

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Janez Sedeljšak

Razvoj DBMS 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 DBMS 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 Poženel

Opis:

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

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

Description:

opis diplome v angleščini

Zahvaljujem se svojemu mentorju doc. dr. Boštjanu Slivniku in somentorju asist. dr. Marku Poženelu za vso strokovno pomoč in svetovanje med implementacijo celotne rešitve ter pisanju diplomskega dela. Prav tako se zahvaljujem svoji družini in prijateljem za pomoč in podporo med študijem. 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	Kje se danes uporabljajo relacijske podatkovne baze?	3
1.2.1	Vloga podatkovne baze v informacijskem sistemu	4
1.3	Motivacija za razvoj lastnega DBMS in ORM vmesnika	5
2	Sorodna dela	7
2.1	Izbor DBMS	7
2.2	Statističen pregled Python ORM knjižnic	8
2.2.1	Slabost uporabe ORM knjižnic na nivoju poizvedb	9
3	Razvoj DBMS in mehanizma za shranjevanje	13
3.1	Mehanizem shranjevanja podatkov	14
3.1.1	Matrika podatkov in zaglavje s kazalci	14
3.1.2	Tipi podatkov	15
3.1.3	Implementacija relaciji	16
3.2	Indeksiranje z uporabo B+ dreves	17
3.2.1	Vstavljanje v B+ drevo	18

3.2.2	Brisanje iz B+ drevesa	19
3.2.3	Posodabljanje v B+ drevesu	20
3.2.4	Implementacija za shranjevanje na disk	20
3.2.5	Implementacija v InnoDB	21
3.2.6	Podprte operacije filtriranja	22
3.2.7	Ostali pristopi indeksiranja	23
3.3	Optimizacije poizvedovanja	24
3.3.1	Gručanje podatkov ob branju iz diska	24
3.3.2	Uporaba prioritete vrste za urejanje zapisov	25
3.3.3	Uporaba podatkovnih okvirjev	27
3.4	Poizvedovanje z uporabo ORM	28
4	Sodoben pristop razvoja programske opreme	35
4.1	Razvoj knjižnice za programski jezik	
	Python	35
4.1.1	Uporaba Python.h za razvoj knjižnice	36
4.1.2	Naprednejša knjižnica za lažji prehod med jezikoma	37
4.2	Končna struktura namenske knjižnice	39
4.3	Testno usmerjen razvoj	39
4.3.1	Testi enot na nivoju DBMS in mehanizma za shranjevanje	40
4.3.2	Integracijsko testiranje funkcionalnosti na nivoju končne knjižnice	40
4.3.3	Performančno testiranje zahtevnejših akcij	41
4.3.4	Integracija avtomatskega testiranja z GitHub	41
5	Analiza uspešnosti	43
5.1	Množično vstavljanje podatkov	44
5.1.1	Vstavljanje z dodatnim indeksiranim poljem	45
5.2	Velikosti podatkovnih baz na disku	46
5.3	Primerjava poizvedb na eni entiteti	47

5.3.1	Branje s filtriranjem, urejanjem in omejevanjem podatkov	48
5.4	Primerjava poizvedb z uporabo relaciji	49
5.5	Izvedba agregacijske poizvedbe	50
5.6	Pohitritve s pomočjo indeksiranja	51
5.6.1	Optimizacija v metodi filtriranja zapisov	52
6	Scenariji oz. primeri uporabe	55
6.1	Strežniško beleženje podatkov	55
6.1.1	Aplikacija, ki služi kot osnova za modul beleženja . . .	55
6.1.2	Kreiranje sheme za beleženje podatkov	56
6.1.3	Dekorator za dinamično dodajanje beleženja	56
6.1.4	Branje zapisanih podatkov	58
6.2	Namizna aplikacija za upravljanje stroškov	60
6.2.1	Zaslonski pregled aplikacije	61
6.2.2	Aplikacija z vidika kode	62
7	Sklepne ugotovitve	65
7.1	Nadaljnji razvoj	65
7.2	Refleksija razvoja	66
	Literatura	69

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
DBMS	database management system	sistem za upravljanje podatkovnih baz
ER	entity relationship (diagram)	(diagram) razmerji med entitetami
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
TCP/IP	Internet protocol, transmission control protocol	internetni protokol in protokol za nadzor prenosa

Povzetek

Naslov: Razvoj DBMS z integracijo v programski jezik Python

Avtor: Janez Sedeljšak

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

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

V zadnjem sklopu diplomskega dela smo pripravili analizo uspešnosti implementacije DBMS na različnih scenarijih, kjer razvito programsko opremo primerjamo z že obstoječima DBMS – SQLite in MySQL.

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

Abstract

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

Author: Janez Sedeljšak

The thesis presents an overview of some of the currently most widely used Database Management Systems (DBMS). Relational databases still largely dominate the landscape of database standards. With this in mind, the thesis focuses on developing a custom DBMS for the Python programming language.

The development of this dedicated library is carried out in the C++ programming language, chosen for its low-level nature, providing high flexibility in memory management. The development encompasses all necessary components for a well-functioning relational database. Throughout the development process, a crucial aspect has been the utilization of efficient data structures and algorithms, optimizing I/O operations provided by the operating system, resulting in a reliable and optimized database performance.

In the final part of the thesis, a performance analysis of the implemented DBMS is conducted across various scenarios. The developed software is compared against existing DBMS solutions – SQLite and MySQL.

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

Poglavje 1

Uvod

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

1.1 Paradigme v svetu podatkovnih baz

1.1.1 Relacijske podatkovne baze

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

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

Kaj hitro, ko začnemo raziskovati interno delovanje relacijskih podatkovnih baz, naletimo na pojem DBMS (sistem za uporabljanje podatkovnih baz). Gre za programsko rešitev, ki je zasnovana za upravljanje in organiziranje podatkov. Uporabnikom omogoča ustvarjanje, spreminjanje in poizvedovanje po bazi podatkov [18].

Kadar govorimo specifično o relacijskih podatkovnih bazah, pa lahko uporabimo tudi kratico RDBMS (relacijski sistem za uporabljanje podatkovnih baz).

DBMS sam po sebi predstavlja vmesnik med uporabnikom, ki dela s podatkovno bazo in mehanizmom za shranjevanje podatkov (ang. storage engine). Mehanizem za shranjevanje podatkov je ponovno programska rešitev, ki skrbi za strukturo shranjevanja podatkov. MySQL v ta namen ponuja dva mehanizma InnoDB (privzet mehanizem od verzije v5.5) in MyISAM. Med drugim ponuja tudi možnost shranjevanja v .csv dateoteke, ki sicer podpira zelo omejen nabor funkcionalnosti ostalih mehanizmov [25].

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 [11]. Prva družina relacijskih podatkovnih baz pa je bila razvita leta 1983 – Db2 [12] izdal jo je IBM. Skozi leta je Db2 doživel, kar nekaj posodobitev – zadnja med njimi je bila leta 2022, ko je DBMS v svoje delovanje dodal še nekaj konceptov iz umetne inteligence za boljšo izrabo virov [13]. Kljub temu je DBMS konceptualno ostal nespremenjen in še vedno uporablja zasnovo relacijskega modela.

1.1.2 Nerelacijske podatkovne baze

Nerelacijske podatkovne baze predstavljajo novo kategorijo baz, ki temeljijo na bistveno drugačnih osnovah kot tradicionalne relacijske podatkovne baze.

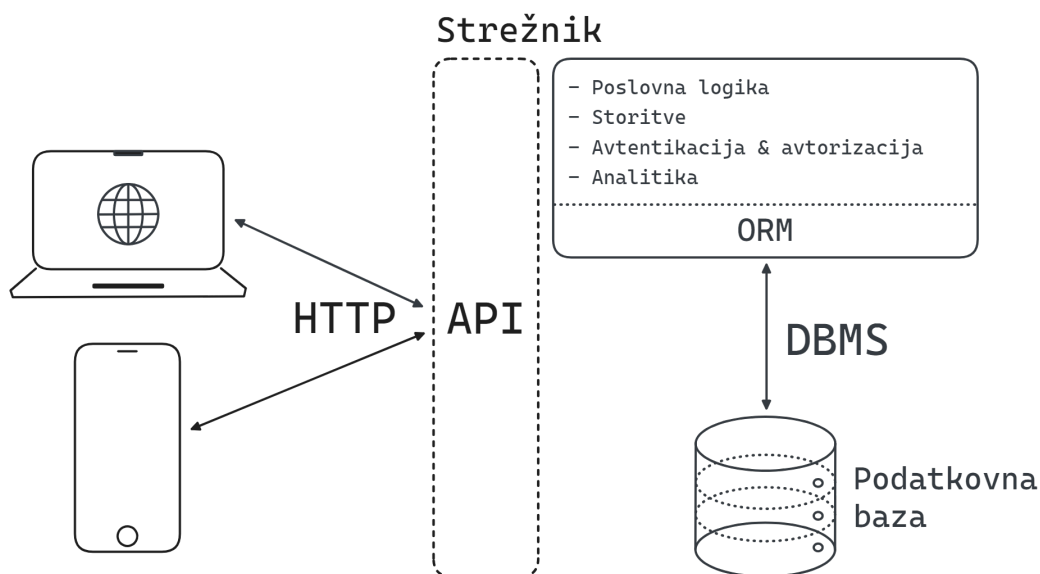
Nova paradigma se je razvila kot odziv na izzive, s katerimi se srečujejo relacijske podatkovne baze. Glavna ovira relacijskih podatkovnih baz izvira iz njihove stroge strukture. V novi kategoriji podatkovnih baz, znanih kot NoSQL, je ključna lastnost prilagodljivost. Organizacija podatkov se bistveno razlikuje, pri čemer tradicionalne entitete in relacije med zapisi nadomeščajo objekti in koncept dedovanja. Ta pristop prinaša tudi eno od glavnih prednosti, ki jih imajo nerelacijske podatkovne baze – sposobnost enkapsulacije posameznih zapisov, ki omogoča učinkovito razporeditev celotne baze na več strežnikov (horizontalno skaliranje podatkov), ki ga tradicionalne relacijske podatkovne baze težje dosežejo.

1.2 Kje se danes uporabljajo relacijske podatkovne baze?

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

1.2.1 Vloga 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 eni strani so odjemalci, ki vključujejo mobilne, spletne aplikacije ... Ti odjemalci s strežniškim delom aplikacije komunicirajo preko standardiziranih HTTP metod na nivoju povezovalne API komponente.

Večji del informacijskega sistema pa se nahaja v uporabniku skritem delu, kjer je realizirana poslovna logika in upravljanje s podatki oz. bazo podatkov s katero opravlja DBMS. V sodobnih informacijskih sistemih za dodaten nivo varnosti in lažjo manipulacijo podatkov pogosto uporabimo še ORM (objektno relacijska preslikava). Gre za programsko rešitev, ki omogoča lažje delo s podatkovno bazo. Entitete se znotraj ORM preslikajo v razrede, zapisi znotraj entitete pa so predstavljeni kot instance teh razredov. ORM predstavlja dodaten nivo abstrakcije, ki programerju med drugim omogoči tudi avtomatske migracije podatkovne baze, ter lažjo integracijo podatkovne baze znotraj uporabljenega programskega jezika.

1.3 Motivacija za razvoj lastnega DBMS in ORM vmesnika

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

Podatkovna baza s seboj prinese visoko raven abstrakcije, ki je podprta z obsežno množico algoritmov in konceptov, ki omogočijo zanesljivo, hitro in varno izvajanje operaciji nad podatki. Cilj razvoja je spoznati tudi te koncepte, ki sicer ostajajo skriti v abstrakciji, ki jo prinašata DBMS in jezik SQL, ki programerju predstavlja način komuniciranja z bazo podatkov.

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

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

Dve bolj pogosti podatkovni strukturi za izvedbo indeksiranja sta zgoščevalne tabele in drevesa. Na področju relacijskih podatkovnih baz se najpogosteje uporablja več nivojsko indeksiranje, ki je realizirano prav z drevesi (v večini primerov gre za B drevo). B drevo je tip drevesa, ki je sestavljeno iz vozlišč, kjer ima vsako največ $b - 1$ zapisov in b kazalcev na nadaljnja vozlišča. S pomočjo dodajanja indeksov lahko linearno iskanje, ki ima časovno kompleksnost $O(N)$, kjer je N število vseh zapisov, bistveno pohitrimo. Z uporabo B drevesa, namreč dosežemo časovno kompleksnost $O(\log_b(N))$, kjer je N

število vseh zapisov v drevesu.

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

Poglavje 2

Sorodna dela

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

2.1 Izbor DBMS

Pri izbiri DBMS se soočimo z različnimi možnostmi, med katerimi izstopajo DBMS-ji, kot so Db2 [12], Oracle [27], ter Microsoft SQL Server (MSSQL) [17], ki predstavljajo največje in najbolj skalabilne DBMS-je. Kljub temu je izbor DBMS-ja, podobno kot pri večini področij v računalništvu, odvisen od zahtev naše aplikacije in narave podatkov, ki jih bomo shranjevali.

Za mnoge primere morda zadostujeta bolj preprosta DBMS-ja, kot sta PostgreSQL [33] in MySQL [21], ki ponujata enostavno upravljanje in predstavljata dobro izbiro tudi za shranjevanje obsežnih količin podatkov.

V določenih scenarijih, ko imamo opravka z bolj preprosto in manj obsežno strukturo podatkov (npr. pri razvoju mobilnih aplikacij), se za najprimernejšo izbiro izkaže SQLite [43]. Gre za minimalističen DBMS, ki za razliko od prej omenjenih izbir za potrebe shranjevanja podatkov uporablja zgolj datoteko, ki jo zlahka enkapsuliramo v izolirano okolje, kot je mobilna naprava.

V področje manjših podatkovnih baz, pa lahko uvrstimo tudi razvit DBMS – Graphenix [41].

2.2 Statističen pregled Python ORM knjižnic

V spodnji tabeli je predstavljena kratka statistična analiza najpogostejše uporabljenih ORM knjižnic, ki so na voljo v programskem jeziku Python. Podatke za število mesečnih in tedenskih prenosov smo pridobili iz spletne platforme PYPI Stats [37], kjer se nahaja sledenje prenosov posameznih paketov znotraj Python ekosistema. Podatki za število GitHub zvezd, pa so pridobljeni direktno iz repozitorijev posameznih knjižnic, ki so dostopne na GitHub platformi.

Knjižnica	GitHub zvezde	Mesečni prenosi	Tedenski prenosi
Django [6]	72 tisoč	10 milijonov	2.4 milijona
SQLAlchemy [42]	7.5 tisoč	83 milijonov	20 milijonov
Peewee [29]	10 tisoč	1.1 milijon	281 tisoč
Tortoise ORM [46]	3.7 tisoč	88 tisoč	22 tisoč
SQLObject [44]	133	27 tisoč	6 tisoč

Glede na zgornjo tabelo izluščimo, da v ekosistemu prevladujejo naslednje SQLAlchemy, Django in Tortoise ORM.

- **Django ORM [6]** je vgrajen del širše Django knjižnice, ki je namenjena izdelavi spletnih aplikaciji. Omogoča zelo širok nabor operaciji nad podatkovno bazo, poleg tega ima vgrajen tudi svoj CLI (vmesnik za ukazno vrstico). S tem celotno ogrodje bistveno pospeši produktivnost razvijalcev, a hkrati kot ogrodje doda kar 8.9 MB dodatne teže. Knjižnica podpira pet DBMS-jev, to so PostgreSQL [33], MariaDB [20], MySQL [21], Oracle [27] in SQLite [43]. Pomembno je izpostaviti, da je število prenosov nekoliko zavarajoče, saj uporaba Django knjižnice v primerih, ko ne potrebujemo trajnega shranjevanja podatkov, ne pomeni uporabe Django ORM.

- **SQLAlchemy** [42] gre za eno najbolj fleksibilnih in uporabljenih knjižnic znotraj programskega jezika. Fleksibilnost pomeni, da so številne operacije, ki so znotraj Django ORM že avtomatizirane, tukaj prepuščene implementaciji razvijalca. Fleksibilnost pa s seboj prinaša tudi bistveno prednost, saj lahko kot razvijalec, ki dobro pozna ozadje delovanja uporabljenega DBMS, izkoristimo različne pristope optimizacije, ki jih znotraj Django ORM ni mogoče realizirati oz. je implementacija bistveno težja. Knjižnica zaradi lahkotne abstrakcije podpira SQLite [43], MySQL [21], Oracle [27], MSSQL [17] in še mnogo drugih DBMS-jev.
- **Peewee** [29] je ena izmed preprostejših knjižnic, ki predstavlja lahko-ten nivo abstrakcije. Knjižnica podpira 3 relacijske podatkovne baze, to so PostgreSQL [33], MySQL [21] in SQLite [43]. Gre za dokaj omejen nabor, saj je knjižnica dokaj minimalistična in s seboj ne prinese kompleksnosti, kot to storita Django ORM in SQLAlchemy.

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

2.2.1 Slabost uporabe ORM knjižnic na nivoju poizvedb

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

Narava relacijskih podatkovnih baz in združevanja tabel je usmerjena v skupen rezultat v obliki ene matrike, kjer imamo kartezičen produkt zapisov različnih entitet, katerega filtriramo glede na relacije, ki so realizirane s pomočjo tujih ključev.

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

Za primer lahko vzamemo naslednjo nalogo: “Za vse uporabnike pridobi njihova opravila in sporočila“. Za nalogo je SQLAlchemy [42] izvedel naslednjo SQL poizvedbo:

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

Zgornji pristop pomeni, da za vsakega uporabnika prenesemo $\max(T_i, 1) \cdot \max(M_i, 1)$ 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 zaradi načina združevanje, saj uporabljamo `OUTER JOIN` in ne `INNER JOIN`, kar posledično pomeni, da bomo za vsakega uporabnika zagotovo prenesli vsaj en zapis.

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

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

Skupno število zapisov, kjer bi vsak zapis iz posamezne entitete pridobili le 1x, pa lahko napišemo kot vsoto

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

Najprej torej potrebujemo N zapisov za vsakega uporabnika, nato pa za vsakega uporabnika pridobimo še T_i opravil in 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.

Medtem ko v primeru druge vsote (2.2), to pomeni prenos bistveno manjše količine podatkov $((100 \cdot (1000 + 200)) = 1.2 \times 10^5$. Kar predstavlja le 0.6% podatkov v primerjavi s prvo vsoto.

V pristopu nabora podatkov s pomočjo ORM se kaže, da končni rezultat in struktura, ki jo vrača podatkovna baza nista v enakem formatu. Zaradi tega mora ORM opraviti še nekaj dela z obdelavo podatkov. Pripravljanje končne strukture, je v vseh treh omenjenih knjižnicah izvedeno na nivoju programskega jezika Python, ki sam po sebi ni optimiziran za filtriranje in združevanje zapisov oz. ne omogoča pristopov, ki jih lahko uporabimo znotraj prevedenih jezikov in tako bistveno pohitrimo kreiranje končne strukture.

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

Poglavje 3

Razvoj DBMS in mehanizma za shranjevanje

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

Vsa izvorna koda je dostopna na GitHub repozitoriju [41].

Shema podatkov za predstavitev funkcionalnosti

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

```
import graphenix as gx

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

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()
    reciever = gx.Field.Link()
```

3.1 Mehanizem shranjevanja podatkov

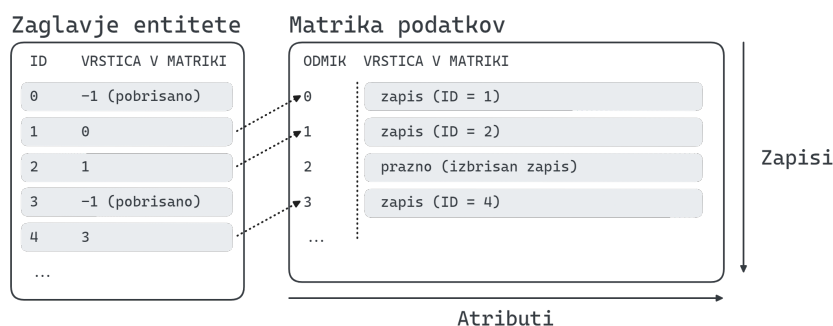
3.1.1 Matrika podatkov in zaglavje s kazalci

Za vsako posamezno entiteto se privzeto kreirata dve datoteki `ix_<naziv_entitete>.bin` in `<naziv_entitete>.bin`.

1. `<naziv_entitete>.bin` (matrika podatkov) – je matrična struktura, ki predstavlja dejanske zapise podatkov. Vsaka vrstica predstavlja en zapis, kjer je dolžina vrstice vsota dolžin vseh atributov, ki se shranjujejo v entiteti. Posamezen stolpec matrike predstavlja posamezne attribute v matriki.
2. `ix_<naziv_entitete>.bin` (glava entitete s kazalci) – gre za vektorsko strukturo, kjer vsaka zaporedna vrstica vsebuje kazalec na dejanski

zapis v matriki podatkov – ta kazalec je realiziran, kot celoštevilski odmik zapisa v matriki. Poleg kazalcev, struktura vsebuje tudi kazalec na prvo prsto vrstico v matriki, ki je definiran po pravilu:

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



Slika 3.1: Struktura podatkovne matrike in glave entitete s kazalci

3.1.2 Tipi podatkov

Znotraj razvitega DBMS podpiramo 7 podatkovnih tipov:

1. Cela števila (**Int**) – v tem primeru gre za vrednost, ki je na nivoju C++ programskega jezika shranjena kot `int64_t`, oziroma predznačeno celo število. Velikost za posamezno vrednost je 8 bajtov. Razpon vrednosti, ki jih lahko shranimo v posamezen zapis je $[-2^{63}, 2^{63} - 1]$.
2. Realna števila (**Float**) – zaradi lažje implementacije je velikost realnih števil enaka kot velikost celih števil – torej 64 bitov. Vrednost je na nivoju podatkov realizirana kot “dvojni float“ oz. “double“. Gre za zaporedje bitov, ki so po standardu IEEE 754 [19] ločeni v predznak, eksponent in mantiso.

3. Nizi (**String**) – Gre za zaporedje znakov, kjer na nivoju posamezne entitete določimo maksimalno dovoljeno dolžino posameznega zapisa. Torej dolžina posameznega zapisa je N bajtov, kjer je N maksimalna dolžina niza.
4. Logične vrednosti (**Bool**) – je najmanjši podatkovni tip, ki ga vsebuje naš DBMS in zavzame vsega 1 bajt; drži pa lahko vrednosti 0/1 oz. True/False, kot je to definirano v programskem jeziku Python.
5. Datumi (**Datetime**) – gre za podatkovni tip, ki je ponovno shranjen, kot vrednost `int64_t` in je predstavljen v EPOCH formatu [8] – gre za časovni odmik trenutnega časa od UTC datuma 1. 1. 1970.
6. Povezava (**Link**) – gre za vgrajen tip relacije, kjer je podatek kazalec na drug zapis v določeni entiteti. V ozadju je to le primarni ključ določenega zapisa, ki se nastavi avtomatsko (ponovno gre za podatek, ki je zapisan kot `int64_t` vrednost).
7. Virtualna povezava (**VirtualLink**) – gre za navidezno polje v entiteti, ki ne zavzame nobenega dodatnega prostora. Polje je pomembno za kreiranje povezave ob poizvedbah, kjer na določen zapis vežemo seznam zapisov iz druge tabele.

3.1.3 Implementacija relaciji

Implementacija relacij med entitetami se izvaja s pomočjo uporabe tujih ključev, ki predstavljajo primarne identifikatorje zapisov, ki jih želimo povezati in omogočajo povezavo med entitetami. Vzpostavitev povezave se izvede pri ustvarjanju modela na ravni programskega jezika Python. Ko na modelu nastavimo povezavo se v polje shrani identifikator zapisa, ki smo ga povezali.

Spodaj je predstavljen preprost primer, kjer neko sporočilo vežemo na dva uporabnika, ki imata vlogo pošiljatelja in prejemnika.

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



```
first_mesage = Message(  
    content = 'Moje prvo sporočilo',  
    date = datetime.now(),  
    sender = john,  
    reciever = jane).make()
```

3.2 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 nova vozišča.

V B+ drevesu vsa interna vozišča vsebujejo kazalce na nadaljnja vozišča znotraj drevesa. Vmesna vozišča 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 vozišči na enakem nivoju, saj to omogoča optimalno iskanje z intervalom in pripelje do časovne kompleksnosti $O(\log_b(N) + k)$, kjer je k širina intervala. Enaka poizvedba bi v klasičnem B drevesu pomenila časovno kompleksnost $O(\frac{k}{b} \cdot \log_b(N))$, saj bi zapise iz intervala iskali ločeno glede na posamezno vozišče [1].

Lastnosti B+ drevesa

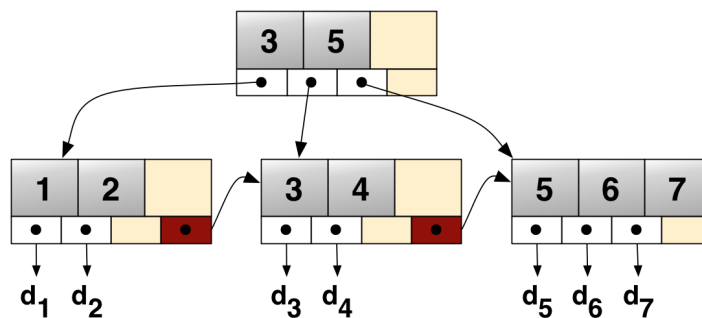
Ko govorimo o definiciji strukture B+ drevesa se moramo držati naslednjih pravil [10]:

- Vsako vozišče ima največ $b - 1$ vrednosti in b kazalcev, kjer je b stopnja vozišča.
- 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 nadaljnja vozlišča.

Primer B+ drevesa

Na sliki 3.2 je prikazan preprost primer B+ drevesa realiziran s stopnjo $b = 4$.



Slika 3.2: Primer B+ drevesa, povzeto po [1]

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 entiteti.

3.2.1 Vstavljanje v B+ drevo

Ko govorimo o kompleksnosti implementacije posamezne operacije znotraj B+ drevesa, sta brisanje in vstavljanje definitivno bistveno bolj kompleksna, kot pa iskanje. Ob dobri definiciji strukture znotraj programskega jezika je iskanje zapisov skoraj trivialen postopek, ki je tudi bolj intuitiven.

Algoritem za vstavljanje je povzet po implementaciji na spletni platformi “GeeksforGeeks” [16]. 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 nadaljnja vozlišča, kot tudi starša (spustimo se v rekurzivno posodabljanja 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.

3.2.2 Brisanje iz B+ drevesa

Pri izvedbi brisanja gre koračno gledano za zelo podoben postopek, kot je bilo to 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 iz starša.

3. Zagotoviti moramo, da so veljavna tudi vsa predhodna vozlišča (od brisane vrednosti, do korenskega vozlišča). V tem postopku ponovno izvajamo rekurziven sprehod in po potrebi posodabljammo drevo.

Ključnega pomena je, da drevo ohrani vse zahtevane lastnosti B+ drevesa, saj s tem ohrani tudi časovno kompleksnost operacij vstavljanja, brisanja in iskanja $O(\log_b(N))$ [10].

3.2.3 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 v vrednosti, ki jo posodabljammo, lahko pogosto vrednost le premaknemo znotraj vozlišča. Optimizacija na časovno kompleksnost nima vpliva, lahko pa ima velik vpliv na številu I/O operaciji.

3.2.4 Implementacija za shranjevanje na disk

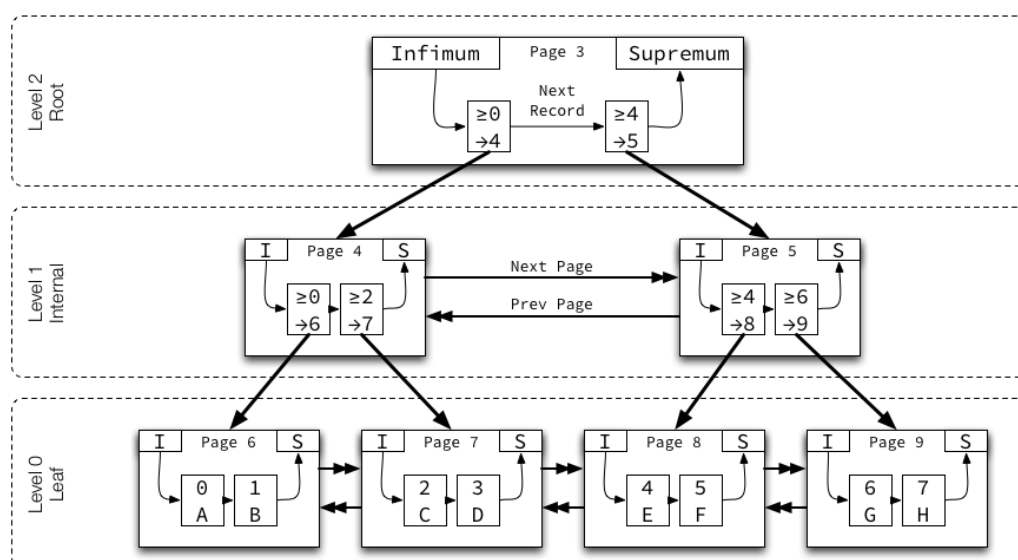
Razlog za uporabo B dreves znotraj podatkovnih baz je dokaj trivialen. Iskalne strukture, ki so realizirane s pomočjo binarnih dreves na področju shranjevanja na disk, namreč vsebujejo izrazito ozko grlo. Težava se pojavi pri branju in pisanju posameznih vozlišč, saj operacija nad enim vozliščem pomeni en dostop do diska oz. datoteke. B drevesa ta problem rešujejo s samo količino podatkov, ki je shranjena na nivoju enega vozlišča. 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 [10].

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 spremenljivko `innodb_page_size`, ki je privzeto nastavljena na 16 384 bajtov (16 KB) [14]. Vrednost spremenljivke je možno določiti le ob kreiranju MySQL instance [15], njeno vrednost pa lahko preverimo z ukazom `SHOW VARIABLES LIKE 'innodb_page_size'`.

3.2.5 Implementacija v InnoDB

InnoDB je eden izmed privzeti mehanizmov za shranjevanje podatkov, ki za indeksiranje uporablja implementacijo B+ drevesa. Struktura je prikazana na sliki 3.3.



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

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 knjižnice 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 kompleksno implementacijo B+ drevesa, ki omogoča še nekaj dodatnih optimizacij z vidika shranjevanja podatkov in izvajanja iskanja.

3.2.6 Podprte operacije filtriranja

Znotraj razvitega DBMS 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 iskanja vozlišča 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 optimizacijah 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 direktno 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, ki uporabljajo filtriranje z indeksiranjem.

V naslednjem koraku kreiramo presek vseh identifikatorjev, ki smo jih dobili iz zgornje akcije. 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.

3.2.7 Ostali pristopi indeksiranja

MySQL omogoča dva bolj pogosta pristopa za indeksiranje podatkov. Prvi pristop je s prej omenjenim B+ drevesom 3.2.5. 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 [5].

DBMS 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 kompleksnost $O(1)$ oz. konstanto časovno kompleksnost.

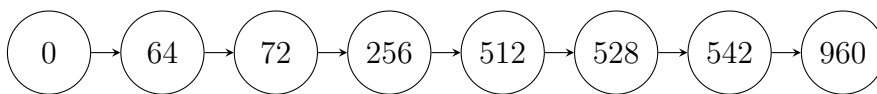
Pristop je tako zelo uporaben, kadar imamo polje, ki vsebuje vrednosti iz omejenega nabora vrednosti, npr. pri indeksiranju pošiljatelja sporočila bi z uporabo zgoščevalne tabele zelo hitro lahko poiskali pošiljatelja, hkrati pa za ta atribut načeloma ne izvajamo ostalih operaciji filtriranja in v večini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

3.3 Optimizacije poizvedovanja

3.3.1 Gručanje podatkov ob branju iz diska

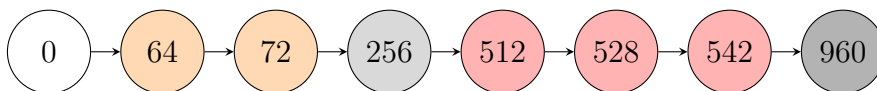
Kot je bilo že omenjeno v prejšnjih poglavjih, je največje ozko grlo, s katerim se srečujemo, I/O operacije (torej pisanje in branje iz/na disk). V ta namen je tudi v segmentu prenosa podatkov iz datoteke v pomnilnik realizirano s čim manj dostopi do diska. Zato je ob prvi fazi nabora dodan algoritem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



Slika 3.4: Zaporedje odmirov za poizvedbo iz matrične datoteke

Iz odmirov lahko hitro razberemo dve gruči, za kateri bi želeli, da jih naš algoritem identificira in branje odmirov izvede v enem koraku – to sta gruči [64, 62], ter [512, 528, 542]. Optimalen pristop branja bi lahko barvno označili po naslednji konfiguraciji:



Slika 3.5: Zaporedje odmirov 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 elementov v matriki. V ta namen smo pripravili preprost algoritem, ki se sprehodi skozi seznam in spremlja odmike, ter združuje elemente, v kolikor ne presežemo zgornje meje velikosti ene gruče in poleg tega zadovoljimo minimalno velikost za gručo.

3.3.2 Uporaba prioritete vrste za urejanje zapisov

Ob implementaciji omejevanja števila rezultatov s pomočjo ukazov ‘limit’ in ‘offset’ moramo poskrbeti za časovno optimalno izvedbo poizvedbe, saj z vidika uporabnika DBMS pričakujemo, da z dodatkom ukaza ‘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 oz. limita, O pa predstavlja odmik oz. število zapisov, ki jih želimo preskočiti.

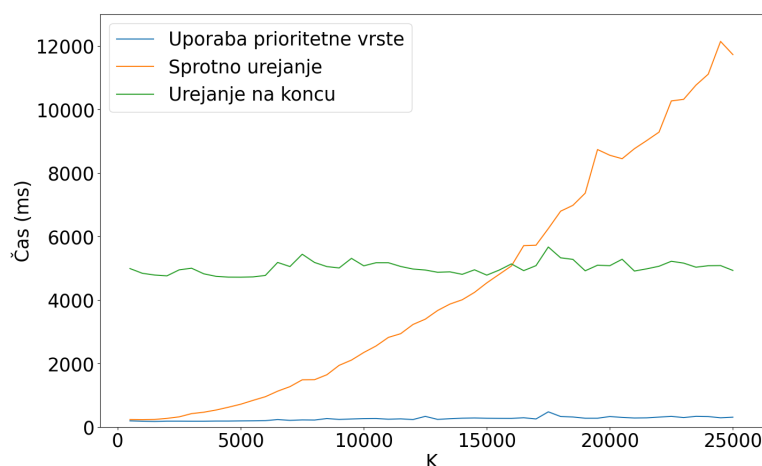
Preizkusili smo tri različne pristope za iskanje K najmanjših elementov v seznamu.

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 $O(N)$, saj moramo v pomnilniku držati vse potencialne elemente.
2. Postopno vstavljanje v urejen seznam, v tem primeru uporabimo vgrajeno `std::vector` [49] 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 postane $O(N)$. Tako je časovna zahtevnost celotnega postopka vstavljanja $O(N^2)$, v najboljšem primeru pa $O(N \cdot \log_2(K))$. Glavna prednost algoritma pred pristopom urejanja na koncu se nahaja v prostorski zahtevnosti. Ta je namreč le $O(K)$, saj v pomnilniku ohranjamo maksimalno K vrstic in jih po potrebi zamenjujemo ob iteraciji skozi vrstice matrike.

3. Za daleč najboljši pristop se je izkazalo vstavljanje v prioriteto vrsto, ki je realizirana kot kopica. Za podatkovno strukturo uporabimo vgrajeno C++ `std::priority_queue` [34]. V tem primeru je časovna zahtevnost posameznega vstavljanja $O(\log_2(K))$, kar pomeni za N vstavljanj $O(N \cdot \log_2(K))$. Prostorska zahtevnost, pa je enako kot v zgornjem primeru $O(K)$. Problem je zastavljen tako, da K ne more nikoli biti večji od N in je posledično časovna zahtevnost tega pristopa najbolj optimalna. Pomemben segment pri uporabi vgrajene prioritete vrste je tudi izbor podatkovne strukture, ki jo vrsta uporablja za shranjevanje podatkov. Za najbolj optimalen pristop se je izkazal vgrajeni `std::vector` [49], ki je na nivoju pomnilnika shranjen v enem kosu, kar omogoči prevajalniku in procesorju, da izrabita še optimizacijo z lokalnostjo.

Za zgornje tri pristope obstaja veliko spremenljivk, ki vplivajo na čas izvajanja algoritma za iskanje K najmanjših elementov v seznamu. Največ vpliva ima vrednost N , ki predstavlja število zapisov, ki jih moramo preveriti in primerjati z ostalimi elementi. Je tudi spodnja meja za vse časovne zahtevnosti, vse imajo namreč časovno zahtevnost vsaj $O(N)$. Bistveno bolj zanimiv pa je vpliv vrednosti K , ki predstavlja tudi razliko v časovnih zahtevnostih med posameznimi algoritmi. Za lažje razumevanje, kako vrednost K vpliva na čas izvajanja, smo pripravili tudi grafično analizo, kjer v izoliranem okolju izvedemo vse tri algoritme. Vrednost N je nastavljena na 1×10^7 , K pa se spreminja na x osi in je definiran na intervalu $[0, 25000]$.



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

Iz zgornjega grafa lahko razberemo, da vrednost K nima bistvenega vpliva na prvi in tretji algoritem. Izjema je drugi algoritem, 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 pa postaja sorazmerno počasneje glede na večanje vrednosti K .

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

3.3.3 Uporaba podatkovnih okvirjev

Akcija za poizvedovanje ob prvi implementaciji deluje po naslednjih korakih:

1. Izgradnja strukture, ki definira vse potrebne parametre za poizvedbo – `QueryObject`.
2. Izvedba branja na nivoju programskega jezika C++ in pretvorba zapisov v seznam, kjer je posamezen zapis predstavljen kot slovar.
3. Pretvorba posameznega slovarja (vrstice) v instanco objekta.

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

Za hitrejšo izvedbo smo na nivoju programskega jezika Python pripravili podatkovno strukturo – pogled (ang. View). Ta struktura omogoča preprečevanje potrebe po tretjem koraku in tudi spreminja predstavitev posameznih vrstic iz strukture slovarja v terko (ang. Tuple). View, pa predstavlja ovoj, ki deluje na podobnem principu, kot delujejo podatkovni okvirji (ang. DataFrame) v knjižnici za podatkovno analitiko [28]. Struktura smo implementirali na nivoju programskega jezika C++, kar še dodatno pospeši čas dostopa do posameznega zapisa.

Optimizacija v številkah

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

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

3.4 Poizvedovanje z uporabo ORM

• Filtriranje podatkov – funkcija filter

Parametri 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(...)`.

```
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)))
```

V zgornjem primeru je sestavljena poizvedba, kjer izvedemo 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 preprosta ukaza, ki sta namenjena omejevanju števila zapisov, ko iščemo le prvih nekaj zapisov oz. prvih nekaj zapisov z določenim odmikom.

Pogost primer uporabe je implementacija strani (ang. *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 entitete oz. atributih ene entitete. Pri tem imamo možnost urejati po poljubnem številu atributov, pri čemer lahko izvajamo bodisi naraščajoče bodisi padajoče urejanje.

V spodnjem primeru je predstavljen 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 zgornjo poizvedbo lahko napišemo tudi ekvivalentno SQL poizvedbo.

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

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

Gre za malo drugačno izvedbo kot je to pri klasičnih relacijskih sistemih, saj v našem primeru podatkov ne združujemo v eno matrično strukturo, temveč so podatki združeni v drevesno strukturo, kjer povezave realiziramo z dedovanjem. Sledečo SQL poizvedbo, kjer podatke naloge in uporabnike združimo preko tujega ključa v tabeli nalog.

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

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

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

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

Algoirtem za poizvedovanje z uporabo relaciji

Spodaj je v obliki psevdokode predstavljen algoritem za poizvedovanje, kjer omogočamo nabor podatkov iz različnih entitet hkrati.

```
fn execute_query(qobject: query_object) -> View
    root_res := execute query on the root entity
    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
```

Algoritem nam omogoča, da izvajamo zapletene poizvedbe na strukturah, ki imajo hierarhično organizacijo podatkov.

1. Začetna točka rekurzivne metode je izvedba na korenski entiteti celotne 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 nadaljnjih poizvedb. Poleg tega pa pripravimo vmesno strukturo za shranjevanje rezultatov teh poizvedb.
3. Nato rekurzivno izvajamo nadaljnje poizvedbe in rezultate shranjujemo v prej pripravljeno združevalno strukturo (gre za zgoščevalno tabelo).
4. V zaključni fazi algoritma vmesne rezultate iz zgoščevalne tabele združimo v rezultat korenske poizvedbe – to je tudi rezultat celotne metode, ki je predstavljen v obliki podatkovnega okvirja.

- **Agregacija oz. združevanje podatkov**

Ko govorimo o agregaciji imamo v mislih poizvedbe, kjer iščemo različne statistične rezultate iz posamezne tabele. MySQL DBMS podpira kar nekaj različnih operaciji za agregacijo [22] (`AVG`, `COUNT`, `MAX`, `MIN`, `SUM`, `STD`, `STDDEV`, `VARIANCE` ...).

V našem DBMS podpiramo prvih pet operaciji iz zgornjega seznama.

- `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 nad striktno eno entiteto 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 grupirano glede na neko polje npr. iščemo najnovejše sporočilo

za vsakega posameznega uporabnika. Kadar govorimo v smislu SQL poizvedb, to pomeni uporabo GROUP BY stavka.

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

Zgornjo SQL poizvedbo lahko v poizvedbo, ki deluje na pripravljeni knjižnici, pretvorimo v naslednji odsek kode:

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

Sintaksa je dokaj preprosta. Za izvedbo agregacijske poizvedbe preprosto uporabimo metodo `.agg(...)` nad eno izmed entitet. 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

Sodoben pristop razvoja programske opreme

V poglavju bomo predstavili izbrane metodologije, ki so ključne za razvoj naše izbrane programske opreme. Jasno se zavedamo, da kakovostna zasnova predstavlja temelj celotnega sistema. Za zagotovitev pravilnega delovanja te zasnove pa se danes pogosto poslužujemo postopka avtomatskega testiranja programske opreme, ki vključuje teste enot in integracijsko testiranje.

4.1 Razvoj knjižnice za programski jezik Python

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

V namen upravljanja z eksternimi moduli programskega jezika se pogosto srečamo z upravljalci paketov (ang. package manager). Gre za orodje, ki omogoča enostaven način nameščanja, posodabljanja in odstranjevanja uvoženih knjižnic. Programski jezik Python v svoji dokumentaciji [35] pre-

dlaga uporabo PIP [31] upravljalnika paketov za distribucijo knjižnic, ki niso del standardnega modula programskega jezika.

4.1.1 Uporaba Python.h za razvoj knjižnice

Programski jezik Python je poznan predvsem po uporabi v področjih umetne inteligence, podatkovne analitike in strežniških aplikacijah. 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. Poleg tega jezik nima striktnih tipov, kar pomeni, da je za vsako spremenljivko ob izvajanju programske kode potrebno preveriti, za kateri tip gre, kar ponovno prinese časovne zakasnitve.

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 nastale knjižnice, ki so v osnovi napisane v jezikih, ki so bistveno hitrejši (prevedeni). Med temi knjižnicami so pogostejše uporabljene:

- **Numpy** [26] – odprtokodna knjižnica, ki v Python prinese podatkovni tip polja (ang. array), in veliko vgrajenih funkcij, ki bistveno pospešijo operacije, ki delajo s polji in matrikami. V področju podatkovne analitike in umetne inteligence, knjižnica s polji predstavlja enega izmed glavnih temeljev. Veliko je tudi knjižnic, ki za svoje delovanje uporabljajo ravno Numpy. Tak primer je npr. Pandas [28] – gre za eno izmed najpogostejše uporabljenih knjižnic za podatkovno analitiko, ki s seboj prinese še nekaj dodatnih abstrakcij za lažje delo s podatkovnimi okviri (ang. data frame).
- **Tensorflow** [45] – odprtokodna knjižnica za strojno učenje. V osnovi je celoten produkt Tensorflow namenjen širšemu spektru programskih

jezikov in ni implementiran le za Python. Gre za splošno namensko knjižnico z že pripravljenimi metodami za kreiranje modelov za strojno učenje, kar olajša delo programerju, saj je veliko korakov avtomatiziranih.

- **UltraJSON** [47] – gre za manj poznano knjižnico, ki dela z JSON strukturami. JSON je postal najbolj popularen format za prenos strukturiranih podatkov preko HTTP [30]. 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 UltraJSON knjižnice.

Vse te knjižnice so v osnovi napisane 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` [38], katerega lahko neposredno uporabljamo in manipuliramo na nivoju programskega jezika C.

4.1.2 Naprednejša knjižnica za lažji prehod med jezikoma

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

Za še lažji prehod med jezikoma smo se odločili za razvoj knjižnice s pomočjo programskega jezika C++, ki ima poleg tega tudi bistveno širši nabor struktur v standardni knjižnici.

Za prehod smo uporabili PyBind11 [36] knjižnico, ki deluje na nivoju višje in za svoje delovanje v ozadju uporablja Python.h. Gre za knjižnico, ki olajša pretvorbe podatkovnih struktur med jezikoma npr. `std::vector` [49],

se samodejno pretvori v strukturo seznama na nivoju programskega jezika Python.

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

Primer uporabe PyBind11 knjižnice

V spodnjem segmentu kode imamo še preprost primer definiranja 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;
}

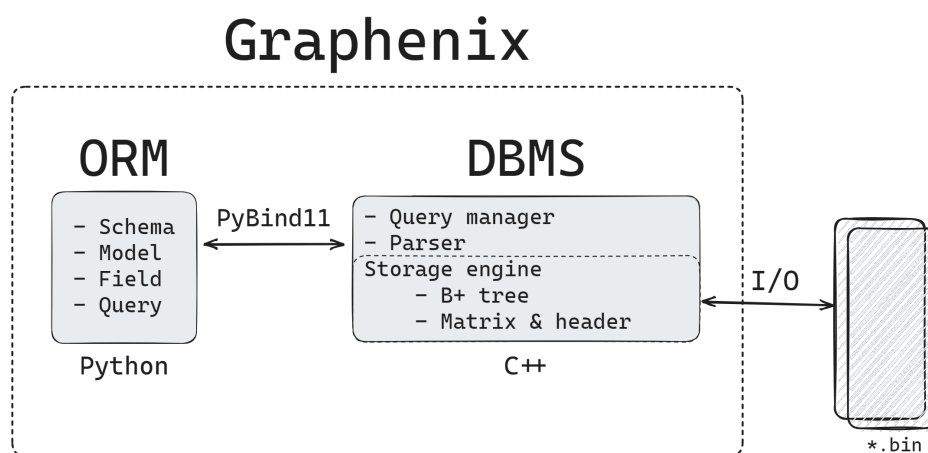
PYBIND11_MODULE(my_math_module, m)
{
    m.def("custom_sum", &add_nums, "Add 2 ints");
}
```

Pod pogojem, da je knjižnica pravilno nameščena, lahko na nivoju programskega jezika Python uporabimo funkcijo `custom_sum`.

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

4.2 Končna struktura namenske knjižnice

Na sliki 4.1 je diagram, ki prikazuje strukturo celotnega sistema z vsemi pomembnimi komponentami in orodji, ki omogočajo komunikacijo in zanesljivo delovanje Graphenix knjižnice.



Slika 4.1: Struktura celotne Graphenix knjižnice

Knjižnica je sestavljena iz dveh ključnih komponent. To sta ORM, ki je napisan s pomočjo programskega jezika Python in predstavlja tudi vhodno točko za uporabnike knjižnice. Na drugi strani pa je jedro DBMS s sistemom za shranjevanje podatkov, ki z ORM komunicira preko prej omenjene PyBind11 [36] knjižnice.

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

4.3 Testno usmerjen razvoj

Da zagotovimo pravilno delovanje nekega bolj kompleksnega sistema, je na področju razvoja programske opreme zelo priporočen testno usmerjen razvoj. Gre za način validacije pravilnega delovanja posameznih komponent sistema

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 segmente – testiranje enot, integracijsko testiranje in performančno testiranje.

4.3.1 Testi enot na nivoju DBMS in mehanizma za shranjevanje

V prvi fazi želimo zagotoviti pravilno delovanje na najnižjem nivoju oz. na nivoju mehanizma za shranjevanje in jedra DBMS. 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 knjižnice Doctest [7]. Gre za testno ogrodje napisano za programski jezik C++, ki omogoča fleksibilen pristop validacije rezultatov iz posameznih funkciji.

4.3.2 Integracijsko testiranje funkcionalnosti na nivoju končne knjižnice

Naslednja faza testiranja je realizirana s pomočjo integracijskih testov. Gre za pristop testiranja, kjer preizkusimo ali prej posamezno testirane enote delujejo tudi na nivoju integracije ene z drugo [4]. Iz praktičnega vidika, če smo prej testirali ali pravilno delujejo posamezne funkcionalnosti B+ dreves, lahko zdaj testiramo ali se B+ drevo kreira pravilno na nivoju vnašanja podatkov v entiteto, kjer imamo za določen atribut nastavljen indeks. Izvedba testiranja je izvedena na nivoju programskega jezika Python z vgrajeno knjižnico `unittest` [48].

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 kopiji podatkovne baze iz produkcijskega strežnika.

V našem primeru se ob vsaki posodobitvi kode izvede akcija za preverjanje pravilnega delovanja knjižnice. Akcija se izvede za dve verziji 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 celotne knjižnice.

1. Zagon virtualnega okolja za programski jezik Python - venv [39].
2. Namestitev knjižnice PyBind11 [36].
3. Namestitev jedra DBMS & klic testne metode jedra.
4. Namestitev knjižnice, ki vsebuje ORM in operira z jedrom.

5. Izvedba nizko nivojskih 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

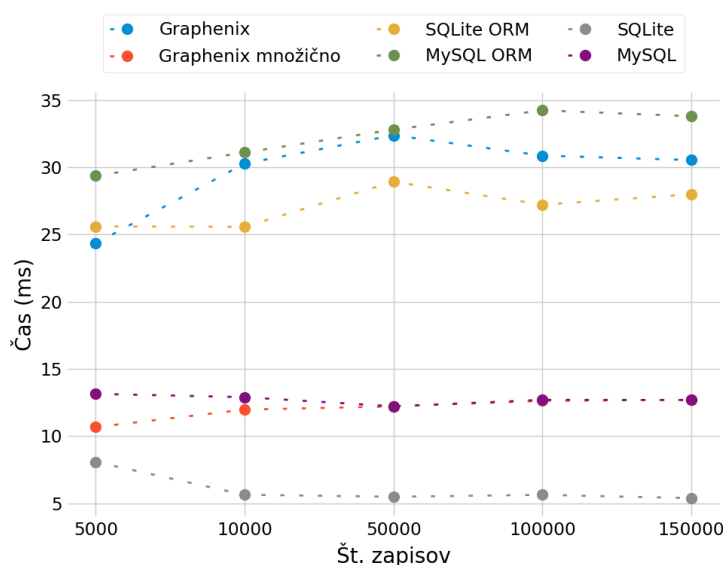
Analiza uspešnosti

V tem poglavju se bomo posvetili temeljni analizi, ki vključuje časovno in prostorsko primerjavo. Osredotočili se bomo na primerjavo med našo razvito knjižnico Graphenix ter obstoječima DBMS-jema SQLite in MySQL. Analizo bomo izvedli na različnih scenarijih, pri čemer se bomo 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 ter slabosti posameznih rešitev v različnih kontekstih.

5.1 Množično vstavljanje podatkov

V okviru prvega scenarija izvajamo postopek vstavljanja podatkov v entiteto 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 je, da je vstavljanje podatkov z uporabo ORM precej počasnejše v primerjavi z neposrednim vstavljanjem preko SQL ukazov. Uporaba ORM zahteva ustvarjanje instanc objektov, 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 DBMS 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 I/O operaciji v SQLite.

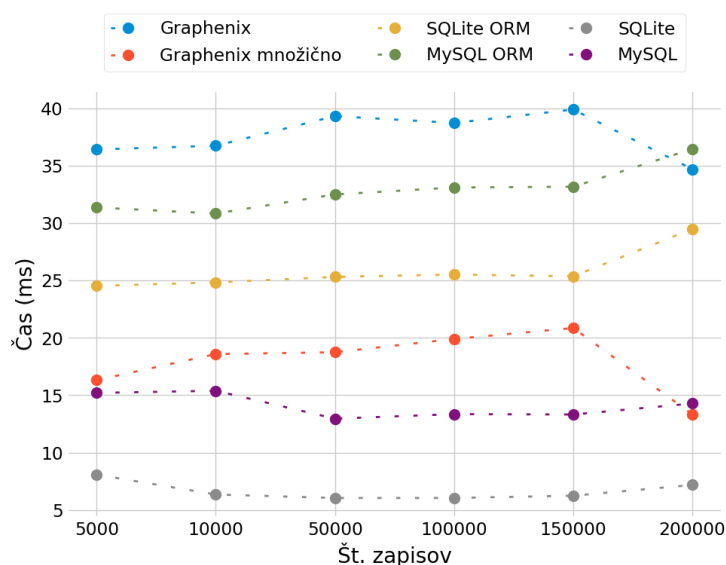
V primeru Graphenix, vstavljanje ni izvedeno v klasični množični obliki,

saj je vstavljanje vsakega zapisa ločen dostop do datoteke. V analizi sta predstavljena dva pristopa vnašanja z uporabo razvite knjižnice. V primeru množičnega vnašanja gre za direkten klic funkcije jedra za vnašanje podatkov in ne potrebujemo kreirati instanc objektov, 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, ki v primeru napake omogočata 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 definicije indeksa na atributu entitete (indeks je definiran na celoštevilskem polju, kjer je vsaka vrednost unikatna).



Slika 5.2: Množično vstavljanje z dodatnim indeksiranim atributom

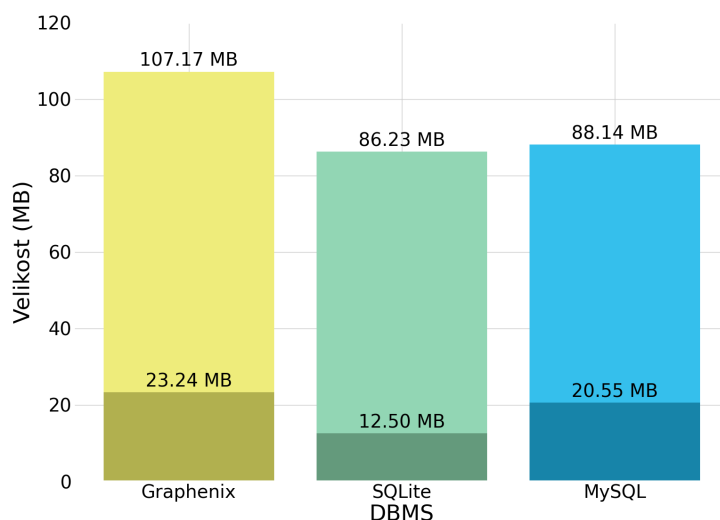
Iz grafa lahko razberemo, da z vidika hitrosti vstavljanja še vedno prevladuje

SQLite. Za razliko od predhodnega scenarija 5.1, pa tokrat MySQL v večini primerov postane bolj učinkovit kot Graphenix.

V vseh treh DBMS 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 DBMS (gre za baze podatkov z 1×10^6 zapisi).



Slika 5.3: Prostorska poraba na nivoju diska posameznega DBMS

Iz grafa 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 po-

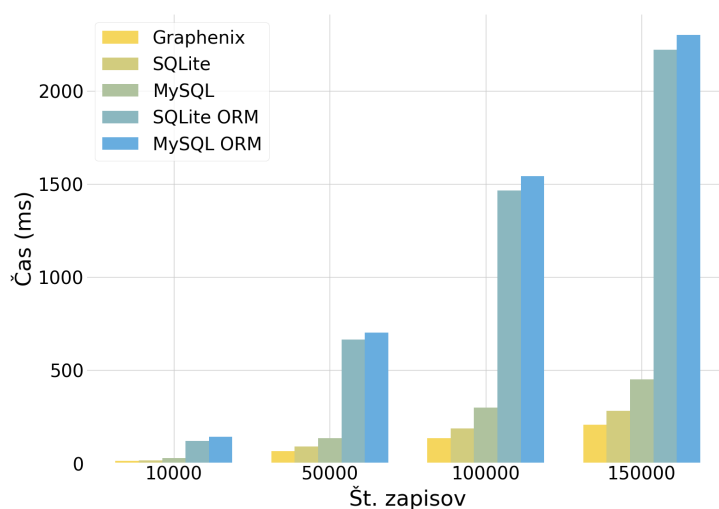
trebno. V primeru Graphenix, pa je dolžina tekstovnih polj točno določena, in v primeru, da polje definiramo kot `String(size=50)`, vsak zapis ne glede na dolžino zavzame 50 bajtov.

V temnem delu stolpca je prikazana razlika velikosti z in brez uporabe indeksne strukture. DBMS 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 entiteti

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

```
SELECT * FROM users
```



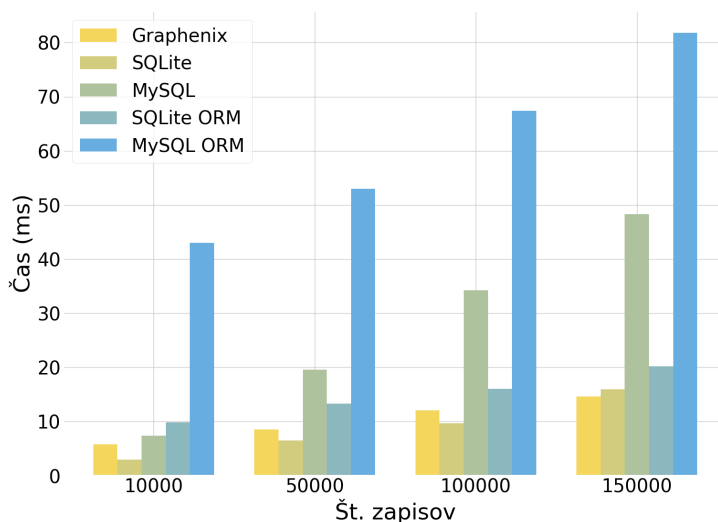
Slika 5.4: Branje brez dodatnih parametrov znotraj poizvedbe

Iz grafa 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 preko TCP/IP protokola.

5.3.1 Branje s filtriranjem, urejanjem in omejevanjem podatkov

V drugem scenariju branja podatkov iz ene entitete 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
```



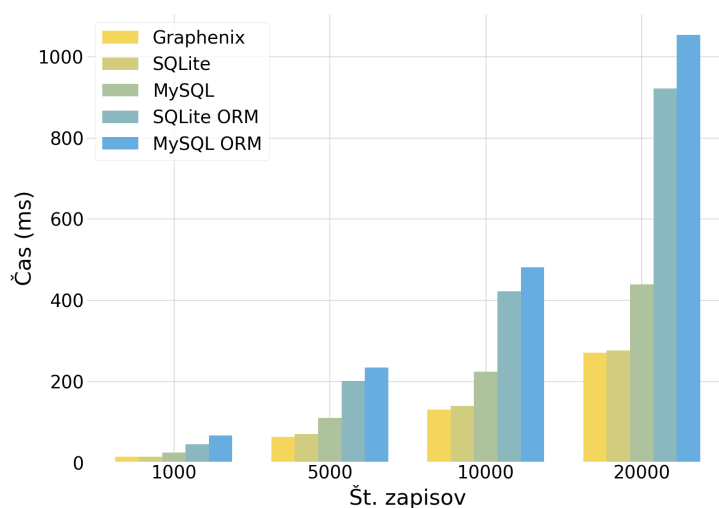
Slika 5.5: Branje z dodanim filtriranjem in urejanjem zapisov

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 videno v primerih uporabe MySQL.

5.4 Primerjava poizvedb z uporabo relaciji

V scenariju definiramo poizvedbo, kjer izvedemo nabor podatkov iz dveh entitet. Št. 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 entiteti nalog.

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



Slika 5.6: Poizvedovanje podatkov iz dveh entitet 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 v obliki kartezičnega produkta oz. matrike in imamo za vsakega uporabnika toliko zapisov, kot ima uporabnik sporočil.

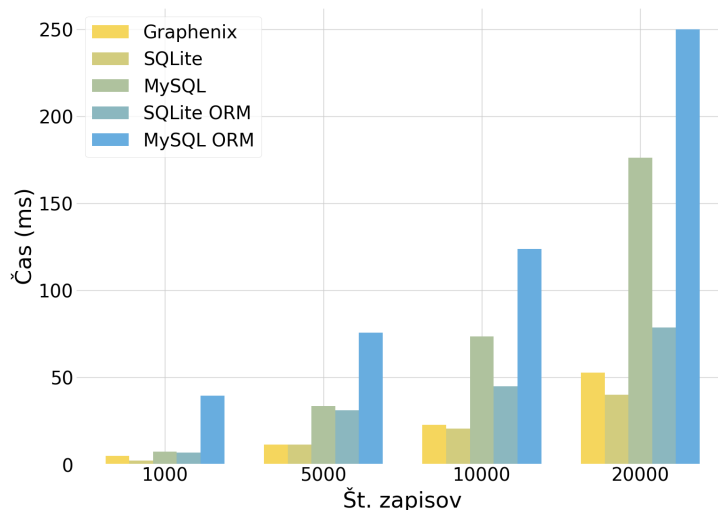
Graphenix deluje po principu ORM in imamo gnezdeno strukturo podatkov. Kljub temu pa zaradi združevanja na nivoju programskega jezika C++ poizvedba steče v primerljivem času s poizvedbami, ki ne uporabljajo ORM.

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 objektov, saj vedno kreiramo prilagojen tip rezultata, ki vsebuje identifikator uporabnika, število nalog in zadnji datum naloge za vezanega uporabnika.

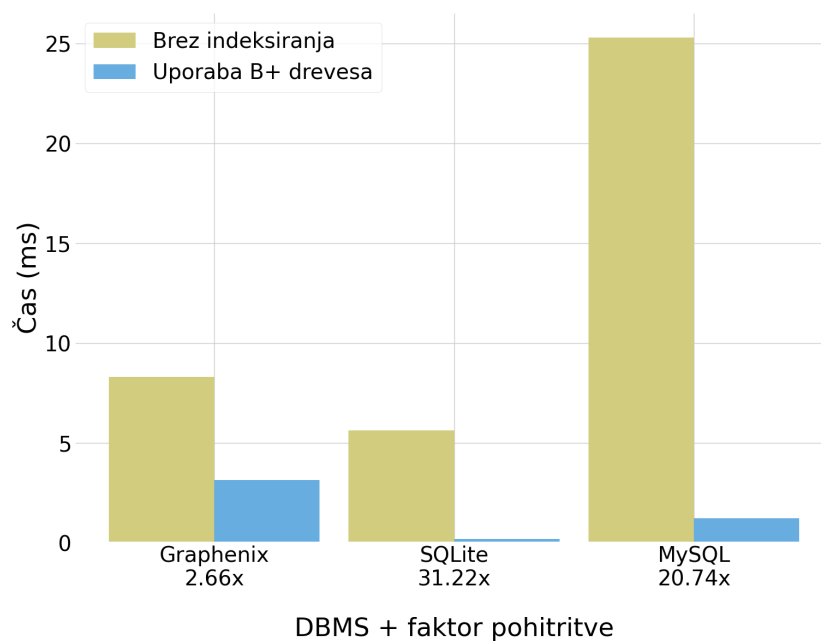
Rezultati sicer dobro sledijo trendom iz predhodnih scenarijev – MySQL zaradi načina prenosa podatkov potrebuje nekoliko več časa, poleg tega pa se ponovno pozna razlika uporabe ORM in uporaba SQL poizvedb. Graphenix pa je ponovno primerljiv uporabi SQLite brez ORM.

5.6 Pohitritve s pomočjo indeksiranja

Kot zadnji scenarij je predstavljen eden izmed najbolj kritičnih segmentov DBMS – 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 nad entiteto z 1×10^5 zapisi.



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

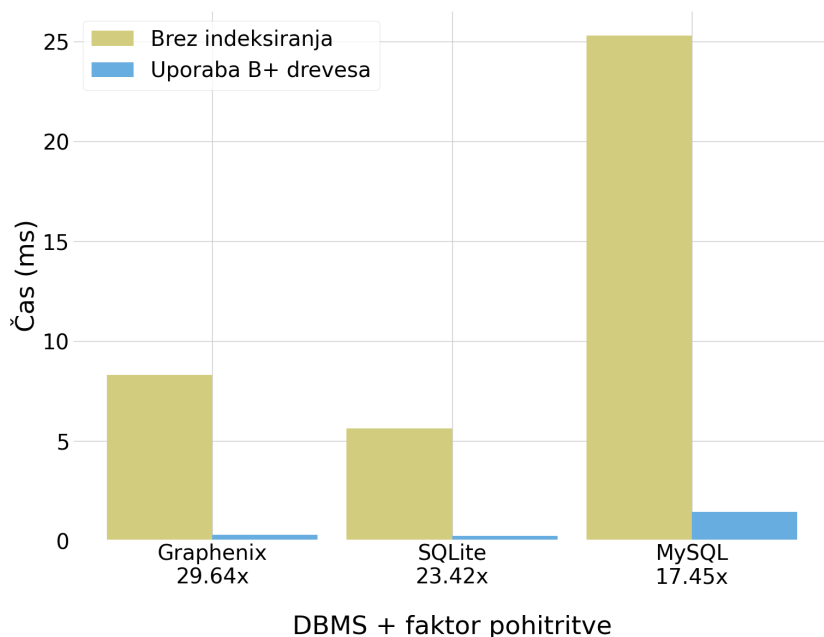
Iz grafa hitro razberemo, da je pohitritev v Graphenix DBMS drastično manjša, kot je to v primeru SQLite in MySQL.

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 entitete. Metoda za branje podatkov iz posamezne entitete je delovala tako, da je najprej prebrala celotno datoteko zaglavja entitete in za posamezen identifikator zapisa pridobila še odmik v matriki zapisov. Izkazalo se je, da je ta proces v zgornjem 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.

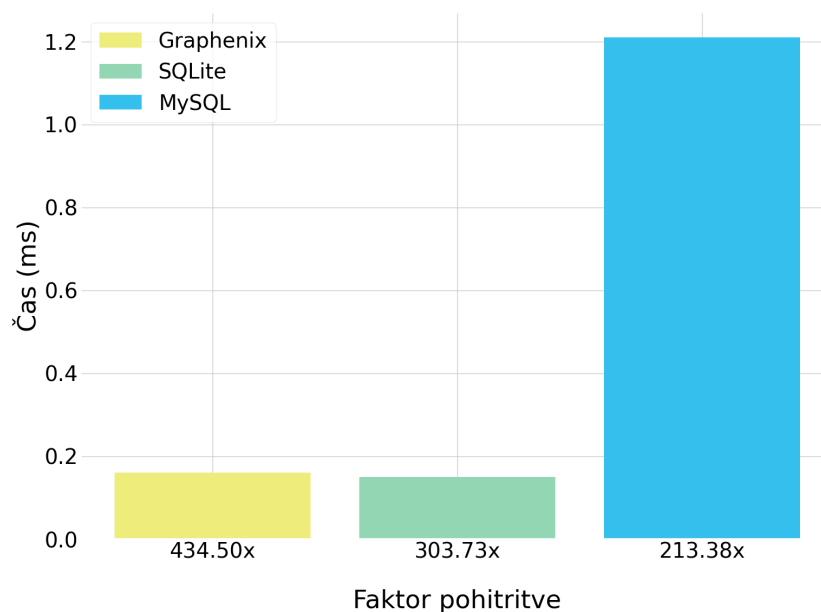


Slika 5.9: Iskanje specifičnega zapisa ($N = 1 \times 10^5$ z optimizacijo)

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

Analiza iskanja nad večjo bazo podatkov

Na grafu 5.10 je prikazana še pohitritev enake poizvedbe, kot v zgornjem 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 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 vseeno pojavi iz strani MySQL, kar je glede na predhodne scenarije pričakovano.

Idealni pogoji za indeksiranje

Pomembno je omeniti, da je scenarij zelo 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 interval vrednosti bi se ponovno oddaljili od časovne zahtevnosti $O(\log_b(N))$ in približali $O(N)$.

Če potegnemo črto je jasno, da se pri odločanju o uporabi indeksov odpira

kompleksno 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 knjižnice Graphenix. V prvem primeru se knjižnica uporablja kot dodatek strežniški aplikacij v namen beleženja podatkov. V drugem scenariju pa je predstavljen razvoj preproste namizne aplikacije, kjer knjižnico uporabljamo za trajno shranjevanje podatkov. Oba primera skušata na praktičen način predstaviti dobre strani uporabe Graphenix knjižnice.

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 knjižnic je prilagojeno beleženje podatkov lahko zelo preprosto.

6.1.1 Aplikacija, ki služi kot osnova za modul beleženja

V prvi fazi si lahko za primer pripravimo preprosto strežniško aplikacijo, ki uporablja Flask [9] ogrodje. Znotraj aplikacije smo definirali eno vhodno točko, in sicer “get-range“, 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()
```

6.1.2 Kreiranje sheme za beleženje podatkov

```
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()
```

V zgornjem kosu programske kode pripravimo razred, ki predstavlja entiteto, ki 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. entitete, ki bodo del naše baze podatkov.

6.1.3 Dekorator za dinamično dodajanje beleženja

V namen beleženja znotraj vhodnih točk strežniške aplikacije bomo 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.

```
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
```

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

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

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

Statističen pregled zahtevkov

Podatke lahko v pripravljenem DBMS tudi grupiramo, in sicer z uporabo `.agg(...)` metode v poizvedbi. 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())
```

Z zgornjo poizvedbo dobimo število zahtevkov grupirano 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 HTTP zahtevkih 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.

```
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()
```

Preslikava v podatkovne okvirje

Za namen podatkovne analitike v programskem jeziku Python najpogosteje uporabljamo knjižnice, ki za svoje delovanje uporabljajo podatkovne okvirje. Ena izmed teh knjižnic je že omenjeni Pandas [28].

Leta 2021 se je pojavila nova knjižnica za podatkovno analitiko - Polars [32]. Za svoje delovanje uporablja prevedeni programski jezik, vendar v tem primeru ne gre za C/C++, temveč za programski jezik Rust [40].

Prednost uporabe Graphenix knjižnice je v načinu nalaganju podatkov, saj podatke zlahka filtriramo in urejamo že pred kreiranjem podatkovnega okvirja, kar klasične `.csv` datoteke, 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()
```

V zgornjem primeru tako v podatkovna okvirja naložimo le zapise, ki so bili ustvarjeni po začetku leta 2020 in od tega vzamemo prvih 10 000 zapisov urejenih po datumu.

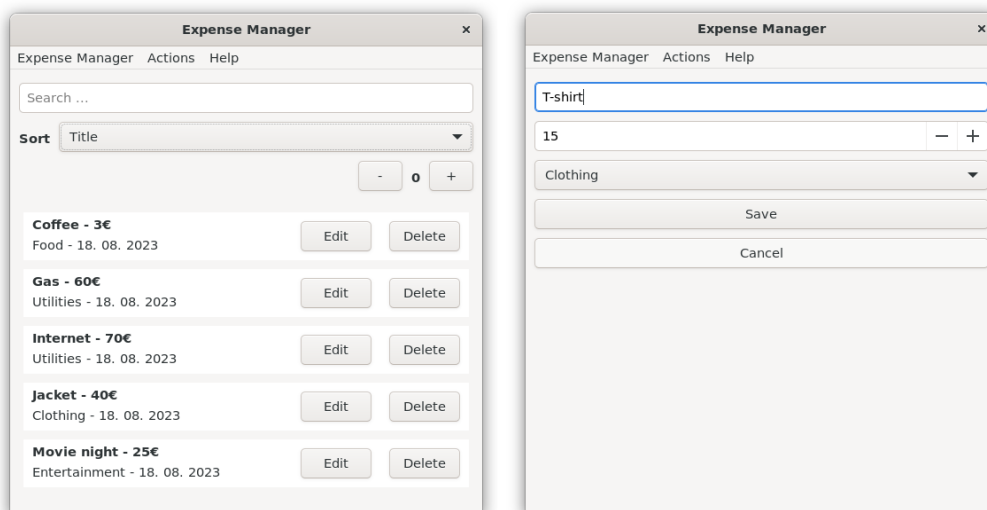
Od te točke dalje pa lahko podatke v celoti obdelujemo s pomočjo knjižnic za podatkovno analitiko.

6.2 Namizna aplikacija za upravljanje stroškov

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

6.2.1 Zasloni pregled aplikacije

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



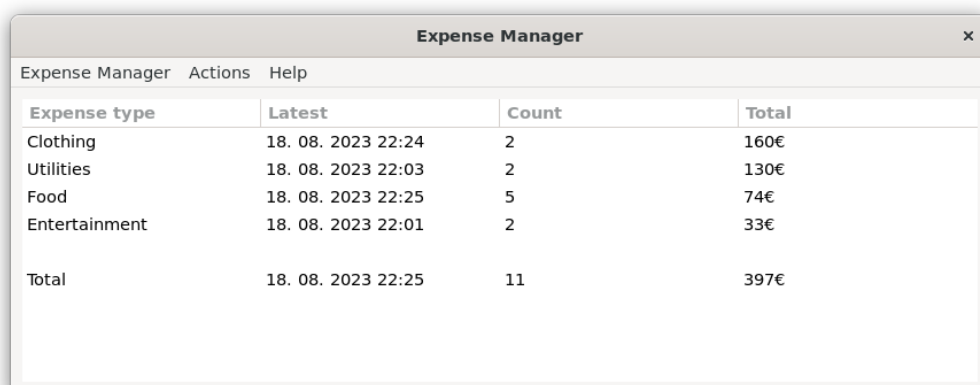
Slika 6.1: Seznam stroškov z možnostjo vstavljanja in urejanja

Za vsak posamezni strošek na skupnem prikazu sta definirani akciji brisanja in urejanja. Poleg tega lahko stroške glede na naziv, ceno in kategorijo tudi urejamo. Omogočeno je tudi filtriranje stroškov po nazivu in prikazovanje po straneh.

Na desnem zaslonu je predstavljen obrazec, do katerega dostopamo preko urejanja ali kreiranja 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 the following data:

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 zelo preprost pristop za manipulacijo podatkov.

Za izris uporabniškega vmesnika aplikacija uporablja BeeWare [3], ki omogoča medplatformski razvoj programske opreme, kar pomeni, da lahko isto kodo uporabimo za več različnih operacijskih sistemov.

Struktura podatkovne baze

```
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)
```

V namen beleženja stroškov je definirana entiteta `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 samodejno napolnijo že ob kreiranju podatkovne baze.

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)
    ...
```

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

Za zaključek podatke uredimo glede na skupno ceno, saj ta funkcionalnost za agregacijske metode v DBMS 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))
```

Za prikazovanje vseh stroškov uporabljamo metodo `display_expenses`, ki prejme parameter za iskanje, urejanje in stran, ki jo mora prikazati. Prikazan je kodni vzorec grajenja, kjer postopoma dodajamo dodatne attribute 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. Določimo stran, ki jo želimo prikazati. V ta namen imamo globalno konstanto `PAGE_SIZE`, ki določa velikost strani in v poizvedbi določimo, da preskočimo `PAGE_SIZE × page` stroškov, kjer `page` določa stran, katero želimo prikazati. 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 izboljšave razvitega DBMS, ki bi knjižnico približale nivoju produkcijske podatkovne baze.

V 7.2 pa je na kratko opisana celotna izkušnja razvoja namenske knjižnice Graphenix.

7.1 Nadaljnji razvoj

1. Razširitev funkcionalnosti indeksiranja

Indeksiranje je v razviti programski opremi sicer implementirano in v nekaterih scenarijih celo doseže primerljive rezultate implementacijam v ostalih DBMS rešitvah.

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

Delno implementirana rešitev znotraj Graphenix, je tudi držanje vozlišč v pomnilniku, kjer branje iz fizične datoteke izvedemo le v primeru, ko odmik branega vozlišča ni znotraj zgoščevalne tabele, ki drži shranjena vozlišča. S pristopom še dodatno optimiziramo postopek iskanja. Problem pa nastane, ko se moramo odločiti, katera vozlišča držati

v pomnilniku in kako dolgo ta vozlišča držati, saj ob nepremišljeni implementaciji zelo hitro preobremenimo pomnilnik in izničimo efekt optimizacije.

2. Implementacija dnevnika

Koncept dnevnika je dokaj preprost – gre za začasen sklad, kjer definiramo spremembe, ki se zgodijo direktno nad bazo podatkov. V primeru različnih izpadov električne energije ali napak pri vstavljanju je mogoče podatkovno bazo obnoviti v prejšnje (delujoče) stanje.

3. Prostorska optimizacija shranjevanja nizov

MySQL in SQLite sta že dva primera rešitev, ki ponujata prostorsko optimizacijo, kot je razvidno iz prostorske analize 5.2.

- SQLite nize definira s tipom `TEXT`, ki besedilo shranjuje v ločeni strukturi od zapisov in po potrebi zapise v tej ločeni strukturi premika.
- MySQL ima možnost uporabe `VARCHAR` [23], kjer lahko definiramo tudi maksimalno dolžino niza, vendar ta dolžina ne pomeni dejanske dolžine, ki jo niz porabi na disku, temveč tudi MySQL DBMS izvaja dinamično alokacijo za nize, ki s seboj poleg boljše izkoriščenega prostora na disku prinese tudi nekaj dodane kompleksnosti in dodatnih korakov za branje teh zapisov.

7.2 Refleksija razvoja

Glavni cilj celotnega diplomskega dela je bila implementacija lastnega DBMS, kjer bi na praktičen način spoznali ozadje delovanja relacijskih podatkovnih baz in obenem izdelali namensko knjižnico za programski jezik Python.

Razvoj dobrega DBMS ni trivialen, kar je mogoče razbrati iz količine programske kode in različnih konceptov, ki jih uporablja npr. odprtokodni DBMS MySQL [24].

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 jedru DBMS omogočiti optimalno delo z datotekami.

Poudarek je bil tudi na implementaciji indeksiranja z uporabo B+ drevesa. S tem smo uporabnikom namenske knjižnice omogočili možnost določanja indeksov, ki bistveno pospešijo operacije v sklopu filtriranja zapisov. Zagotovo gre za najbolj kompleksen del celotnega sistema, saj je implementaciji drevesa z uporabo C++ kazalcev dodana še kompleksnost shranjevanja v datoteki. Posledično ostaja veliko funkcionalnosti, ki znotraj sklopa indeksiranja niso dobro ali sploh niso implementirane.

Poleg tega smo tekom razvoja dodali še lasten ORM, ki neposredno komunicira z operacijami jedra, kar bistveno zmanjša kompleksnost, ki jo sicer prinese ORM, kjer moramo skrbeti še za dodatne preslikave med modeli in SQL poizvedbami. V našem DBMS je veliko operaciji prilagojenih objektnemu pristopu programiranja. V tem segmentu se delno oddaljimo od klasičnih relacijskih podatkovnih baz in se približamo NoSQL bazam podatkov.

Razvoj kot celota je bil uspešen, z dodatkom različnih testnih scenarijev pa smo zagotovili pravilno delovanje. Analiza uspešnosti 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] *B+ tree* - *Wikipedia*. URL: https://en.wikipedia.org/wiki/B%2B_tree (pridobljeno 20. 8. 2023).
- [2] *B+Tree index structures in InnoDB*. URL: <https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/> (pridobljeno 20. 8. 2023).
- [3] *BeeWare knjižnica*. URL: <https://beeware.org/> (pridobljeno 18. 8. 2023).
- [4] 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.
- [5] *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).
- [6] *Django*. 2023. URL: <https://github.com/django/django> (pridobljeno 20. 7. 2023).
- [7] *Doctest*. 2023. URL: <https://github.com/doctest/doctest> (pridobljeno 2. 7. 2023).
- [8] *Epoch format*. URL: <https://www.maketecheasier.com/what-is-epoch-time/> (pridobljeno 2. 7. 2023).

-
- [9] *Flask*. 2023. URL: <https://github.com/pallets/flask> (pridobljeno 30. 7. 2023).
- [10] Michael T Goodrich, Roberto Tamassia in David M Mount. *Data structures and algorithms in C++*. John Wiley & Sons, 2011, str. 470–474.
- [11] *Relational Database*. URL: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/> (pridobljeno 29. 6. 2023).
- [12] *IBM Db2*. URL: <https://www.ibm.com/products/db2> (pridobljeno 19. 8. 2023).
- [13] *IBM Db2 13*. URL: <https://www.ibm.com/docs/en/announcements/db2-13-zos-delivers-leading-edge-ai-innovations-enhancements-reinforcing-it-as-foundation-enterprise-computing-within-hybrid-cloud-digital-world?region=US> (pridobljeno 20. 8. 2023).
- [14] *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).
- [15] *InnoDB Startup Configuration*. URL: <https://dev.mysql.com/doc/refman/8.0/en/innodb-init-startup-configuration.html> (pridobljeno 22. 8. 2023).
- [16] *Insertion in a B+ tree*. URL: <https://www.geeksforgeeks.org/insertion-in-a-b-tree/> (pridobljeno 20. 8. 2023).
- [17] *Introducing SQL Server 2022*. URL: <https://www.microsoft.com/en-us/sql-server> (pridobljeno 19. 8. 2023).

- [18] *Introduction of DBMS*. URL: <https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/> (pridobljeno 19. 8. 2023).
- [19] 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.
- [20] *MariaDB Server*. URL: <https://mariadb.org/> (pridobljeno 20. 8. 2023).
- [21] *MySQL*. URL: <https://www.mysql.com/> (pridobljeno 19. 8. 2023).
- [22] *MySQL - Aggregate Function Descriptions*. URL: <https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html> (pridobljeno 13. 8. 2023).
- [23] *MySQL - The CHAR and VARCHAR Types*. URL: <https://dev.mysql.com/doc/refman/8.0/en/char.html> (pridobljeno 20. 8. 2023).
- [24] *MySQL Repository*. 2023. URL: <https://github.com/mysql/mysql-server> (pridobljeno 18. 8. 2023).
- [25] *MySQL Storage Engines*. URL: <https://www.w3resource.com/mysql/mysql-storage-engines.php> (pridobljeno 22. 8. 2023).
- [26] *Numpy*. 2023. URL: <https://github.com/numpy/numpy> (pridobljeno 15. 7. 2023).
- [27] *Oracle*. URL: <https://www.oracle.com/database/> (pridobljeno 19. 8. 2023).
- [28] *Pandas*. 2023. URL: <https://github.com/pandas-dev/pandas> (pridobljeno 15. 7. 2023).
- [29] *Peewee*. 2023. URL: <https://github.com/coleifer/peewee> (pridobljeno 20. 7. 2023).

-
- [30] 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>.
- [31] *PIP*. URL: <https://pypi.org/project/pip/> (pridobljeno 20. 8. 2023).
- [32] *Polars*. URL: <https://pypi.org/project/polars/> (pridobljeno 20. 8. 2023).
- [33] *PostgreSQL*. URL: <https://www.postgresql.org/> (pridobljeno 19. 8. 2023).
- [34] *C++ std::priority_queue*. URL: https://en.cppreference.com/w/cpp/container/priority_queue (pridobljeno 22. 7. 2023).
- [35] *Python docs - Installing Packages*. URL: <https://packaging.python.org/en/latest/tutorials/installing-packages/> (pridobljeno 9. 8. 2023).
- [36] *PyBind11*. 2023. URL: <https://github.com/pybind/pybind11> (pridobljeno 15. 7. 2023).
- [37] *PyPI Stats*. URL: <https://pypistats.org/> (pridobljeno 20. 7. 2023).
- [38] *Python - Common Object Structures*. URL: <https://docs.python.org/3/c-api/structures.html> (pridobljeno 19. 8. 2023).
- [39] *Python - venv*. URL: <https://docs.python.org/3/library/venv.html> (pridobljeno 22. 7. 2023).
- [40] *Rust*. URL: <https://www.rust-lang.org/> (pridobljeno 20. 8. 2023).

-
- [41] Janez Sedeljšak. *Graphenix*. 2023. URL: <https://github.com/JanezSedeljsak/graphenix> (pridobljeno 2. 7. 2023).
- [42] *SQLAlchemy*. 2023. URL: <https://github.com/sqlalchemy/sqlalchemy> (pridobljeno 20. 7. 2023).
- [43] *SQLite*. URL: <https://www.sqlite.org/index.html> (pridobljeno 20. 8. 2023).
- [44] *SQLObject*. 2023. URL: <https://github.com/sqlobject/sqlobject> (pridobljeno 20. 7. 2023).
- [45] *TensorFlow*. 2023. URL: <https://github.com/tensorflow/tensorflow> (pridobljeno 15. 7. 2023).
- [46] *Tortoise ORM*. 2023. URL: <https://github.com/tortoise/tortoise-orm> (pridobljeno 20. 7. 2023).
- [47] *UltraJSON*. 2023. URL: <https://github.com/ultrajson/ultrajson> (pridobljeno 15. 7. 2023).
- [48] *unittest* — *Unit testing framework*. URL: <https://docs.python.org/3/library/unittest.html> (pridobljeno 20. 7. 2023).
- [49] *C++ std::vector*. URL: <https://cplusplus.com/reference/vector/vector/> (pridobljeno 20. 7. 2023).