Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Pascal Palmer

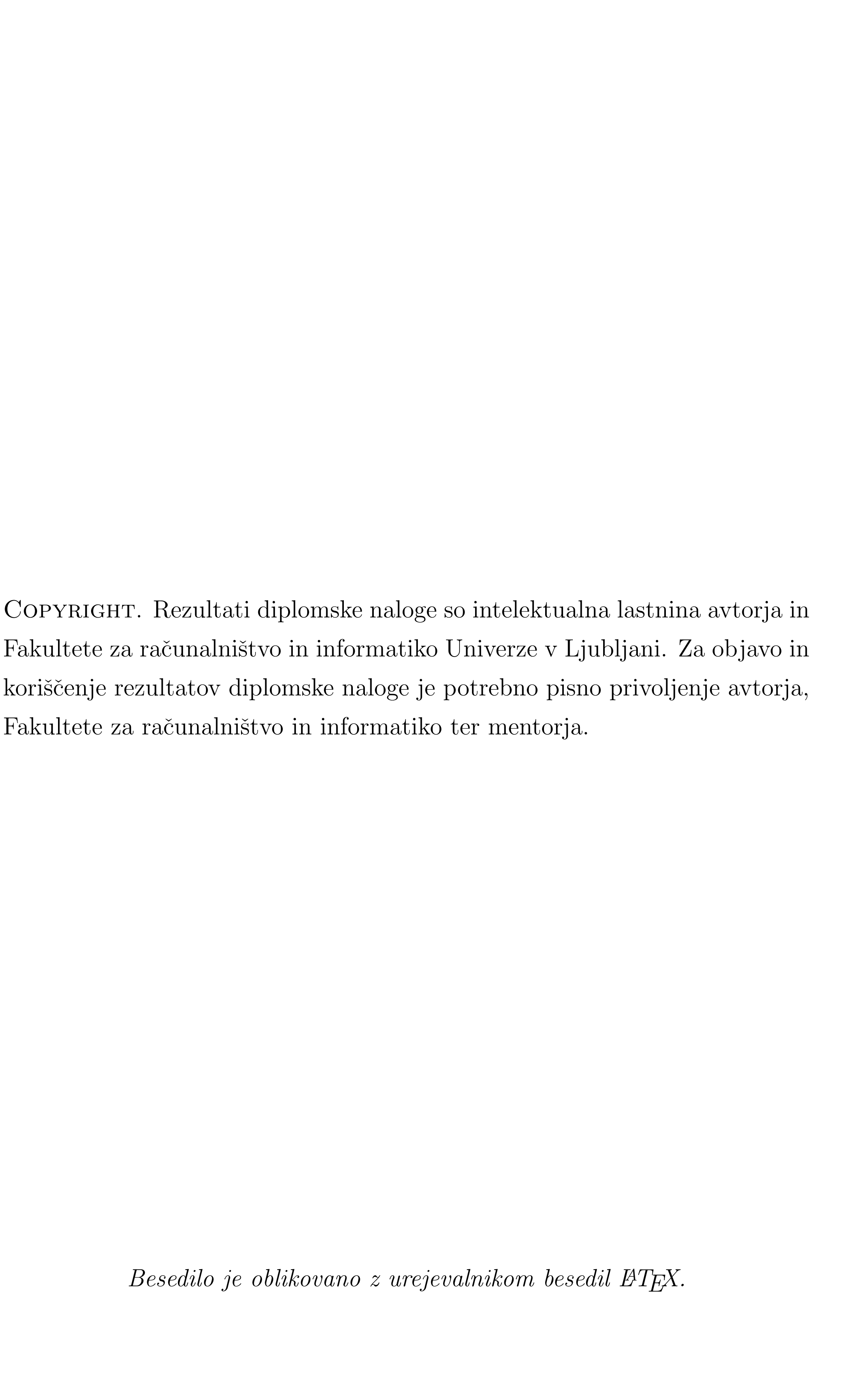
Primerjava prostorske podatkovne baze Tile38 z relacijskimi sistemi

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Matjaž Kukar

Ljubljana, 2018



COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednje delo: Primerjava prostorske podatkovne baze Tile38 z relacijskimi sistemi

Tematika dela:

Preučite specializirano prostorsko podatkovno bazo Tile38. Opišite njene zmožnosti in lastnosti, ter jih primerjate s prostorskimi razširitvami relacijskih podatkovnih sistemov (PostgreSQL z razširitvijo PostGIS in MySQL). Primerjavo predstavite na konkretnem primeru, ter kvalitativno in kvantitativno ovrednotite hitrost delovanja in splošno uporabnost sistemov.

Zahvaljujem se mojemu mentorju izr. prof. dr. Matjažu Kukarju, ki mi je omogočil spoznati še nekako drugačne vrste podatkov, in to so bili prostorski podatki.

Kazalo

Povzetek

Abstract

1 Uvod 1

2 Upravljanje s prostorskimi podatki v relacijskih sistemih 3

2.1 PostgreSQL 3
2.2 PostGIS 4
2.3 MySQL 4
2.4 Prostorski podatkovni tipi 5
2.5 Koordinatni sistemi 7
2.6 Primeri uporabe prostorskih ukazov PostGIS in MySQL 8

3 Tile38 13

3.1 Namestitev strežnika 13
3.2 Značilnosti 14
3.3 GeoJSON 14
3.4 Poizvedbe in ukazi 17

4 OpenStreetMap 23

4.1 Uvažanje OSM-ja v podatkovno bazo 25
4.2 QGIS 30

5 Prikaz dela s prostorskimi podatki **33**
5.1 API Trola.si 33
5.2 Na kateri postaji bo avtobus v kratkem 34
5.3 Prikaz s QGIS 38
5.4 Geografska ograja s Tile38 38

6 Primerjava orodij **47**
6.1 Splošne ugotovitve 47
6.2 Kreiranje podatkovnih struktur in vnos podatkov 49
6.3 Primerjava funkcionalnosti na primerih poizvedb 49

7 Sklepne ugotovitve **55**

Literatura **57**

Seznam uporabljenih kratic

kratica	angleško	slovensko
GPS	Global Positioning System	globalni sistem pozicioniranja
GIS	Geographic(al) Information System	geografski informacijski sistem
JSON	JavaScript Object Notation	notacija za opis JavaScript objektov
OSM	OpenStreetMap	OpenStreetMap
API	Application programming interface	aplikacijski programski vmesnik
SRS	Spatial reference system	prostorski referenčni sistem
SRID	Spatial reference identifier	prostorski referenčni identifikator
WKT	Well-known text	predstavitev prostorskih podatkov v tekstovnem formatu
WKB	Well-known binary	format za predstavitev geometrij v binarnem formatu

Povzetek

Naslov: Primerjava prostorske podatkovne baze Tile38 z relacijskimi sistemi

Avtor: Pascal Palmer

V diplomski nalogi predstavljamo specializirano podatkovno bazo za prostorske podatke Tile38, ki jo primerjamo z relacijskima podatkovnima bazama PostgreSQL in MySQL s katerima tudi lahko upravljamo prostorske podatke. Predstavimo v kakšnih formatih in na kakšen način lahko vnašamo prostorske podatke v izbrane podatkovne baze. Pri vsaki bazi smo opisali nekaj ključnih metod in ukazov za upravljanje prostorskih podatkov. Razvili smo spletno aplikacijo za delo s prostorskimi podatki OpenStreetMap in Trola.si API-jem, kjer pokažemo, kako se Tile38 obnese v primerjavi s PostgreSQL.

Ključne besede: prostorski podatki, Tile38, PostGIS, OpenStreetMap, geografska ograja, GeoJSON.

Abstract

Title: Comparison of spatial database Tile38 with relational systems

Author: Pascal Palmer

In this thesis, we present a specialized database Tile38 for spatial data. For comparison, we chose two relational databases, PostgreSQL and MySQL, that can also be used to manage spatial data. We will show how we can insert spatial data into these databases and in what format. For each database, we have described some of the key methods and commands for managing spatial data. We developed web application to work with OpenStreetMap spatial data and Trola.si API, where we show how Tile38 does compared to PostgreSQL.

Keywords: spatial data, Tile38, PostGIS, OpenStreetMap, geofence, GeoJSON.

Poglavje 1

Uvod

Iz leta v leto imamo več prostorskih podatkov in posledično tudi večjo potrebo po upravljanju take vrste podatkov. Prostorski podatki, ki jim pravimo tudi geoprostorski podatki, predstavljajo lokacijo, obliko in velikost objektov na Zemlji. Ti objekti so na primer: ceste, reke in stavbe. Prostorski podatki vsebujejo tudi lastnosti o posameznem objektu (ulična številka, ime znamenitosti ...) [1].

Z geografskimi informacijskimi sistemi (GIS) upravljamo, analiziramo, vizualiziramo in dostopamo do prostorskih podatkov [2]. Namen diplome je predstaviti in primerjati več (prostorskih) podatkovnih baz, ki ponujajo funkcionalnosti za podporo GIS.

Glavna orodja, ki jih predstavljamo, so:

- PostgreSQL (ver. 10.4), PostGIS (ver. 2.4.4)
- MySQL (ver. 8.0)
- Tile38 (ver. 1.12.3)
- QGIS (ver. 3.2.2)

Uporabljali bomo odprte prostorske podatke OpenStreetMap (OSM) in prikazali delo z njimi. Pogledali si bomo, v kakšnem formatu so ti podatki

oziroma kako jih vnesti v izbrane podatkovne baze, ki sprejemajo tudi prostorske podatke.

Podatkovne baze bomo primerjali predvsem funkcionalno, ker predpostavljamo dejstvo, da lahko enako poizvedbo napišemo na več različnih načinov in tako pride do ne absolutnih meritev časov poizvedb.

Manj znani specializirani prostorski podatkovni bazi Tile38 smo namenili večje poglavje za predstavitev ključnih funkcionalnosti in formata GeoJSON, ki ga baza uporablja za opis prostorskih objektov.

Poglavje 2

Upravljanje s prostorskimi podatki v relacijskih sistemih

2.1 PostgreSQL

PostgreSQL je sistem za upravljanje z objektno relacijskimi podatkovnimi bazami (ORDBMS), ki je baziran na Postgres, projekt razvit na oddelku za računalništvo na Univerzi Berkeley v Kaliforniji. PostgreSQL je odprtokodni naslednik Postgresove kode. Podpira velik del standarda SQL in ponuja mnogo modernih funkcij [3]:

- kompleksne poizvedbe,
- tuji ključi,
- prožilci,
- možnost posodabljanja podatkov v pogledih *(ang. view)*,
- integriteta transakcij,
- upravljanje z več različicami za sočasni dostop.

PostgreSQL lahko uporabnik razširi z različnimi dodatki, ki dodajo:

- podatkovne tipe,

- funkcije,
- operatorje,
- agregatne funkcije,
- metode za indeksiranje,
- proceduralne jezike.

2.2 PostGIS

PostGIS je dodatek za PostgreSQL, ki omogoča dodatne metode in podatkovne tipe za prostorske podatke. Podpira vse prostorske metode in podatkovne tipe opisane v standardu OpenGIS [4], ki ga predpisuje odprt geoprostorski konzorcij (OGC). Ker PostGIS razširja standard OpenGIS, mu pravimo, da je supermnožica funkcionalnosti standarda OpenGIS [5].

Zmožnosti PostGISa so:

- nastavitev koordinatnega sistema podatkom,
- transformacija podatkov v druge koordinatne sisteme,
- vnašanje prostorskih podatkov v formatu WKT, WKB ali GeoJSON,
- pretvarjanje med formati,
- iskanje, kjer se dva prostorska podatka sekata, sta v bližini, ali eden vsebuje drugega,
- računanje površine,
- ter nestandardne prostorske metode (rotiranje in skaliranje prostorskih podatkov).

2.3 MySQL

MySQL je odprtokoden sistem za upravljanje relacijske podatkovne baze. Ponuja enak nabor funkcionalnosti kot PostgreSQL (kompleksne poizvedbe,

tuji ključi, prožilci ...), nima pa dodatkov, s katerimi bi razširili MySQL. Ampak ima že sam po sebi razširitev za prostorske podatke, ki vsebuje metode in podatkovne tipe za prostorske podatke, ki jih najdemo v standardu OpenGIS.

Prostorske zmožnosti je dosti manj kot pri PostGIS, ker podpira le standard OpenGIS. MySQL je brez možnosti za transformacijo podatkov v druge koordinatne sisteme in brez naprednih metod. Na primer, nima metode za rotiranje prostorskih podatkov, kot jo ima PostGIS.

2.4 Prostorski podatkovni tipi

V MySQL in PostGIS vnašamo prostorske podatke v stolpce s podatkovnim tipom geometrija (angl. geometry). Oba sprejemata enak nabor geometrij (prostorskih podatkov), ki jih opišemo v formatu za predstavitev geometrij v tekstovni obliki (WKT) ali formatu za predstavitev geometrij v binarni obliki (WKB).

PostGIS ima tudi prostorski tip geografija *(angl. geography)*, ki je podtip geometrije in je nastavljen tako, da privzeto računa prostorske operacije kot, da bi bili prostorski podatki na sferoidu. Ker je geografija podtip in nima prednosti pred tipom geometrija, bomo uporabljali samo tip geometrija in ga nastavili, da bo tudi ta izvajal prostorske operacije na sferoidu, ko se bomo ukvarjali s geografskimi podatki [6].

2.4.1 Format WKT

Format WKT (Well-known Text) je tudi opisan v standardu OpenGIS, z njim predstavimo geometrije z navadnim tekstom. Večina podatkovnih baz, ki sledi standardom OGC, podpira format WKT.

Z WKT-jem lahko predstavimo točke, črte in poligone. Te geometrijske strukture so dovolj za predstavitev podatkov zemljevida. V tabeli 2.1 smo navedli najbolj pogoste tipe geometrij v formatu WKT in jih tudi grafično predstavili.

geometrijski tip	grafični prikaz	format WKT
Točka		POINT(-122.349 47.651)
Črtni niz (angl. Linestring)		LINESTRING(-122.360 47.656, -122.343 47.656, ...)
Poligon (angl. Polygon)		POLYGON((-122.358 47.653, -122.348 47.649, -122.348 47.658, -122.358 47.658, -122.358 47.653))
Točke (angl. MultiPoint)		MULTIPOINT(-122.360 47.656, -122.343 47.656)
Črtni nizi (angl. MultiLine-String)		MULTILINESTRING ((-122.358 47.653, -122.348 47.649, -122.348 47.658), (-122.357 47.654, -122.357 47.657, -122.349 47.657, -122.349 47.650))
Poligoni (angl. MultiPo-lygon)		MULTIPOLYGON(((-122.358 47.653, -122.348 47.649, -122.358 47.658, -122.358 47.653)), ((-122.341 47.656, -122.341 47.661, -122.351 47.661, -122.341 47.656)))
Zbirka geometrij (angl. GeometryCollection)		GEOMETRYCOLLECTION (POINT(-122.34900 47.65100), LINESTRING(-122.360 47.656, -122.343 47.656), ...)

Tabela 2.1: Prostorski podatki zapisani v formatu WKT [7].

2.4.2 Format WKB

Format WKB (Well-Known Binary) tudi najdemo v standardu OpenGIS, ki se uporablja za predstavitev geometrij v binarni obliki.

Struktura formata WKB je naslednja:

1. Prvi bajt pove vrstni red bajtov (angl. little/big endian).
2. Sledi 4 bajtno število, ki predstavlja, za kateri geometrijski tip gre. S števili od 1 do 7 povemo, da gre za: *Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection*.
3. Nato sledijo koordinate, ki so 8 bajtna števila z dvojno natančnostjo v formatu IEEE 754.

Primer zapisa geometrije *POINT(1 2)*, predstavljeno s hexadecimalnim številom v formatu WKB:

0x0101000000000000000000F03F0000000000000040

To zaporedje lahko zdaj razstavimo na naslednje komponente:

- vrstni red bajtov : 0x01
- geometrijski tip : 0x01000000
- X : 0x000000000000F03F
- Y : 0x0000000000000040

Vrednost 0x01 pomeni, da gre za zapis z malim koncem (angl. little endian) in 0x00 bi pomenilo, da gre za zapis z velikim koncem (angl. big endian). Sledi geometrijski tip z vrednostjo 0x01000000, kar pomeni, da gre za točko in nato sledijo koordinati točke (X, Y) [8].

2.5 Koordinatni sistemi

Koordinatni sistem je pomembni element GIS-a, s katerim določimo, na kakšen način so predstavljeni podatki in v kakšnem razmerju so si.

Pri PostGIS-u in MySQL-u imamo možnost določiti, v katerem koordinatnem sistemu so naši podatki. Vendar pri MySQL-u nimamo možnosti za transformacijo podatkov v druge koordinatne sisteme, kot pri PostGIS.

Ko dodamo razširitev PostGIS, se nam v bazi pojavi tabela *spatial_ref_sys*. Tabela vsebuje podatke za različne koordinatne sisteme oziroma prostorske referenčne sisteme *(angl. Spatial reference system)* in njihove prostorske referenčne identifikatorje (SRID). V kontekstu prostorskih podatkovnih baz, SRS opisuje prostor (enote, meje in obliko koordinatnega sistema), v katerem so geometrije. Definira tudi kako podatke transformiramo v drug prostorski referenčni sistem.

EPSG (European Petrouleum Survey Group) je register, ki skrbi za opise prostorskih referenčnih sistemov, zato pogosto zasledimo zapis prostorskega referenčnega sistema s predpono EPSG. Na primer EPSG:4326, pomeni, da gre za prostorski referenčni sistem s SRID-jem 4326.

Najbolj pogost je elipsoidni prostorski referenčni sistem EPSG:4326, uporablja ga navigacijski sistem GPS, za koordinate uporablja zemljepisno širino in dolžino. Ta drugi znan prostorski referenčni sistem je EPSG:3857. Zato, ker je oblika sistema EPSG:3857 ravnina, se uporablja za projekcijo podatkov na 2D površino. To projekcijo oziroma prostorski referenčni sistem uporabljajo spletni zemljevidi (Google Maps, Bing Maps, OpenStreetMap).

2.6 Primeri uporabe prostorskih ukazov PostGIS in MySQL

2.6.1 Primeri PostGIS

Ko kreiramo bazo v PostgreSQL-u, razširitev PostGIS še ni omogočena. Omogočimo z ukazom:

```
create extension postgis;
```

Zdaj lahko kreiramo tabele s prostorskimi stolpci in vnašamo geometrije, ki so v formatu WKT ali WKB.

```
CREATE TABLE ulicne_svetilke (
   ime varchar,
   geolokacija geometry
);

INSERT INTO ulicne_svetilke (ime, geometrija)
VALUES ('SVETILKA1', ST_GeometryFromText('POINT(50.5 60.7)', 4326));
```

Kreirali smo tabelo s prostorskim stolpcem tipa geometrija in vanjo vnesli točko v formatu WKT s SRID-jem 4326, ki predstavlja ulično svetilko. Metoda **ST_GeometryFromText(string wkt, int srid)** nam pretvori format WKT v WKB, ker so geometrije shranjene v bazi v binarni obliki.

Nakažemo še nekaj prostorskih metod, ki so definirane v standardu OpenGIS:

ST_Distance(geometry g1, geometry g2) Vrne razdaljo med dvema geometrijama

ST_Within(geometry A, geometry B) Vrne vrednost *true*, če je A vsebovan v B

ST_Intersects(geometry geomA , geometry geomB) Vrne vrednost *true*, če se geometriji A in B sekata

Primer za izračun razdalje med dvema točkama z metodo ST_Distance:

```
SELECT ST_Distance(ST_GeomFromText('POINT(1 0)'),
ST_GeomFromText('POINT(1 1)')) as Razdalja;
```

PostGIS ima zmožnost za transformiranje podatkov v druge koordinatne sisteme z metodo **ST_Transform(geometry g, int srid)**.

Za dostop do baze uporabljamo grafični vmesnik pgAdmin 4, ki se namesti skupaj s PostgreSQL-om. Programsko lahko dostopamo do baze s paketom *psycopg2* za Python [9].

Primer programa za dostop do baze v PostgreSQL-u, s katerim izpišemo vse vrstice, ki so v tabeli ulične svetilke:

```
import psycopg2

#PostgreSQL connection
try:
    conn = psycopg2.connect("dbname='slovenia4326' user='postgres'
host='localhost' password='root'")
except:
    print ("Unable to connect")

postgre_cursor = conn.cursor()
postgre_cursor.execute("SELECT * FROM ulicne_svetilke")
rows = postgre_cursor.fetchall()

for row in rows:
    print(row)
```

Slika 2.1: Primer programa za dostop do baze v PostgreSQL-u, s katerim izpišemo vse vrstice, ki so v tabeli ulične svetilke

2.6.2 Primeri MySQL

MySQL prav tako podpira formata WKT in WKB [10].

Tabelo za prostorske podatke v MySQL-u kreiramo tako:

```
CREATE TABLE ulicne_svetilke (
    ime varchar(120),
    geolokacija geometry SRID 4326
);
```

Pri kreiranju tabele smo dodali podatkovnemu tipu geometrija še SRID, ki pomeni, da bo tabela sprejemala samo prostorske podatke s SRID-jem 4326. V zgodnjih verzijah MySQL-a 5.7 SRID ni imel posebne vloge, zato ni bil upoštevan pri prostorskih operacijah nad podatki, takrat so bili izračuni

narejeni kar na navadnem kartezičnem koordinatnem sistemu. MySQL 8.0 zdaj podpira parameter SRID zato verziji 5.7 in 8.0 nista popolnoma kompatibilni [11]. V tabelo vnašamo enako kot pri PostGIS-u:

```
INSERT INTO ulicne_svetilke (ime, geometrija)
VALUES ('SVETILKA1', ST_GeometryFromText('POINT(50.5 60.7)', 4326));
```

Ker MySQL sledi standardu OpenGIS, podpira tudi metode **ST_Distance, ST_Within, ST_Intersects**.

Za upravljanje podatkovne baze uporabljamo grafični vmesnik MySQL Workbench, programsko pa dostopamo s pytonskim paketom *mysql-connector-python* [12].

Poglavje 3

Tile38

Tile38 se je pojavil na GitHubu v letu 2016, zato ji lahko rečemo, da je ena od novejših baz za prostorske podatke, namenjena aplikacijam za iskanje bližnjih objektov, iskanje objektov glede na vsebovanje ali sekanje z drugimi objekti in zaznavanje premikov objektov [13, 14]. Zaenkrat Tile38 nima grafičnega vmesnika. Za izvajanje poizvedb imamo tako na voljo ukazno vrstico. Ker je Tile38 baziran na projektu Redis, lahko skoraj v katerem koli programskem jeziku, kjer imamo na voljo knjižnico Redis, dostopamo do podatkovne baze Tile38.

Redis je odprtokoden projekt, ki hrani podatke v delovnem pomnilniku, uporablja se ga kot podatkovno bazo in posrednika sporočil. Redisu lahko rečemo, da je NoSQL podatkovna baza, ki shranjuje po ključ-vrednost principu [15].

Za dostop do Tile38 strežnika smo uporabljali Python in paket *redis-py* [16].

3.1 Namestitev strežnika

Delovanje strežnika je možno na operacijskem sistemu Windows ali Linux. Namestitev strežnika je precej poenostavljena. Ko strežnik prenesemo na računalnik, je že pripravljen na delovanje, vse kar je potrebno narediti, je

zagnati zagonsko datoteko strežnika. Da je strežnik dostopen preko svetovnega spleta iz našega privatnega omrežja, pa moramo na modemu nastaviti, na kateri lokalni IP-naslov in vrata naj preusmeri *(angl. Port forwarding).*

3.2 Značilnosti

Tile38 omogoča hitro iskanje, ker se vsi podatki nahajajo v delovnem pomnilniku. Strežnik privzeto odgovarja v formatu JSON. V Tile38 lahko shranjujemo geometrije, ki so točke in pravokotniki. Ostale geometrije vnesemo v formatu GeoJSON. Baza privzeto uporablja sistem EPSG:4326 in ne moremo spreminjati prostorskega referenčnega sistema.

Najbolj značilne funkcionalnosti oziroma ukaze, ki jih baza ponuja, so:

- **NEARBY** je ukaz za iskanje objektov v bližini.
- **WITHIN** je ukaz, s katerim poiščemo objekte, ki so popolnoma znotraj nekega objekta.
- **INTERSECTS** je ukaz, s katerim poiščemo objekte, ki se sekajo z nekim objektom.
- **Geografska ograja** *(angl. geofence)*, je funkcionalnost, s katero strežniku povemo, da naj opazuje premike objektov na nekem območju.

3.3 GeoJSON

GeoJSON je format za izmenjavo prostorskih podatkov, ki je podoben formatu JSON. Format uporabljamo za zapis različnih geografskih objektov oziroma geometrij. Podpira naslednje tipe geometrij: *Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon* [17]. Povemo lahko tudi ali gre za objekt *Feature* ali zbirko objektov *FeatureCollection.* Pri tem lahko dodamo objektom tudi lastnosti *(angl. properties).*

3.3.1 Primer objekta zapisanega z GeoJSON

Na takšen način zapišemo ulično svetilko, ki ima ime in atribut ali je prižgana.

```
{
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [102.0, 0.5]
            },
            "properties": {
                "ime": "Ulicna svetilka 0",
                "prizgana" : "ja"
            }
}
```

3.3.2 Primer zbirke objektov

Če imamo geografski objekt, ki je sestavljen iz več objektov (na primer park), uporabimo tip *FeatureCollection*.

```
{
        "type": "FeatureCollection",
        "features": [{
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [
                        [40.0, 40.0],
                        [50.0, 40.0],
                        [50.0, 50.0],
                        [40.0, 50.0],
```

```
                    [40.0, 40.0]
                ]
            ]
        },
        "properties": {
            "opis": "Park",
        }
    },{
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": [55.0, 42.0]
        },
        "properties": {
            "opis": "klopca",
           "barva" : "bela"
        }]
}
```

Zgornje primere Tile38 lepo sprejme, lahko pa uporabljamo tudi skrajšan format. Primer zapisa točke (x: 100, y: 0).

```
{
         "type": "Point",
         "coordinates": [100.0, 0.0]
}
```

V *type* navedemo tip prostorskega podatka, in v *coordinates* zapišemo koordinati točke v vrstnem redu x, y.

3.3.3 Opis cest in rek

Opis cest, rek in podobnih geografskih objektov lahko opišemo s črtnim nizom *(angl. LineString)*.

```
{
        "type": "LineString",
        "coordinates": [
            [100.0, 0.0],
            [101.0, 1.0]
        ]
}
```

3.3.4 Opis oblike območja ali stavb

Območja in oblike stavb opišemo s tipom poligon *(angl. Polygon)*.

```
{
        "type": "Polygon",
        "coordinates": [
            [
                [100.0, 0.0],
                [101.0, 0.0],
                [101.0, 1.0],
                [100.0, 1.0],
                [100.0, 0.0]
            ]
        ]
}
```

3.4 Poizvedbe in ukazi

Podatkovni model pri Tile38 je preprost in temelji na množici parov (ključ, identifikator), ki se preslikajo v prostorski objekt (točka, mejni pravokotnik ali objekt zapisan v formatu GeoJSON). Ključ je v tem kontekstu kot nekakšna zbirka, identifikator pa ime objekta. Pri tem podatkovnem modelu imamo naslednje vrste ukazov:

SET key id spatial_object S ključno besedo SET povemo, da želimo narediti zapis v bazi. Navedemo pod kateri ključ *(angl. key)* in identifikator *(angl. id)*, bomo shranili prostorski podatek.

FIELD name value FIELD uporabimo v kombinaciji s SET, da dodamo številske atribute objektov pri vnosu v bazo.

FSET key id field_name value Če je objekt že v bazi, dodamo številski atribut s FSET.

GET key id GET nam vrne objekt iz zbirke.

SCAN key SCAN nam izpiše vse objekte v zbirki.

NEARBY key spatial_object Ukaz za iskanje objektov v bližini (slika 3.1).

WITHIN key spatial_object Ukaz za iskanje objektov, ki so popolnoma vsebovani znotraj drugega objekta (slika 3.2).

INTERSECTS key spatial_object Iskanje objektov, ki se sekajo z nekim objektom (slika 3.3).

WHERE field_name min max Ukaz WHERE uporabimo v kombinaciji z drugimi ukazi (SCAN, NEARBY, WITHIN ...), s katerim filtriramo rezultate glede na dodatne številske atribute, ki smo jih dodali s FIELD/FSET.

Podatke vnašamo v zbirke, ki se samodejno kreirajo, ko vanjo dodamo prvi podatek:

```
SET hoteli hotel_lev POINT 46.056206 14.502361
```

V zbirko hoteli smo dodali točko z zemljepisno širino 46.056206 in zemljepisno dolžino 14.502361, ki predstavlja lokacijo hotela lev.

Primer s ključno besedo BOUNDS, ki predstavlja objekt pravokotne oblike, predstavljen z najbolj jugozahodno in severovzhodno točko.

```
SET parkirisca parkirisce_lev BOUNDS 46.055976
14.502514 46.056153 14.502672
```

Ker ima Tile38 definirana samo dva prostorska objekta (točka in pravokotnik), moramo vse ostale vrste prostorskih objektov (geometrij) vnesti v formatu GeoJSON. Pri tem uporabimo ključno besedo OBJECT

```
SET svetilke svetilka2 OBJECT { "type": "Point","coordinates":
 [14.502361, 46.056206]}
```

Da dobimo podatke za določeni objekt, uporabimo ukaz GET, če pa želimo izpisati vse objekte v zbirki pa SCAN

```
127.0.0.1:9851> GET svetilke svetilka2
{"ok":true,"object":{"type":"Point","coordinates":[14.502361,46.056206]},
"elapsed":"0s"}
127.0.0.1:9851> SCAN svetilke
{"ok":true,"objects":[{"id":"svetilka2","object":{"type":"Point",
"coordinates":[14.502361,46.056206]}},{"id":"svetilka3","object":
{"type":"Point","coordinates":[14.502361,46.057]}}],"count":2,
"cursor":0,"elapsed":"994.7µs"}
```

Podatkom dodamo številski atribut s ključno besedo FIELD

```
SET parkirisca parkirisce_lev FIELD max_kapaciteta 20
 POINT 46.056206 14.502361
```

Če je objekt že v bazi, dodamo številski atribut s ključno besedo FSET

```
FSET parkirisca parkirisce_lev max_kapaciteta 20
```

V Tile38 je možno delno filtriranje rezultatov glede na dodatna polja, ki smo jih vnesli z ukazi FIELD/ FSET. Z naslednjim ukazov poiščemo vsa parkirišča, ki imajo kapaciteto med 20 in 40.

```
SCAN parkirisca WHERE max_kapaciteta 20 40
```

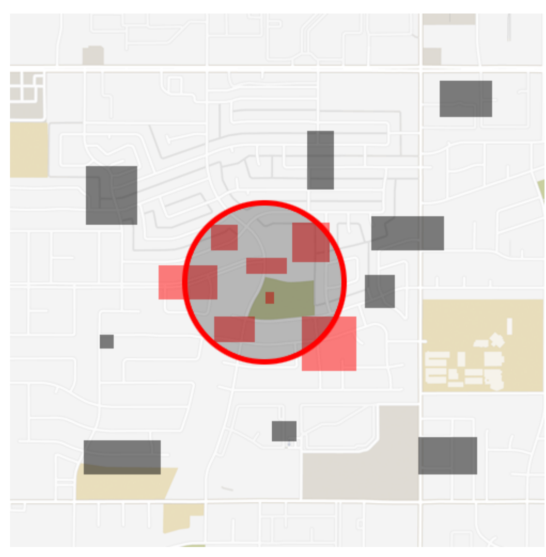
Ukaz za iskanje bližnjih objektov okoli točke (v radiju 100 metrov).

```
NEARBY hoteli POINT 46.048811 14.508545 100
```

Primer odgovora, ko iščemo bližnje objekte:

```
127.0.0.1:9851> NEARBY hoteli POINT 46.056 14.502 100
{"ok":true,"objects":[{"id":"hotel_lev","object":
{"type":"Point","coordinates":[14.502361,46.056206]}}],
"count":1,"cursor":0,"elapsed":"1.0282ms"}
```

V zbirki hoteli smo v radiju točke našli objekt z identifikatorjem (imenom) *hotel_lev*.

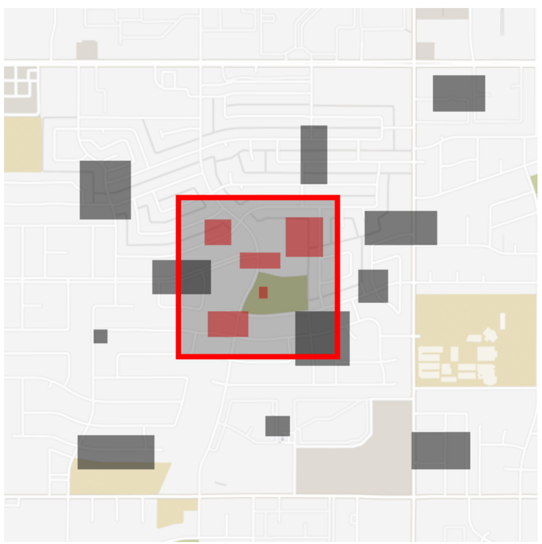


Slika 3.1: Grafični prikaz ukaza NEARBY. (vir: Tile38)

Primer iskanja objektov, ki so popolnoma znotraj nekega objekta. Z ukazom WITHIN in BOUNDS smo v tem primeru poiskali vse objekte, ki so znotraj pravokotnika oziroma mejnega pravokotnika *(angl. bounding box)*.

```
WITHIN hoteli BOUNDS 46.005 14.398 46.091 14.609
```

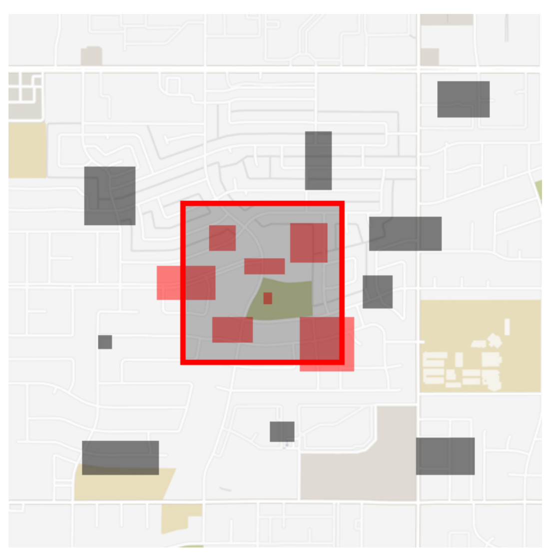
Z ukazom INTERSECTS poiščemo vse objekte, ki so se v našem primeru sekali s pravokotnikom.



Slika 3.2: Grafični prikaz ukaza WITHIN. (vir: Tile38)

```
INTERSECTS hoteli BOUNDS 46.005 14.398 46.091 14.609
```

Ukaza WITHIN in INTERSECTS dopuščata, da namesto ključne besede BOUNDS uporabimo OBJECT, in tako opišemo bolj poljuben objekt v formatu GeoJSON.



Slika 3.3: Grafični prikaz ukaza INTERSECTS. (vir: Tile38)

Poglavje 4

OpenStreetMap

Da smo dobili občutek za našteta orodja in kaj prostorski podatki pravzaprav so, ter kako jih upravljati v različnih podatkovnih bazah, smo si za namene diplome izbrali prostorske podatke, ki jih nudi OpenStreetMap (OSM). To je projekt odprtih geografskih podatkov za zemljevide. Je zelo velika baza podatkov, ki poleg pozicij in oblik objektov vsebuje tudi druge pomembne lastnosti, s katerimi povemo na primer, ali gre za pošto ali trgovino. Podatki OSM so predstavljeni z datoteko XML, ki ima naslednje elemente [18]:

- **<bounds />** element ima atribute, ki opisujejo, kakšno geografsko območje predstavljamo

  ```
  <bounds minlat="54.0889580" minlon="12.2487570"
  maxlat="54.0913900" maxlon="12.2524800"/>
  ```

- **<node />** je najbolj pomemben element, ki predstavlja vozlišča, oziroma točke v prostoru.

  ```
  <node id="261728686" lat="46.048800" lon="14.508695"/>
  ```

  Vozliščem in drugim elementom dodajamo lastnosti z elementom <tag />, ki ima atributa k (ključ) in v (vrednost).

```
<node id="1831881213" lat="46.048800" lon="14.508695">
  <tag k="name" v="Ljubljanski grad"/>
  <tag k="historic" v="castle"/>
 </node>
```

- **<way>... </way>** element ni namenjena samo opisu poti, kot bi to lahko sklepali iz imena, ampak z njim opisujemo tudi poligone, meje, ceste in reke. Temu elementu dodajamo elemente <nd />, ki se sklicujejo na vozlišča, ki smo jih že navedli nekje prej v datoteki.

```
<way id="26659127">
  <nd ref="292403538"/>
  <nd ref="298884289"/>
   ...
   ...
  <nd ref="261728686"/>
  <tag k="name" v="Tržaška cesta"/>
  <tag k="oneway" v="no"/>
</way>
```

- **<relation />** element uporabimo običajno na koncu, ko želimo navesti, v kakšnem razmerju so si neki elementi.

```
<relation id="56688">
  <member type="node" ref="294942404" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  ...
  <member type="node" ref="249673494" role=""/>
  <tag k="name" v="Park Tivoli"/>
  <tag k="leisure" v="park"/>
 </relation>
```

4.1 Uvažanje OSM-ja v podatkovno bazo

Ker je OSM napisan z XML-om je prednost ta, da lahko kdorkoli naredi sintaksni analizator za procesiranje datoteke. Slabost tega pa je, da so ne stisnjene datoteke precej velike. Ne stisnjena datoteka OSM ima pripono .osm, stisjena datoteka pa .bz2 ali .pbf. Datoteke OSM dobimo na spletni strani `http://download.geofabrik.de/` (Dostopano 28.8.2018). Pomen pripon je naslednji:

- **.osm** navadna tekstovna datoteka, ki vsebuje XML,
- **.bz2** s programom bzip2 stisnjena datoteka .osm,
- **.pbf** datoteka v binarnem formatu in jo lahko beremo z določenimi knjižicami, na primer s PyOsmium za Python.

V tabeli 4.1 primerjamo velikosti datotek različnih pripon.

ime datoteke	velikost
slovenia-latest.osm.pbf	203 MB
slovenia-latest.osm.bz2	381MB
slovenia-latest.osm	4.88 GB
spain-latest.osm.pbf	664 MB
spain-latest.osm.bz2	1.1 GB
spain-latest.osm	12.9 GB

Tabela 4.1: Primerjava datotek OSM s priponami .osm, .bz2 in .pbf za Slovenijo in Španijo

4.1.1 PyOsmium

Če ne želimo napisati lastnega sintaksnega analizatorja za procesiranje datotek OSM, lahko uporabimo PyOsmium. PyOsmium je pythonski paket za

procesiranje .osm in .pbf datotek [19]. Z malo kode lahko na primer izpišemo vsa imena lekarn za območje, ki ga preberemo iz datoteke (slika 4.1).

4.1.2 Uvoz v PostgreSQL

Za uvoz v PostgreSQL, smo uporabili orodje osm2pgsql, ki sprejme datoteke .bz2, .osm in .pbf. Podatke neposredno vnese v bazo in ustvari tabele. Preden zaženemo ukaz moramo v našo bazo dodati še razširitev *postgis*. Običajni ukaz za uvoz podatkov je naslednji [20]:

```
osm2pgsql -c -W -U postgres -d slovenia  -S C:\osm2pgsql\default.style
C:\osmdata\slovenia.pbf
```

Pomen stikal je naslednji:

-c izbris obstoječih podatkov iz baze.

-W zahtevamo, da nas program vpraša po geslu za dostop bazo.

-U uporabniško ime za dostop do baze.

-d baza, kjer želimo uvoziti OSM.

-S stilska datoteka, ki definira katere stolpce želimo imeti v tabelah.

Na koncu še navedemo katero datoteko želimo uvoziti v bazo.

Pri tem nam orodje ustvari naslednje tabele:

- planet_osm_line (vsebuje vse objekte, ki so sestavljeni iz črt, na primer reke)
- planet_osm_point (vsebuje geolokacije objektov, na primer lokacije lekarn, vsebino tabele prikažemo na sliki 4.2)
- planet_osm_polygon (vsebuje vse objekte, ki so sestavljeni iz poligonov, na primer stavbe)
- planet_osm_roads (vsebuje podmnožico elementov iz planet_osm_line)

```
import osmium

class StevecLekarn(osmium.SimpleHandler):
    def __init__(self):
        osmium.SimpleHandler.__init__(self)
        self.stevilo_lekarn = 0

    def obdelaj(self, tags):
        if 'amenity' in tags and 'name' in tags:
            if tags['amenity'] == 'pharmacy':
                print(tags['name'])
                self.stevilo_lekarn += 1



    def node(self, n):
        self.obdelaj(n.tags)

    def way(self, w):
        self.obdelaj(w.tags)

    def relation(self, r):
        self.obdelaj(r.tags)

if __name__ == '__main__':
    h = StevecLekarn()
    h.apply_file("slovenia.pbf")
    print("Stevilo lekarn v sloveniji je: %d" % h.stevilo_lekarn)
```

Slika 4.1: Primer kode, ki prešteje in izpiše vsa imena lekarn na območju Slovenije

Atribute, ki imajo te tabele so nekatere izmed lastnosti elementov, ki jih opišemo s `<tag />`. Ker nimajo vsi objekti enakih lastnosti imamo veliko praznih vrednosti. Za večino atributov razberemo njihov pomen iz imena, imamo pa atribut z_order, ki pomeni v kakšnem vrstnem redu naj programi za izrisovanje izrišejo elemente, da se pravilno prekrivajo. Geometrije so shranjene v stolpcu (atributu) z imenom *way*, ki ga ima vsaka tabela. (Slika 4.3).

	osm_id bigint	access text	addr:housename text	addr:housenumber text	addr:interpolation text	admin_level text	aerialway text	aeroway text	amenity text	area text	ba te
1	40498285	[null]	[null]	26d	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
2	88459663	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
3	40456913	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
4	79803512	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
5	40456914	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
6	09848261	[null]	[null]	4/b	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
7	19025992	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
8	59059705	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
9	02961690	[null]	[null]	21	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
10	12843076	[null]	[null]	35	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
11	10551588	[null]	[null]	[null]	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu
12	02203327	[null]	[null]	8	[null]	[null]	[null]	[null]	pharmacy	[null]	[nu

Slika 4.2: Vsebina tabele planet_osm_point. (vir: lastni)

Če dodamo stikalo --slim oziroma -s nam orodje ustvari še te tri dodatne tabele:

- **planet_osm_nodes** (vsebuje vsa vozlišča, vsebino tabele prikažemo na sliki 4.4)
- **planet_osm_ways** (vsebuje vse objekte, ki so sestavljeni iz vozlišč)
- **planet_osm_rels** (vsebuje podatke o relacijah med objekti)

Iz teh treh tabel so izpeljane tabele na sliki 4.1. Te tri tabele predstavljajo podatke OSM v surovem formatu in ne vsebujejo stolpca s podatkovnim tipom *geometry*. Imajo pa zato stolpec s podatkovnim tipom *hstore* v katerega lahko shranjujemo vrednosti po ključ-vrednost principu. Zato se vse lastnosti, ki se pojavijo pri nekem objektu nahajajo v tej podatkovni strukturi. Tako se lahko izognemo večinoma praznim stolpcem.

	name text	way geometry
1	[null]	0101000020110F000014AE4761CE1537418FC2F59836C1...
2	Lekarna Simonov zaliv	0101000020110F0000A4703D4A842F374185EB5128B9C...
3	Obalne lekarne	0101000020110F00009A9999193D3437413D0AD7F35BC...
4	Lekarna Olmo	0101000020110F0000713D0AD7D35137415C8FC2D57A...
5	Tosama	0101000020110F000085EB5138345137411F85EB5119C...
6	[null]	0101000020110F00007B14AE87E1543741333333E3CAC...
7	Lekarna / Farmacia	0101000020110F000085EB5138205437413333333F3B2C...
8	Lekarna Ankaran	0101000020110F00009A999999EA55374152B81EE54AC...
9	[null]	0101000020110F000052B81E4583AF37416666661684CF...
10	Goriska lekarna Dobrovo	0101000020110F00007B14AEC742FA364114AE4711840...

Slika 4.3: V vsaki tabeli imamo stolpec *way*, kjer so geometrije zapisane z šestnajstiškim nizom v formatu WKB. (vir: lastni)

Ukaz, ki smo ga uporabili:

```
osm2pgsql -c --slim -W -U postgres -d slovenia C:\osmdata\slovenia.pbf
-S C:\osm2pgsql\default.style
```

	id bigint	lat integer	lon integer	tags text[]
1	56064345	578948839	161805284	{name,"Lekarna Moste",amenity,pharmacy,ope...
2	72009871	578783367	161265932	{name,"Lekarna Vič",amenity,pharmacy,operat...
3	88459663	570609581	152070110	{name,"Obalne lekarne Koper",amenity,pharm...
4	04183105	579229436	161527520	{name,"Kamilica zeliščna prodajalna",source,su...
5	20398990	585334292	174488901	{name,"Lekarna Rače",amenity,pharmacy}
6	22975005	585568094	175488098	{amenity,pharmacy}
7	47525395	586721874	174111224	{addr:housenumber,1,addr:street,"Ulica Eve Lo...

Slika 4.4: Vsebina tabele planet_osm_nodes. (vir: lastni)

4.1.3 Uvoz v MySQL

Za MySQL ne obstaja veliko skript, da bi uvozili OSM in tudi tiste, ki obstajajo, jih brez večjega napora ne bi mogli uporabljati, ker so že zastarele. Zato

smo naredili pythonsko skripto in OSM prenesli iz baze, ki smo jo ustvarili v PostgreSQL s pomočjo orodja osm2pgsql. Tako smo preizkusili, če lahko dosežemo enako shemo podatkov kot v PostgreSQL bazi. Izkazalo se je, da lahko, razen dodatnih tabel (tistih, ki jih dobimo, če vključimo stikalo -s), ker MySQL nima podatkovnih tipov, kot je *hstore* oziroma polj *(angl. array)*. Ampak teh dodatnih tabel tako nismo potrebovali za namene diplome.

Če bi želeli neposredno uvažati OSM, bi to naredili s prej omenjeno pytonsko knjižnico *pyosmium*, kjer bi neposredno brali datoteko OSM, vendar bi potrebovali kar nekaj časa preden bi iz elementov *node* in *way* sestavili dejanske geometrije.

4.1.4 Uvoz v Tile38

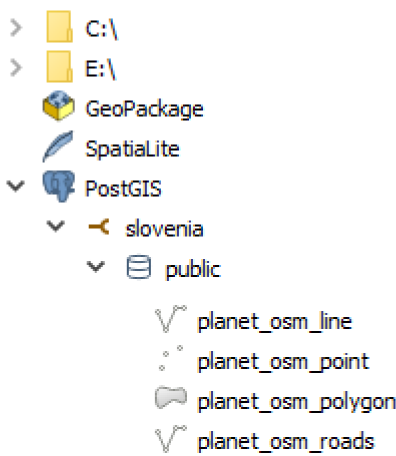
Tudi uvoz v Tile38 smo naredili s pythonsko skripto in podatke prenesli iz PostgreSQL baze. Drugače bi večino časa porabili za razčlenjevanje *(angl. parsing)* datoteke OSM. Ker v Tile38 ne moremo na enostaven način dodajati atributov objektom, smo vse atribute oziroma lastnosti dali v polje *properties*, ki ga podpira format GeoJSON. Prenesli smo tabele planet_osm_point, planet_osm_line, planet_osm_roads in planet_osm_polygon. Na enak način smo poimenovali zbirke v bazi Tile38. Do strežnika Tile38 smo dostopali s knjižnico *redis-py*.

4.2 QGIS

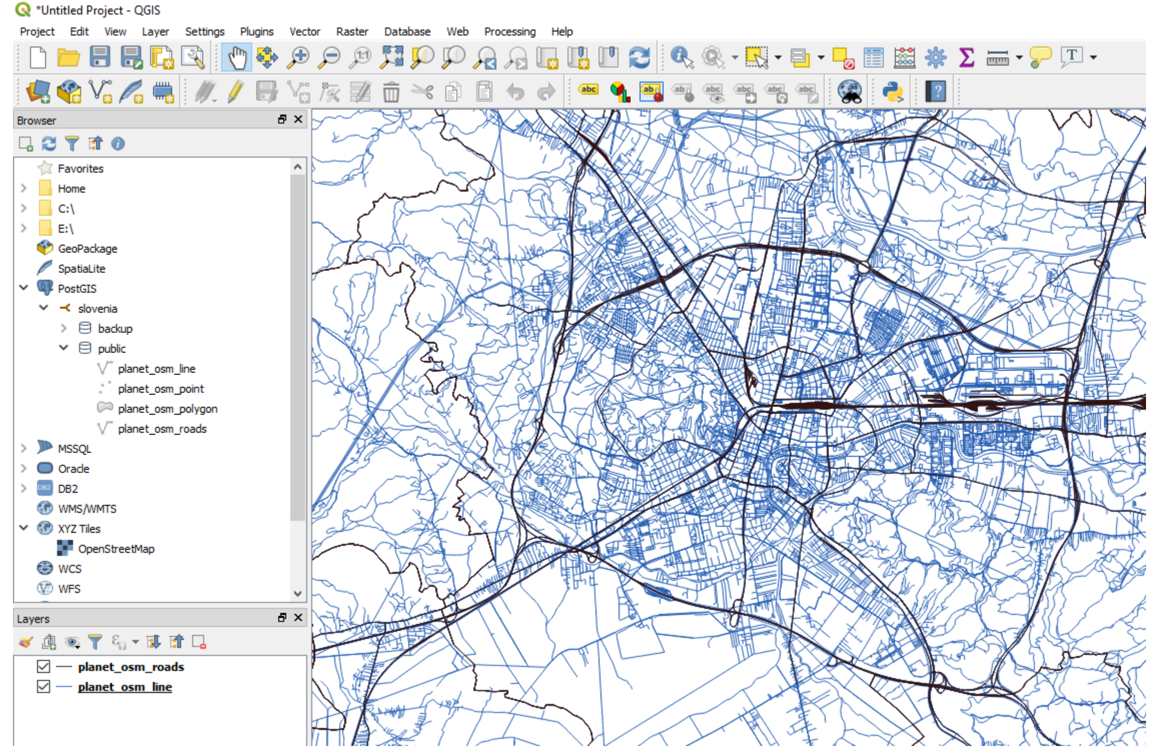
QGIS je odprtokodno orodje za prikazovanje, urejanje in ustvarjanje prostorskih podatkov [21]. Obstajajo podobna orodja, na primer ArcGIS, vendar je plačljivo [22]. QGIS ima vgrajeno možnost, da se povežemo na PostgreSQL ali MySQL bazo. Ne podpira pa povezave s Tile38, zato bi morali prenesti podatke v kakšno bazo, ki jo QGIS podpira in z njo prikazati podatke. Orodje zna prikazovati podatke tipa *geometry*. Zato samodejno zazna tabele, kjer so stolpci tipa *geometry* in jih prikaže v raziskovalcu projekta *(Slika 4.6)*.

```
import redis

client = redis.Redis(host='127.0.0.1', port=9851)

geojson_object = '{"type": "Feature", ' \
                 '"geometry": {"type": "Point", ' \
                 '"coordinates": [14.4971719, 46.0604639]}, ' \
                 '"properties": {"osm_id": 1640764737, ' \
                           '"highway": "bus_stop", ' \
            '              "name": "801011 ~ Tivoli / Ljubljana Tivoli"}}'

result = client.execute_command('SET', 'planet', 'point303',
                       'OBJECT', geojson_object)
print(result)
```

Slika 4.5: Primer kako se povežemo s knjižnico *redis-py* na Tile38



Slika 4.6: Raziskovalec v QGIS. (vir: lastni)



Slika 4.7: Prikaz Ljubljane s podatki OSM v QGIS-u. (vir: lastni)

Poglavje 5

Prikaz dela s prostorskimi podatki

Ker je bil uvoz OSM-ja v PostgreSQL, MySQL in Tile38 dokaj preprost, smo si zamislili aplikacijo, ki nam pokaže oziroma izpiše lokacijo postaje, na katero bo avtobus v kratkem prišel. Tako smo dobili boljši občutek za uporabo orodij.

5.1 API Trola.si

API Trola.si nam pove, čez koliko minut bo avtobus prišel na postajo. Iz časov prihodov avtobusov smo določili na katero postajo bo avtobus v kratkem prispel oziroma na kateri stoji. Lahko bi tudi rekli, da smo določevali med katerima postajama se avtobus nahaja.

API-ju pošiljamo zahtevke HTTP GET. Odgovor na zahtevek dobimo v formatu JSON ali HTML. Zahtevek lahko API-ju podamo na naslednja URL-ja:

- https://www.trola.si/ime_postaje/ (zahtevek z imenom postaje).
- https://www.trola.si/id_postaje/ (zahtevek z identifikacijsko številko postaje).

```
from requests import get


ime_postaje = "brnčičeva"
request = get(f"https://www.trola.si/{ime_postaje}/",
    headers={'Accept': 'application/json'}).json()
print(request)
```

Slika 5.1: Prikaz zahtevka HTTP v Pythonu.

Privzeto se nam prikaže spletna stran oziroma dobimo odgovor v HTML-ju, če pa želimo dobiti odgovor v formatu JSON, moramo v glavo zahtevka HTTP dodati polje 'Accept' : 'application/json' [23]. Primer zahtevka HTTP v Pythonu prikažemo na sliki ??.

Odgorovor, ki ga dobimo:

```
{'stations': [{'number': '204121', 'name': 'BRNČIČEVA',
'buses': [{'direction': 'Brnčičeva', 'number': '8',
 'arrivals': [0, 19, 31]},{'direction': 'Gameljne',
'number': '8', 'arrivals': [1, 15, 30]}]}]}
```

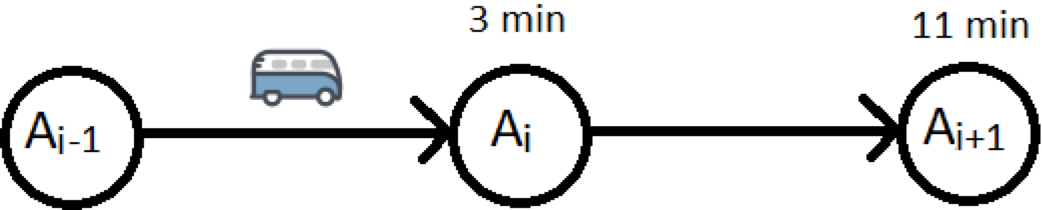
Struktura odgovora je naslednja: API nam vrne odgovor v formatu JSON, ki ima tabelo z opisi postaj pod ključem 'stations'. Opis postaje je sestavljen iz številke, imena in tabele avtobusov, ki vozijo čez to postajo. Avtobus je opisan s številko linije, v katero smer potuje in njegovimi prihodi na postajo. V primeru imamo tri prihode, ki pomeni, da od trenutka, ko smo dobili odgovor, prvi avtobus pride čez minuto, drugi čez 15 minut in tretji čez 30 minut.

5.2 Na kateri postaji bo avtobus v kratkem

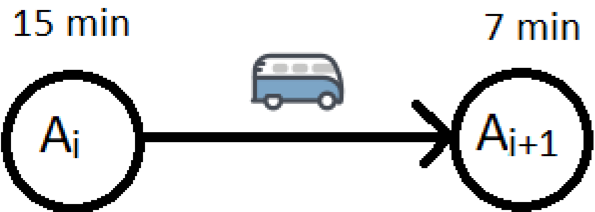
Da smo določili med katerima postajama se avtobus nahaja oziroma, na katero postajo se približuje smo nad potjo avtobusa naredili zahtevke po vrsti

od začetne postaje A_0 do končne postaje A_k. Vrstni red in identifikacijske številke postaj smo pridobili s pomočjo iskalnika avtobusnih linij na strani LPP [24].

Torej, če je čas prihod na postajo A_i manjši od A_{i+1} pomeni, da se avtobus premika proti postaji A_i, to ilustriramo s sliko 5.2, in če je čas prihoda na A_i večji od A_{i+1} pomeni, da je avtobus nekje med postajo A_i in A_{i+1}, kot kaže slika 5.3.



Slika 5.2: Ko je čas prihoda na postajo A_i manjši od A_{i+1}, se avtobus nahaja nekje pred postajo A_i. (vir ikone: Bus Icon)



Slika 5.3: Ko je čas prihoda na postajo A_i večji od A_{i+1}, se avtobus nahaja nekje med postajo postajo A_i in A_{i+1}. (vir ikone: Bus Icon)

V Pythonski kodi (slika 5.4) še ponazorimo prej povedano. V spremenljivki postaje so, identifikacijske številke postaj v vrstnem redu od začetne do končne postaje. Na koncu je izpis lokacije postaje na kateri bo avtobus v kratkem in številke avtobusa.

S prej navedeno kodo pridobimo podatke na katere lokacije avtobus prihaja. Kodo smo nato prilagodili, da je iterirala po izbranih avtobusih in

```
postaje = [104221, ..., 204121]
st_avtobusa = 8

for i in range(0, len(postaje) - 1):
    a = requests.get("https://www.trola.si/" + postaje[i] + "/",
                     headers={'Accept': 'application/json'}).json()
    a_naslednja = requests.get("https://www.trola.si/" + postaje[i+1]
                    + "/", headers={'Accept': 'application/json'}).json()

    bus1_arrival = None
    bus2_arrival = None
    lokacija_postaje = getLocation(postaje[i])

for busi in a['stations'][0]['buses']:
    if busi['number'] == st_avtobusa:
        bus1_arrival = busi['arrivals']
for busi in a_naslednja['stations'][0]['buses']:
    if busi['number'] == st_avtobusa:
        bus2_arrival = busi['arrivals']
if bus1_arrival > bus2_arrival:
    print(lokacija_postaje + " " + str(st_avtobusa))
```

Slika 5.4: Koda za pridobitev pozicij avtobusa

vnašala podatke o pozicijah avtobusov v tabelo z imenom *avtobusi*. Lokacije postaj smo pridobili z poizvedbo nad tabelo planet_osm_point v bazi postgresql-slovenia:

```
SELECT * FROM
(SELECT
ST_X(ST_Transform(ST_GeomFromWKB(ST_AsBinary(way), 3857),4326)) as lon,
ST_Y(ST_Transform(ST_GeomFromWKB(ST_AsBinary(way), 3857),4326)) as lat,
name, ST_asText(way) FROM planet_osm_point WHERE highway = 'bus_stop'
AND name LIKE any(array['%104221%', '%104211%', ... , '%204112%'
, '%204121%'])) as coordinates WHERE coordinates.lat > 45.947163
AND coordinates.lat < 46.155775 AND coordinates.lon > 14.321393
AND coordinates.lon < 14.719027
```

Naredili smo ugnezdeno poizvedbo, kjer smo najprej izluščili zemljepisno širino in dolžino s metodama ST_X in ST_Y. Bazo smo imeli v koordinatnem sistemu EPSG:3857, ki ga spletni zemljevidi uporabljajo za risanje geometrij na 2D površino, zato smo z metodo ST_Transform pretvorili geometrije v sistem EPSG:4326, kjer imamo koordinate zapisane z zemljepisno širino in dolžino. V stavku WHERE pri ugnezdeni poizvedbi smo določili, da iščemo avtobusne postaje *(angl. bus station)*, in da iščemo postaje s točno določenimi identifikacijskimi številkami v njihovem imenu. Zunanji SELECT stavek smo s stavkom WHERE omejili iskanje na območje Ljubljane in tako pridobili pozicije postaj.

V Tile38 bi tako poizvedbo precej težko naredili, ker podpira samo iskanje po številskih atributih in ne podpira iskanja po lastnostih, ki jih definiramo s formatom GeoJSON. Kar lahko ponovimo od poizvedbe je iskanje na območju Ljubljane. Primer bi bil takšen:

```
WHITHIN planet_osm_point BOUNDS 45.947163 14.321393 46.155775 14.719027
```

Z ukazoma WITHIN in BOUNDS smo pridobili vse prostorske podatke na območju Ljubljane, sedaj bi morali rezultate poizvedbe prebrati z drugim programom in sami filtrirati rezultate, glede na atribute, ki smo jih vnesli v polje *properties*.

5.3 Prikaz s QGIS

Baza OSM nima samo podatkov, da narišemo zemljevid z njo, ampak še druge podatke na primer podatki o poti, ki jo opravlja določen avtobus. Da je bolj pregledno, kje se nahajajo naši avtobusi sem podatke vnesel v drugo tabelo z imenom linije. Poizvedbo, ki sem jo naredil nad tabelo planet_osm_line v bazi postgresql-slovenia je bila:

```
SELECT name, operator, ref, way FROM planet_osm_line
WHERE operator = 'Ljubljanski Potniški Promet'
AND ref in('1', '3', '6', '8', '11')
```

Tabela planet_osm_line ima stolpec operator, ki predstavlja prevoznika in stolpec ref, ki je v našem primeru predstavljal številko linije.

Tudi te poizvedbe v Tile38 ne bi mogli narediti, ker smo precej omejeni z ukazom WHERE, ki išče po številskih atributih. Tukaj vidimo, da bi bilo potrebno Tile38 kombinirati še z dodatno aplikacijo oziroma bazo.

Na sliki 5.5 prikažemo linije in trenutno stanje avtobusov.

5.4 Geografska ograja s Tile38

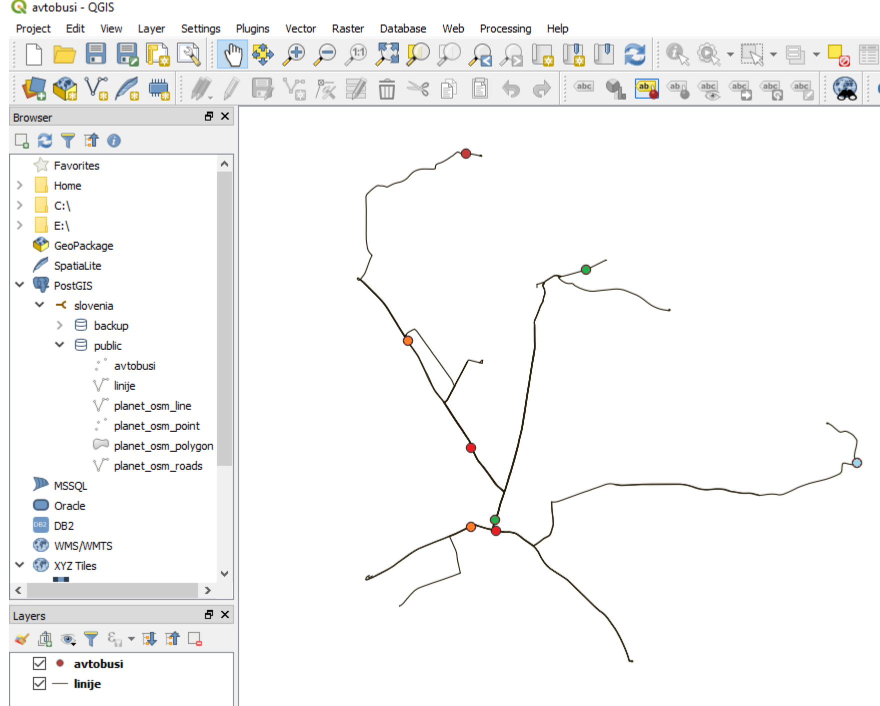
V tem razdelku predstavimo posebno funkcionalnost prostorske baze Tile38, s katero postavimo geografsko ograjo. S to funkcionalnostjo zaznavamo premike objektov na območju, ki ga označimo s geografsko ograjo.

Primer kako naredimo geografsko ograjo v obliki kroga z radijem 100 metrov.

```
NEARBY planet_osm_point FENCE POINT 46.05180 14.50331 100
```

S ključno besedo FENCE smo določili, da gre za geografsko ograjo. Odgovori, ki jih lahko dobimo, če se dogodek zgodi na geografski ograji:

- objekt X je v območju,
- objekt X ni v območju,



Slika 5.5: Prikaz avtobusnih linij in pozicij avtobusov. (vir: lastni)

- objekt X je odšel iz območja,
- objekt X je prišel v območje,
- objekt X je prečkal območje.

Ukaza WITHIN in INTERSECTS dopuščata, da lahko poleg ključnih besed POINT in BOUNDS uporabimo tudi OBJECT, s katerim specificiramo bolj poljuben objekt v formatu GeoJSON.

```
WITHIN avtobusi FENCE OBJECT {"type":"Polygon","coordinates":
[[[14.4985642798, 46.06259242086], [14.49320488124, 46.06049636553],
[14.49642052038, 46.05765642170], [14.50207224979, 46.05927926463],
[14.4985642798, 46.06259242086]]]}
```

Pri zgrornjem ukazu smo s poligonom opisali območje Tivolija. Če bi imeli kakšen tak objekt že v neki zbirki, bi ga lahko neposredno naslovili z ukazom GET.

```
WITHIN avtobusi FENCE GET ograje obmocje_tivoli
```

Ko aktiviramo geografsko ograjo s FENCE, se strežnik spremeni v način, ki ne sprejema več ukazov ampak lahko poslušamo samo odgovore, ko so na primer, kateri objekti so prišli in odšli iz območja. Ker paket *redis-py* ni imel funkcionalnosti, kjer samo posluša, smo tokrat uporabili omrežni protokol Telnet, da smo se povezali na strežnik, zagnali ukaz in poslušali odgovore. Koda s katero poslušamo dogodke na geografski ograji je na sliki 5.6.

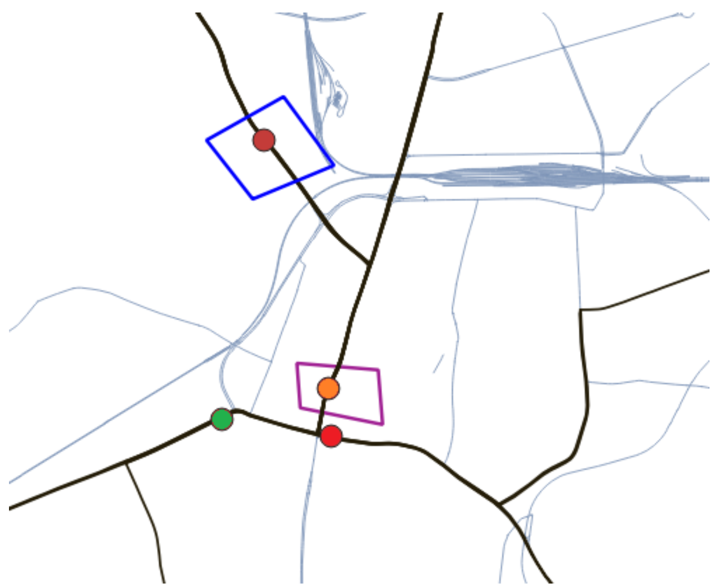
Ko strežnik zazna dogodek na geografski ograji nam pošlje odgovor v formatu JSON, v katerem so navedeni premiki nekega objekta. Opazujemo ali je objekt vstopil ali izstopil iz območja.

Če želimo poslušati več območij naenkrat, moramo uporabiti spletni kavelj (angl. Webhook). To je funkcionalnost, ki omogoča samodejno pošiljanje sporočil drugim aplikacijam, z njo lahko nastavimo, da poslušamo več območij hkrati. Ko se bo dogodek na ograji zgodil, bo podatke poslal drugi aplikaciji, v našem primeru je bil to strežnik napisan v Pythonu (slika 5.9) na

```
import telnetlib
import json

command = "WITHIN avtobusi FENCE GET ograje obmocje_tivoli\n"

tn = telnetlib.Telnet(host="127.0.0.1", port=9851)
tn.write(command.encode("ascii"))
res = tn.read_until("\n".encode("ascii"))

while True:
    expected_bytes = tn.read_until("\n".encode("ascii"))
    json_string = tn.read_until("\n".encode("ascii")).decode('ascii')
                                                           .strip()
    json_object = json.loads(json_string)
    if json_object["detect"] == "enter":
        print("avtobus je vstopil v območje")
    elif json_object["detect"] == "exit":
        print("avtobus je iztopil iz območje")
```

Slika 5.6: Koda za poslušanje dogodkov na geografski ograji.

Slika 5.7: Z modro označeno območje, ki ga opazujemo. (vir: lastni)



Slika 5.8: Opazujemo več območij hkrati s pomočjo webhookov. (vir: lastni)

naslovu `http://127.0.0.1:8888/obvestilo` [25]. Primer izpisov strežnika je na sliki 5.10

Tako naredimo spletna kavlja za območje Tivolija in Drame:

```
SETHOOK tivoli http://127.0.0.1:8888/obvestilo WITHIN avtobusi
FENCE GET ograje obmocje_tivoli
SETHOOK drama http://127.0.0.1:8888/obvestilo WITHIN avtobusi
FENCE GET ograje obmocje_drama
```

5.4.1 Potujoča geografska ograja

Funkcionalnost, ki je nismo uporabljali, pa je vseeno zanimiva je potujoča geografska ograja. Ograja potuje skupaj s premiki objekta. Funkcionalnost je primerna za aplikacijo, ko se dva uporabnika želita srečati, tako jima strežnik pove, da sta si v bližini, in ni potrebno periodično poizvedovati njuni poziciji v bazi. Tudi, ko nove objekte vnesemo v zbirko dobijo ograjo.

V zbirko uporabniki z naslednjim ukazom dodamo vsem objektom potujočo geografsko ograjo, ki išče objekte v radiju 100 metrov:

```
NEARBY uporabniki FENCE ROAM uporabniki * 100
```

Z zvezdica je v tem primeru regularni izraz s katerim povemo, da želimo nastaviti potujočo geografsko ograjo vseem objektom v zbirki uporabniki.

Na sliki 5.11, vidimo potujočo geografsko ograjo na primeru avtobusov.

```
import socket
import json


listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
listen_socket.bind(('127.0.0.1', 8888))
listen_socket.listen(1)
print('Webhook listener on port %s ...' % 8888)
while True:
    client_connection, client_address = listen_socket.accept()
    request = client_connection.recv(1024)
    request_list = request.decode('utf-8').split("\n")
    json_object = json.loads(request_list[len(request_list)-1])
    if json_object['detect'] == "exit":
        print("Objekt {} je odšel iz območja {}".format(json_object['id'],
                          json_object['hook']))
    elif json_object['detect'] == "enter":
        print("Objekt {} je vstopil v območje {}".format(json_object['id'],
                         json_object['hook']))
    http_response = """HTTP/1.1 200 OK\n\n"""
    client_connection.sendall(http_response.encode())
    client_connection.close()
```

Slika 5.9: Koda za strežnik, ki sprejema sporočila spletnih kavljev.

```
>python webhook_listener.py
Webhook listener on port 8888 ...
Objekt bus_1_vizmarje je vstopil v območje tivoli
Objekt bus_1_vizmarje je odšel iz območja tivoli
```

Slika 5.10: Primer izpisov strežnika, ki je poslušal sporočila spletnih kavljev.

Slika 5.11: Potujoča geografska ograja na primeru avtobusov. (vir: lastni)

Poglavje 6

Primerjava orodij

Na primeru aplikacije s prostorskimi podatki OSM in API Trola.si smo ugotovili, da se s Tile38 težko išče objekte po njihovih lastnostih. Pri Tile38 lahko objekte iščemo večinoma le s prostorskimi metodami, kot je na primer, iskanje objektov v bližini nekega objekta. V splošnem bi to težavo lahko rešili tako, da bi Tile38 kombinirali, še s kakšno drugo podatkovno bazo.

6.1 Splošne ugotovitve

6.1.1 PostgreSQL/PostGIS

Ugotovitve:

- PostGIS ima zelo veliko metod in dokumentacije za upravljanje s prostorskimi podatki.

- Skupnost *(angl. community)* za PostGIS je precej velike, ko se nam kaj zatakne, običajno najdemo odgovore na internetu.

- PostgreSQL in PostGIS sta odprtokodna projekta, kar pomeni večjo vpletenost uporabnikov in razvijalcev. Ker je PostGIS dodatek za PostgreSQL, je pri tem možnost neodvisnega izdajanja različic (verzij).

6.1.2 MySQL

Ugotovitve:

- Manj metod in funkcionalnosti, vendar še vedno primerljivo s Tile38.
- Razširitev za upravljanje prostorskih podatkov je del MySQL, zato moramo počakati do naslednje verzije MySQL-a za nove funkcionalnosti.
- Funcionalnost za transformiranje podatkov v druge prostorske referenčne sisteme ne deluje, četudi je bila opisana v dokumentaciji.

6.1.3 Tile38

Ugotovitve:

- Enostavna namestitev in uporaba
- Primerno orodje, če se ukvarjamo s podatki GPS.
- Filtriranje po atributih, ki smo jih dodali v ključ *properties* pri GeoJSON, ni možno, ampak samo po številskih atributih, ki jih dodamo s FIELD/FSET.
- Ni možnosti spremeniti oziroma določiti kateri prostorski referenčni sistem bomo uporabljali. Tile38 privzeto uporablja sistema EPSG:4326.
- Ker so vsi podatki v delovnem pomnilniku se morajo podatki po zagonu strežnika naložiti iz diska preden je strežnik dostopen za odjemalce. Za podatke Slovenije je strežnik potreboval 80 sekund (processor AMD FX-8320E in SSD SAMSUNG 860 EVO), ter porabljal okoli 4GB delovnega pomnilnika.

6.1.4 Primerjava podprtih funkcionalnosti

Naredili smo tabelo podprtih funkcionalnosti, ki smo jih zasledili pri uporabi orodij. Pri tem ugotovimo, da ima PostgreSQL/PostGIS največji nabor prostorskih funkcionalnosti.

	Tile38	PostgreSQL	MySQL
Iskanje v bližini	✓	✓	✓
Iskanje znotraj	✓	✓	✓
Iskanje objektov, ki se sekajo	✓	✓	✓
določitev koordinatnega sistema		✓	✓
transformiranje podatkov (projekcija)		✓	
rotiranje geometrij		✓	
skaliranje geometrij		✓	
WKT		✓	✓
WKB		✓	✓
GeoJSON	✓	✓	✓

Tabela 6.1: Primerjava podprtih funkcionalnosti

6.2 Kreiranje podatkovnih struktur in vnos podatkov

Kreiranje tabel in vnos podatkov pri MySQL in PostgreSQL je enako, saj oba podpirata prostorskih tip geometrija, ter enake formate (WKT, WKB, GeoJSON) in metode (ST_GeomFromText) za vnos geometrij. Tile38 je zasnovan tako, da se zbirka ustvari, ko vnesemo prvi podatek vanjo. Ima nekaj definiranih prostorskih objektov kot sta točka in pravokotnik, ostale geometrije pa vnesemo v formatu GeoJSON.

6.3 Primerjava funkcionalnosti na primerih poizvedb

Primerjamo kako bi enako funkcionalnost Tile38 izvedli v relacijskih bazah.

6.3.1 Iskanje znotraj objekta

Eksperiment: Izpisati vse objekte, ki so v celoti znotraj mejnega pravokotnika.

Ukaz v **Tile38**:

```
WITHIN planet_osm_point BOUNDS 45.947163 14.321393 46.155775 14.719027
```

Z ukazom WITHIN smo določili, da iščemo znotraj objekta BOUNDS, ki predstavlja pravokotnik.

Poizvedba v **PostgreSQL**:

```
SELECT name, way FROM planet_osm_point
WHERE ST_Within(ST_Transform(way, 4326), ST_GeometryFromText(ST_AsText(
ST_Envelope(ST_GeometryFromText('LINESTRING(14.321393 45.947163,
 14.719027 46.155775)'))), 4326))
```

Pri poizvedbi smo uporabili metodo ST_Envelope, ki nam iz *LineString* naredi mejni pravokotnik, enak kot pri Tile38 z ukazom BOUNDS. Potrebno je bilo tudi transformirati geometrije v sistem EPSG:4326 s metodo ST_Transform, ker smo jih imeli bazo v sistemu EPSG:3857. Nato smo z metodo ST_Within poiskali objekte znotraj pravokotnika.

Poizvedba v **MySQL**:

```
SELECT name, way FROM planet_osm_point WHERE
ST_Within(way, ST_GeomFromText(ST_AsText(ST_Envelope(ST_GeomFromText(
'LINESTRING(14.321393 45.947163, 14.719027 46.155775)'))),4326));
```

Poizvedba v MySQL je praktično identična prejšnji, razlika je, da tukaj nismo uporabili metode ST_Transform, ker je MySQL (še) nima oziroma so stolpci z geometrijami že uporabljali sistem EPSG:4326.

6.3.2 Sekanje z objektom

Eksperiment: Kako bi izpisali vse objekte, ki se sekajo z določenim objektom.

Ukaz v **Tile38**:

```
INTERSECTS planet_osm_point BOUNDS 45.947163 14.321393 46.155775 14.719027
```

Z INTERSECTS smo določili, da Tile38 poišče vse objekte, ki se sekajo z mejnim pravokotnikom.

Poizvedba v **PostgreSQL**:

```
SELECT name, way FROM planet_osm_point
WHERE ST_Intersects(ST_Transform(way, 4326),
ST_GeometryFromText(ST_AsText(ST_Envelope(ST_GeometryFromText(
'LINESTRING(14.321393 45.947163, 14.719027 46.155775)'))), 4326));
```

Iskanje objektov, ki se sekajo z drugim objektom, je enako kot iskanje znotraj objekta. Spremenili smo to, da namesto metode ST_Within uporabimo metodo ST_Intersects.

Poizvedba v **MySQL**:

```
SELECT name, way FROM planet_osm_point WHERE
ST_Intersects(way, ST_GeomFromText(ST_AsText(ST_Envelope(
ST_GeomFromText('LINESTRING(14.321393 45.947163, 14.719027 46.155775)'))),
4326));
```

MySQL poizvedba za sekanje z objektom je enaka kot pri PostgreSQL.

6.3.3 Objekti v bližini

Eksperiment: Poiskati vse objekte v razdalji od točke, ki je manjša ali enaka 100 metrov.

Ukaz v **Tile38**:

```
NEARBY planet_osm_point POINT 46.048811 14.508545 100
```

Z NEARBY smo poiskali objekte, ki so oddaljeni manj ali enako 100 metrov od točke.

Poizvedba v **PostgreSQL**:

```
SELECT name, way FROM planet_osm_point WHERE
ST_Distance(ST_GeometryFromText('POINT(14.508545 46.048811)', 4326),
                      ST_Transform(way, 4326), true) <= 100
```

Za izračun razdalje med objektoma uporabimo metodo ST_Distance, ki ima pri PostGIS še dodaten (boolean) parameter, s katerim povemo, da naj izvaja izračune na sferoidu in ne na sferi (krogli). Brez uporabe parametra za sferoid bodo izračuni hitrejši, vendar pri tem izgubimo del natančnosti izračunov.

Poizvedba v **MySQL**:

```
SELECT name, way FROM planet_osm_point WHERE
ST_Distance(ST_GeometryFromText('POINT(14.508545 46.048811)',
 4326), way) <= 100
```

Poizvedba je enaka kot pri PostgreSQL. ST_Distance nima parametra za sferoid, ker ga že sama upošteva in izvaja izračune na sferoidu.

6.3.4 Primerjava časov izvajanja

V tabelo 6.2 smo vnesli povprečne vrednosti časov izvajanja. Pri prostorskih operacijah je bil Tile38 najhitrejši. Pozna se, da so vsi podatki, ki jih vnesemo v Tile38, v delovnem pomnilniku. Najbolj impresiven rezultat je dosegel pri iskanju objektov v bližini. Medtem, ko je imel MySQL največ težav pri tem, saj je porabil skoraj 6 sekund.

	Tile38	PostgreSQL	MySQL
Iskanje znotraj objekta	60 ms	254 ms	2.3 s
Sekanje z objektom	61 ms	238 ms	2.4 s
Objekti v bližini	1 ms	652 ms	5.6 s

Tabela 6.2: Povprečne vrednosti časov izvajanja

Poglavje 7

Sklepne ugotovitve

Tile38 v trenutni fazi razvoja ni najbolj primeren, da bi ga uporabili pri izdelavi geografskega informacijskega sistema. Na primeru dela z OSM smo videli, da ima Tile38 dosti težav z iskanjem prostorskih podatkov po njihovih atributih. Vendar pa je primeren, ko upravljamo s prostorskimi podatki (na primer GPS) in ima nekaj ključnih ukazov za prostorsko iskanje objektov. Posebna lastnost prostorske baze Tile38 je, da strežnik sam obvešča o dogodkih, ki se zgodijo na geografskih ograjah, tako ni potrebno periodično poizvedovati o položajih objektov. Orodje je še v razvoju, šele v zadnjem letu se je začel resno razvijati, in pričakujemo, da bo v prihodnosti postal precej zmogljiv sistem. Ena od planiranih funkcionalnosti je sprejemanje geometrij v formatu WKT.

Če želimo napredno upravljanje s prostorskimi podatki, je po mojem mnenju najboljše z razširitvijo PostGIS, ki se razvija neodvisno od PostgreSQL-a. Ima največ metod in dokumentacije od primerjanih orodij. Zato ga umestimo za najzmogljiveše orodje za upravljanje s prostorskimi podatki.

MySQL ima manj metod za upravljanje s prostorskimi podatki kot PostGIS, ki ima metode še za skaliranje in rotiranje objektov. Prenosljivost SQL poizvedb iz MySQL v PostGIS je v večini primerov enostavna, ker PostGIS podpira celoten standard OpenGIS, ki mu sledi MySQL. Obratna prenosljivost iz PostGIS na MySQL je slabša, saj MySQL nima več kot samo

standardnih funkcij za upravljanje s prostorskimi podatki.

Ko smo že pri prenosljivosti SQL poizvedb med sistemi, omenimo še relacijsko podatkovno bazo MariaDB, ki so jo ustvarili nekateri razvijalci MySQL, v novejših verzijah podpira tudi nabor funkcionalnosti za prostorske podatke. Prenosljivost SQL poizvedb med MySQL in MariaDB, bi pa povsem šla, saj oba podpirata enak nabor prostorskih metod, kar je razvidno iz dokumentacije MariaDB, kjer primerjajo prostorske zmožnosti MySQL in MariaDB [26]. Upravljanje prostorskih podatkov v MySQL bi umestili med PostGIS in Tile38.

V kakšnih primerih bi se odločili za določeno orodje? Torej, če imamo bolj enostavno aplikacijo, ki uporablja na primer podatke GPS, potem bi se odločili za Tile38, če je potrebno bolj napredno iskanje prostorskih podatkov, tudi po njihovih lastnostih (metapodatkih), bi se odločili za MySQL, če pa bi želeli imeti polno kontrolo nad prostorskimi podatki z naprednimi metodami (rotiranje in skaliranje), pa bi se odločili za PostGIS/PostgreSQL.

Literatura

[1] "Spatial data." Dosegljivo: https://searchsqlserver.techtarget.com/definition/spatial-data. [Dostopano 13. 6. 2018].

[2] A. Marquez, *PostGIS Essentials*. Community experience distilled, Packt Publishing, 2015.

[3] "PostgreSQL 10.5 Documentation." Dosegljivo: https://www.postgresql.org/files/documentation/pdf/10/postgresql-10-A4.pdf. [Dostopano 13. 6. 2018].

[4] "OpenGIS standard — Simple Feature Access — OGC." Dosegljivo: http://www.opengeospatial.org/standards/sfs. [Dostopano 29. 8. 2018].

[5] "PostGIS 2.4.6dev Manual. Chapter 4. Using PostGIS: Data Management and Queries." Dosegljivo: https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html. [Dostopano 29. 8. 2018].

[6] "PostGIS 2.4.6dev Manual. Chapter 14. PostGIS Special Functions Index." Dosegljivo: https://postgis.net/docs/manual-2.4/PostGIS_Special_Functions_Index.html#PostGIS_TypeFunctionMatrix. [Dostopano 29. 8. 2018].

[7] "Well Known Text Module." Dosegljivo: https://msdn.microsoft.com/en-us/library/mt712880.aspx. [Dostopano 29. 8. 2018].

[8] "Well-known Binary (WKB) Format." Dosegljivo: http://ftp.nchu.edu.tw/MySQL/doc/refman/5.4/en/gis-wkb-format.html. [Dostopano 29. 8. 2018].

[9] "PostgreSQL + Python — Psycopg." Dosegljivo: http://initd.org/psycopg/. [Dostopano 29. 8. 2018].

[10] "MySQL 8.0 Reference Manual :: 12.15.6 Geometry Format Conversion Functions." Dosegljivo: https://dev.mysql.com/doc/refman/8.0/en/gis-format-conversion-functions.html. [Dostopano 29. 8. 2018].

[11] "Upgrading MySQL to version 8.0." Dosegljivo: https://mysqlserverteam.com/upgrading-to-mysql-8-0-with-spatial-data/. [Dostopano 13. 6. 2018].

[12] "mysql-connector-python · PyPI." Dosegljivo: https://pypi.org/project/mysql-connector-python/. [Dostopano 29. 8. 2018].

[13] "Tile38." Dosegljivo: http://tile38.com. [Dostopano 13. 6. 2018].

[14] "Tile38 Github repository." Dosegljivo: https://github.com/tidwall/tile38. [Dostopano 29. 8. 2018].

[15] "Redis." Dosegljivo: https://redis.io/. [Dostopano 13. 6. 2018].

[16] "andymccurdy/redis-py: Redis Python Client." Dosegljivo: https://github.com/andymccurdy/redis-py. [Dostopano 29. 8. 2018].

[17] "RFC 7946 - The GeoJSON Format." Dosegljivo: https://tools.ietf.org/html/rfc7946. [Dostopano 13. 6. 2018].

[18] "OSM XML - OpenStreetMap Wiki." Dosegljivo: https://wiki.openstreetmap.org/wiki/OSM_XML. [Dostopano 29. 8. 2018].

[19] "Basic Usage — Pyosmium 2.14.3 documentation." Dosegljivo: https://docs.osmcode.org/pyosmium/latest/intro.html. [Dostopano 29. 8. 2018].

[20] "Osm2pgsql - OpenStreetMap Wiki." Dosegljivo: https://wiki.openstreetmap.org/wiki/Osm2pgsql. [Dostopano 29. 8. 2018].

[21] "Welcome to the QGIS project!." Dosegljivo: https://qgis.org/en/site/. [Dostopano 29. 8. 2018].

[22] "ArcGIS — Main." Dosegljivo: https://www.arcgis.com/index.html. [Dostopano 29. 8. 2018].

[23] "trola.si API's documentation." Dosegljivo: https://trolasi.readthedocs.io/en/latest/. [Dostopano 29. 8. 2018].

[24] "Iskanje postajeliščnega voznega reda." Dosegljivo: http://www.lpp.si/sites/default/files/lpp_vozniredi/iskalnik/. [Dostopano 29. 8. 2018].

[25] "Let's Build A Web Server. Part 1. - Ruslan's Blog." Dosegljivo: https://ruslanspivak.com/lsbaws-part1/. [Dostopano 29. 8. 2018].

[26] "MySQL/MariaDB Spatial Support Matrix." Dosegljivo: https://mariadb.com/kb/en/library/mysqlmariadb-spatial-support-matrix/. [Dostopano 29. 8. 2018].