejo na sistemih z zelo majhnimi pomnilniškimi in diskovnimi viri. Zaradi učinkovitosti implementirajte osnovo SUPB v programskem jeziku C++.

Title: Development of a DBMS with integration into the Python programming language

Description:

Develop an ultra-lightweight relational SUPB that can be used in simple Python applications running on systems with very limited memory and disk resources. For efficiency, implement the core of SUPB in the C++ programming language.

Zahvaljujem se svojemu mentorju doc. dr. Boštjanu Slivniku in somentorju asist. dr. Marku Poženelu za vso strokovno pomoč in svetovanje med implementacijo rešitve ter pisanju diplomskega dela.

Prav tako se zahvaljujem svoji družini in prijateljem za pomoč, podporo in obilico dobre volje tekom študija. Hvala vsem!

Kazalo

Povzetek

Abstract

1	Uvod	1
1.1	Paradigme v svetu podatkovnih baz	1
1.1.1	Relacijske podatkovne baze	1
1.1.2	Nerelacijske podatkovne baze	2
1.2	Področja uporabe relacijskih podatkovnih baz	3
1.2.1	Vloga podatkovne baze v informacijskem sistemu . . .	3
1.3	Motivacija za razvoj lastnega SUPB in ORM vmesnika	4
2	Sorodna dela	7
2.1	Izbor SUPB	7
2.2	Statističen pregled Python paketov ORM	8
2.2.1	Slabost uporabe paketov ORM	10
3	Razvoj SUPB in mehanizma za shranjevanje	13
3.1	Arhitektura SUPB	13
3.2	Shema podatkov za predstavitev funkcionalnosti	15
3.3	Mehanizem shranjevanja podatkov	16
3.3.1	Matrika podatkov in zaglavje s kazalci	16
3.3.2	Tipi podatkov	17
3.3.3	Implementacija relacij	18

3.4	Indeksiranje z uporabo B+ dreves	19
3.4.1	Iskanje v B+ drevesu	21
3.4.2	Vstavljanje v B+ drevo	21
3.4.3	Brisanje iz B+ drevesa	22
3.4.4	Posodabljanje v B+ drevesu	23
3.4.5	Implementacija za shranjevanje na disk	23
3.4.6	Implementacija v InnoDB	24
3.4.7	Podprte operacije filtriranja	25
3.4.8	Ostali pristopi indeksiranja	26
3.5	Optimizacije poizvedovanja	27
3.5.1	Gručanje podatkov ob branju iz diska	27
3.5.2	Uporaba prioritete vrste za urejanje zapisov	28
3.5.3	Uporaba podatkovnih okvirjev	30
3.6	Poizvedovanje z uporabo ORM	31
4	Pristop razvoja programske opreme	39
4.1	Razvoj paketa za programski jezik	
	Python	39
4.1.1	Uporaba Python.h za razvoj paketa	40
4.1.2	Naprednejša rešitev za lažji prehod med jezikoma	41
4.2	Arhitektura rešitve Graphenix	43
4.3	Testno usmerjen razvoj	44
4.3.1	Testi enot na nivoju SUPB in mehanizma za shranjevanje	44
4.3.2	Integracijsko testiranje funkcionalnosti	44
4.3.3	Performančno testiranje zahtevnejših akcij	45
4.3.4	Integracija avtomatskega testiranja z GitHub	45
5	Analiza delovanja	47
5.1	Množično vstavljanje podatkov	47
5.1.1	Vstavljanje z dodatnim indeksiranim poljem	49
5.2	Velikosti podatkovnih baz na disku	50
5.3	Primerjava poizvedb na eni tabeli	51

5.3.1	Branje z omejitvami	52
5.4	Primerjava poizvedb z uporabo relacij	53
5.5	Izvedba agregacijske poizvedbe	54
5.6	Pohitritve s pomočjo indeksiranja	55
5.6.1	Optimizacija v metodi filtriranja zapisov	56
5.6.2	Analiza iskanja nad večjo bazo podatkov	57
6	Scenariji oz. primeri uporabe	59
6.1	Strežniško beleženje podatkov	59
6.1.1	Modul beleženja	59
6.1.2	Kreiranje sheme za beleženje podatkov	60
6.1.3	Dekorator za dinamično dodajanje beleženja	61
6.1.4	Branje zapisanih podatkov	62
6.2	Namizna aplikacija za upravljanje stroškov	65
6.2.1	Zaslonski pregled aplikacije	65
6.2.2	Aplikacija z vidika kode	66
7	Sklepne ugotovitve	71
7.1	Nadaljnji razvoj	71
7.2	Refleksija razvoja	72
	Literatura	75

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	application programming interface	aplikacijski programski vmesnik
CI/CD	continuous integration, continuous delivery	neprekinjena integracija in postavitve
CLI	command-line interface	vmesnik za ukazno vrstico
CRUD	create, read, update, delete	ustvarjanje, branje, posodabljanje in brisanje
CSV	comma-separated values	vrednosti, ločene z vejico
GNU	General public license	splošna javna licenca
HTTP	hypertext transfer protocol	protokol za prenos hiperteksta
I/O	input/output operations	vhodno/izhodne operacije
JSON	JavaScript object notation	objektna notacija za JavaScript
NoSQL	nonrelational databases	nerelacijske podatkovne baze
ORM	object-relational mapping	objektno relacijska preslikava
SQL	structured query language	strukturiran jezik poizvedb
SUPB	database management system	sistem za upravljanje podatkovnih baz
TCP/IP	Internet protocol, transmission control protocol	internetni protokol in protokol za nadzor prenosa
UTC	coordinated universal time	univerzalni koordinirani čas

Povzetek

Naslov: Razvoj SUPB z integracijo v programski jezik Python

Avtor: Janez Sedeljšak

V diplomskem delu so predstavljeni nekateri izmed najbolj uveljavljenih sistemov za upravljanje podatkovnih baz (SUPB). Kljub napredku na tem področju ostajajo relacijske podatkovne baze v veliki meri osrednja paradigma. V okviru diplomskega dela smo se osredotočili na razvoj in predstavitev lastnega relacijskega SUPB za programski jezik Python. Razvoj namenske rešitve je realiziran v programskem jeziku C++. Gre za nizkonivojski jezik, kjer imamo visoko fleksibilnost pri upravljanju s pomnilnikom. Predstavljen je razvoj vseh potrebnih segmentov za dobro delujočo relacijsko podatkovno bazo. Ključnega pomena tekom razvoja je bila uporaba podatkovnih struktur in algoritmov, ki dobro izkoristijo vhodno/izhodne operacije in omogočajo zanesljivo in optimalno delovanja podatkovne baze. V zadnjem delu diplomskega dela smo izvedli primerjalno analizo našega razvitega SUPB-ja z že uveljavljenima rešitvama SQLite in MySQL, ter predstavili dosežene rezultate.

Ključne besede: Podatkovne baze, C++, Python, B+ drevesa, SQL, SUPB, ORM.

Abstract

Title: Development of a DBMS with integration into the Python programming language

Author: Janez Sedeljšak

The thesis introduces some of the most established Database Management Systems (DBMS) available. Despite advancements in the field, relational databases remain a prominent paradigm. Consequently, our focus within the study was directed towards developing and presenting our dedicated DBMS solution tailored for the Python programming language. The development of this specialized solution was implemented using the C++ programming language due to its low-level nature, offering high flexibility in memory management. The development encompasses all the necessary components for a well-functioning relational database. A crucial aspect during development was the utilization of data structures and algorithms that efficiently leverage the input/output operations provided. This approach ultimately ensures reliable and optimal database performance. In the final section of the thesis, we conducted a comparative analysis of our developed DBMS with the established solutions SQLite and MySQL, presenting the achieved results.

Keywords: Databases, C++, Python, B+ trees, SQL, DBMS, ORM.

Poglavje 1

Uvod

Živimo v dobi, v kateri se spopadamo z izzivom obdelave velikih količin podatkov, ki jih poznamo kot velepodatke (angl. *big data*) in predstavljajo dragocen vir informacij.

Za trajno shranjevanje podatkov uporabljamo podatkovne baze. Glede na organizacijo podatkov, podatkovne baze v grobem delimo v dve skupini – relacijske (angl. *relational databases*) in nerelacijske podatkovne baze (angl. *non-relational or NoSQL databases*).

1.1 Paradigme v svetu podatkovnih baz

1.1.1 Relacijske podatkovne baze

Relacijske podatkovne baze temeljijo na strogi strukturi podatkov, ki opredeljuje tabele in attribute. Posebej pomembne so logične povezave med posameznimi zapisi, ki omogočajo boljšo organizacijo in interpretacijo podatkov. Te povezave se vzpostavijo z uporabo koncepta tujih ključev, ki na nivoju tabele predstavlja atribut, ki drži vrednost primarnega ključa vezanega zapisa [8].

Kaj je SUPB in kakšna je njegova vloga v bazi podatkov?

Kaj hitro, ko začnemo raziskovati interno delovanje relacijskih podatkovnih baz, naletimo na pojem sistem za uporabljanje podatkovnih baz – SUPB (angl. *Database Management System*). SUPB je programska rešitev za upravljanje in organiziranje podatkov. Uporabnikom omogoča ustvarjanje, spreminjanje in poizvedovanje po bazi podatkov [8].

SUPB predstavlja vmesnik med uporabnikom, ki dela s podatkovno bazo (angl. *Data store*), in mehanizmom za shranjevanje podatkov (angl. *storage engine*). Mehanizem za shranjevanje podatkov je programska rešitev, ki skrbi za strukturo shranjenih podatkov. MySQL nudi dva primarna mehanizma: MyISAM in InnoDB (privzet od različice v5.5). MySQL med drugim ponuja mehanizem za shranjevanje podatkov v datoteke .csv, kar omogoča vpogled v podatke s standardnimi orodji za obdelavo podatkov (npr. Excel). Pomanjkljivost tega pristopa je omejen nabor funkcij SUPB [32].

Začetki relacijskih podatkovnih baz

Gre za paradigmo podatkovnih baz, ki se je prvič pojavila leta 1970, ko je model za shranjevanje predstavil Edgar F. Codd – matematik izobražen na univerzi Oxford [14]. Prva družina relacijskih podatkovnih baz pa je bila razvita leta 1983 – Db2 [16] v podjetju IBM. IBM je Db2 vseskozi posodabljal. Kljub temu je ta konceptualno ostal nespremenjen in še vedno uporablja zasnovno relacijskega modela iz leta 1970.

1.1.2 Nerelacijske podatkovne baze

Nerelacijske podatkovne baze predstavljajo novo kategorijo podatkovnih baz, ki za razliko od relacijskih podatkovnih baz, uporabljajo model shranjevanja, ki je prilagojen specifičnim zahtevam shranjevanih podatkov (npr. dokumenti, grafi, ključ-vrednost, ipd). Ta pristop prinaša tudi eno glavnih prednosti nerelacijskih podatkovnih baz, to je sposobnost enkapsulacije posameznih zapisov. Ti omogočajo učinkovito razporeditev celotne podatkovne

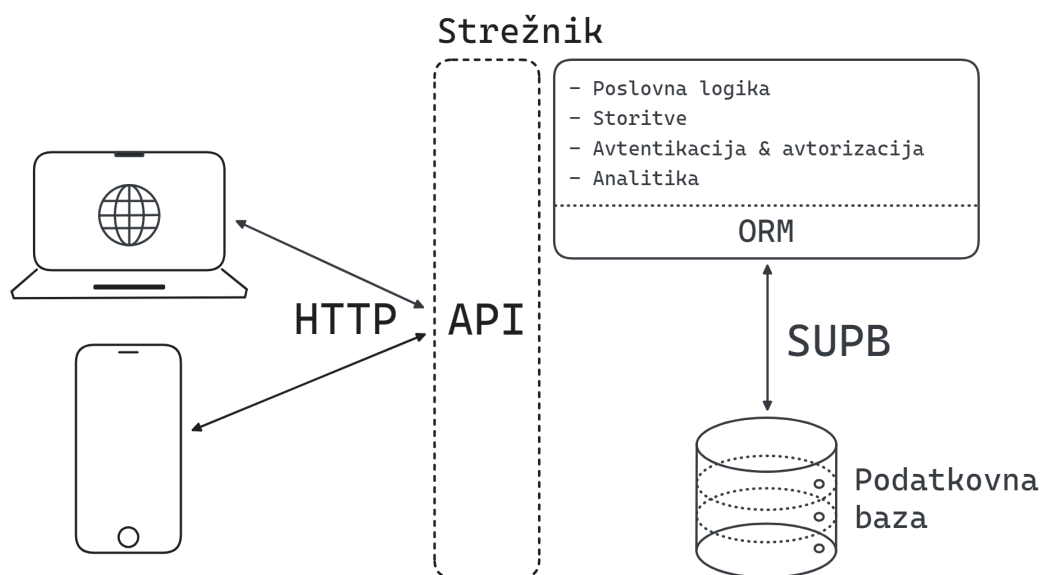
baze na več strežnikov (tj. horizontalno skaliranje podatkov), kar pri relacijskih podatkovnih bazah težje izvedemo [37].

1.2 Področja uporabe relacijskih podatkovnih baz

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 posodobljenih konceptov se relacijske podatkovne baze uporabljajo v številnih informacijskih sistemih, kjer nudijo zanesljivost, možnost zahtevnejših analiz podatkov, ipd.

1.2.1 Vloga podatkovne baze v informacijskem sistemu

Na sliki 1.1 je predstavljena standardna arhitektura za večino modernih informacijskih sistemov.



Slika 1.1: Arhitektura informacijskega sistema.

Arhitekturo sestavljajo tri ključne komponente. Na levi strani slike so odje-

malci (mobilne, spletne aplikacije, itd). S strežniškim delom sistema komunicirajo s pomočjo protokola HTTP.

Večji del informacijskega sistema se nahaja v uporabniku skritem delu, kjer je realizirana poslovna logika in upravljanje s podatki oz. bazo podatkov, s katero upravlja SUPB. V sodobnih informacijskih sistemih za dodaten nivo varnosti in lažjo manipulacijo podatkov pogosto uporabimo še objektno relacijsko preslikavo – ORM (angl. *Object-relational mapping*) [48]. Gre za programsko rešitev, ki s pomočjo preslikave tabel omogoča lažje delo s podatkovno bazo. Tabele se znotraj ORM preslikajo v razrede, zapisi znotraj tabele pa so predstavljeni kot instance teh razredov. ORM predstavlja dodaten nivo abstrakcije, ki programerju omogoči lažje upravljanje s podatkovno bazo znotraj uporabljenega programskega jezika.

1.3 Motivacija za razvoj lastnega SUPB in ORM vmesnika

Glavna motivacija za razvoj lastnega SUPB je boljše razumevanje delovanja podatkovnih baz, ter njihovega vpliva na delovanje modernih aplikacij. Programerji pogosto več pozornosti posvečajo implementaciji novih funkcij sistema, zanemarjajo pa vidike in vzorce, ki jih uporabljajo za razvoj podatkovne baze [21].

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. Eden izmed ciljev našega dela je spoznati v abstrakciji skrite koncepte, ki jih prinašata SUPB in jezik SQL, ki programerju predstavlja način komuniciranja z bazo podatkov.

Pomembnost podatkovnega sloja, ki vključuje podatkovno bazo in del aplikacije, ki je odgovoren za delo s podatki, postane očitna predvsem, ko se aplikacija začne odzivati počasneje. Pogost razlog je obsežna količina podatkov v posamezni tabeli, kar za uporabnika hitro pomeni nesprejemljiv čas odzivnosti celotnega sistema.

Za izboljšanje hitrosti poizvedb, lahko med drugim uporabimo mehanizem indeksiranja. Mehanizem ni omejen zgolj na relacijske podatkovne baze, temveč se mehanizem z uporabo iskalnih struktur pojavi še na ostalih področjih računalništva. Osnovna ideja za optimizacijo je dobra organizacija podatkov s pomočjo iskalne strukture. Dve bolj pogosti podatkovni strukturi za izvedbo indeksiranja sta zgoščevalne tabele in drevesa. Na področju relacijskih podatkovnih baz se najpogosteje uporablja večnivojsko indeksiranje, ki običajno uporablja B drevo. B drevo je sestavljeno iz vozlišč, kjer ima vsako vozlišče največ $b - 1$ zapisov in b kazalcev na naslednja vozlišča. S pomočjo dodajanja indeksov lahko linearno iskanje, ki ima časovno zahtevnost $O(N)$, kjer je N število vseh zapisov, bistveno pohitrilo. Z uporabo B drevesa dosežemo časovno zahtevnost iskanja $O(\log_b(N))$, kjer je N število vseh zapisov v drevesu.

Poudarimo, da cilj implementacije lastne rešitve ne vključuje razvoja dovršenega SUPB, ki bi se primerjal z naprednimi in uveljavljenimi rešitvami. Naš cilj je razviti SUPB, ki bo vseboval temeljne funkcionalnosti in bo primeren za uporabo v manjših aplikacijah. Osredotočili se bomo predvsem na izgradnjo preprostega in intuitivnega načina za upravljanje podatkov.

Poglavje 2

Sorodna dela

Prve podatkovne baze so bile razvite pred več kot štiridesetimi leti, ob tem pa tudi ogromno različnih pristopov za komunikacijo med programskimi jeziki in SUPB [4]. V računalništvu je izbor orodja ključnega pomena in izbor tehnologij znotraj podatkovnega sloja nikakor ne predstavlja izjeme.

2.1 Izbor SUPB

Pri izbiri SUPB za rešitve, ki upravljajo večje količine podatkov, imamo na voljo različne možnosti, med katerimi izstopajo: Db2 [16], Oracle [15] in Microsoft SQL Server (MSSQL) [26], ki predstavljajo največje in najbolj skalabilne SUPB-je. Kljub temu je izbor SUPB-ja, podobno kot pri večini področij v računalništvu, odvisen od zahtev naše aplikacije in narave podatkov, ki jih bomo shranjevali.

V mnogih primerih zadostujeta že preprostejša sistema za upravljanje s podatkovno bazo, kot sta PostgreSQL [39] in MySQL [28], ki ravno tako ponujata enostavno upravljanje podatkov 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 aplikacij), se za najprimernejšo izbiro izkaže SQLite [46]. Gre za minimalističen SUPB, 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 skupino manjših sistemov za upravljanje s podatkovno bazo lahko uvrstimo tudi rešitev Graphenix, ki smo jo razvili v okviru tega diplomskega dela [44].

2.2 Statističen pregled Python paketov ORM

V tabeli 2.1 je predstavljena kratka statistična analiza najpogostejše uporabljenih paketov ORM, ki so na voljo za programski jezik Python. Podatke za število mesečnih in tedenskih prenosov smo pridobili iz spletne platforme PYPI Stats [42], ki hrani podatke o prenosih posameznih paketov za programski jezik Python. Podatke za število zvezd GitHub, pa smo pridobili iz repositorijev posameznih paketov, ki so dostopni na platformi GitHub.

Paket	GitHub zvezde	Mesečni prenosi	Tedenski prenosi
Django [9]	72 tisoč	10 milijonov	2.4 milijona
SQLAlchemy [45]	7.5 tisoč	83 milijonov	20 milijonov
Peewee [35]	10 tisoč	1.1 milijon	281 tisoč
Tortoise ORM [49]	3.7 tisoč	88 tisoč	22 tisoč
SQLObject [47]	133	27 tisoč	6 tisoč

Tabela 2.1: Statistična primerjava ORM paketov za programski jezik Python.

Iz tabele 2.1 izluščimo, da za uporabo v programskem jeziku Python prevladujejo paketi SQLAlchemy, Django in Peewee.

- **Django ORM [9]** je del širšega orodja za izdelavo spletnih aplikacij Django. Omogoča zelo širok nabor operaciji nad podatkovno bazo, poleg tega ima vgrajen tudi svoj vmesnik za ukazno vrstico – CLI (angl. *Command-line interface*). S tem celotno ogrodje bistveno pospeši produktivnost razvijalcev. ORM podpira pet sistemov za upravljanje s

podatkovno bazo, to so PostgreSQL [39], MariaDB [23], MySQL [28], Oracle [15] in SQLite [46]. Izpostavimo, da je število prenosov lahko nekoliko zavajajoče, saj uporaba Django ogrođa v primerih, ko ne potrebujemo trajnega shranjevanja podatkov, ne pomeni uporabe Django ORM.

- **SQLAlchemy** [45] gre za enega najbolj fleksibilnih in uporabljenih paketov znotraj programskega jezika. Fleksibilnost pomeni, da so številne operacije, ki so znotraj Django ORM že avtomatizirane, tukaj prepuščene implementaciji razvijalca. Fleksibilnost s seboj prinaša tudi bistveno prednost, saj lahko kot razvijalec, ki dobro pozna ozadje delovanja uporabljenega SUPB, izkoristimo različne pristope optimizacije, ki jih ni mogoče realizirati znotraj Django ORM oz. je implementacija bistveno težja. Paket med drugim podira tudi prej omenjene SQLite [46], MySQL [28], Oracle [15], MSSQL [26] in še druge SUPB rešitve.
- **Peewee** [35] je eden izmed preprostejših paketov, ki ima lahkoten nivo abstrakcije. Paket podpira tri relacijske podatkovne baze, to so PostgreSQL [39], MySQL [28] in SQLite [46]. Gre za dokaj omejen nabor, saj je paket dokaj minimalističen in ne ponuja toliko funkcionalnosti kot Django ORM in SQLAlchemy.

Cilj vseh zgoraj navedenih paketov je pospešiti proces razvoja aplikacij in zagotoviti dodatno varnost pri delu s podatkovno bazo. Omeniti velja, da ti paketi omogočajo tudi druge koristne funkcionalnosti, kot je lažja manipulacija s podatki, poenostavljeno vzdrževanje in enostavna migracija strukture (preoblikovanje obstoječe sheme v že nameščeni podatkovni bazi). S svojo abstrakcijo in intuitivnimi vmesniki odpravljajo potrebo po neposrednem ročnem kovanju z zapletenimi poizvedbami SQL ter s tem razbremenjujejo razvijalce in omogočajo hitrejši razvoj aplikacij [48].

2.2.1 Slabost uporabe paketov ORM

Kljub vsem prednostim omenjenih paketov, pa tudi vsak izmed njih prinaša dodaten nivo abstrakcije in razvijalcu skrije veliko možnosti za optimizacijo delovanja SUPB. V algoritmih se moramo pogosto odločiti za kompromis med hitrostjo in porabo prostora, ko pa govorimo o paketih ORM in programskih jezikih, pa višji nivo abstrakcije lahko prinaša tudi nižje hitrosti delovanja.

SUPB za pridobitev končnega rezultata uporablja mehanizme kartezičnega produkta tabel, projekcije (izbira atributov) in selekcije (izbira vrstic). Končen rezultat poizvedbe je matrika, kjer vrstice predstavljajo izbrane zapise in stolpci predstavljajo attribute teh zapisov [22].

Kot ciljni uporabniki pogosto želimo drugačno strukturo podatkov, ki pa ni v skladu z osnovno idejo relacijskih baz. V ta namen ORM rešitve izvedejo več različnih poizvedb in interno združujejo zapise in kreirajo ciljno strukturo ali pa izvedejo eno zahtevnejšo poizvedbo in pridobljene podatke nato preoblikujejo v končno strukturo.

V primeru poizvedbe **“Za vse uporabnike pridobi njihova opravila in sporočila”** SQLAlchemy v ozadju izvede naslednjo poizvedbo SQL:

```
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
```

Za vsakega uporabnika prenesemo $\max(T_i, 1) \cdot \max(M_i, 1)$ zapisov, kjer je T_i število opravil i -tega uporabnika in M_i število sporočil i -tega uporabnika. N pa določa skupno število uporabnikov. Razlog za uporabo funkcije \max je v načinu združevanja, saj uporabljamo `OUTER JOIN`, ki vrne ujemaajoče se vrednosti kot tudi prazne vrednosti iz vezanih tabel.

Končno število zapisov, ki jih pridobimo iz podatkovne baze, je torej

$$\sum_{i=1}^N \max(T_i, 1) \cdot \max(M_i, 1). \quad (2.1)$$

Skupno število zapisov, kjer vsak zapis iz posamezne tabele pridobimo le enkrat, lahko napišemo kot vsoto

$$N + \sum_{i=1}^N (T_i + M_i). \quad (2.2)$$

Za izhodišče vzamemo vse uporabnike (N zapisov). Nadalje za vsakega uporabnika pridobimo njegova opravila (T_i opravil) in sporočila (M_i sporočil).

Če predpostavimo, da imamo podatkovno bazo, kjer imamo 100 uporabnikov, in ima vsak izmed njih 1000 sporočil, ter 200 opravil, pomeni da bomo v primeru prve vsote (2.1) prenesli kar $(100 \cdot 1000 \cdot 200) = 2 \times 10^7$ zapisov.

V primeru drugega načina poizvedbe, ki je predstavljena z vsoto (2.2) prenašamo bistveno manjšo količino podatkov $((100 \cdot (1000 + 200)) = 1.2 \times 10^5$. Kar predstavlja le 0.6 % podatkov v primerjavi s prvo vsoto (2.1).

V pristopu nabora podatkov s pomočjo ORM se kaže, da končni rezultat in struktura, ki jo vrača podatkovna baza, nista v enakem formatu. Preoblikovanje podatkov v končno obliko je pri uporabi omenjenih paketov izvedeno znotraj Python kode, ki ne izkoristi nizkonivojskega pristopa, kot ga omogoča programski jezik C++.

Pri implementaciji namenske rešitve bomo skušali format podatkov, ki jih vrača SUPB, kar se da dobro prilagoditi za ORM del in se posledično pri branju količinsko gledano približati vsoti (2.2).

Poglavje 3

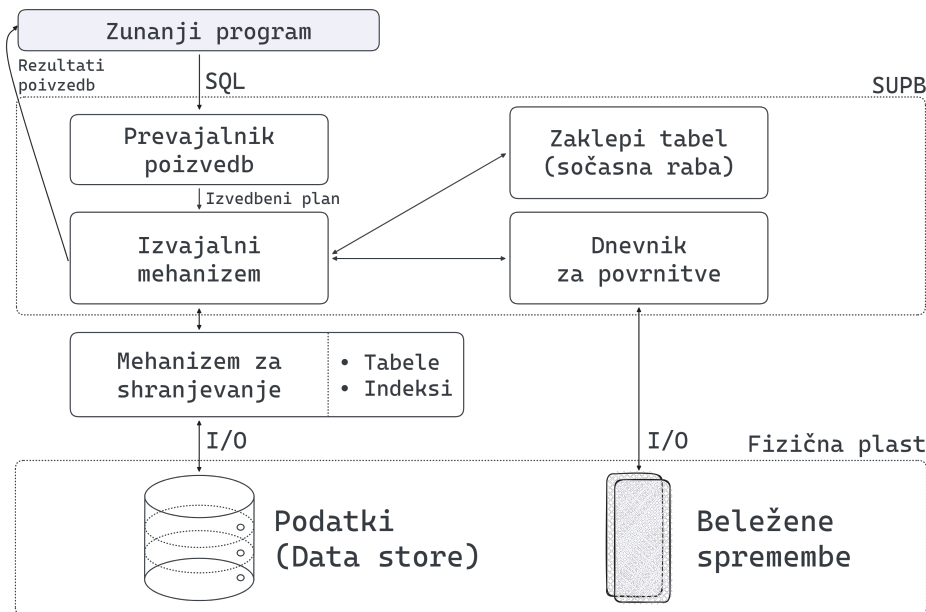
Razvoj SUPB in mehanizma za shranjevanje

V poglavju predstavimo razvoj SUPB in mehanizma za shranjevanje s pomočjo programskega jezika C++. V prvi fazi je predstavljena arhitektura uveljavljenih SUPB rešitev. Sledi predstavitev strukture shranjevanje podatkov in razvoja drevesne strukture B+ za optimalno indeksiranje podatkov. Nato predstavimo optimizacije oz. algoritme, ki omogočajo hitrejše branje podatkov in za zaključek predstavimo funkcionalnosti, ki jih podpira naš SUPB.

Vsa izvorna koda rešitve Graphenix je prosto dostopna (pod splošno javno licenco – GNU) na repozitoriju GitHub [44].

3.1 Arhitektura SUPB

Za boljše razumevanje delovanja SUPB je na sliki 3.1 predstavljena poenostavljena arhitektura sistema za upravljanje s podatkovno bazo. Poudarimo, da arhitektura ni točno določena in se v različnih implementacijah lahko razlikuje.



Slika 3.1: Arhitektura SUPB, prirejeno po knjigi: A First Course in DATABASE SYSTEMS, 2007 [50].

- **Prevajalnik poizvedb**

Vhodna točka za zunanji program je prevajalnik poizvedb (angl. *Query compiler*), ki prejme ukaz SQL, katerega pretvori v izvedbeni plan (angl. *Execution plan*). Poleg tega se v fazi kreiranja strukture izvedejo tudi optimizacije, kjer glede na statistične podatke v podatkovni bazi preuredimo strukturo za hitrejšo izvedbo.

- **Izvajalni mehanizem**

Izvajalni mehanizem (angl. *Execution engine*) je odgovoren za izvedbo vseh korakov definiranih v izvedbenem planu. Mehanizem predstavlja jedro SUPB, saj komunicira z večino komponent znotraj sistema.

- **Dnevnik za povrnitve**

Koncept dnevnika za povrnitev (angl. *rollback journal*) je dokaj preprosto. Gre za mehanizem, ki v fizično datoteko zapisuje vse spremembe, ki se zgodijo nad podatkovno bazo. V primeru izpadov električne energije ali napak pri vstavljanju zapisov je mogoče podatkovno bazo ob-

noviti v prejšnje (delujoče) stanje.

- **Mehanizem za shranjevanje**

Fizični podatki so shranjeni v sekundarnem pomnilniku (npr. SSD disk). Ko želimo fizične podatke spreminjati, moramo te najprej prenesti v glavni pomnilnik. Nalogo opravlja mehanizem za shranjevanje (angl. *Storage engine*), ki s pomočjo konfiguracijske datoteke določa format in način dostopa do shranjenih podatkov (angl. *Data store*).

- **Nadzor nad sočasno uporabo**

Nadzor nad sočasno uporabo (angl. *Concurrency control*) je način upravljanja več hkratnih poizvedb nad podatkovno bazo, ki vstavljajo, brišejo ali posodablajo zapise. Upravljalnik mora zagotoviti izvajanje v izolaciji in v času pisanja nastaviti ključavnico, ki zaklene dostop drugim poizvedbam. Po koncu poizvedbe se ključavnica odklene in s tem omogoči dostop naslednji poizvedbi.

Komponente realizirane v rešitvi Graphenix

Rešitev smo implementirali z nekoliko drugačno arhitekturo prikazano v razdelku 4.2. Največja razlika se pojavi v sami komunikaciji med uporabnikom in SUPB, saj ne uporabljamo klasičnih poizvedb SQL, temveč do podatkovne baze dostopamo z uporabo ORM.

Poleg tega pa v rešitev nista vključena nadzor nad sočasno uporabo in dnevnik za povrnitve.

3.2 Shema podatkov za predstavitev funkcionalnosti

Vsi primeri v tem poglavju uporabljajo shemo z uporabniki, nalogami in sporočili, prikazanimi v programski kodi 3.1. Uporabnik ima lahko več nalog in sporočil – pri sporočilih pa imamo še dodatno vlogo, saj je lahko uporabnik pošiljatelj ali prejemnik.

```
class User(gx.Model):
    name = gx.Field.String(size=100)
    tasks = gx.Field.VirtualLink("user")
    sent_msgs = gx.Field.VirtualLink("sender")
    recieved_msgs = gx.Field.VirtualLink("receiver")

class Task(gx.Model):
    content = gx.Field.String(size=100)
    user = gx.Field.Link()

class Message(gx.Model):
    content = gx.Field.String(size=50)
    date = gx.Field.DateTime().as_index()
    sender = gx.Field.Link()
    receiver = gx.Field.Link()
```

Programska koda 3.1: Shema podatkov za predstavitev funkcionalnosti.

3.3 Mehanizem shranjevanja podatkov

3.3.1 Matrika podatkov in zaglavje s kazalci

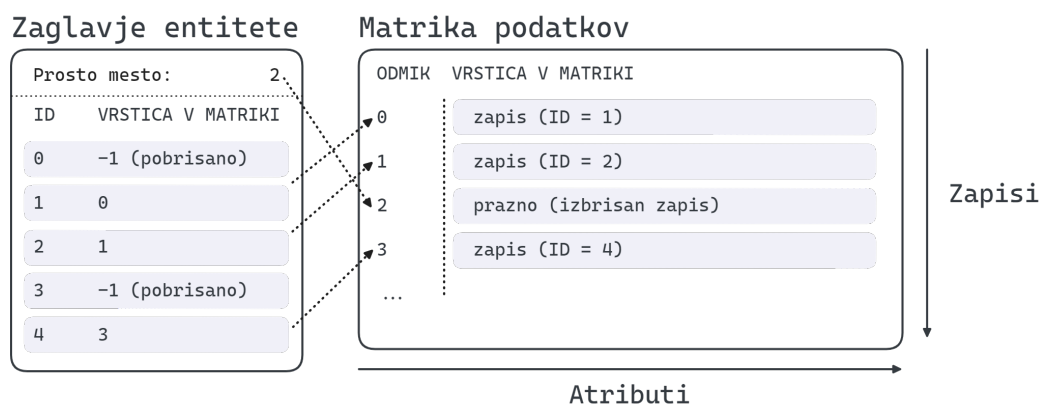
Za vsako posamezno tabelo se privzeto kreirata dve datoteki

`ix_<naziv_tabele>.bin` in `<naziv_tabele>.bin`.

- `<naziv_tabele>.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 shranjujejo v tabeli. Posamezen stolpec matrike predstavlja posamezen atribut v tabeli.
- `ix_<naziv_tabele>.bin` (glava tabele 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 prsto vrstico v matriki, ki je definiran po pravilu:

1. Matrika je polna – kazalec na konec datoteke.
2. V matriki podatkov obstaja fragmentacija (v matriki je prazen prostor, ki je nastal zaradi brisanja zapisa) – kazalec na prvo prsto vrstico v matriki.



Slika 3.2: Struktura matrike podatkov in zaglavja s kazalci.

3.3.2 Tipi podatkov

Znotraj razvitega SUPB podpiramo sedem podatkovnih tipov:

- Cela števila (**Int**) – gre za vrednost, ki je na nivoju programskega jezika C++ shranjena kot podatkovni tip `int64_t`, oziroma predznačeno celo število. Velikost za posamezno vrednost je 8 bajtov. Razpon vrednosti, ki jih lahko shranimo v posamezen zapis je določen z intervalom $[-2^{63}, 2^{63} - 1]$.
- Realna števila (**Float**) – zaradi lažje implementacije je velikost realnih števil enaka kot velikost celih števil – torej 8 bajtov. Vrednost je na nivoju podatkov realizirana kot “dvojni float“ oz. “double“. Gre za

zaporedje bitov, ki so po standardu IEEE 754 [20] ločeni v predznak, eksponent in mantiso.

- Nizi (**String**) – Gre za zaporedje znakov, kjer pri definiciji atributa določimo maksimalno dovoljeno dolžino posameznega niza. Torej dolžina posameznega niza je N bajtov, kjer je N enak maksimalni dolžini.
- Logične vrednosti (**Bool**) – je najmanjši podatkovni tip, ki ga vsebuje naš SUPB in zavzame vsega 1 bajt; drži pa lahko vrednosti 0/1 oz. **True/False**, kot je to definirano v programskem jeziku Python.
- Datumi (**Datetime**) – gre za podatkovni tip, ki je ponovno shranjen, kot vrednost `int64_t` in je predstavljen v EPOCH formatu [11] – gre za časovni odmik trenutnega časa od UTC datuma 1. 1. 1970.
- Povezava (**Link**) – gre za vgrajen tip relacije, kjer je podatek kazalec na drug zapis v določeni tabeli. V ozadju je to le primarni ključ določenega zapisa, ki se nastavi avtomatsko (ponovno gre za podatek, ki je zapisan kot `int64_t` vrednost).
- Virtualna povezava (**VirtualLink**) – gre za navidezno polje v tabeli, ki ne zavzame nobenega dodatnega prostora. Polje je pomembno za vezave zapisov ob poizvedbah, kjer na določen zapis vežemo seznam zapisov iz druge tabele.

3.3.3 Implementacija relacij

Implementacija relacij med tabelami se izvaja s pomočjo uporabe tujih ključev, ki predstavljajo primarne identifikatorje zapisov, ki jih želimo povezati. Vzpostavitev povezave se izvede pri ustvarjanju zapisa na ravni programskega jezika Python. Ko na zapisu nastavimo povezavo, se v polje shrani identifikator povezanega zapisa.

V programski kodi 3.2 je predstavljen preprost primer, kjer sporočilo vežemo na dva uporabnika, ki imata vlogi pošiljatelja in prejemnika.

```
john = User(name='John Doe').make()
jane = User(name='Jane Doe').make()

first_mesage = Message(
    content = 'Hello, World!',
    sender = john,
    reciever = jane
).make()
```

Programska koda 3.2: Povezovanje zapisov v implementiranem SUPB.

3.4 Indeksiranje z uporabo B+ dreves

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

V B+ drevesu vsa vmesna vozlišča vsebujejo kazalce na naslednja vozlišča znotraj drevesa. Vmesna vozlišča služijo le kot povezave, ter omogočajo učinkovitejše iskanje in navigacijo po strukturi, medtem pa so podatki shranjeni le v listih drevesa.

Sama struktura zahtevna tudi striktno uravnoteženost drevesa, kar pomeni, da morajo biti vsi listi drevesa na enakem nivoju.

Še ena izmed karakteristik B+ drevesa je povezava med vozlišči na enakem nivoju, saj to omogoča optimalno iskanje na intervalu in pripelje do časovne zahtevnosti $O(\log_b(N) + k)$, kjer je k širina intervala. Enaka poizvedba bi v klasičnem B drevesu pomenila časovno zahtevnost $O(\frac{k}{b} \cdot \log_b(N))$, saj bi zapise iz intervala iskali ločeno glede na posamezno vozlišče [2].

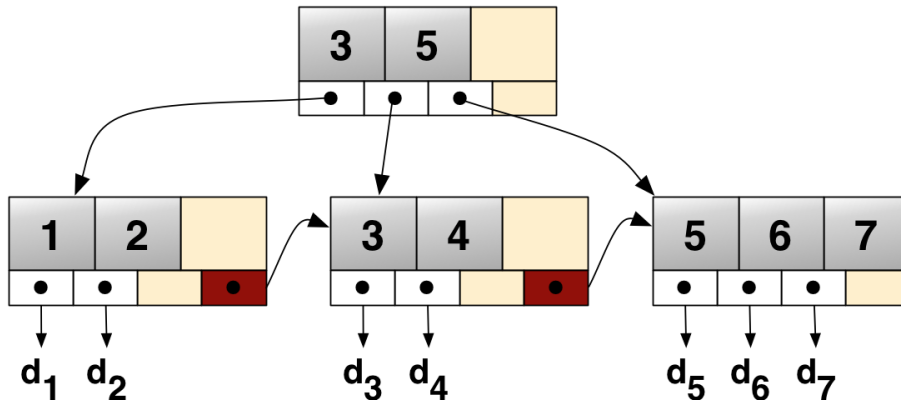
Lastnosti B+ drevesa

Ko govorimo o definiciji strukture drevesa B+ se moramo držati naslednjih pravil [13]:

- Vsako vozlišče ima največ $b - 1$ vrednosti in b kazalcev, kjer je b stopnja vozlišča.
- Listi drevesa so vsi na enakem nivoju in tvorijo povezan seznam.
- Urejenost – vse vrednosti, ki se nahajajo levo morajo biti manjše ali enake trenutni. Vse vrednosti desno, pa morajo biti večje ali enake trenutni.
- Vozlišče je veljavno, kadar ima vsaj $\lceil \frac{b-1}{2} \rceil$ zapisov. V primeru, da gre za notranje vozlišče mora to imeti vsaj $\lceil \frac{b}{2} \rceil$ kazalcev na naslednja vozlišča.

Primer B+ drevesa

Na sliki 3.3 je prikazan preprost primer B+ drevesa realiziran s stopnjo $b = 4$.



Slika 3.3: Primer B+ drevesa, povzeto po Wikipediji [2].

Drevo vsebuje sedem različnih vrednosti, pri čemer ima vsaka izmed vrednosti še kazalec na zapis, katerega označimo z d_i . V primeru uporabe B+ drevesa za indeksiranje v podatkovni bazi, ti kazalci kažejo na posamezen zapis v tabeli, kjer se nahaja indeksiran atribut.

Časovne zahtevnosti glavnih operaciji

Iskanje, vstavljanje in brisanje imajo v B+ drevesu časovno zahtevnost $O(\log_b(N))$, kjer je b stopnja vozlišča in N število zapisov v drevesu.

Vendar časovna zahtevnost velja le v primeru, ko sta razcep, ki se zgodi ob prekoračitvi velikosti vozlišča ob vstavljanju in združevanje, do katerega pride ob spodnji prekoračitvi velikosti vozlišča (pri brisanju), implementirana s časovno zahtevnostjo $O(1)$ [13]. Poleg tega pa se mora implementacija drevesa držati vseh lastnosti definiranih v razdelku 3.4.

3.4.1 Iskanje v B+ drevesu

Iskanje v B+ drevesu sledi algoritmu v ostalih iskalnih drevesih. Ključna razlika se pokaže v koraku iskanja znotraj posameznega vozlišča, kjer moramo v seznamu zapisov poiskati prvo večjo ali enako vrednost glede na iskani ključ. To določa odmik, po katerem se premaknemo k naslednjemu vozlišču v B+ drevesu [13].

V implementaciji B+ drevesa je ena izmed ključnih lastnosti omogočanje večkratnega shranjevanja istega ključa. Posledično je lahko rezultat več kot en zapis, zato rezultat funkcije predstavimo kot množica $\{d_1, d_2, d_3, \dots\}$, kjer z d_i označujemo vrednosti najdenih zapisov.

3.4.2 Vstavljanje v B+ drevo

Ko govorimo o zahtevnosti implementacije posamezne operacije znotraj B+ drevesa, sta brisanje in vstavljanje definitivno bolj zahtevni operaciji od iskanja. Ob dobri definiciji strukture znotraj programskega jezika je iskanje zapisov relativno lahek postopek, ki je tudi bolj intuitiven.

Algoritem za vstavljanje je povzet po implementaciji na spletni platformi GeeksforGeeks [19]. Pristop vstavljanja je narejen na primeru, ki podpira le celoštevilsko indeksiranje, hkrati pa drevo shranjuje le na nivoju pomnilnika.

V naši implementaciji je bilo potrebno pokriti oba scenarija. Težavo z omejitvijo podatkovnih tipov smo rešili s pristopom uporabe "generikov", ki

omogočajo izbiro poljubnega podatkovnega tipa pri kreiranju drevesa. Kljub temu smo morali ročno pokriti še branje in pisanje v datoteko, saj pristop za pisanje števil in nizov ni povsem enak.

Koraki za vstavljanje

1. S pristopom iskanja poiščemo mesto, kamor bi morali vstaviti vrednost.
2. Poskusimo z vnašanjem, če je list poln, vozlišče razdelimo na dva dela in kreiramo novo vozlišče, katerega povežemo na starša.
3. Če pride do preliva v vmesnem vozlišču, to vozlišče ponovno razdelimo na dva vozlišča in prevežemo naslednja vozlišča, kot tudi starša (izvedemo rekurzivno posodobitev drevesa).

Koraki za vstavljanje so zelo okrnjeni in poenostavljeni, saj celoten algoritem za vstavljanje vsebuje tudi rekurzivne dele metode, ki skrbijo za uravnoteženost drevesa in pravilno povezavo med zaporednimi vozlišči.

3.4.3 Brisanje iz B+ drevesa

Pri izvedbi brisanja gre koračno gledano za zelo podoben postopek, kot pri vstavljanju zapisa.

1. Poiščemo vozlišče, ki vsebuje vrednost, ki jo bomo izbrisali, če ta ne obstaja, zaključimo.
2. Iz vozlišča odstranimo vrednost. V primeru, da v vozlišču obstaja vsaj še $\lceil \frac{b-1}{2} \rceil$ zapisov, je brisanje končano. V scenariju, kjer pride do spodnje prekoračitve, moramo izvesti postopek združevanja vozlišč. Zatem moramo pravilno nastaviti tudi kazalec starša.
3. Zagotoviti moramo, da so veljavna tudi vsa predhodna vozlišča (od brisanega vozlišča do korenskega vozlišča). V tem postopku ponovno izvajamo rekurzivno iteracijo in po potrebi posodabljammo drevo.

3.4.4 Posodabljanje v B+ drevesu

Najbolj preprost način izvedbe posodabljanja v B+ drevesu je izvedba brisanja in nato vstavljanja nove vrednosti. Iz vidika števila operaciji lahko izvedemo bistveno optimizacijo, saj v primeru manjše spremembe vrednosti, ki jo posodabljam, lahko pogosto vrednost le premaknemo znotraj vozlišča. Optimizacija na časovno zahtevnost nima vpliva, lahko pa ima velik vpliv na število I/O operaciji.

3.4.5 Implementacija za shranjevanje na disk

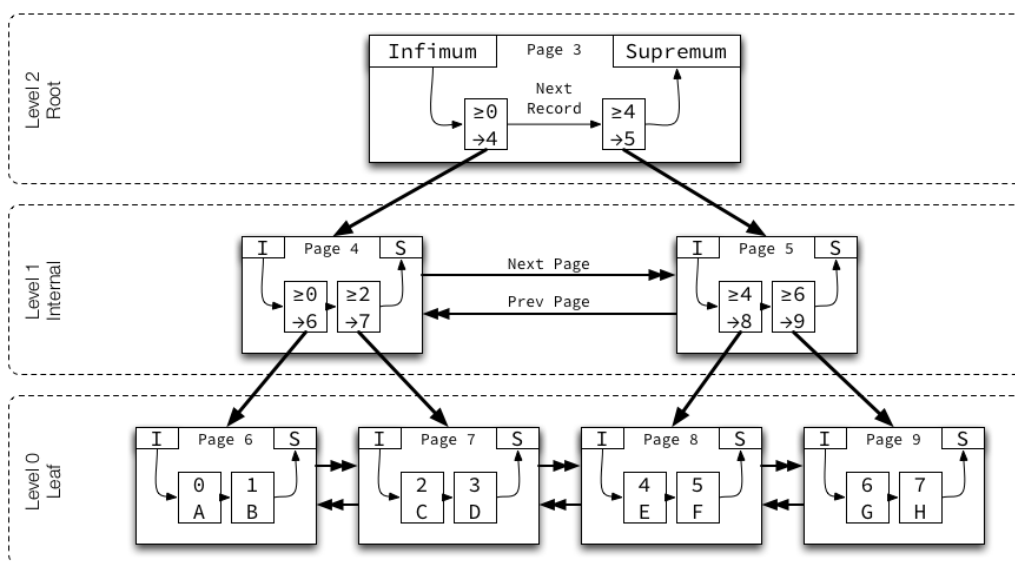
Razlog za uporabo B dreves znotraj podatkovnih baz je dokaj trivialen. Iskalne strukture, ki so realizirane s pomočjo binarnih dreves na področju shranjevanja na disk, namreč vsebujejo izrazito ozko grlo. Težava se pojavi pri branju in pisanju posameznih vozlišč, saj operacija nad enim vozliščem pomeni en dostop do diska oz. datoteke. B drevesa ta problem rešujejo s samo količino podatkov, ki je shranjena v enem vozlišču. Ob shranjevanju na klasične trde diske je bila velikost enega vozlišča določena, kot velikost sektorja na disku. To je pomenilo, da je dostop do posameznega vozlišča predstavljal le en dostop na disk, ki je omogočal branje/pisanje b vrednosti hkrati [13].

Klasičen način določanja velikosti posameznih vozlišč se je delno spremenil odkar so standard za shranjevanje podatkov postali hitrejši tipi diskov kot so SSD in v tem primeru velikost posameznega vozlišča ni več določena izključno na podlagi strojne opreme.

V primeru InnoDB se velikost posameznega vozlišča nastavi glede na sistemsko konstanto `innodb_page_size`, ki je privzeto nastavljena na 16384 bajtov (16 KB) [17]. Vrednost konstante je možno določiti le ob kreiranju MySQL instance [18]. Vrednost pa lahko preberemo z ukazom `SHOW VARIABLES LIKE 'innodb_page_size'`.

3.4.6 Implementacija v InnoDB

InnoDB je privzeti mehanizem za shranjevanje podatkov v MySQL SUPB, ki med drugim za indeksiranje uporablja implementacijo B+ drevesa. Struktura je prikazana na sliki 3.4.



Slika 3.4: Primer B+ drevesa, povzeto po InnoDB [3].

Znotraj InnoDB implementacije B+ drevesa, vsi nivoji vsebujejo kazalec na prejšnje in naslednje vozlišče na enakem nivoju. Enak pristop je implementiran tudi v mehanizmu za shranjevanje v rešitvi Graphenix.

Vsaka stran oz. vozlišče vsebuje "Infimum", ki predstavlja vrednost manjšo kot kateri koli zapis v vozlišču. Na drugi strani pa je "Supremum", ki predstavlja vrednost večjo kot kateri koli zapis v vozlišču.

Gre še za malo bolj napredno implementacijo B+ drevesa, ki omogoča nekaj dodatnih optimizacij z vidika shranjevanja podatkov in izvajanja iskanja [3].

3.4.7 Podprte operacije filtriranja

Znotraj razvitega SUPB B+ drevo uporabljamo za tri različne operacije filtriranja.

- Najbolj preprosta izmed operaciji je iskanje zapisa po enakosti, kjer v drevesu po vrednosti preprosto poiščemo zapise in vrnemo identifikatorje teh zapisov oz. zapisa.
- Preverjanje ali je element v seznamu podanih elementov, kjer za vsak element v seznamu izvedemo iskanje zapisa znotraj B+ drevesa (večkrat izvedemo iskanje po enakosti).
- Zadnja podprta operacija je iskanje vrednosti na intervalu, kjer podamo spodnjo in zgornjo mejo. Ob filtriranju najprej izvedemo iskanje po spodnji meji, nato izvedemo sprehod skozi povezana vozlišča, dokler ne presežemo zgornje meje. Postopek je reliziran z metodo, ki prejme delni rezultat v obliki vektorja vozlišč in desno mejo `load_up_to_right(nodes, right_limit)`. Metoda tekom iteracije skozi liste drevesa, identifikatorje dodaja v vektor `nodes`.

Pri uporabi indeksiranja ostaja ogromno nepokritih operaciji, ki jih sicer klasični mehanizmi za shranjevanje podpirajo. Vendar zaradi osnovne strukture pri nekaterih operacijah ne bi dosegli bistvene izboljšave in bi v določenih scenarijih celo upočasnili poizvedbo.

Filtriranje pred branjem matrične datoteke

Pred začetkom branja matrične datoteke moramo najprej dobiti vse odmike. V primeru, da nimamo nobenih omejitev, preprosto preberemo vse odmike in preberemo vse veljavne zapise znotraj matrike podatkov. Za optimizacijo branja, pa imamo tri scenarije, ki lahko že pred začetkom branja izločijo neveljavne zapise.

1. Pogoji nad primarnim identifikatorjem zapisov.

2. Že definirani identifikatorji – `qobject.ix_constraints` (kadar gre za poizvedbo, ki ni v korenu drevesa poizvedb).
3. Kadar imamo pogoje nad atributi, ki uporabljajo indekse.

V naslednjem koraku kreiramo presek vseh identifikatorjev, ki smo jih pridobili kot rezultat dodatnih filtriranj. Vse akcije namreč vračajo zgoščeno tabelo identifikatorjev. Za konec moramo pred branjem le še preveriti, kateri identifikatorji se pojavijo v preseku in lahko nadaljujemo na branje matrične datoteke, ki vsebuje dejanske zapise.

3.4.8 Ostali pristopi indeksiranja

InnoDB omogoča dva bolj pogosta pristopa za indeksiranje podatkov. Prvi pristop je s prej omenjenim B+ drevesom 3.4.6. Z uporabo B+ drevesa omogočimo optimizacijo poizvedb, kjer uporabljamo pogoje `=`, `>`, `>=`, `<`, `<=`, `BETWEEN` delno, pa je pokrit tudi operator `like`, kjer je iskanje s pomočjo drevesa izvedeno v primerih, ko iščemo začetek niza [7].

Mehanizem za shranjevanje poleg dreves podpira tudi indeksiranje s pomočjo zgoščevalnih tabel. Gre za pristop, ki se uporablja le za preverjanje enakosti, saj zgoščene vrednosti ne moremo primerjati po velikosti. Prednost uporabe zgoščevalnih tabel se skriva v hitrosti iskanja po specifični vrednosti, saj ima ta proces časovno zahtevnost $O(1)$ oz. konstanto časovno zahtevnost.

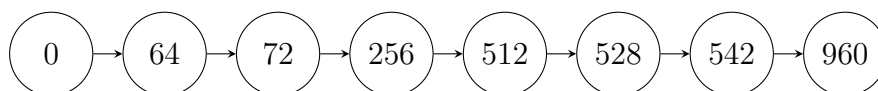
Pristop je tako zelo uporaben, kadar imamo polje, ki vsebuje omejen nabor vrednosti, npr. pri indeksiranju pošiljatelja sporočila bi z uporabo zgoščevalne tabele zelo hitro poiskali sporočila iskanega pošiljatelja, hkrati pa za ta atribut načeloma ne izvajamo ostalih operaciji filtriranja in v večini scenarijev le iščemo zapise točno določenega uporabnika.

3.5 Optimizacije poizvedovanja

3.5.1 Gručanje podatkov ob branju iz diska

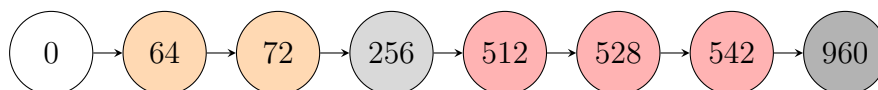
Kot je bilo že omenjeno v prejšnjih poglavjih, je največje ozko grlo, s katerim se srečujemo, I/O operacije (branje in pisanje na disk). V ta namen je tudi prenos podatkov iz datoteke v pomnilnik realiziran s čim manj dostopi do diska. Zato je ob prvi fazi nabora dodan algoritem za gručenje podatkov.

Definicija problema je zelo preprosta: **“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 vektor odmikov v matrični datoteki prikazan na sliki 3.5.



Slika 3.5: Zaporedje odmikov za poizvedbo iz matrične datoteke.

Iz odmikov lahko hitro razberemo dve gruči, za kateri bi želeli, da jih naš algoritem identificira in branje odmikov izvede v enem koraku – to sta gruči $[64, 72]$, ter $[512, 528, 542]$. Optimalen pristop branja bi lahko barvno označili po konfiguraciji, ki je prikazana na sliki 3.6.



Slika 3.6: Zaporedje odmikov 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 opravimo v linearnem času, saj ob testiranju ostalih algoritmov v večini primerov samo gručenje traja dlje, kot pa branje posameznih zapisov v matriki. V ta namen smo pripravili preprost algoritem, ki se sprehodi skozi seznam in spremlja odmike, ter jih združuje, v kolikor ne presežemo zgornje meje velikosti ene gruče (`MAX_CLUSTER_SIZE`) in poleg tega zadovoljimo minimalno velikost za gručo (`MIN_CLUSTER_SIZE`).

3.5.2 Uporaba prioritete vrste za urejanje zapisov

Ob implementaciji omejevanja števila rezultatov s pomočjo funkciji `limit` in `offset` moramo poskrbeti za časovno optimalno izvedbo poizvedbe, saj z vidika uporabnika SUPB pričakujemo, da z uporabo funkcije `limit` čas izvajanja skrajšamo.

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 , pri čemer je L maksimalno število elementov, O pa predstavlja odmik oz. število zapisov, ki jih želimo preskočiti”**.

Preizkusili smo tri različne pristope za iskanje K najmanjših elementov v seznamu.

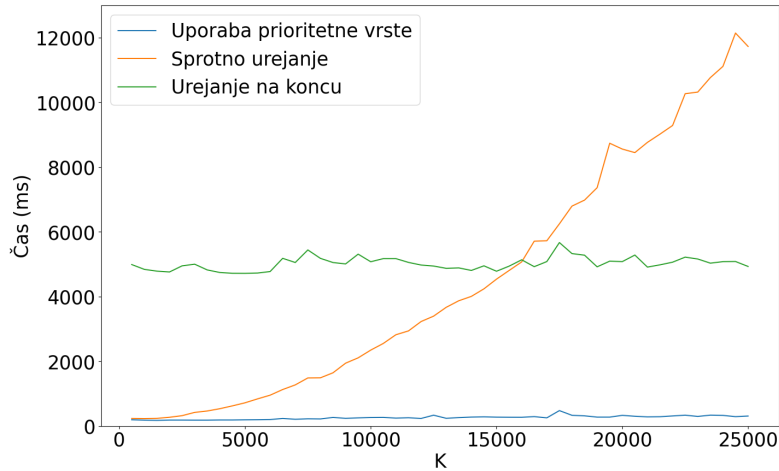
1. Klasično vstavljanje v seznam in ureditev elementov ob koncu vnašanja, ter rezanje elementov od indeksa O , do vključno $O + L$. Časovna zahtevnost tega pristopa je $O(N \cdot \log_2(N))$, prostorska zahtevnost pa je $O(N)$, kjer je N število vseh zapisov, saj moramo v pomnilniku držati vse potencialne elemente.
2. Postopno vstavljanje v urejen seznam, v tem primeru uporabimo vgrajeno `std::vector` [53] 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. Čeprav je časovna zahtevnost binarnega iskanja $O(\log_2(N))$ zaradi premikanja elementov časovna zahtevnost vstavljanja 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 v primerjavi z urejanjem 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 prioriteto vrsto, ki je realizirana kot kopica (angl. *heap*). Za podatkovno strukturo uporabimo vgrajeno prioriteto vrsto (`std::priority_queue`) [40]. 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 predhodnem 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 prioritete vrste je tudi izbor podatkovne strukture, ki jo vrsta uporablja za shranjevanje podatkov. Za najbolj optimalen pristop se je izkazal vgrajeni `std::vector` [53], ki je na nivoju pomnilnika shranjen v enem kosu, kar prevajalniku in procesorju omogoči, da izrabita še optimizacijo lokalnosti.

Za vse tri pristope obstaja veliko spremenljivk, ki vplivajo na čas izvajanja 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. 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 nastala

vljena na 1×10^7 , K pa se spreminja na x osi in je definiran na intervalu $[100, 25000]$.



Slika 3.7: Časovna primerjava pristopov za iskanje K najmanjših elementov v seznamu.

Iz grafa 3.7 lahko razberemo, da vrednost K nima bistvenega vpliva na prvi in tretji algoritem. Izjema je drugi algoritem, ki 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 je razvidno iz dobre korelacije glede na vrednost K .

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

3.5.3 Uporaba podatkovnih okvirjev

Akcija za poizvedovanje je ob prvi implementaciji delovala 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 razreda.

Ob profiliranju akcije za poizvedovanje podatkov smo opazili, da velik delež izvajalnega časa vzame 3. točka (pretvorba posameznega zapisa v instanco razreda).

Za hitrejšo izvedbo smo na nivoju programskega jezika C++ pripravili podatkovno strukturo – pogled (angl. *View*). Ta struktura odstrani potrebo po tretjem koraku in spremeni predstavitev posameznih vrstic iz strukture slovarja v terko (angl. *Tuple*). *View* predstavlja ovoj, ki deluje na podobnem principu, kot delujejo podatkovni okvirji (angl. *DataFrame*) v paketih za podatkovno analitiko [25].

Optimizacija v številkah

Za izmeritev učinkovitosti implementacije smo uporabili scenarij iz performančnih testov celotnega sistema (branje 1×10^6 vrstic in shranjevanje celotnega seznama v pomnilnik).

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

3.6 Poizvedovanje z uporabo ORM

• Filtriranje podatkov – funkcija *filter*

Argumenti te funkcije so drevo, kjer je posamezno vozlišče drevesa predstavljeno z enim ali več pogoji, kjer lahko vozlišče zahteva izpolnjenost vseh ali vsaj enega pogoja.

Vozlišče v katerem želimo, da vsi pogoji držijo, kreiramo s pomočjo `gx.every(...)` metode, medtem ko za kreiranje vozlišča z vezavo “ali” uporabimo metodo `gx.some(...)`.

Primer filtriranja podatkov je predstavljen s primerom programske kode 3.3, kjer izvedemo zahtevnejšo obliko filtriranja nad tabelo sporočil.

```
john = User(name='John Doe').make()
jane = User(name='Jane Doe').make()
...
query = Message.filter(
    Message.date.greater(
        datetime.now() - timedelta(days=5)
    ),
    gx.some(
        Message.sender.equals(john),
        Message.reciever.equals(john),
        Message.reciever.is_not(jane)
    )
)
```

Programska koda 3.3: Filtriranje podatkov znotraj Graphenix rešitve.

Poizvedba izvede nabor sporočil, ki so bila poslana v zadnjih petih dneh, poleg tega pa želimo, da je izpolnjen vsaj eden izmed pogojev – pošiljatelj/prejemnik je “John” in prejemnik ni “Jane”.

V sklopu filtriranja podpiramo 11 različnih operaciji. To so: je enako, ni enako, večje, večje ali enako, manjše, manjše ali enako, **regex**, **iregex** (neobčutljiv **regex** na velike in male črke), je v (element je v seznamu), ni v (element ni v seznamu), med (vrednost je v intervalu).

- **Omejitev števila zapisov in določanje odmikov – funkciji `limit` in `offset`**

Gre za funkciji, 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 (angl. *paging*). Razlog za implementacijo je optimizacija prikaza na način, da omejimo število zapisov, ki jih naenkrat prikažemo uporabniku. Lahko pa gre tudi za čisto preprosto poizvedbo, kjer iščemo prvih nekaj zapisov glede na določene kriterije npr. zadnjih pet sporočil določenega uporabnika.

- **Urejanje zapisov – funkcija `order`**

Izvedba urejanja podatkov je omejena na urejanje znotraj ene tabele. Pri tem imamo možnost urejati po poljubnem številu atributov, pri čemer lahko izvajamo bodisi naraščajoče bodisi padajoče urejanje.

Predstavljen je enostaven primer, kjer uredimo sporočila. Urejamo padajoče glede na datum sporočila in naraščajoče glede na vsebino.

```
query = Message.order(  
    Message.date.desc(),  
    Message.content  
)
```

Za poizvedbo ORM lahko napišemo tudi ekvivalentno poizvedbo SQL.

```
SELECT * FROM messages  
ORDER BY date DESC, content
```

- **Uporaba relacij znotraj poizvedbe – funkcija `link`**

Gre za malo drugačno izvedbo kot je to pri relacijskih SUPB rešitvah, saj v našem primeru podatkov ne združujemo v eno matrično strukturo, temveč so podatki združeni v drevesno strukturo.

```
SELECT * FROM users  
INNER JOIN tasks ON tasks.id_user = users.id
```

Ker moramo povezavo definirati že na nivoju zapisa, je združevanje nekoliko drugačno.

```
query = User.link(tasks=Task)
```

V primeru izvedbe, bi za vsakega uporabnika dobili še seznam njegovih nalog.

Algoirtem za poizvedovanje z uporabo relacij

Algoirtem za poizvedovanje je predstavljen s pomočjo psevdokode 3.4, kjer omogočamo nabor podatkov iz različnih tabel hkrati.

1. Začetna točka rekurzivne metode je izvedba nabora podatkov na korenski tabeli poizvedbe. V primeru, da gre za poizvedbo, ki ne vsebuje povezovanja, rekurzijo zaključimo – robni pogoj rekurzije.
2. V drugi točki pripravimo strukture za dodatno filtriranje znotraj naslednjih poizvedb. Poleg tega pa pripravimo vmesno strukturo za shranjevanje rezultatov teh poizvedb.
3. Nato rekurzivno izvajamo naslednje poizvedbe in rezultate shranjujemo v prej pripravljeno združevalno strukturo (gre za zgoščevalno tabelo, ki vezan identifikator preslika v dejanski zapis).
4. V zaključni fazi algoritma vmesne rezultate iz zgoščevalne tabele združimo v rezultat korenske poizvedbe – to je tudi rezultat celotne metode, ki je tipa **View** predstavljen v razdelku 3.5.3.

```
fn execute_query(qobject: query_object) -> View
    root_res := execute query on the root table
    if no qobject.subquery:
        return root_res

    subquery_map := map
    for link in root_res:
        subquery_map[link] := empty View

    for subquery in qobject.subquery:
        subquery.filters.apply(subquery_map)
        subquery_res := execute_query(subquery)
        for link, view in subquery_res:
            subquery_map[link].add(view)

    for link, view in subquery_map:
        root_res[link] := view

    return root_res
end fn
```

Programska koda 3.4: Algoritem za izvedbo poizvedbe z uporabo relacij.

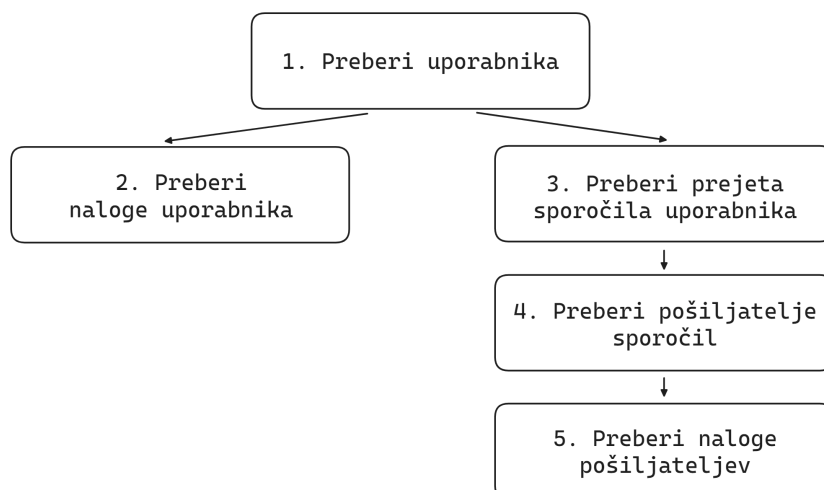
Globoko vgnezdene poizvedbe

Razvit SUPB s funkcijo `execute_query` 3.6 omogoča tudi globoko vgnezdene poizvedbe, ki v eni poizvedbi omogočajo nabor vgnezdene strukture podatkov (kar klasične poizvedbe SQL ne omogočajo). V programski kodi 3.8 je predstavljen primer globoko vgnezdene poizvedbe, ki je sestavljena iz štirih nivojev.

```
user_view = User.link(  
    tasks = Task.limit(N),  
    recieved_msgs = Message  
        .link(sender = User.link(  
            tasks = Task.limit(N))  
        )  
        .order(Message.date.desc())  
        .limit(N)  
)\  
.filter(User.equals(user_id))\  
.first(as_view=True)
```

Programska koda 3.5: Primer globoko vgnezdene poizvedbe.

Poizvedba za uporabnika, ki ga poiščemo preko primarnega identifikatorja pridobi še zadnjih N nalog in sporočil. Za nabor sporočil se pomaknemo globlje v rekurzivno funkcijo in za vsako sporočilo pridobimo še podatke pošiljateljev, nato pa še za vsakega izmed pošiljateljev preberemo N nalog.



Slika 3.8: Koraki za nabor podatkov globoko vgnezdene poizvedbe.

Ob naboru podatkov izvedemo pristop od zgoraj navzdol, saj najprej preberemo podatke za korensko tabelo in se nato spuščamo v nižje nivoje.

Vsako vozlišče drevesa iz slike 3.8 predstavlja ločeno branje matrične datoteke podatkov. Prikazan pristop nabora v klasičnih poizvedbah SQL ni mogoč, saj rezultat poizvedbe SQL ne more vsebovati vgnезdenih zapisov.

Vezavo zapisov izvedemo na nivoju programskega jezika C++, kjer uporabimo strukturo zgoščevalnih tabel za grupiranje zapisov po identifikatorju starša. Posledično sestava končne strukture steče bistveno hitreje, kot ob pristopu združevanja v programskem jeziku Python.

- **Agregacija oz. združevanje podatkov**

Ko govorimo o agregaciji imamo v mislih poizvedbe, kjer iščemo različne statistične rezultate iz posamezne tabele. MySQL SUPB podpira kar nekaj različnih operaciji za agregacijo [29] (AVG, COUNT, MAX, MIN, SUM, STD, STDDEV, VARIANCE ...).

V našem SUPB podpiramo prvih pet operaciji:

- SUM – operacija za seštevanje vrednosti v stolpcu.
- MIN – iskanje najmanjše vrednosti v stolpcu.
- MAX – iskanje maksimalne vrednosti v stolpcu.
- COUNT – štetje zapisov.

Vse operacije so omejene na eno tabelo in en stolpec, kar pomeni, da lahko v eni poizvedbi poiščemo npr. iskanje najnovejšega sporočila med zapisi. V večini primerov agregaciji pa ne iščemo globalno gledano zapisa, ki najbolj ustreza našim kriterijem, ampak iščemo zapis grupiran glede na določen kriterij npr. iščemo najnovejše sporočilo za vsakega posameznega uporabnika. Kadar govorimo v smislu poizvedb SQL, to pomeni uporabo GROUP BY stavka.

```
SELECT MAX(messages.date) AS 'latest'  
FROM messages  
GROUP BY tasks.user_id
```

Poizvedbo SQL lahko pretvorimo v enakovredno poizvedbo, ki deluje z uporabo paketa Graphenix.

```
query = Message.agg(  
    by=Message.user,  
    latest=gx.AGG.max(Message.date)  
)
```

Sintaksa je dokaj preprosta. Za izvedbo agregacijske poizvedbe uporabimo metodo `.agg(...)`. Kot parametre pošljemo polje, po katerem združujemo (nadomestilo za `GROUP BY`). Nato pa lahko dodamo še poljubno število parametrov, ki predstavljajo različne združevalne akcije.

Poglavje 4

Pristop razvoja programske opreme

V poglavju bomo predstavili izbrane metodologije, ki so bile 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 omogoča lažje odkrivanje napak in višjo kakovost rešitve.

4.1 Razvoj paketa za programski jezik Python

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

V namen upravljanja s paketi, ki niso del standardne knjižnice programskega jezika se pogosto srečamo z upravljalci paketov (angl. *package manager*). Gre za orodje, ki omogoča enostaven način nameščanja, posodabljanja

in odstranjevanja uvoženih paketov.

4.1.1 Uporaba Python.h za razvoj paketa

Programski jezik Python je poznan predvsem po uporabi v področjih umetne inteligence, podatkovne analitike in strežniških aplikacij. 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 prevedenih jezikov.

Področji umetne inteligence in podatkovne analitike temeljita na obdelavi masovnih podatkov, kar posledično pomeni, da Python ni najbolj optimalen jezik za izvedbo teh nalog. Zaradi teh omejitev so nastali paketi, ki so v osnovi napisani v jezikih, ki so bistveno hitrejši (prevedeni). Med pogosteje uporabljene pakete uvrščamo:

- **Numpy** [33] – odprtokodni paket, ki v Python prinese podatkovni tip polja (angl. *array*), in veliko vgrajenih funkcij, ki bistveno pospešijo operacije, ki delajo s polji in matrikami. V področju podatkovne analitike in umetne inteligence, paket s polji predstavlja enega izmed glavnih temeljev. Veliko je tudi paketov, ki za svoje delovanje uporabljajo ravno Numpy. Tak primer je npr. Pandas [34] – gre za enega izmed najpogosteje uporabljenih paketov za podatkovno analitiko, ki s seboj prinese še nekaj dodatnih abstrakcij za lažje delo s podatkovnimi okviri (angl. *data frame*).
- **Tensorflow** [1] – odprtokodni paket za strojno učenje. V osnovi je celoten produkt Tensorflow namenjen širšemu spektru programskih jezikov in ni implementiran le za Python. Gre za splošno namensko rešitev z že pripravljenimi metodami za kreiranje modelov strojnega učenja, kar olajša delo programerju, saj je veliko korakov avtomatiziranih.

- **UltraJSON** [51] – gre za manj poznan paket, ki dela z JSON strukturami. JSON je postal najbolj popularen format za prenos strukturiranih podatkov preko HTTP [36]. Gre za format, kjer je shema strukture enkapsulirana med podatke, zaradi česar je format prostorsko zelo potraten, hkrati pa je zaradi nedefinirane strukture tudi obdelava relativno počasna. Ravno zaradi tega je kakršnakoli optimizacija v smislu obdelave podatkov zelo dobrodošla in to je tudi glavni cilj paketa.

Vsi paketi so v osnovi napisani s pomočjo Python.h vmesnika. Lahko bi rekli, da gre za most, ki poveže jezika Python in C. Na nivoju programskega jezika C je vsak tip iz programskega jezika Python predstavljen s strukturo `PyObject` [43], katerega lahko neposredno uporabljamo in manipuliramo na nivoju programskega jezika C.

4.1.2 Naprednejša rešitev za lažji prehod med jezikoma

Izdelava paketov s pomočjo Python.h vmesnika pomeni uporabo programskega jezika C, kar pomeni, da je velik del implementacij različnih struktur prepuščen razvijalcu.

Za lažjo implementacijo rešitve smo se odločili za razvoj v programskem jeziku C++, ki ima bistveno širši nabor struktur in funkcij v standardni knjižnici.

Za prehod smo uporabili PyBind11 [41], ki deluje na nivoju višje in za svoje delovanje v ozadju uporablja Python.h. Gre za rešitev, ki olajša pretvorbe podatkovnih struktur med jezikoma npr. `std::vector` [53], se samodejno pretvori v strukturo seznama na nivoju programskega jezika Python.

Poleg avtomatskih pretvorb, pa imamo možnost uporabe vseh tipov in metod, ki so vključene v Python.h vmesnik.

Primer uporabe PyBind11

V segmentu kode 4.1 imamo še preprost primer funkcije za seštevek dveh celih števil z uporabo PyBind11.

```
#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");
}
```

Programska koda 4.1: Primer uporabe PyBind11 za seštevanje dveh celih števil.

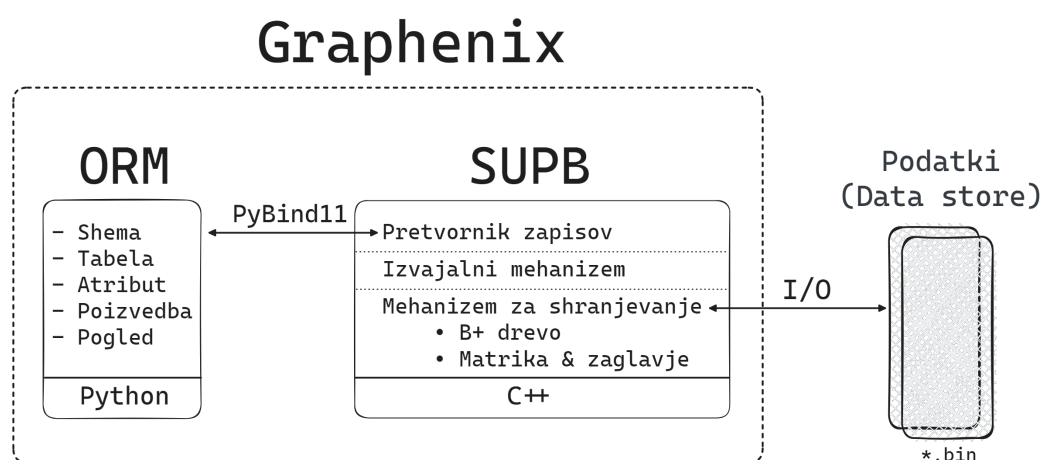
Pod pogojem, da je paket pravilno nameščen, lahko na nivoju programskega jezika Python uporabimo funkcijo `custom_sum` 4.2.

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

Programska koda 4.2: Uporaba paketa na nivoju programskega jezika Python.

4.2 Arhitektura rešitve Graphenix

Na sliki 4.1 je diagram, ki prikazuje arhitekturo rešitve z vsemi pomembnimi komponentami in orodji, ki omogočajo komunikacijo in zanesljivo delovanje Graphenix rešitve.



Slika 4.1: Arhitektura paketa Graphenix.

Paket je sestavljen iz dveh ključnih komponent. To sta ORM, ki je napisan v programskem jeziku Python in predstavlja tudi vhodno točko za uporabnike paketa. Na drugi strani pa je jedro SUPB s sistemom za shranjevanje podatkov. ORM in SUPB komunicirata preko prej omenjenega paketa PyBind11 [41]. Za preslikavo zapisov in poizvedb na nivoju SUPB skrbi pretvornik, ki sestavi strukture s katerimi potem upravlja izvajalni mehanizem znotraj jedra.

SUPB vsebuje vse performančno gledano kritične operacije in podatke s pomočjo mehanizma za shranjevanje drži v datotekah *.bin, ki predstavljajo zunanjo komponento sistema.

4.3 Testno usmerjen razvoj

Da zagotovimo pravilno delovanje nekega bolj zahtevnega sistema, je na področju razvoja programske opreme zelo priporočen testno usmerjen razvoj. Gre za način validacije pravilnega delovanja posameznih komponent sistema najprej kot ločene enote in nato kot del večjega sistema, kjer testiramo usklajenost delovanja več komponent hkrati.

V primeru razvoja izbrane programske opreme smo proces testiranja razdelili na tri faze: testiranje enot, integracijsko testiranje in performančno testiranje.

4.3.1 Testi enot na nivoju SUPB in mehanizma za shranjevanje

V prvi fazi želimo zagotoviti pravilno delovanje na najnižjem nivoju oz. na nivoju mehanizma za shranjevanje in jedra SUPB. Gre za testiranje najbolj osnovnih funkciji, ki zagotavljajo pravilno delovanje manjših komponent. V tem segmentu je bilo največ pozornosti na testiranju vstavljanja in iskanja v implementaciji B drevesa. Avtomatsko testiranje je na tem nivoju realizirano s pomočjo paketa Doctest [10]. Gre za testno ogrodje napisano za programski jezik C++, ki omogoča fleksibilen pristop validacije rezultatov iz posameznih funkciji.

4.3.2 Integracijsko testiranje funkcionalnosti

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 [6]. 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 tabelo, kjer imamo za določen atribut nastavljen indeks. Testiranje je izvedeno na nivoju programskega jezika Python z vgrajenim paketom

`unittest` [52].

4.3.3 Performančno testiranje zahtevnejših akciji

Izvedba tretje faze je sicer zelo podobna fazi integracijskega testiranja, z izjemo, da ne posvetimo toliko pozornosti 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. Vsak test se izvede petkrat, kjer ob koncu izračunamo povprečni čas izvajanja.

4.3.4 Integracija avtomatskega testiranja z GitHub

Platforma GitHub omogoča izvajanje različnih akcij ob spremembah znotraj repozitorija. V praksi to najpogosteje počnemo, kadar izvajamo združevanje vej ali kar ob vsaki posodobitvi kode. Poleg avtomatskega testiranja na tem področju se lahko izvajajo tudi avtomatske posodobitve strežniške aplikacije ali različne periodične naloge, kot npr. kreiranje varnostnih kopij podatkovne baze iz produkcijskega strežnika.

V našem primeru se ob vsaki posodobitvi kode izvede akcija za preverjanje pravilnega delovanja paketa. Akcija se izvede za dve različici programskega jezika Python (v3.10 in v3.11). Postavitev okolja pa se izvede na virtualnem Linux okolju.

Ob vzpostavitvi okolja izvedemo zaporedje manjših akcij, ki postopno validirajo pravilno delovanje celotnega paketa.

1. Zagon virtualnega okolja za programski jezik Python - venv [54].
2. Namestitev paketa PyBind11 [41].
3. Namestitev jedra SUPB, mehanizma za shranjevanje & klic testne metode.

4. Namestitev paketa, ki vsebuje ORM in operira z jedrom.
5. Izvedba nizkonivojskih testov oz. testov enot v programskem jeziku C++.
6. Izvedba integracijskih testov na nivoju programskega jezika Python.
7. Izvedba performančnih testov.

Takoj, ko se katera izmed akcij 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 le-ta temelji na hitrem in varnem razvoju, ki razvijalca hitro obvesti o morebitnih napakah, ki se pojavljajo znotraj sistema.

Poglavje 5

Analiza delovanja

V tem poglavju se bomo posvetili temeljni analizi, ki vključuje časovno in prostorsko primerjavo. Osredotočili se bomo na primerjavo med razvitim paketom Graphenix ter ustaljenima rešitvama SQLite in MySQL. Analizo bomo izvedli na različnih scenarijih, pri čemer se bomo osredotočili predvsem na branje podatkov. Poleg tega bomo preučili 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 in slabosti posameznih rešitev v različnih kontekstih.

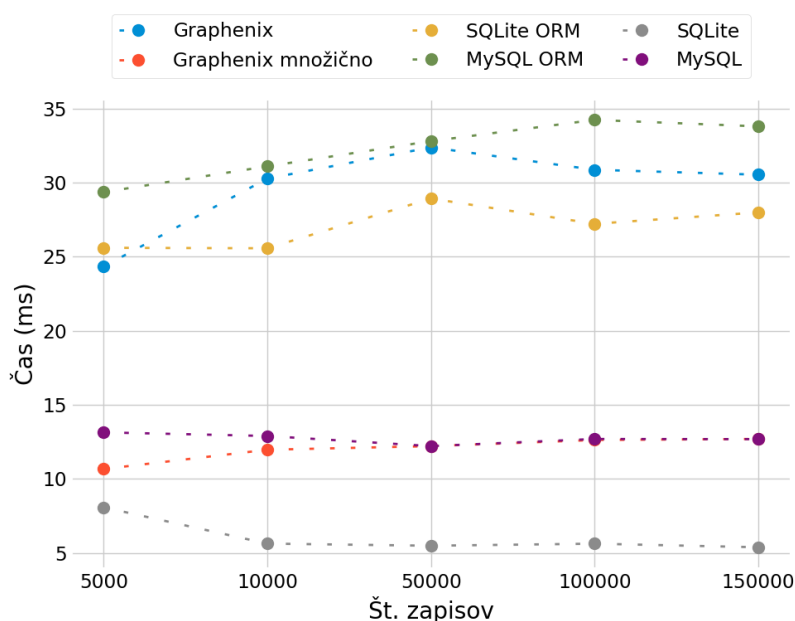
Za izvajanje različnih testnih scenarijev je pripravljena skripta, ki vsak scenarij izvede na različno velikih podatkovnih bazah. Hkrati pa skripta izvede tudi izračun povprečnega časa, kjer med desetimi izvedbami izloči dve najpočasnejši in dve najhitrejši, ter iz preostalih meritev izračuna povprečen čas izvajanja.

5.1 Množično vstavljanje podatkov

V okviru prvega scenarija izvajamo postopek vstavljanja podatkov v tabelo uporabnikov. Skozi postopek povečujemo število zapisov, ki jih sistematično vnašamo v podatkovno bazo.

Na grafu 5.1 so prikazani povprečni časi vstavljanja. Za vsak pristop je

prikazan čas vstavljanja 1000 zapisov, izražen v milisekundah.



Slika 5.1: Množično vstavljanje podatkov.

Prva ugotovitev, ki jo lahko izpeljemo iz grafa 5.1 je, da je vstavljanje podatkov z uporabo ORM precej počasnejše v primerjavi z neposrednim vstavljanjem preko SQL ukazov. Uporaba ORM zahteva ustvarjanje instanc razredov, kar vodi v dodatne časovne zakasnitve.

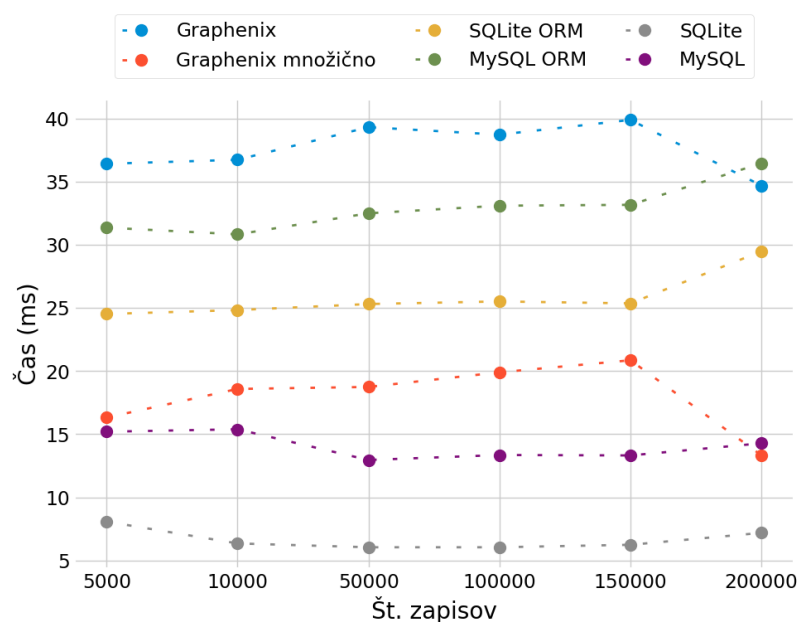
Na splošno lahko opazimo, da je SQLite prevladujoč pri hitrosti vstavljanja. Razlog, da MySQL ni tako hiter, je dokaj enostaven. MySQL teče na strežniku in vso komunikacijo opravlja preko vgrajenega protokola, kjer se podatki prenašajo z uporabo TCP/IP paketov, kar ni tako učinkovito, kot uporaba vhodno/izhodnih operaciji v SQLite.

V primeru Graphenix, vstavljanje ni izvedeno v klasični množični obliki, saj je vstavljanje vsakega zapisa ločen dostop do datoteke. V analizi sta predstavljena dva pristopa vnašanja z uporabo Graphenix. V primeru množičnega vnašanja gre za direkten klic funkcije jedra za vnašanje podatkov in ne potrebujemo kreirati instanc razredov, kar je bilo tudi glavno ozko grlo pri vnašanju podatkov z uporabo ORM.

Pomemben dejavnik pri vnašanju podatkov je tudi dodatna varnost, ki jo zagotavljata SQLite in zlasti MySQL. Oba sistema imata dnevnike za povrnitev (angl. *rollback journal*), ki v primeru napake omogočajo povrnitev celotne baze podatkov v stanje pred vnašanjem. Posledično v primeru napake ali prekinitve v Graphenix tvegamo imeti neveljavne zapise v bazi podatkov.

5.1.1 Vstavljanje z dodatnim indeksiranim poljem

V drugem scenariju gre za enakovredno vstavljanje prvemu, vendar z dodatkom uporabe indeksa na atributu tabele (indeks je definiran na celoštevilskem polju, kjer je vsaka vrednost unikatna).



Slika 5.2: Množično vstavljanje z dodatnim indeksiranim atributom.

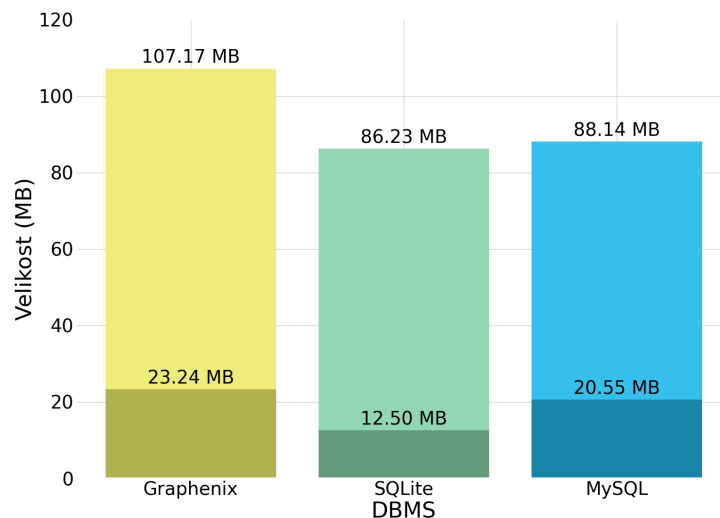
Iz grafa 5.2 lahko razberemo, da z vidika hitrosti vstavljanja še vedno prevladuje SQLite. Za razliko od predhodnega scenarija 5.1, pa je tokrat MySQL v večini primerov postal bolj učinkovit kot Graphenix.

V vseh treh SUPB je indeksiranje realizirano z uporabo B drevesa. Kljub temu pa je razlika v načinu vstavljanja podatkov, saj naša implementacija

B drevesa ne izkorišča algoritma za množično vstavljanje v drevo. MySQL in SQLite za vstavljanje uporabljata algoritem množičnega vstavljanja in posledično čas vstavljanja v drevo nima bistvenega vpliva na celoten čas vstavljanja.

5.2 Velikosti podatkovnih baz na disku

Glede na prva dva scenarija smo pripravili tudi stolpčni grafikon 5.3, kjer primerjamo porabo prostora na disku posameznega SUPB (gre za baze podatkov z 1×10^6 zapisi).



Slika 5.3: Prostorska poraba na nivoju diska posameznega SUPB.

Iz grafa 5.3 razberemo, da je Graphenix najbolj potraten z vidika porabe prostora. Razlog se skriva v načinu shranjevanja tekstovnih vrednosti. SQLite in MySQL uporabljata podatkovne strukture, ki ob shranjevanju prinesejo dodatno optimizacijo porabe prostora, saj uporabita le toliko prostora kot je potrebno. V primeru Graphenix, pa je dolžina tekstovnih polj točno določena, in v primeru, da polje definiramo kot `String(size=50)`, vsak zapis ne glede na dolžino zavzame 50 bajtov.

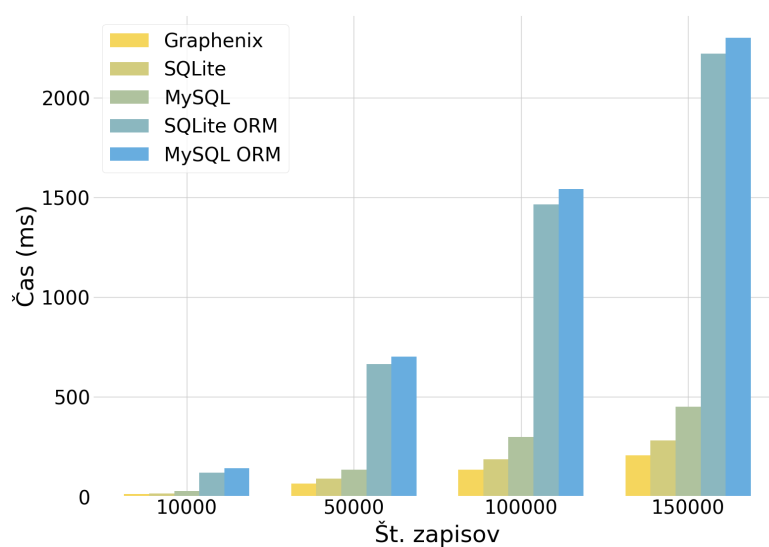
V temnem delu stolpca je prikazana razlika velikosti z in brez uporabe

indeksne strukture. SUPB z najnižjo porabo prostora z vidika velikosti indeksne strukture je SQLite. Razlog za nižjo porabo se nahaja v velikosti posameznega vozlišča in algoritmu vstavljanja v drevo, ki vpliva tudi na delitev vozlišč tekom vstavljanja.

5.3 Primerjava poizvedb na eni tabeli

Za tretji scenarij je pripravljena analiza, kjer izmerimo čas izvajanja na preprosti poizvedbi branja brez kakršnihkoli omejitev.

```
SELECT * FROM users
```



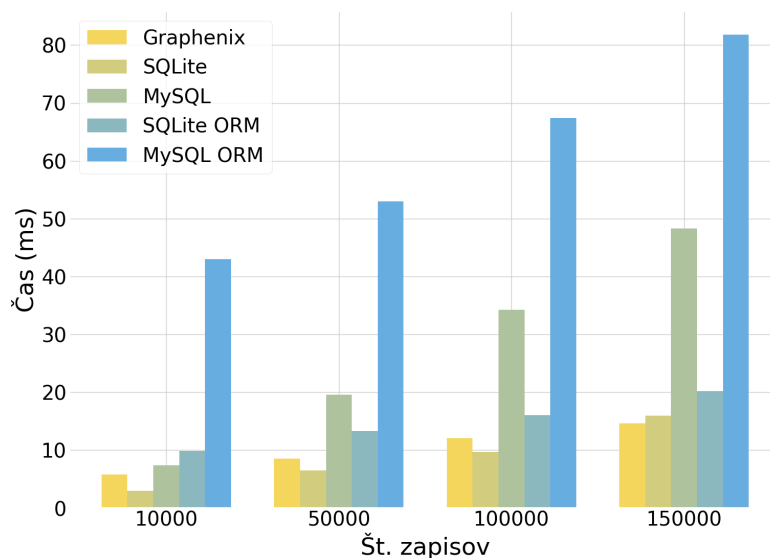
Slika 5.4: Branje brez dodatnih parametrov znotraj poizvedbe.

Iz grafa 5.4 je jasno razvidno, da uporaba ORM ni dobro pripravljena za masovno nalaganje podatkov, saj se ponovno kreirajo instance kar predstavlja več kot 70 % delež celotnega branja. V primeru MySQL se ponovno pojavi tudi nekaj dodatne zakasnitve zaradi načina prenosa podatkov s TCP/IP protokolom.

5.3.1 Branje z omejitvami

V drugem scenariju branja podatkov iz ene tabele dodamo še omejitev na tip uporabnikov (želimo le administratorje). Poleg tega pa je dodano tudi urejanje podatkov in omejitev števila zapisov (želimo prvih 500 zapisov urejenih po točkah uporabnika).

```
SELECT * FROM users WHERE is_admin = 1  
ORDER BY points, id LIMIT 500
```



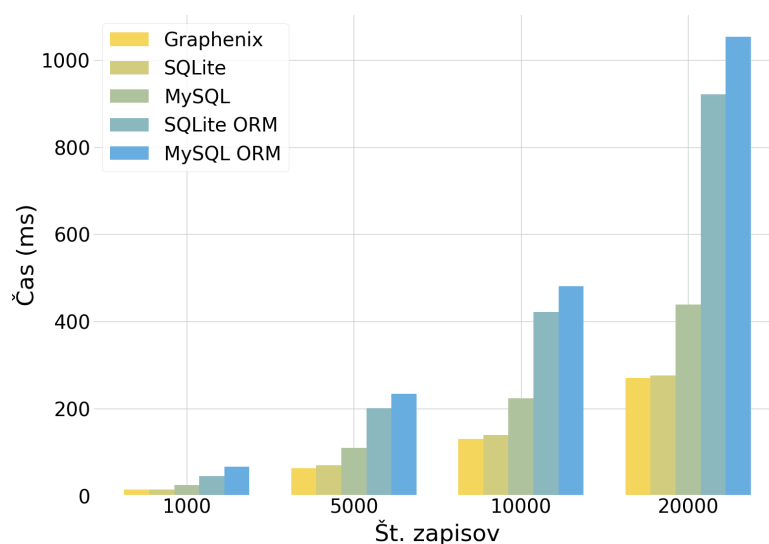
Slika 5.5: Branje z dodanim filtriranjem in urejanjem zapisov.

Čas izvajanja posameznega branja je relativno nizek, v vseh primerih je bilo branje izvedeno v manj kot 100 ms. Opazimo, da zakasnitve kreiranja instanc pri ORM tukaj ne pridejo do izraza, saj je potrebno kreirati le 500 instanc. Do večjega izraza pridejo zakasnitve, ki jih za prenos podatkov prinaša TCP/IP protokol, kar je na grafu 5.5 videno v primerih uporabe MySQL.

5.4 Primerjava poizvedb z uporabo relacij

V scenariju definiramo poizvedbo, kjer izvedemo nabor podatkov iz dveh tabel. Število zapisov je določeno s številom uporabnikov. Za vsakega uporabnika pa je bilo vnesenih še 10 nalog, ki so vezane preko tujega ključa `user_id` v tabeli `nalog` (`Task`).

```
SELECT * FROM users
INNER JOIN tasks ON tasks.user_id = users.id
```



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

Ponovno se izkaže, da ORM s seboj prinese dodatne zakasnitve zaradi kreiranja instanc. Za razliko od prejšnjih scenarijev pride tukaj do razlike v sami strukturi rezultata.

- ORM – podatki so v gnezdjeni obliki, kjer imamo za vsakega izmed uporabnikov seznam njegovih nalog.
- Brez – podatki so predstavljeni v obliki matrike, kjer imamo za vsakega uporabnika toliko zapisov, kot ima uporabnik sporočil.

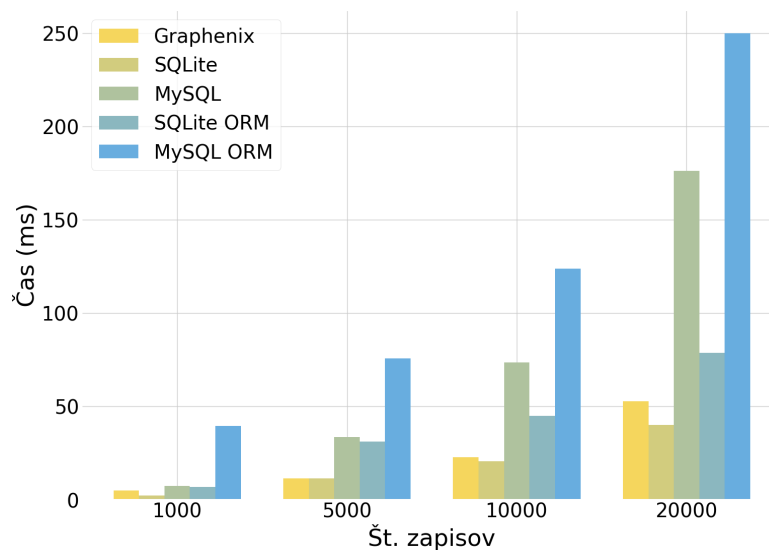
Graphenix deluje po principu ORM in imamo gnezdeno strukturo podatkov. Kljub temu pa zaradi združevanja na nivoju programskega jezika C++ poizvedba steče v primerljivem času s poizvedbami, ki ne uporabljajo ORM.

5.5 Izvedba agregacijske poizvedbe

V scenariju izvedbe agregacije uporabljamo strukturo, ki smo jo pripravili v prejšnjem scenariju 5.4.

V poizvedbi naloge združujemo po uporabnikih in za vsakega uporabnika pridobimo število nalog, ki mora biti v vseh primerih 10. Poleg tega pa glede na atribut `created_at` pridobimo tudi datum zadnje naloge uporabnika.

```
SELECT user_id, COUNT(*) AS 'count',  
       MAX(created_at) AS 'latest'  
FROM tasks GROUP BY user_id
```



Slika 5.7: Uporaba agregacijskih metod.

V tem scenariju rezultat poizvedbe ni predstavljen kot instance razredov, saj vedno kreiramo prilagojen tip rezultata, ki vsebuje identifikator uporabnika, število nalog in zadnji datum naloge za vezanega uporabnika.

Rezultati sicer dobro sledijo trendom iz predhodnih scenarijev – MySQL zaradi načina prenosa podatkov potrebuje nekoliko več časa, poleg tega pa se ponovno pozna razlika uporabe ORM in uporaba poizvedb SQL. Graphenix pa je ponovno primerljiv uporabi SQLite brez ORM.

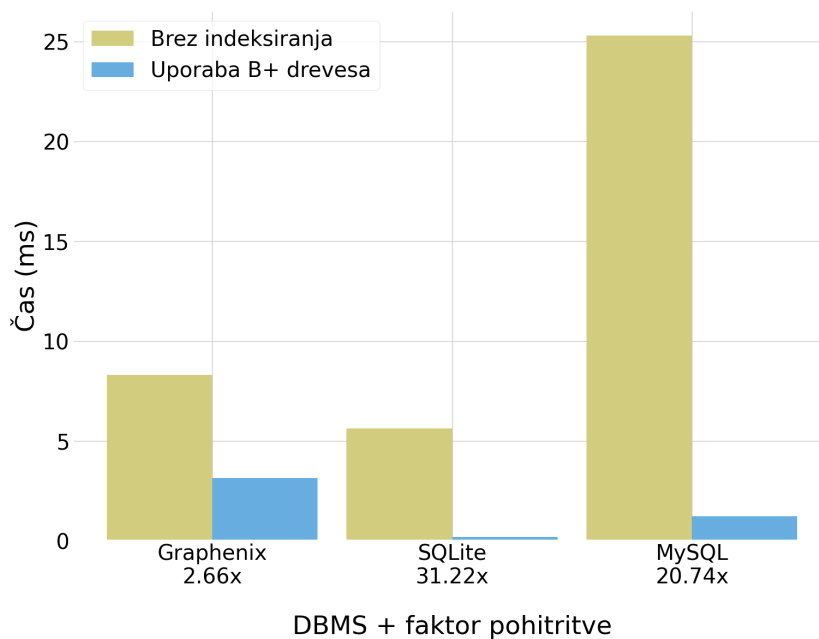
5.6 Pohitritve s pomočjo indeksiranja

Kot zadnji scenarij je predstavljen eden izmed najbolj kritičnih segmentov SUPB – pohitritev iskanja s pomočjo indeksiranja podatkov. V scenariju primerjamo pristop brez in z uporabo indeksiranja.

```
SELECT * FROM users
```

```
WHERE points = 5432
```

Na grafu 5.8 je prikazano iskanje zapisa v tabeli z 1×10^5 zapisi.



Slika 5.8: Iskanje specifičnega zapisa ($N = 1 \times 10^5$).

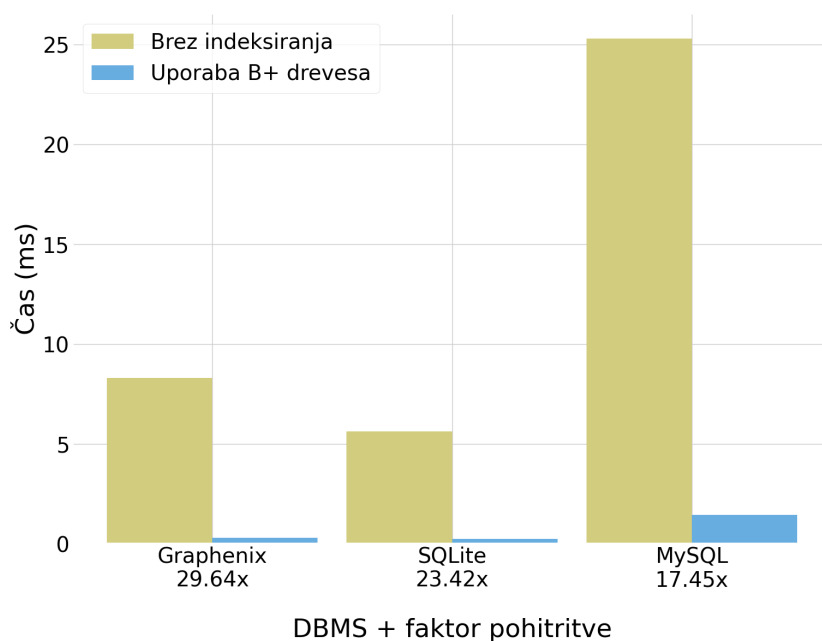
Iz grafa 5.8 hitro razberemo, da je pohitritev v Graphenix SUPB drastično manjša, kot je to v primeru SQLite in MySQL.

5.6.1 Optimizacija v metodi filtriranja zapisov

Zaradi bistveno slabšega rezultata pri iskanju smo se lotili profiliranja posameznih segmentov programske kode za nabor podatkov.

Kaj hitro smo prišli do ugotovitve, da težava ni bil proces iskanja v B drevesu, temveč je bila težava branje zaglavja tabele. Metoda za branje podatkov iz posamezne tabele je delovala tako, da je najprej prebrala celotno datoteko zaglavja tabele in za posamezen identifikator zapisa pridobila še odmik v matriki zapisov. Izkazalo se je, da je ta proces v scenariju predstavljal nekje $\approx 85\%$ izvajalnega časa.

V namen hitrejšega izvajanja se branja celotnega zaglavja, v primeru, da imamo že vnaprej določene identifikatorje, ki jih iščemo in je teh manj kot `IX_THRESHOLD`, izognemo. Konstanta predstavlja mejo za branje celotnega zaglavja in je v trenutni implementaciji nastavljena na 100.

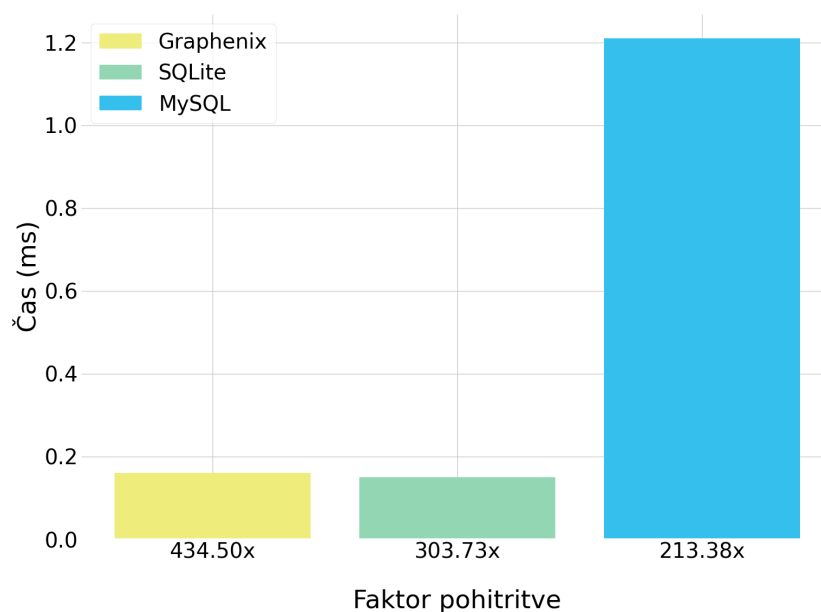


Slika 5.9: Iskanje specifičnega zapisa ($N = 1 \times 10^5$) z optimizacijo.

Po implementirani optimizaciji opazimo bistveno izboljšavo časa iskanja. Sedaj se čas in faktor pohitritve lahko primerjata z MySQL in SQLite.

5.6.2 Analiza iskanja nad večjo bazo podatkov

Na grafu 5.10 je prikazana še pohitritev enake poizvedbe, kot v predhodnem scenariju. V tem primeru pa je število vseh uporabnikov 1×10^6 .



Slika 5.10: Iskanje specifičnega zapisa ($N = 1 \times 10^6$) z optimizacijo.

Iz grafa 5.10 razberemo, da v vseh treh izvedbah zapis najdemo v manj kot 2 ms, kar tudi v vseh treh scenariji predstavlja več 100x pohitritev. Nekaj zakasnitve se sicer pojavi iz strani MySQL, kar je glede na predhodne scenarije pričakovano.

Idealni pogoji za indeksiranje

Omenimo, da je scenarij prilagojen uporabi indeksiranja, saj v scenariju iskanja, kjer vrednosti niso unikatne, učinkovitost indeksa zelo hitro pade. Poleg tega je bilo iskanje izvedeno kot iskanje specifične vrednosti v drevesu, medtem ko če bi iskali širši nabor vrednosti bi časovna zahtevnost $O(\log_b(N))$ postala $O(K \cdot \log_b(N))$, kjer je K število vrednosti, ki se ujemajo z iskalnim kriterijem.

Če potegnemo črto je jasno, da se pri odločanju o uporabi indeksov odpira zahtevno vprašanje, ki zahteva preiščeno obravnavo. Odgovornost za to odločitev se prepušča razvijalcem, pri čemer je ključnega pomena, da se kot razvijalci zavedamo optimizaciji in pasti, ki izvirajo iz uporabe B dreves za indeksiranje. Poleg tega je nujno, da poznamo naravo podatkov in zahtev aplikacije, saj je optimizacija z indeksiranjem smiselna le v določenih kontekstih.

Poglavje 6

Scenariji oz. primeri uporabe

V tem poglavju sta predstavljena dva primera uporabe paketa Graphenix. V prvem primeru se paket uporablja kot dodatek strežniški aplikacij v namen beleženja podatkov (angl. *logging*). V drugem scenariju pa je predstavljen razvoj preproste namizne aplikacije, kjer paket uporabljamo za trajno shranjevanje podatkov. Oba primera skušata na praktičen način predstaviti dobre strani uporabe rešitve Graphenix.

6.1 Strežniško beleženje podatkov

Pogosta uporaba programskega jezika Python je na strani strežniških aplikacij. Na tem področju je beleženje sprememb in zahtev kritično. Z uporabo pripravljenih rešitev je prilagojeno beleženje podatkov lahko zelo preprosto.

6.1.1 Modul beleženja

V prvi fazi smo si za primer pripravili preprosto strežniško aplikacijo, ki uporablja Flask [12] ogrodje. Znotraj aplikacije smo definirali eno vhodno točko, in sicer `get-range`, ki je predstavljena v programski kodi 6.1, kjer lahko poljubno nastavljamo tri HTTP GET parametre, to so `start`, `end` in `step`, kjer ima vsak od teh parametrov tudi privzete vrednosti.

```
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()
```

Programska koda 6.1: Vhodna metoda za strežniško aplikacijo.

6.1.2 Kreiranje sheme za beleženje podatkov

V namen beleženja podatkov smo si najprej pripravili tabelo in shemo za shranjevanje podatkov, kar je prikazano v programski kodi 6.2.

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

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

Programska koda 6.2: Shema za beleženje podatkov.

Tabela ReqInfo skupaj z definiranimi atributi predstavlja strukturo podatkov, ki jih bomo tekom delovanja naše aplikacije beležili. V drugi fazi pa moramo kreirati še shemo, v kateri definiramo naziv in vse razrede oz. tabele, ki bodo del naše baze podatkov.

6.1.3 Dekorator za dinamično dodajanje beleženja

V namen beleženja znotraj vhodnih točk strežniške aplikacije smo uporabili pristop z uporabo dekoratorja. Dekorator je vzorec, ki omogoča dinamično dodajanje funkcionalnost obstoječi funkciji, ne da bi spreminjali njeno strukturo. Med izvedbo notranje funkcije lahko argumente in različne segmente izvajanja prestrežemo in jih prilagodimo glede na naše zahteve [27].

```
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
```

Programska koda 6.3: Dekotratore za implementacijo avtomatskega beleženja.

V primeru našega dekoratorja `@route_with_log` vhodni točki dodamo še

funkcionalnost beleženja podatkov.

6.1.4 Branje zapisanih podatkov

V končni fazi je glavna prednost uporabe pripravljene rešitve dobra struktura shranjenih podatkov, ki omogoča preprosto obdelavo in manipulacijo. Posledično nam je omogočeno tudi bistveno več načinov pridobivanja statističnih podatkov, saj lahko le-te poljubno filtriramo, grupiramo, združujemo in razvrščamo. Na nivoju administratorske aplikacije lahko nato prožimo različne poizvedbe nad našo bazo podatkov.

Izpis zadnjih treh zahtevkov

```
_, reqs = ReqInfo.order(ReqInfo.timestamp.desc()).limit(3).all()
...
# izpis zahtevkov
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 SUPB tudi grupiramo, in sicer z uporabo `.agg(...)` metode v poizvedbi. Znotraj metode definiramo polje po katerem grupiramo podatke, kot tudi agregacije, ki jih želimo izvesti.

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

S poizvedbo dobimo število zahtevkov grupiranih po vhodni točki aplikacije.

Izpis napak v zadnjem dnevu

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

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

Programska koda 6.4: Poizvedba za nabor napak v zadnjem dnevu.

Preslikava v podatkovne okvirje

Za namen podatkovne analitike v programskem jeziku Python najpogosteje uporabljamo rešitve, ki za svoje delovanje uporabljajo podatkovne okvirje. Ena taka rešitev je že omenjeni Pandas [34].

Leta 2021 se je pojavil nov paket za podatkovno analitiko – Polars [38]. Za svoje delovanje uporablja prevedeni programski jezik, vendar v tem primeru ne gre za C/C++, temveč za programski jezik Rust [24].

Prednost uporabe rešitve Graphenix za beleženje je v načinu branja podatkov, saj podatke zlahka filtriramo in urejamo že preden ustvarimo podatkovni okvir, kar klasične datoteke `.csv`, ki pogosto predstavljajo format shranjenih podatkov, ne omogočajo.

```
import pandas as pd
import polars as pl

_, qview = ReqInfo.order(ReqInfo.timestamp)\
    .filter(ReqInfo.timestamp.greater_or_equal(
        datetime(2020, 1, 1)))\
    .limit(10000)\
    .all()

pandas_df : pd.DataFrame = qview.as_pandas_df()
polars_df : pl.DataFrame = qview.as_polars_df()
```

Programska koda 6.5: Preslikava rezultata poizvedbe v podatkovna okvirja.

V programski kodi 6.5 podatkovna okvirja napolnimo z zapisi, ki so bili ustvarjeni po začetku leta 2020 in od tega vzamemo prvih 10 000 zapisov urejenih po datumu.

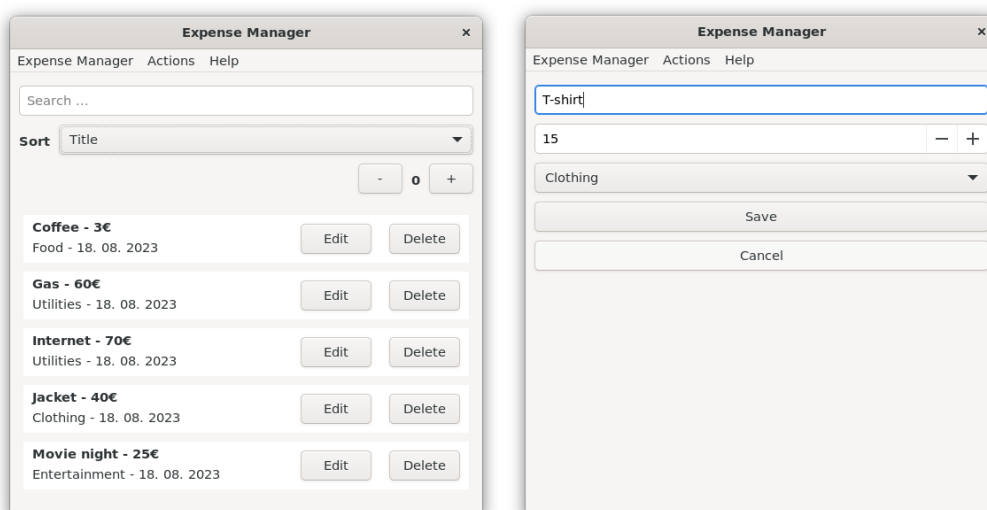
Od te točke dalje pa lahko podatke v celoti obdelujemo s pomočjo paketov za podatkovno analitiko.

6.2 Namizna aplikacija za upravljanje stroškov

V tem razdelku je predstavljen razvoj namizne aplikacije za upravljanje stroškov. Aplikacija omogoča pregled stroškov, vstavljanje, urejanje, brisanje in statističen pregled, kjer so stroški grupirani glede na kategorije.

6.2.1 Zaslonski pregled aplikacije

Na sliki 6.1 je na levi prikazan začetni zaslon, kjer imamo v obliki seznama pregled vseh stroškov uporabnika in ob tem še zaslon za urejanje ali kreiranje novega stroška.



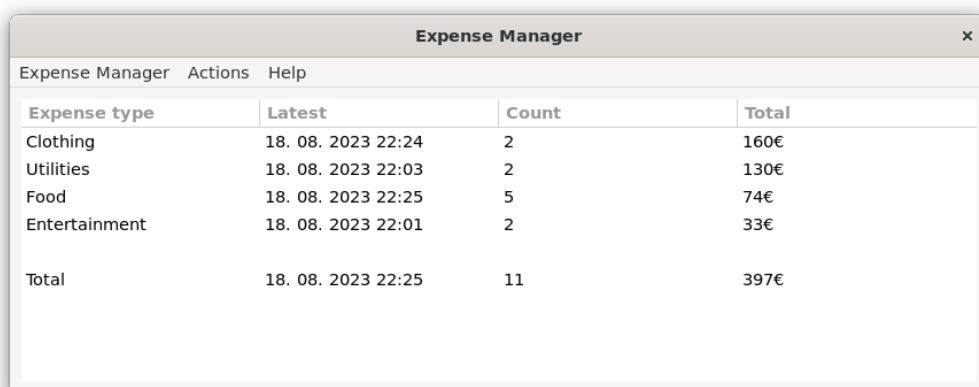
Slika 6.1: Seznam stroškov z možnostjo vstavljanja in urejanja.

Za vsak posamezen strošek sta na skupnem prikazu definirani akciji za brisanje in urejanje. Poleg tega lahko stroške glede na naziv, ceno in kategorijo tudi uredimo. Omogočeno je tudi filtriranje po nazivu in prikazovanje po straneh.

Na desnem zaslonu je prikazan obrazec, do katerega dostopamo preko akciji za urejanje in kreiranje novega stroška. Gre za preprost obrazec, kjer določimo naziv, ceno in kategorijo posameznega stroška.

Statističen pregled stroškov

Na sliki 6.2 je prikazan statistični pregled stroškov, kjer so ti grupirani glede na kategorijo. Gre za tabelarni prikaz, kjer imamo za vsako kategorijo naziv, datum zadnjega stroška, število stroškov in vsoto cen vseh stroškov posamezne kategorije.



The screenshot shows a window titled "Expense Manager" with a menu bar containing "Expense Manager", "Actions", and "Help". Below the menu bar is a table with four columns: "Expense type", "Latest", "Count", and "Total". The table contains five rows of data, including a total row at the bottom.

Expense type	Latest	Count	Total
Clothing	18. 08. 2023 22:24	2	160€
Utilities	18. 08. 2023 22:03	2	130€
Food	18. 08. 2023 22:25	5	74€
Entertainment	18. 08. 2023 22:01	2	33€
Total	18. 08. 2023 22:25	11	397€

Slika 6.2: Statističen pregled stroškov.

6.2.2 Aplikacija z vidika kode

Grahepnix je dobra izbira, saj gre za relativno nezahtevno aplikacijo, kjer ni potrebno skrbeti za več hkratnih dostopov do baze podatkov, poleg tega pa pripravljen ORM omogoča preprost pristop za manipulacijo podatkov.

Za izris uporabniškega vmesnika aplikacija uporablja BeeWare [5], ki omogoča medplatformski (angl. *Cross-platform*) razvoj programske opreme, kar pomeni, da lahko isto kodo uporabimo za več različnih operacijskih sistemov.

Struktura podatkovne baze

V namen beleženja stroškov je definiran razred *Invoice*, kjer določimo naziv, ceno, datum in kategorijo posameznega stroška. Poleg tega pa imamo še tabelo, ki služi kot šifrant za kategorije stroškov. Podatki v kategorijah se napolnijo že ob kreiranju podatkovne baze z avtomatskim uvozom šifrantov.

```
class Invoice(gx.Model):
    title = gx.Field.String(size=20)
    amount = gx.Field.Int()
    day = gx.Field.DateTime()
    expense_type = gx.Field.Link()
```

```
class ExpenseType(gx.Model):
    name = gx.Field.String(size=30)
```

Programska koda 6.6: Struktura za uporabo v namizni aplikaciji.

Metoda za statistični nabor podatkov

```
def refresh_stats(self):
    agg_data = Invoice.agg(by=Invoice.expense_type,
                          count=gx.AGG.count(),
                          amount=gx.AGG.sum(Invoice.amount),
                          latest=gx.AGG.max(Invoice.day))

    data = sorted(agg_data, key=lambda x: -x.amount)
    ...
```

Programska koda 6.7: Metoda za osvežitev statističnega prikaza.

V metodi 6.7 najprej izvedemo agregacijsko poizvedbo, kjer podatke grupiramo glede na kategorijo in pridobimo število zapisov, datum zadnjega stroška in vsoto cen pri posamezni kategoriji.

Za zaključek podatke ločeno uredimo glede na skupno ceno, saj ta funkcionalnost za agregacijske metode v Graphenix SUPB ni podprta.

Nabor podatkov za glavni prikaz stroškov

```
def display_expenses(self, search=None, order_by=None, page=0):
    query = Invoice.link(expense_type=ExpenseType).filter(
        Invoice.title.iregex(f'.*{search or ""}.*'))

    match order_by:
        case 'Title':
            query = query.order(Invoice.title)
        case 'Amount':
            query = query.order(Invoice.amount.desc())
        case _:
            query = query.order(Invoice.day.desc())

    _, invoices = query.offset(PAGE_SIZE * page)\
        .limit(PAGE_SIZE).all()

    for invoice in invoices:
        self.items_box.add(self.make_card(invoice))
```

Programska koda 6.8: Metoda za nabor in prikaz stroškov.

Za prikazovanje vseh stroškov uporabljamo metodo `display_expenses` 6.8, ki prejme parameter za iskanje, urejanje in stran, ki jo mora prikazati. Prikazan je kodni vzorec grajenja (angl. *builder pattern*), kjer postopoma dodajamo dodatne lastnosti na objekt poizvedbe.

1. Na stroške povežemo kategorije in za iskanje uporabimo neobčutljiv **regex** na male in velike črke, kjer pogoj zahteva, da naziv stroška vsebuje iskani niz `search`.
2. Parameter za urejanje preslikamo v določeno ureditev ali kot privzeto vrednost vzamemo padajoče urejanje po datumu.

3. S pomočjo argumenta `page` določimo stran, ki jo želimo prikazati. Nato v poizvedbi preskočimo $\text{PAGE_SIZE} \times \text{page}$ stroškov, kjer je `PAGE_SIZE` konstanta, ki določa velikost strani. Hkrati pa dodamo še omejitev števila zapisov (vzamemo le prvih `PAGE_SIZE` zapisov).

Za konec se le še sprehodimo skozi rezultat poizvedbe in za vsak strošek kreiramo element na uporabniškem vmesniku.

Poglavje 7

Sklepne ugotovitve

V zaključenem poglavju so najprej na kratko predstavljene možnosti za izboljšavo paketa Graphenix, ki bi rešitev približale nivoju, ki je potreben za uporabo v produkcijskem okolju.

V razdelku 7.2 pa je na kratko opisana celotna izkušnja razvoja programske rešitve.

7.1 Nadaljnji razvoj

Glede na širino področja upravljanja podatkovnih baz imamo na voljo obsežen spekter izboljšav. V okviru našega dela smo našli številne možnosti za nadgradnjo, ki bi lahko obogatile funkcionalnost in zmogljivosti celotne rešitve.

1. Izboljšane indeksiranja

Indeksiranje je v razviti programske opreme sicer implementirano in doseže primerljive rezultate z uveljavljenimi sistemi za upravljanje baz podatkov, ki smo jih vzeli za primerjavo.

Kljub temu pa ostaja še ogromno scenarijev, ki jih z našo implementacijo nismo podprli. To je najbolj razvidno v množičnih akcijah, kot npr. iskanje več zapisov in množično vstavljanje v B+ drevo.

2. Prostorska optimizacija shranjevanja nizov

Iz prostorske analize v razdelku 5.2 je razvidno, da tako MySQL kot tudi SQLite ponujata prostorsko optimizacijo pri shranjevanju nizov.

- SQLite nize definira s podatkovnim tipom `TEXT`, ki besedilo shranjuje v ločeni strukturi od zapisov in po potrebi zapise v tej ločeni strukturi premika.
- MySQL ima možnost uporabe podatkovnega tipa `VARCHAR` [30], kjer lahko definiramo maksimalno dolžino niza. Ta dolžina ne pomeni dejanske dolžine, ki jo niz porabi na disku, temveč MySQL SUPB izvaja dinamično alokacijo za nize. Uporabljen mehanizem s seboj poleg boljše izkoriščenega prostora na disku prinese tudi nekaj dodane zahtevnosti in dodatnih korakov za branje takih zapisov.

7.2 Refleksija razvoja

Glavni cilj celotnega diplomskega dela je bila implementacija lastnega relacijskega sistema za upravljanje s podatkovno bazo, kjer bi na praktičen način spoznali ozadje delovanja relacijskih podatkovnih baz in obenem izdelali celovito rešitev za programski jezik Python.

Razvoj dobrega SUPB ni lahka naloga, kar lahko razberemo iz količine programske kode in različnih konceptov, ki jih uporablja npr. odprtokodni MySQL [31].

Spoznali smo, kako organizirati podatke za učinkovito izvedbo vseh operacij CRUD (ustvarjanje, branje, posodabljanje in brisanje). Za uspešno izvajanje je ključnega pomena tudi dobro poznavanje operacijskega sistema in načina, kako mehanizmu za shranjevanje omogočiti optimalno delo z datotekami.

Poudarek je bil tudi na implementaciji indeksiranja z uporabo B+ drevesa. S tem smo uporabnikom paketa Graphenix omogočili možnost uporabe indeksov, ki ob določenih pogojih (v poizvedbi uporabljamo iskanje) bistveno

pospešijo poizvedbe. Zagotovo gre za najbolj zapleten del celotnega sistema. Implementacija B+ drevesa je realizirana z rekurzivno strukturo in uporabo kazalcev, kjer smo implementirali tudi shranjevanje strukture v datoteko. Za naslednje različice pa smo identificirali možne izboljšave, ki bi pohitrile mehanizem indeksiranja.

Poleg tega smo tekom razvoja dodali še lasten ORM, ki neposredno komunicira z operacijami SUPB, kar bistveno zmanjša zahtevnost, ki jo sicer prinese ORM, kjer moramo skrbeti še za dodatne preslikave med objekti in poizvedbami SQL. Pri združevanju zapisov iz različnih tabel smo se delno oddaljili od relacijskih podatkovnih baz in uporabili objektne pristope, ki jih uporabljajo nerelacijske podatkovne baze.

Razvoj prototipa je bil uspešen. Z vključitvijo testnih primerov smo poskrbeli za hitrejšo sprotno odkrivanje napak in višjo kakovost rešitve. Analiza delovanja je v nekaterih scenarijih pokazala celo primerljive rezultate z MySQL in SQLite, kar je bil tudi eden izmed glavnih ciljev pred začetkom razvoja.

Literatura

- [1] Martin Abadi. “TensorFlow: learning functions at scale”. V: *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*. 2016.
- [2] *B+ tree* - Wikipedia. URL: https://en.wikipedia.org/wiki/B%2B_tree (pridobljeno 20. 8. 2023).
- [3] *B+Tree index structures in InnoDB*. URL: <https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/> (pridobljeno 20. 8. 2023).
- [4] Charles W Bachman. “The origin of the integrated data store (IDS): The first direct-access DBMS”. V: *IEEE Annals of the History of Computing* 31.4 (2009), str. 42–54. DOI: 10.1109/MAHC.2009.110.
- [5] *BeeWare knjižnica*. URL: <https://beeware.org/> (pridobljeno 18. 8. 2023).
- [6] 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.
- [7] *Comparison of B-Tree and Hash Indexes*. URL: <https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html> (pridobljeno 21. 8. 2023).

-
- [8] Thomas M Connolly in Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005, str. 15–21.
 - [9] *Django*. 2023. URL: <https://github.com/django/django> (pridobljeno 20. 7. 2023).
 - [10] *Doctest*. 2023. URL: <https://github.com/doctest/doctest> (pridobljeno 2. 7. 2023).
 - [11] *Epoch format*. URL: <https://www.maketecheasier.com/what-is-epoch-time/> (pridobljeno 2. 7. 2023).
 - [12] *Flask*. 2023. URL: <https://github.com/pallets/flask> (pridobljeno 30. 7. 2023).
 - [13] Michael T Goodrich, Roberto Tamassia in David M Mount. *Data structures and algorithms in C++*. John Wiley & Sons, 2011, str. 437–474.
 - [14] Burton Grad. “Relational Database Management Systems: The Formative Years [Guest editor’s introduction]”. V: *IEEE Annals of the History of Computing* 34.4 (2012), str. 7–8.
 - [15] Rick Greenwald, Robert Stackowiak in Jonathan Stern. *Oracle essentials: Oracle database 12c*. ”O’Reilly Media, Inc.”, 2013.
 - [16] *IBM Db2*. URL: <https://www.ibm.com/products/db2> (pridobljeno 19. 8. 2023).
 - [17] *InnoDB – innodb_page_size*. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_page_size (pridobljeno 22. 8. 2023).
 - [18] *InnoDB Startup Configuration*. URL: <https://dev.mysql.com/doc/refman/8.0/en/innodb-init-startup-configuration.html> (pridobljeno 22. 8. 2023).

-
- [19] *Insertion in a B+ tree*. URL: <https://www.geeksforgeeks.org/insertion-in-a-b-tree/> (pridobljeno 20. 8. 2023).
- [20] William Kahan. “IEEE standard 754 for binary floating-point arithmetic”. V: *Lecture Notes on the Status of IEEE 754.94720-1776* (1996), str. 2–19.
- [21] Bill Karwin. *SQL antipatterns*. in the United States of America, 2010.
- [22] Arthur M Keller. “Algorithms for translating view updates to database updates for views involving selections, projections, and joins”. V: *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1985, str. 154–163.
- [23] *MariaDB Server*. URL: <https://mariadb.org/> (pridobljeno 20. 8. 2023).
- [24] Nicholas D Matsakis in Felix S Klock. “The rust language”. V: *ACM SIGAda Ada Letters* 34.3 (2014), str. 103–104.
- [25] Wes McKinney in sod. “Data structures for statistical computing in python”. V: *Proceedings of the 9th Python in Science Conference*. Zv. 445. 1. Austin, TX. 2010, str. 51–56.
- [26] Ross Mistry in Stacia Misner. *Introducing Microsoft SQL Server 2014*. Microsoft Press, 2014.
- [27] Huaxin Mu in Shuai Jiang. “Design patterns in software development”. V: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE. 2011, str. 323–324. DOI: 10.1109/ICSESS.2011.5982228.
- [28] *MySQL*. URL: <https://www.mysql.com/> (pridobljeno 19. 8. 2023).
- [29] *MySQL - Aggregate Function Descriptions*. URL: <https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html> (pridobljeno 13. 8. 2023).

-
- [30] *MySQL - The CHAR and VARCHAR Types*. URL: <https://dev.mysql.com/doc/refman/8.0/en/char.html> (pridobljeno 20. 8. 2023).
- [31] *MySQL Repository*. 2023. URL: <https://github.com/mysql/mysql-server> (pridobljeno 18. 8. 2023).
- [32] *MySQL Storage Engines*. URL: <https://www.w3resource.com/mysql/mysql-storage-engines.php> (pridobljeno 22. 8. 2023).
- [33] *Numpy*. 2023. URL: <https://github.com/numpy/numpy> (pridobljeno 15. 7. 2023).
- [34] *Pandas*. 2023. URL: <https://github.com/pandas-dev/pandas> (pridobljeno 15. 7. 2023).
- [35] *Peewee*. 2023. URL: <https://github.com/coleifer/peewee> (pridobljeno 20. 7. 2023).
- [36] Felipe Pezoa in sod. "Foundations of JSON Schema". V: *Proceedings of the 25th International Conference on World Wide Web*. WWW '16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, str. 263–273. ISBN: 9781450341431. DOI: 10.1145/2872427.2883029. URL: <https://doi.org/10.1145/2872427.2883029>.
- [37] Jaroslav Pokorný. "NoSQL databases: a step to database scalability in web environment". V: *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*. 2011, str. 278–283.
- [38] *Polars*. URL: <https://pypi.org/project/polars/> (pridobljeno 20. 8. 2023).
- [39] *PostgreSQL*. URL: <https://www.postgresql.org/> (pridobljeno 19. 8. 2023).

- [40] *C++ std::priority_queue*. URL: https://en.cppreference.com/w/cpp/container/priority_queue (pridobljeno 22. 7. 2023).
- [41] *PyBind11*. 2023. URL: <https://github.com/pybind/pybind11> (pridobljeno 15. 7. 2023).
- [42] *PyPI Stats*. URL: <https://pypistats.org/> (pridobljeno 20. 7. 2023).
- [43] *Python - Common Object Structures*. URL: <https://docs.python.org/3/c-api/structures.html> (pridobljeno 19. 8. 2023).
- [44] Janez Sedeljšak. *Graphenix*. 2023. URL: <https://github.com/JanezSedeljsak/graphenix> (pridobljeno 2. 7. 2023).
- [45] *SQLAlchemy*. 2023. URL: <https://github.com/sqlalchemy/sqlalchemy> (pridobljeno 20. 7. 2023).
- [46] *SQLite*. URL: <https://www.sqlite.org/index.html> (pridobljeno 20. 8. 2023).
- [47] *SQLObject*. 2023. URL: <https://github.com/sqlobject/sqlobject> (pridobljeno 20. 7. 2023).
- [48] Alexandre Torres in sod. "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design". V: *information and software technology* 82 (2017), str. 1–18.
- [49] *Tortoise ORM*. 2023. URL: <https://github.com/tortoise/tortoise-orm> (pridobljeno 20. 7. 2023).
- [50] Jeffrey D Ullman. *A first course in database systems*. Pearson Education India, 2007, str. 5–13.
- [51] *UltraJSON*. 2023. URL: <https://github.com/ultrajson/ultrajson> (pridobljeno 15. 7. 2023).

- [52] *Unittest — Unit testing framework*. URL:
<https://docs.python.org/3/library/unittest.html> (pridobljeno
20. 7. 2023).
- [53] *C++ std::vector*. URL:
<https://cplusplus.com/reference/vector/vector/> (pridobljeno
20. 7. 2023).
- [54] Chao Wang in sod. “smartPip: A Smart Approach to Resolving
Python Dependency Conflict Issues”. V: *Proceedings of the 37th
IEEE/ACM International Conference on Automated Software
Engineering*. 2022, str. 1–12.