

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Janez Sedeljšak

# **Razvoj vgrajenega podatkovnega sistema**

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM  
PRVE STOPNJE  
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2023

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](http://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<sup>A</sup>T<sub>E</sub>X.*

**Kandidat:** Janez Sedeljšak

**Naslov:** Razvoj vgrajenega podatkovnega sistema

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

**Mentor:** doc. dr. Boštjan Slivnik

**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 an embeded database system

**Description:**

opis diplome v angleščini



# Kazalo

**Povzetek**

**Abstract**

<b>1</b>	<b>Uvod</b>	<b>1</b>
1.1	Različni tipi podatkovnih baz . . . . .	1
1.1.1	Relacijske podatkovne baze . . . . .	1
1.1.2	Nerelacijske podatkovne baze . . . . .	1
1.2	Kje uporabljamo relacijske podatkovne baze in kako delujejo?	2
1.3	Motivacija za razvoj lastnega relacijskega sistema . . . . .	3
1.3.1	Težava s katero se srečujemo pri uporabi modernih ORM knjižnic . . . . .	4
	Statističen pregled Python ORM knjižnic . . . . .	4
<b>2</b>	<b>Sodoben pristop razvoja knjižnice za programski jezik Python</b>	<b>9</b>
2.1	Kako pohitriti izvajanje v programskem jeziku Python . . . . .	9
2.1.1	Primer uporabe Python.h API vmesnika . . . . .	11
2.2	Naprednejša knjižnica z dodatnimi strukturami za lažji prehod med jezikoma . . . . .	11
2.2.1	Primer uporabe PyBind11 knjižnice . . . . .	12
<b>3</b>	<b>Razvoj jedra podatkovnega sistema</b>	<b>13</b>
3.1	Struktura shranjevanja podatkov . . . . .	14

3.1.1	Datotečna struktura za shranjevanje podatkov znotraj entitet . . . . .	14
3.1.2	Tipi podatkov . . . . .	15
3.1.3	Realizacija relaciji med entitetami . . . . .	16
3.2	Indeksiranje z uporabo B+ dreves . . . . .	17
3.2.1	B+ drevesa in njihove značilnosti . . . . .	17
3.2.2	Pravila, ki jih zahteva B+ drevesna struktura . . . . .	18
3.2.3	Implementacija iskanja v MySQL in SQLite DMBS . . . . .	18
3.2.4	Implementacija B+ dreves za shranjevanje na disk . . . . .	18
3.2.5	Uporaba B+ dreves v bazi podatkov (iskanje po indeksiranih podatkih) . . . . .	19
3.2.6	Dinamično nalaganje in ohranjanje posameznih segmentov drevesa v pomnilnik . . . . .	20
3.2.7	Slabosti uporabe indeksov . . . . .	20
3.2.8	Ostali pristopi indeksiranja podatkov . . . . .	20
3.3	Pomembni gradniki za optimalno izvedbo poizvedb . . . . .	21
3.3.1	Gradnja dreves za pogojni del poizvedb . . . . .	21
3.3.2	Izvedba poizvedb na več entitetah hkrati . . . . .	21
3.3.3	Optimizacija poizvedb z gručenjem (ang. clustering) . . . . .	22
	Razlaga postopka gručenja . . . . .	22
	Linearna izvedba gručenja nad urejenim seznamom . . . . .	23
3.3.4	Optimizacija na nivoju urejanja podatkov ob izvedbi nabora nad posamezno entiteto . . . . .	24
3.3.5	Uporaba podatkovnih okvirjev za nalaganje podatkov . . . . .	27
3.4	Podprte funkcionalnosti na nivoju vgrajenega ORM sistema . . . . .	27
<b>4</b>	<b>Analiza</b>	<b>29</b>
4.1	Testno usmerjen razvoj . . . . .	29
4.1.1	Testi enot na nivoju jedra podatkovnega sistema . . . . .	29
4.1.2	Integracijsko testiranje funkcionalnosti na nivoju končne knjižnice . . . . .	30
4.1.3	Performančno testiranje kritičnih segmentov . . . . .	30

4.1.4	Integracija avtomatskega testiranja z GitHub . . . . .	30
4.2	Primerjava z ostalimi relacijskimi sistemi . . . . .	32
4.2.1	Množično vnašanje podatkov . . . . .	32
4.2.2	Velikosti podatkovnih sistemov na disku . . . . .	33
4.2.3	Primerjava poizvedb na eni entiteti . . . . .	34
4.2.4	Primerjava poizvedb z uporabo relaciji . . . . .	35
4.2.5	Filtriranje podatkov s pomočjo indeksiranja . . . . .	36
<b>5</b>	<b>Scenariji oz. primeri uporabe</b>	<b>37</b>
5.1	Strežniško beleženje podatkov . . . . .	37
5.1.1	Aplikacija, ki služi kot osnova za modul beleženja . . .	37
5.1.2	Kreiranje sheme za beleženje podatkov . . . . .	38
5.1.3	Dekorator za dinamično dodajanje beleženja . . . . .	39
5.1.4	Branje zapisanih podatkov . . . . .	40
	Izpis zadnjih treh zahtevkov . . . . .	40
	Statističen pregled zahtevkov . . . . .	41
	Izpis napak v zadnjem dnevu . . . . .	41
	Izvoz vseh podatkov v .csv datoteko . . . . .	41
5.2	Manjši strežniški API . . . . .	42
5.2.1	Podatkovna shema za aplikacijo . . . . .	42
5.2.2	Pretvornik za serializacijo podatkov nad shemo . . . .	43
5.2.3	Kreiranje novega zapisa . . . . .	43
5.2.4	Nabor podatkov profesorjev po laboratorijih . . . . .	43
5.2.5	Nabor podatkov z dinamičnim filtriranjem . . . . .	43
<b>6</b>	<b>Sklepne ugotovitve</b>	<b>45</b>
	<b>Literatura</b>	<b>47</b>





# Seznam uporabljenih kratic

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



# Povzetek

**Naslov:** Razvoj vgrajenega podatkovnega sistema

**Avtor:** Janez Sedeljšak

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

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

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

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



# Abstract

**Title:** Diploma thesis template

**Author:** Janez Sedeljšak

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

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

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

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



# Poglavje 1

## Uvod

Živimo v obdobju, kjer velepodatkov. Gre za ogromne količine podatkov, ki so shranjeni na različnih strežniških storitvah. Kadar gre za trajno shranjevanje podatkov govorimo o podatkovnih bazah. Trenutno se omenjeno področje deli na dve večji skupini – relacijske in nerelacijske podatkovne baze.

### 1.1 Različni tipi podatkovnih baz

#### 1.1.1 Relacijske podatkovne baze

Trenutno so standard na trgu še vedno relacijske podatkovne baze. Gre za striktno strukturo entitet, ki vsebujejo smiselne povezave – relacije z implementacijo tujih ključev. Gre za standard, ki se je prvič pojavil leta 1970, ko ga je razvil IBM [6]. Razvita je bila prva družina relacijskih podatkovnih baz DB2, katero je razvil Edgar F. Codd – matematik izobražen na univerzi Oxford. **Razširi zgodovino relacijskih podatkovnih baz**

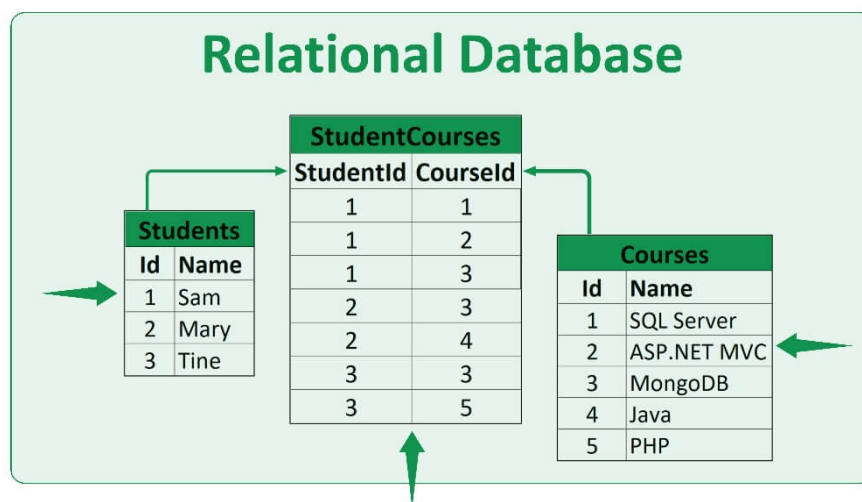
#### 1.1.2 Nerelacijske podatkovne baze

Gre za novo skupino podatkovnih baz, ki temeljijo na čisto drugačni osnovi kot relacijske podatkovne baze. Pojavile so se kot odgovor na težave, s katerimi se srečujemo pri relacijskih podatkovnih bazah. Kot je zapisal viš. pred.

Aljaž Zrnc "Podatkovne baze NoSQL niso bile razvite z namenom popolne zamenjave relacijskih baz" [23]. Glavna težava pri relacijskih podatkovnih bazah je striktna struktura, ki se je moramo držati. V novi skupini podatkovnih baz (NoSQL) je prioriteta fleksibilnost. Sama struktura podatkov je bistveno drugačne, saj entitete in relacije med zapisi zamenjajo objekti in dedovanje. Vse skupaj prinese še eno veliko prednost, ki jo imajo nerelacijske podatkovne baze – zaradi enkapsulacije posameznih zapisov lahko celotno bazo porazdelimo na več računalnikov (tako imenovana horizontalna skalabilnost, ki je relacijske podatkovne baze ne podpirajo). [Najdi boljše članke](#)

## 1.2 Kje uporabljamo relacijske podatkovne baze in kako delujejo?

Relacijske podatkovne baze se uporabljajo povsod, kjer imamo velike količine strukturiranih podatkov, ki jih želimo obdržati za trajno shranjevanje. Preprost primer relacijske podatkovne baze:



Slika 1.1: Preprost primer relacijske podatkovne baze.

Zamenjaj sliko z lastno verzijo - sprememba sheme na User, Messages, Tasks



Na primeru je prikazana struktura, kjer shranjujemo zapise uporabnikov in , ki se jih udeležujejo. Vsak zapis v relacijski podatkovni bazi ima svoj primarni ključ, preko katerega potekajo vse relacije znotraj podatkovne baze. Entiteti "Students" in "Courses" sta starševski entiteti. "StudentCourses", pa predstavlja povezovalno entiteto med prej omenjenima. Namreč vsak študent je lahko v več predmetih in enako velja za predmete (lahko vključujejo več študentov).

### 1.3 Motivacija za razvoj lastnega relacijskega sistema

Koncept shranjevanja podatkov je v osnovi dokaj preprost. Različni sistemi za shranjevanje podatkov bodisi relacijski ali nerelacijski sistemi za shranjevanje podatkov s seboj prinesejo ogromno abstrakcije.

Uporabniki podatkovnih baz se zares začnejo zavedati težav, ko sistem za shranjevanje podatkov ni več odziven kot bi si želeli. Tekom razvoja lastnih aplikaciji in preprostih API-jev, se le redko posvetimo optimalnemu delovanju naše podatkovne baze. Težave se začnejo pojavljati, kadar v posamezni entiteti pride do velike količine podatkov in poizvedbe nad podatki in samo iskanje posameznih zapisov postane zakasnjeno.

Na tej točki so potrebne optimizacije same strukture naše podatkovne baze. Eden najpomembnejših konceptov za hitro iskanje po posameznih atributih je postavitvev indeksov. Indeksiranje podatkov je koncept, ki se pojavlja povsod v računalništvu in ne le v relacijskih podatkovnih bazah. Gre za pripravo iskalne strukture, ki nam omogoča bistveno hitrejšo iskanje podatkov s pomočjo dobre podatkovne strukture. V praksi se izkaže, da sta nekako najbolj pogosta pristopa indeksiranje z zgoščevalnimi strukturami in drevesnimi strukturami. V smislu relacijskih podatkovnih baz so najpogostejše uporabljata več nivojsko indeksiranje, ki je realizirano prav z drevesi (v večini primerov gre za B drevesa). Gre za skupino dreves, kjer ima vsako vozlišče lahko  $M$  zapisov in  $M + 1$  kazalcev na nova vozlišča. S pomočjo po-

stavitve indeksov lahko linearno iskanje skozi zapise spremenimo v binarno iskanje (oz. iz aproksimacijske notacije  $O(N)$  v  $O(\log(N))$ ).

### 1.3.1 Težava s katero se srečujemo pri uporabi modernih ORM knjižnic

#### 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 [13], kjer imamo sledenje prenosov posameznih paketov znotraj Python ekosistema. Podatki za število GitHub zvezd, pa so pridobljeni direktno iz repositorijev posameznih knjižnic, ki so dostopne direktno na GitHub platformi.

Knjižnica	GitHub zvezde	Mesečni prenosi	Tedenski prenosi
Django [2]	72 tisoč	10 milijonov	2.4 milijona
SQLAlchemy [16]	7.5 tisoč	83 milijonov	20 milijonov
Peewee [10]	10 tisoč	1.1 milijon	281 tisoč
Tortoise ORM [19]	3.7 tisoč	88 tisoč	22 tisoč
SQLObject [17]	133	27 tisoč	6 tisoč

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

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

- **SQLAlchemy** [16] gre za eno najbolj fleksibilnih in uporabljenih knjižnic znotraj programskega jezika. Sama fleksibilnost pomeni, da številne operacije, ki so znotraj Django ORM že avtomatizirane so tukaj prepuščene implementaciji razvijalca. Fleksibilnost, pa s seboj prinaša tudi bistveno prednost, saj lahko kot razvijalec, ki dobro pozna ozadje delovanja uporabljenega relacijskega podatkovnega sistema uporabimo različne pristope optimizaciji, ki jih znotraj Django ORM ni mogoče realizirati oz. je implementacija bistveno težja. Knjižnica zaradi lahкотne abstrakcije podpira SQLite, MySQL, Oracle, MS-SQL in še ostale DBMS sisteme.
- **Peewee** [10] je ena izmed preprostejših knjižnic, ki predstavlja lahкотen nivo abstrakcije. Knjižnica podpira 3 relacijske podatkovne sisteme, to so PostgreSQL, MySQL in SQLite. Gre za dokaj omejen nabor podprtih relacijskih podatkovnih sistemov, vendar zaradi svoje preprostosti knjižnica ima svojo ciljno skupino razvijalcev, ki razvijajo preproste sisteme in ne potrebujejo kompleksnosti, ki jih s seboj prineseta Django ORM in SQLAlchemy.

Vsem omenjenim knjižnicam je skupna ciljna ideja – pospešiti postopek razvoja aplikacije, ki bo upravljala s podatki in zagotoviti dodatno varnost nad poizvedbami, ki se izvajajo nad posameznimi entitetami, ter dodatne potrebne validacije na nivoju vnašanja podatkov, ki jih sicer klasični DMBS sistemi ne omogočajo.

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

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

pomočjo tujih ključev. Pogosto, pa kot ciljni uporabniki ne želimo tega, temveč želimo drugačno strukturo podatkov, ki pa ni v skladu z osnovno idejo relacijskih baz. V ta namen ORM knjižnice izvedejo več različnih poizvedb in potem znotraj jedra knjižnice združujejo zapise in kreirajo ciljno strukturo ali pa izvedejo eno kompleksnejšo poizvedbo in pridobljene podatke nato pretvorijo v končno strukturo. Primer, take preproste naloge je: "Ža vse uporabnike pridobi njihove naloge in zadnjih pet sporočil, ki jih je vsak izmed teh uporabnikov poslal".

Za nalogo je SQLAlchemy jedro pripravilo naslednjo poizvedbo:

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

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

**Izrek 1.1** *Skupno število zapisov, ki jih pridobimo je definirano s pomočjo naslednje vsote*

$$\sum_{i=1}^N T_i \cdot M_i \quad (1.1)$$

**Izrek 1.2** *Skupno število zapisov, ob idealnih pogojih (brez podvajanja podatkov)*

$$N + \sum_{i=1}^N T_i + M_i \quad (1.2)$$

Torej najprej potrebujemo  $N$  zapisov za vsakega uporabnika, nato pa za vsakega uporabnika pridobimo še  $T_i$  nalog in  $M_i$  nalog. Če predpostavimo, da imamo podatkovno bazo, kjer imamo sto uporabnikov in ima vsak izmed njih tisoč sporočil, ter dvesto nalog pomeni, da bomo v primeru prve poizvedbe (1.1) prenesli, kar  $100 \cdot 1000 \cdot 200$  oz. 20 000 000 zapisov.

Medtem, ko v primeru druge poizvedbe (1.2) to pomeni prenos bistveno manjše strukture ( $100 \cdot (1000 + 200)$  oz. 120 000. Kar predstavlja le 0.6% količine podatkov, ki jih je potrebno prenesti iz podatkovnega sistema do naše ORM implementacije.

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



## Poglavje 2

# Sodoben pristop razvoja knjižnice za programski jezik Python

### 2.1 Kako pohitriti izvajanje v programskem jeziku Python

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

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

Področji umetne inteligence in podatkovne analitike, pa temeljita na obdelavi masovnih podatkov. Kar pa posledično pomeni, da Python ni naj-

bolj optimalen jezik za izvedbo teh nalog. Zaradi teh omejitev so nastale knjižnice, ki so v osnovi napisane v jezikih, ki so bistveno hitrejši. Tako se funkcije in strukture, ki jih potem lahko uporabljamo na nivoju programskega jezika Python prevedejo že v strojno kodo in jih potem lahko kličemo direktno iz programskega jezika Python. Med temi knjižnicami so pogostejše uporabljene:

- **Numpy** [8] – odprto kodna knjižnica, ki v Python prinese podatkovni tip polja (ang. array), in veliko vgrajenih funkcij, ki bistveno pospešijo operacije delo s polji, kot tudi matrikami, ki v področju podatkovne analitike in umetne inteligence predstavljajo enega izmed glavnih temeljev. Veliko je tudi knjižnic, ki za delovanje uporabljajo ravno Numpy. Tak primer je npr. Pandas [9] – gre za eno izmed najpogostejše uporabljenih knjižnic za podatkovno analitiko, ki s seboj prinese še nekaj dodatnih abstrakcij za lažje delo s podatkovnimi okviri (ang. data frame).
- **Tensorflow** [18] – odprto kodna knjižnica za strojno učenje, v osnovi je celoten produkt Tensorflow namenjen širšemu spektru programskih jezikov in ni implementiran le za Python. Gre za splošno namensko knjižnico z že pripravljenimi metodami za kreiranje modelov za strojno učenje, kar bistveno olajša delo programerju, saj je velikosti postopkov avtomatiziranih.
- **UltraJSON** [20] – gre za manj poznano knjižnico, ki dela z JSON strukturami. Glavni funkcionalnosti knjižnice sta pretvorba Python strukture v JSON in obratno. JSON je v zadnjih letih postal glavni standard za prenos podatkov preko HTTP zahtevkov, posledično je vsaka optimizacija na nivoju obdelave JSON strukture zelo dobrodošla.

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



### 2.1.1 Primer uporabe Python.h API vmesnika

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

```
#include <Python.h>

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

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

Funkcija je sestavljena tako, da v prvi fazi argumente shrani na naslove spremenljivk, ki so deklarirane na nivoju programskega jezika C (to sta v zgornjem primeru *int64\_t a, b;*). Argumenti so poslani kot terka, za katero moramo najprej povedati strukturo, v zgornjem primeru je to *ii*, kar pomeni dve celi števil. Seveda, lahko kot argumente beremo tudi ostale strukture vendar branje postane bistveno bolj kompleksno, ko želimo brati sestavljene tipe. V tem primeru je v večini primerov lažje preprosto pretvoriti vse v primitivne tipe in na nivoju programskega jezika C operirati direktno s temi.

## 2.2 Naprednejša knjižnica z dodatnimi strukturami za lažji prehod med jezikoma

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

PyBind11 [12]. Gre za knjižnico, ki bistveno olajša pretvorbe podatkovnih struktur med jezikoma npr. kadar želimo, kot parameter poslati strukturo seznama v C++ lahko to preprosto definiramo, kot tip vector in pretvorba iz seznama v vektor bo avtomatska. Knjižnica deluje s pomočjo C++ verzije 11, ki je izšla septembra 2011.

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

### 2.2.1 Primer uporabe PyBind11 knjižnice

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

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

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

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

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

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

## Poglavje 3

# Razvoj jedra podatkovnega sistema

V poglavju predstavimo razvoj jedra za relacijski podatkovni sistem. V prvi fazi je predstavljena celotna strukturiranje podatkov in način shranjevanja na disk. Za tem predstavimo razvoj B+ drevesne strukture za optimalno indeksiranje podatkov. Nato, pa je predstavljena še optimizacija posameznih segmentov in sama strukture kode na nivoju jedra podatkovnega sistema. Vsa izvorna koda je dostopna na GitHub repozitoriju [15].

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

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

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

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

## 3.1 Struktura shranjevanja podatkov

### 3.1.1 Datotečna struktura za shranjevanje podatkov znotraj entitet

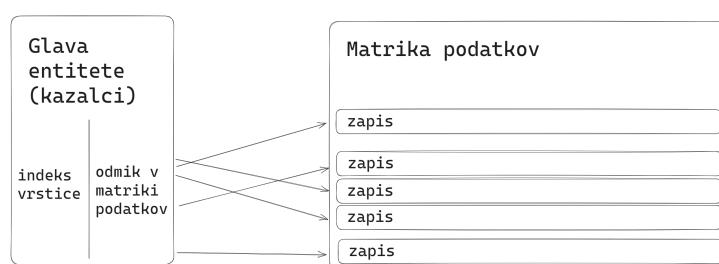
Za vsako posamezno entiteto se privzeto kreirata 2 datoteki *ix\_ < naziv\_entitete > .bin* in *< naziv\_entitete > .bin*.

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

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

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

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



Slika 3.1: Struktura podatkovne matrike in glave entitete s kazalci

Popravi sliko - izvedba v tikz Latex

### 3.1.2 Tipi podatkov

Znotraj našega vgrajenega podatkovnega sistema podpiramo 5 podatkovnih tipov:

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

2. Realna števila (ang. Float) – zaradi lažje implementacije je velikost realnih števil enaka, kot velikost celih števil – torej 64 bitov. Vrednost je na nivoju podatkov realizirana, kot dvojni "float" oz. "double". Gre za zaporedje bitov, ki so po standardu IEEE 754 [7] ločeni v predznak, eksponent in mantiso.
3. Nizi (ang. String) – Gre za zaporedje znakov, kjer na nivoju posamezne entitete določimo maksimalno dovoljeno dolžino posameznega zapisa. Torej velikost posameznega zapisa je  $N * 1B$ , kjer je  $N$  maksimalna dolžina podatka.
4. Logične vrednosti (ang. Bool) – je najmanjši podatkovni tip, ki ga vsebuje naš podatkovni sistem in zavzame vsega 1B; drži pa lahko vrednosti 0/1 oz. True/False, kot to definira Python programski jezik.
5. Datumi (ang. Datetime) – gre za podatkovni tip, ki ponovno shranjen, kot vrednost `int64_t` in je predstavljen v EPOCH formatu – gre za časovni odmik trenutnega časa od UTC datuma 1. 1. 1970 [4].
6. Povezava (ang. Link) – gre za vgrajen tip relacije, kjer je podatek kazalec na drug zapis v določeni entiteti. V ozadju je to le primarni ključ določenega zapisa, ki se nastavi avtomatsko (ponovno gre za podatek, ki je zapisan kot `int64_t` vrednost).
7. Virtualna povezava (ang. VirtualLink) – gre za navidezno polje v entiteti, ki ne zavzame nobenega dodatnega prostora za shranjevanje podatkov. Polje je pomembno za kreiranje povezave, kjer na določen zapis večemo seznam zapisov iz druge tabele.

### 3.1.3 Realizacija relaciji med entitetami

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

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

## 3.2 Indeksiranje z uporabo B+ dreves

### 3.2.1 B+ drevesa in njihove značilnosti

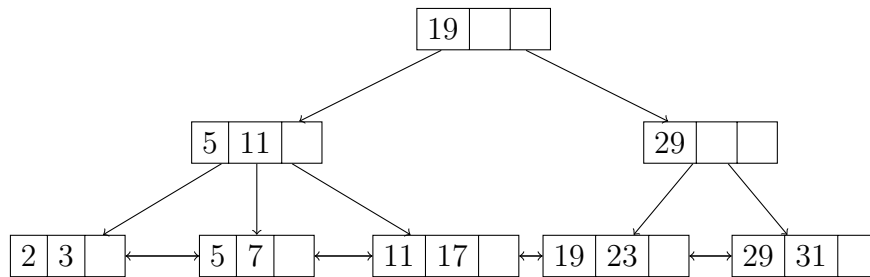
B+ drevesa so podatkovna struktura, uporabljena za učinkovito indeksiranje in iskanje podatkov. Vsako vozišče ima lahko največ  $M$  podatkov in  $M + 1$  kazalcev na nova vozlišča.

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

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

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

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



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

### 3.2.2 Pravila, ki jih zahteva B+ drevesna struktura

1. Vsako vozlišče ima največ  $M$  podatkov in  $M + 1$  kazalcev.
2. Listi drevesa so vsi na enakem nivoju.
3. Urejenost - vse vrednosti, ki se nahajajo levo morajo biti manjše ali enake trenutni; vse vrednosti desno, pa morajo biti večje ali enake.
4. Vmesna vozlišča morajo vsebovati  $X + 1$  kazalcev, kjer je  $X$  število podatkov v vozlišču.
5. Vozlišče je veljavno, kadar ima vsaj  $C/2$  zapisov, kjer je  $C$  kapaciteta oz. maksimalno število podatkov, ki jih lahko ima vozlišče.

### 3.2.3 Implementacija iskanja v MySQL in SQLite DMBS

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

### 3.2.4 Implementacija B+ dreves za shranjevanje na disk

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



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

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

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

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

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

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

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

### 3.2.6 Dinamično nalaganje in ohranjanje posameznih segmentov drevesa v pomnilnik

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

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

### 3.2.7 Slabosti uporabe indeksov

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

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

- Časovna zahtevnost vnašanja in posodabljanja zapisov v posamezno entiteto se poveča iz  $O(1)$  na  $O(\log(N) * X)$ , kjer je  $X$  število postavljenih indeksov.  $N$  pa predstavlja število že obstoječih zapisov v posamezni entiteti.
- Za vsak postavljen indeks se bistveno poveča poraba prostora na disku, saj shranjevanje drevesne strukture s seboj prinese podvajanje atributa, kot tudi dodatek kazalca na glavo entitete.

### 3.2.8 Ostali pristopi indeksiranja podatkov

Indeksiranje s pomočjo zgoščevalnih tabel ...

### 3.3 Pomembni gradniki za optimalno izvedbo poizvedb

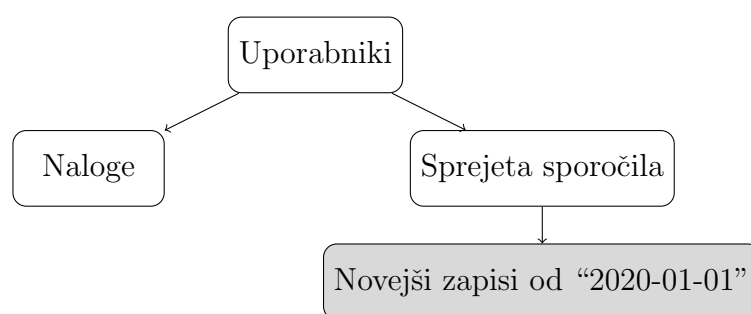
#### 3.3.1 Gradnja dreves za pogojni del poizvedb

Iskanje zapisov glede na strukturo pogoja in kako sestavimo drevesno strukturo

#### 3.3.2 Izvedba poizvedb na več entitetah hkrati

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

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



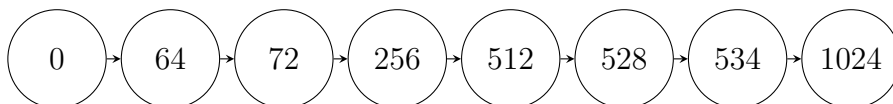
Slika 3.3: Drevesi prikaz poizvedbe, ki se izvaja usmerjeno od vozlišča proti listom

### 3.3.3 Optimizacija poizvedb z gručenjem (ang. clustering)

#### Razlaga postopka gručenja

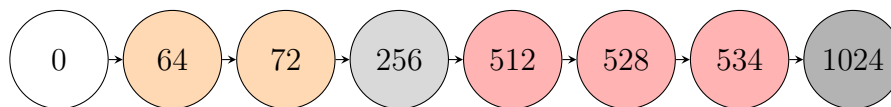
Kot je bilo že omenjeno v prejšnjih poglavjih je največje ozko grlo, s katerim se srečujemo I/O operacije (torej pisanje in branje iz/na disk). V ta namen je tudi v segmentu prenosa podatkov iz datoteke v pomnilnik realizirano s čim manj dostopi do diska. V ta namen je ob prvi fazi nabora dodan algoritem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

Sam problem je zelo preprost imamo urejen vektor celih števil, kjer želimo vrednosti grupirati glede na posamezne odmike in najti skupine oz. gručice 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, kateri bi želeli, da naš algoritem odkrije in branje nad odmiroma izvede v enem branju - to sta gruči [64, 62], ter [512, 528, 540]. Optimalen pristop branja bi lahko barvno označili po naslednji konfiguraciji:



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

### Linearna izvedba gručenja nad urejenim seznamom

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

Spodaj je predstavljen algoritem za gručanje elementov v urejenem seznamu:

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

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

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

    i := j + 1
  end while

  return clusters
end function
```

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

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

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

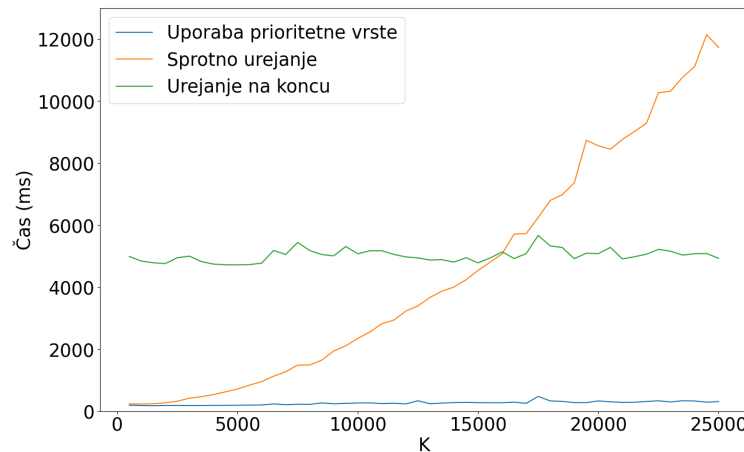
Tekom razvoja smo testirali nekaj različnih pristopov in podatkovnih struktur. Naš problem lahko definiramo, kot iskanje  $K$  najmanjših elementov, kjer je  $K$  sestavljen iz dveh komponent  $L$  in  $O$ , kjer je  $L$  maksimalno število elementov oz. limita,  $O$  pa predstavlja odmik oz. število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

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

1. Klasično vstavljanje v seznam in ureditev elementov ob koncu vnašanja, ter rezanje elementov od indeksa  $O$ , do vključno  $O + L$ . Časovna zahtevnost tega pristopa je  $O(N \cdot \log(N))$ , prostorska zahtevnost, pa je tukaj  $O(N)$ , saj moramo v pomnilniku držati vse potencialne elemente.
2. Postopno vstavljanje v urejen seznam, v tem primeru uporabimo vgrajeno `std::vector` [22] strukturo, kjer vstavljanje izvajamo s pomočjo binarnega iskanja. Težava pristopa se skriva v vstavljanju v začetni del seznama, saj moramo potem vse elemente zamikati za eno mesto v desno. Posledično, čeprav je časovna zahtevnost binarnega iskanja  $O(\log(N))$  zaradi premikanja elementov časovna zahtevnost postane  $O(N)$ . Tako je časovna zahtevnost celotnega postopka vstavljanja  $O(N^2)$  v najboljšem primeru pa  $O(N \cdot \log(K))$ . 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` [11]. V tem primeru je časovna zahtevnost posameznega vstavljanja  $O(\log(K))$ , kar pomeni za  $N$  vstavljanj  $O(N \cdot \log(K))$ . 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` [22], ki je na nivoju pomnilnika shranjen v enem kosu, kar omogoči prevajalniku in procesorju, da izrabita še optimizacijo z lokalnostjo.

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



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



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

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

### 3.3.5 Uporaba podatkovnih okvirjev za nalaganje podatkov

## 3.4 Podprte funkcionalnosti na nivoju vgrajenega ORM sistema

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

za malo drugačno izvedbo, kot je to pri klasičnih relacijskih sistemih, saj v našem primeru podatkov ne združujemo v eno matrično strukturo, temveč so podatki združeni v drevesno strukturo, kjer povezave realiziramo z dedovanjem. Na primer, če bi imeli sledečo poizvedbo na nivoju relacijske podatkovne baze:

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

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

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

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

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

# Poglavje 4

## Analiza

### 4.1 Testno usmerjen razvoj

Da zagotovimo pravilno delovanje nekega bolj kompleksnega sistema je na področju razvoja programske opreme zelo priporočen testno usmerjen razvoj. Gre za način validacije pravilnega delovanja posameznih komponent sistema, kot ločeni segmenti in kot celotna enota. V ta namen testno usmerjen razvoj delimo na tri testne faze.

#### 4.1.1 Testi enot na nivoju jedra podatkovnega sistema

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

### 4.1.2 Integracijsko testiranje funkcionalnosti na nivoju končne knjižnice

Naslednja faza testiranja je realizirana s pomočjo integracijskih testov. Gre za pristop testiranja, kjer preizkusimo ali prej posamezno testirane enote delujejo tudi na nivoju integracije ene z drugo [1]. Iz praktičnega vidika, če smo prej testirali ali pravilno delujejo posamezne funkcionalnosti B+ dreves, lahko zdaj testiramo ali se B+ drevo kreira pravilno na nivoju vnašanja podatkov v entiteto, kjer imamo določen atribut definiran kot indeksiran. Sama izvedba testiranja je izvedena na nivoju programskega jezika Python z vgrajeno knjižnico Unittest [21].

### 4.1.3 Performančno testiranje kritičnih segmentov

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

### 4.1.4 Integracija avtomatskega testiranja z GitHub

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

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

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

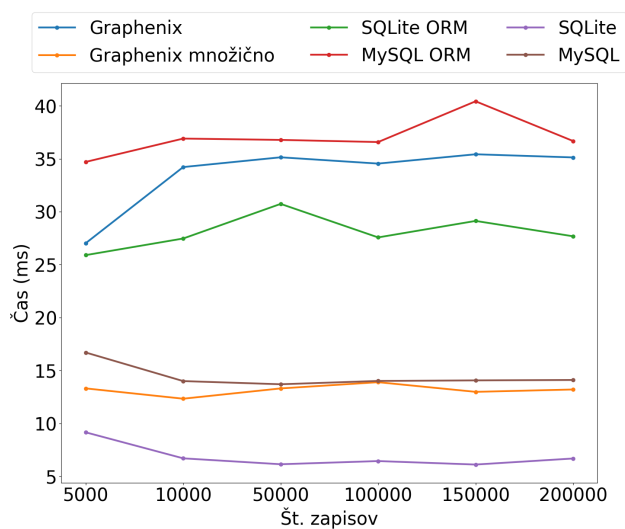
1. Zagon virtualnega okolja za Python programskega okolja (venv [14]).
2. Postavitev potrebnih knjižnic za delovanje jedra (PyBind11 [12]).
3. Namestitev jedra podatkovnega sistema in knjižnice na nivoju programskega jezika Python.
4. Poskus uvoza jedra podatkovnega sistema v CLI, ki je vgrajen v programski jezik Python.
5. Poskus uvoza celotne knjižnice
6. Izvedba nizko nivojskih testov oz. testov enot v programskem jeziku C++
7. Izvedba integracijskih testov na nivoju programskega jezika Python
8. Izvedba performančnih testov

Takoj, ko se katera izmed akcij ne izvede pravilno se ustavi celoten cevovod izvajanja in se javi napaka, ki je vidna na zavihku GitHub akciji ali direktno na spremembi kode, ki je povzročila težavo.

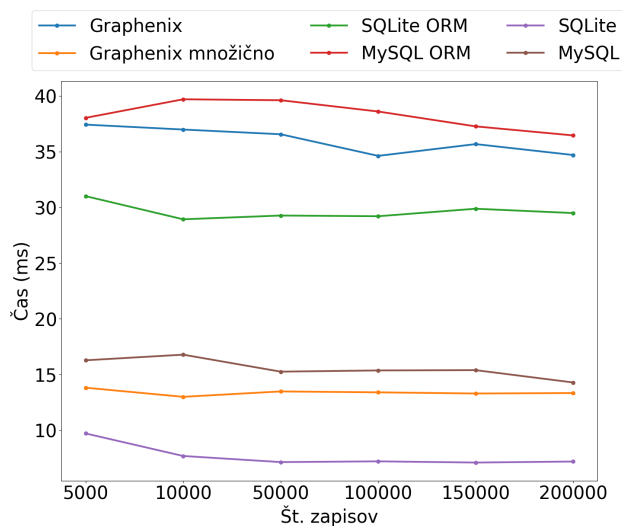
Pristop avtomatskega testiranja in posodabljanja imenujemo tudi CI/CD, ki temelji na hitrem in varnem razvoju, ki razvijalce hitro obvesti o morebitnih napakah, ki se pojavljajo znotraj aplikacije in omogoča, tako avtomatsko testiranje kot tudi avtomatsko posodobitev produkcijskih verziji aplikacije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

## 4.2 Primerjava z ostalimi relacijskimi sistemi

### 4.2.1 Množično vnašanje podatkov

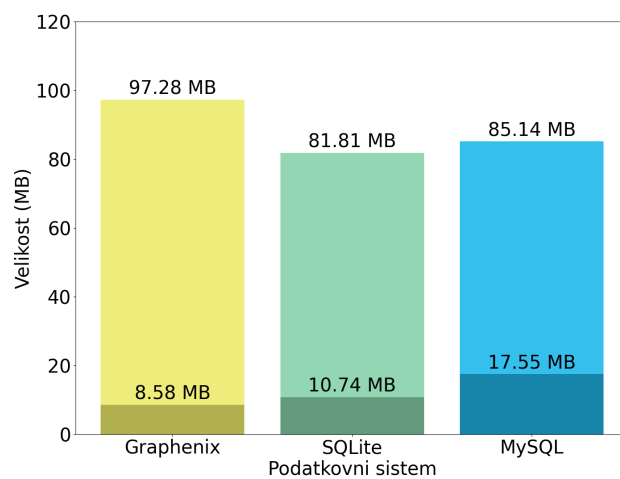


Slika 4.1: Preprost primer relacijske podatkovne baze.



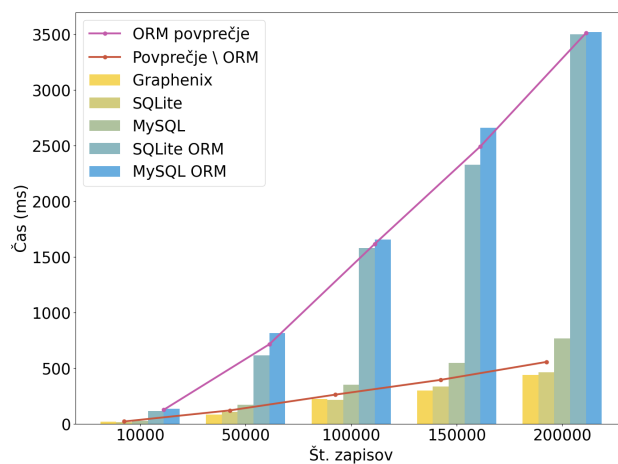
Slika 4.2: Preprost primer relacijske podatkovne baze.

### 4.2.2 Velikosti podatkovnih sistemov na disku

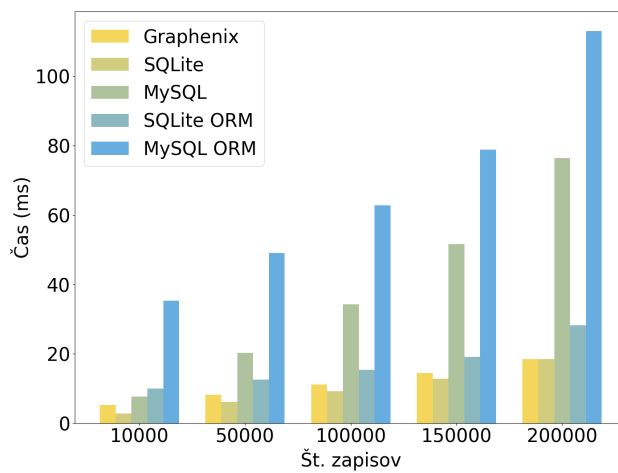


Slika 4.3: Preprost primer relacijske podatkovne baze.

### 4.2.3 Primerjava poizvedb na eni entiteti



Slika 4.4: Preprost primer relacijske podatkovne baze.



Slika 4.5: Preprost primer relacijske podatkovne baze.



#### 4.2.4 Primerjava poizvedb z uporabo relaciji

#### 4.2.5 Filtriranje podatkov s pomočjo indeksiranja

# Poglavje 5

## Scenariji oz. primeri uporabe

### 5.1 Strežniško beleženje podatkov

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

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

#### 5.1.1 Aplikacija, ki služi kot osnova za modul beleženja

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

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

### 5.1.2 Kreiranje sheme za beleženje podatkov

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

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

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

### 5.1.3 Dekorator za dinamično dodajanje beleženja

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

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

            return response

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

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

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

### 5.1.4 Branje zapisanih podatkov

V končni fazi je glavna prednost uporabe pripravljene knjižnice za beleženje podatkov struktura shranjenih podatkov. Namreč z uporabo relacijskega podatkovnega sistema dosežemo fiksno strukturo, ki je v primeru klasičnih datotek za beleženje nimamo. Omogočeno nam je tudi bistveno več načinov pridobivanja statistik na nivoju naše strežniške aplikacije, saj lahko podatke filtriramo in po potrebi dodamo še indeksiranje na stolpce, po katerih pogosteje iščemo in urejamo podatke npr. čas zahtevka (timestamp). Na nivoju administratorske aplikacije lahko nato prožimo različne poizvedbe nad *ReqInfo* entiteto.

#### Izpis zadnjih treh zahtevkov

```
# izpis zadnjih treh zahtevkov
```

```
_, reqs = ReqInfo.order(ReqInfo.timestamp.desc()).limit(3).all()
```

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

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

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

### Statističen pregled zahtevkov

Podatke lahko v vgrajenem podatkovnem sistemu tudi grupiramo, in sicer z uporabo `.agg(...)` metode v poizvedbi, lahko definiramo polje po katerem grupiramo podatke, kot tudi agregacije, ki jih želimo izvesti.

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

### Izpis napak v zadnjem dnevu

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

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

### Izvoz vseh podatkov v .csv datoteko

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

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

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

## 5.2 Manjši strežniški API

Strežniške aplikacije, ki zahtevajo fleksibilno in hitro delo s podatkovnim sistemom postajajo vse bolj pogoste. Python predstavlja enega izmed najpogostejše uporabljenih programskih jezikov na omenjenem področju. Obstaja tudi že ogromno pripravljenih knjižnic za izdelavo API aplikaciji, kot tudi za sam podatkovni sistem, ki se uporablja znotraj aplikacije.

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

### 5.2.1 Podatkovna shema za aplikacijo

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

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

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



### 5.2.2 Pretvornik za serializacijo podatkov nad shemo

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

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

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

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

### 5.2.3 Kreiranje novega zapisa

### 5.2.4 Nabor podatkov profesorjev po laboratorijih

### 5.2.5 Nabor podatkov z dinamičnim filtriranjem



## Poglavje 6

# Sklepne ugotovitve

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



# Literatura

- [1] Hanmeet Kaur Brar in Puneet Jai Kaur. “Differentiating integration testing and unit testing”. V: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE. 2015, str. 796–798.
- [2] *Django*. 2023. URL: <https://github.com/django/django> (pridobljeno 20. 7. 2023).
- [3] *Doctest*. 2023. URL: <https://github.com/doctest/doctest> (pridobljeno 2. 7. 2023).
- [4] *Epoch format*. URL: <https://www.maketecheasier.com/what-is-epoch-time/> (pridobljeno 2. 7. 2023).
- [5] *Flask*. 2023. URL: <https://github.com/pallets/flask> (pridobljeno 30. 7. 2023).
- [6] *Relational Database*. URL: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/> (pridobljeno 29. 6. 2023).
- [7] William Kahan. “IEEE standard 754 for binary floating-point arithmetic”. V: *Lecture Notes on the Status of IEEE 754.94720-1776* (1996), str. 11.
- [8] *Numpy*. 2023. URL: <https://github.com/numpy/numpy> (pridobljeno 15. 7. 2023).

- 
- [9] *Pandas*. 2023. URL: <https://github.com/pandas-dev/pandas> (pridobljeno 15. 7. 2023).
  - [10] *Peewee*. 2023. URL: <https://github.com/coleifer/peewee> (pridobljeno 20. 7. 2023).
  - [11] *C++ std::priority\_queue*. URL: [https://en.cppreference.com/w/cpp/container/priority\\_queue](https://en.cppreference.com/w/cpp/container/priority_queue) (pridobljeno 22. 7. 2023).
  - [12] *PyBind11*. 2023. URL: <https://github.com/pybind/pybind11> (pridobljeno 15. 7. 2023).
  - [13] *PyPI Stats*. URL: <https://pypistats.org/> (pridobljeno 20. 7. 2023).
  - [14] *Python - venv*. URL: <https://docs.python.org/3/library/venv.html> (pridobljeno 22. 7. 2023).
  - [15] Janez Sedeljšak. *Graphenix*. 2023. URL: <https://github.com/JanezSedeljsak/graphenix> (pridobljeno 2. 7. 2023).
  - [16] *SQLAlchemy*. 2023. URL: <https://github.com/sqlalchemy/sqlalchemy> (pridobljeno 20. 7. 2023).
  - [17] *SQLObject*. 2023. URL: <https://github.com/sqlobject/sqlobject> (pridobljeno 20. 7. 2023).
  - [18] *TensorFlow*. 2023. URL: <https://github.com/tensorflow/tensorflow> (pridobljeno 15. 7. 2023).
  - [19] *Tortoise ORM*. 2023. URL: <https://github.com/tortoise/tortoise-orm> (pridobljeno 20. 7. 2023).
  - [20] *UltraJSON*. 2023. URL: <https://github.com/ultrajson/ultrajson> (pridobljeno 15. 7. 2023).

- 
- [21] *unittest* — *Unit testing framework*. URL:  
<https://docs.python.org/3/library/unittest.html> (pridobljeno  
20. 7. 2023).
- [22] *C++ std::vector*. URL:  
<https://cplusplus.com/reference/vector/vector/> (pridobljeno  
20. 7. 2023).
- [23] Aljaž Zrnec in sod. “Podatkovne baze nosql”. V: (2011).