SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-52598

VYUŽITIE GRAFOVEJ DATABÁZY V PRAXI

BAKALÁRSKA PRÁCA

2017 **Juraj Kubričan**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-52598

VYUŽITIE GRAFOVEJ DATABÁZY V PRAXI

BAKALÁRSKA PRÁCA

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Maroš Čavojský

Bratislava 2017 **Juraj Kubričan**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2016/2017
Evidenčné číslo: FEI-5382-52598

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Juraj Kubričan**
ID študenta:	52598
Študijný program:	aplikovaná informatika
Študijný odbor:	9.2.9. aplikovaná informatika
Vedúci práce:	Ing. Maroš Čavojský
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Využitie grafovej databázy v praxi**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

V dnešnej dobe sa okrem tradičných zaužívaných relačných databáz, využívajú aj menej známe grafové databázy, v ktorých sú dáta uložené odlišným spôsobom ako v relačných databázach. Cieľom práce je oboznámiť sa s jednotlivými predstaviteľmi grafových databáz, vybrať jedného a navrhnúť a implementovať využitie vybranej grafovej databázy na reálnom príklade.

Úlohy:
1. Naštudujte si literatúru ohľadom jednotlivých predstaviteľov grafových databáz
2. Vyberte jedného predstaviteľa grafových databáz
3. Navrhnite reálny príklad pre implementáciu grafovej databázy
4. Implementujte reálny príklad pre implementáciu grafovej databázy
5. Zhodnoťte a uveďte výhody použitia grafovej databázy oproti iným typom databáz (relačné, dokumentové,...) v implementovanom reálnom príklade

Zoznam odbornej literatúry:

1. Ian Robinson, Jim Webber, and Emil Eifrem: Graph Databases, O'Reilly Media, Inc. USA 2015, p.224, ISBN: 978-1-491-93200-1

Riešenie zadania práce od: 19. 09. 2016

Dátum odovzdania práce: 19. 05. 2017

Juraj Kubričan
študent

prof. RNDr. Otokar Grošek, PhD.
vedúci pracoviska

prof. Dr. Ing. Miloš Oravec
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Juraj Kubričan
Bakalárska práca:	Využitie grafovej databázy v praxi
Vedúci záverečnej práce:	Ing. Maroš Čavojský
Miesto a rok predloženia práce:	Bratislava 2017

Bakalárska práca sa zameriava na opis výhod využitia grafovej databázy v praxi. V prvej časti sa nachádza porovnanie grafových a iných druhov databáz, predstavenie troch najpopulárnejších predstaviteľov grafových databáz, a výber jednej z nich pre ďalšie použitie v našej práci. Následne sa v práci nachádza špecifikácia a návrh aplikácie, ktorá využíva nami vybranú databázu Neo4J. V ďalšej časti je opis implementácie navrhnutej aplikácie na plánovanie a optimalizáciu trasy cestovateľa. Na reálnych dátach vygenerovaných aplikáciou boli následne vykonané testy porovnávajúce rýchlosť relačnej a grafovej databázy. Výsledky týchto testov a zhodnotenie ostatných výhod použitia grafovej databázy v praxi sa nachádzajú s poslednej kapitole.

Kľúčové slová: grafová databáza, OGM, Cypher, Neo4j

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Juraj Kubričan
Bachelor Thesis:	Graph database use in a real world application
Supervisor:	Ing. Maroš Čavojský
Place and year of submission:	Bratislava 2017

This thesis' main focus is on the advantages of using graph database management system in a real world application.

The first chapter of this thesis focuses on the comparison of graph databases to relational and document oriented databases. Next the thesis contains the specification and design of an application that uses Neo4J graph database. Next chapter focuses on the implementation of a web application for planning and optimization of travel routes. The following chapter contains tests done on the real world data generated by the application. The tests compare the speed of graph traversal in relational and graph databases and show the latency advantage the use of graph databases provide. Another benefits of using a graph database is the database model and tooling suitable for use with graph-like datasets.

Keywords: graph database, OGM, Cypher, Neo4j

Vyhlásenie autora

Podpísaný Juraj Kubričan čestne vyhlasujem, že som bakalársku prácu Využitie grafovej databázy v praxi vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Vedúcim mojej bakalárskej práce bol Ing. Maroš Čavojský.

Bratislava, dňa 4.7.2017

.....

podpis autora

Poďakovanie

Chcel by som poďakovat vedúcemu záverečnej práce, ktorým bol Ing. Maroš Čavojský, za odborné vedenie, rady a pripomienky, ktoré mi pomohli pri vypracovaní tejto bakalárskej práce.

Osobitné poďakovanie patí mojej rodine a priateľom za ich podporu, trpezlivosť a pomoc pri vypracovaní bakalárskej práce.

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek a značiek

TSP - problmém pocestného predajcu (Travelling Salesman Problem)
UML - Unified Modeling Language verzia 2.5
ORM - Mapovanie objektov na relácie (Object to Relational Mapping)
OGM - Mapovanie obejktov na graf (Object to Graph Mapping)
UX - (User Experience)
AJAX - Asynchrónny JavaScript dopyt (Asynchronous Javascript and XML)
MVC - Model, Zobrazenie, Ovládač (Model View Controller)
CSRF - Cross-site Request Forgery, typ útoku API - rozhranie pre programovanie aplikácií(Aplication programming interface)

Zoznam algoritmov

Úvod

Okrem zaužívaných relačných databáz sa v súčastnosti využívajú aj menej známe grafové databázy. Relačné databázy sú vhodné na prehľadávanie oddelených samostatných údajov. Pokiaľ však potrebujeme prehľadávať vzťahy v husto poprepájanej sade dát, relačné databázy narážajú na svoje limity. Podobné sady dát sa však v reálnom svete často vyskytujú, a preto vznikla potreba grafových databáz.

Grafové databázy sú natívne prispôsobené na prácu s grafmi. Neukladajú údaje v tabuľkách, ale vo vrcholoch a hranách. Tento spôsob uložena dát implicitne poskytuje výhody pri práci so sadami dát podobmými grafu.

V našej práci sa oboznámime s hlavnými predstaviteľmi grafových databáz. Z nich jedného vyberieme. Potom navrnhneme a budeme implementovať aplikáciu, ktorá bude vybranú databázu používať. Nakoniec zhodnotíme výhody, ktoré nám poskytlo použitie grafovej databázy v našej aplikácii.

1 Analýza problému

V tejto kapitole si predstavíme koncept grafových databáz a opíšeme, ako natívne riešia vzťahy medzi dátami. Ďalej si priblížime, ako riešia relačné vzťahy a dokumentové databázy. Následne si predstavíme troch najpopulárnejších predstaviteľov grafových databáz a vyberieme z nich jednu, s ktorou budeme ďalej pracovať. Nakoniec si predstavíme syntax dopytovacieho jazyka vybranej databázy.

1.1 Grafové databázy

Modelovanie grafovej databázy prirodzene zapadá do spôsobu, akým bežne abstrahujeme problémy pri vývoji softvéru. Pri návrhu softvéru objekty opisujeme obdĺžnikmi alebo kruhmi a súvislosti medzi nimi šípkami či čiarami. Moderné grafové databázy sú, viac ako akákoľvek iná databázová technológia, vhodné na takúto reprezentáciu, lebo to, čo namodelujeme na papier vieme priamo implementovať v našej grafovej databáze [3, 25–27].

Grafové databázy využívajú model, ktorý pozostáva z hrán, vrcholov, atribútov a značiek. Vrcholy obsahujú atribúty a sú označené jednou alebo viacerými značkami. Tieto značky zoskupujú vrcholy, ktoré zastávajú rovnakú rolu v rámci aplikácie. Hrany v grafových databázach spájajú vrcholy a budujú štruktúru grafu. Hrana grafu má vždy smer, názov, začiatočný vrchol a koncový vrchol. To, že hrany musia mať smer a názov pridáva sémantickú prehľadnosť do grafu. Ak zvolíme správne názov, vieme rýchlo identifikovať štruktúru grafu a význam vzťahov. Hrany môžu rovnako, ako vrcholy obsahovať aj atribúty. Atribúty v hranách môžu byť použité na pridanie kvalitatívnych dát (napr. váha, vzdialenosť) ku vzťahom. Tieto dáta sa potom môžu použiť pri prehľadávaní grafu.

Natívne grafové databázy používajú bezindexovú priľahlosť. To znamená, že prepojenia sa riešia pomocou ukazovateľov a nie cudzích kľúčov, ako v relačných databázach. Absencia cudzích kľúčov v praxi znamená, že pri prehľadávaní vzťahov je výpočtová zložitosť prehľadania jednej hrany $O(1)$. Táto rýchlosť je dosiahnutá tak, že všetky hrany sú uložené s priamymi ukazovateľmi na vrcholy, ktorých vzťah reprezentujú. Taktiež, vo vrcholoch sú uložené priame ukazovatele na všetky hrany vychádzajúce z nich a mieriace do dotyčného vrcholu. Takáto štruktúra poskytuje už spomínanú výpočtovú zložitosť $O(1)$ v oboch smeroch hrany, takže nielen v smere zo začiatočného bodu do koncového, ale aj opačným smerom[3, 149–158]. Pri relačnej databáze by toto muselo byť riešené reverzným vyhľadávaním v cudzích kľúčoch.

1.2 Vzťahy v relačných databázach

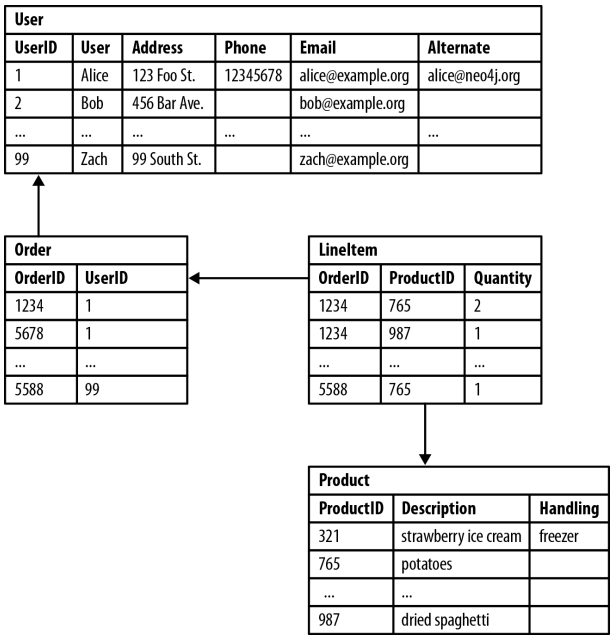
Podľa E. F. Codda z roku 1970 je relačná databáza taká, v ktorej sú údaje uložené podľa relačného databázového modelu. Relačný model je založený na matematickom aparáte relačných množín predikátovej logiky. Databázová relácia sa od matematickej líši v tom, že využíva pomocný aparát nazývaný schéma. V databáze, schéma definuje názov relácie (tabuľky), koľko obsahuje stĺpcov a ich dátový typ[2].

Vývojári databáz sa už desaťročia snažia relačné databázy prispôsobiť na to, aby boli schopné efektívne pracovať s husto prepojenými sadami dát. Keďže štruktúra relačných databáz bola pôvodne navrhnutá na ukladanie formulárov a tabuľkových štruktúr, vznikajú problémy pri snahe zahrnúť vzťahy, ktoré sa vyskytujú v dátach v reálnom svete [3, 11–12].

Vzťahy sa v relačných databázach vyskytujú iba vo fáze návrhu. V reálnej implementácii tieto vzťahy nahrádzajú spájacie tabuľky a cudzie kľúče. Týmto prístupom rýchlo narastá komplexnosť riešenia a znižuje sa prehľadnosť dátového modelu. Pridáva sa záťaž na databázu vznikom veľkých spájacích tabuliek a stĺpcov s cudzími kľúčmi, ktoré môžu byť riedko zaplnené, ale zaberajú priestor na disku[3, 11–12].

Každé prepojenie pri prehľadávaní vzťahov v relačnej databáze pridáva výpočtovú zložitosť. V každej ďalšej prepojenej tabuľke treba vyhľadať záznam s požadovaným kľúčom so zložitosťou $O(log(n))$. Používané relačné databázové systémy riešia tento výpočtový problém použitím indexov a inými optimalizáciami. Ale pokiaľ je dátová sada štruktúrovaná s veľkým množstvom vzťahov, systém sa spravidla spomalí.

Na obrázku č.1 vidíme prístup riešenia vzťahov v relačnej databáze.



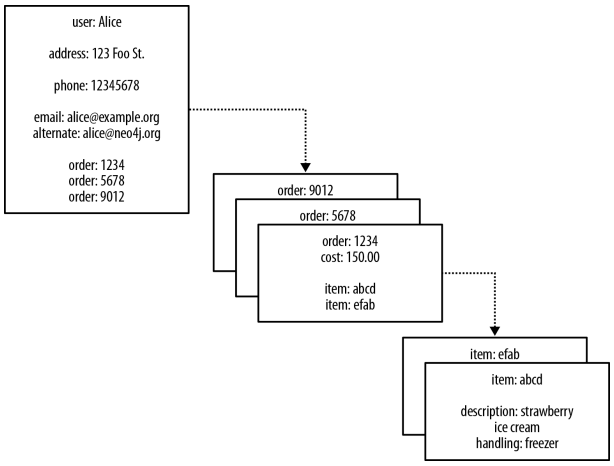
Obrázok 1: Ukážka riešenia vzťahov v relačnej databáze [3, 12]

1.3 Vzťahy v dokumentových databázach

Dokumentové databázy patria rovnako, ako grafové do kategórie NoSQL databáz. V dokumentovej databáze sa údaje ukladajú, ako štruktúrované objekty, ktoré sú adresované pomocou unikátnych kľúčov. Sémanticky vieme dokumenty rozlíšiť do kolekcií podľa ich významu v databáze. Štruktúra grafovej databázy je primárne určená na dopyt podľa kľúča a neposkytuje výhodu pri prehľadávaní cez vzťahy oproti relačným databázam. Rôzne implementácie dokumentových databáz implementujú odlišné prístupy k riešeniu vzťahov.

Bežný spôsob na pridanie vzťahov do štruktúry dokumentovej databázy je pridanie atribútu do dokumentu, ktorý bude odkazovať na kľúč a kolekciu dokumentu, s ktorým chceme vytvoriť vzťah. Týmto spôsobom vieme použiť koncept cudzích kľúčov z relačných databáz. Tento prístup sa však stretáva s rovnakými problémami v rýchlosti, ako pri relačných databázach s pridanou zložitosťou v tom, že tento dopyt musíme často implementovať na strane aplikácie.

Na obrázku č.2 vidíme príklad prístupu implementácie vzťahov v dokumentovej databáze.



Obrázok 2: Ukážka riešenia vzťahov v dokumentovej databáze [3, 15]

1.4 Výber grafovej databázy

Vybrali sme si troch najpopulárnejších predstaviteľov grafových databáz podľa rebríčka DB-Engines.com. V nasledujúcej časti v skratke priblížime históriu každej grafovej databázy, ich výhody a nevýhody.

1.4.1 Neo4J

Prvá verzia Neo4J bola vydaná v roku 2007. Od vtedy sa stala dlhodobo najpoužívanejšou grafovou databázou. Je vyvíjaná Neo Technology, Inc. Neo4j je ponúkaná v dvoch variantoch: Neo4j Community a Neo4j Enterprise. Neo4j Community je open source (GPLv3) grafová databáza obsahujúca všetky základné funkcie. Ďalej budeme spomínať len túto verziu. A Neo4j Enterprise edícia, ktorá má rozšírené funkcie, ako zdieľanie vyrovnávacej pamäte, rozšírené monitorovanie a zálohovanie za behu[1] [**?**]

.

Medzi hlavné výhody Neo4J patria:

- Podľa rebríčka db-rank[1], je dlhodobo najpopulárnejšo u grafovou databázou. To znamená, že máme k dispozícii dobrú dokumentáciu, podporu na internete a väčšiu podporu pre integráciu s populárnymi frameworkami.
- Dopytovací jazyk Cypher, ktorý táto databáza podporuje. Tento jazyk je vizuálne zrozumiteľný a navrhnutý primárne pre prácu z grafovou databázou. Ďalej sa mu venujeme v kapitole 1.5.

Nevýhodami Neo4J sú:

- Nižšia rýchlosť oproti OrientDB.

- Podporuje iba grafový model ukladania údajov.

1.4.2 OrientDB

OrientDB je vyvíjané od roku 2010 firmou OrientDB LTD. Databáza OrientDB rýchlo nabrala na popularite a v roku 2015 sa dostala na druhé miesto v rebríčku DB-engines. OrientDB sa rovnako, ako Neo4j distribuuje v dvoch edíciách: Community - open source (Apache Licence 2.0) so základnými funkciami a Enterprise edícia s podporou migrácie, a synchronizácie na Neo4J, a pridanými analytickými nástrojmi [1] [**?**].

Medzi hlavné výhody OrientDB patrí:

- Podporuje okrem grafového modelu ukladania dát aj dokumentové úložisko typu kľúč-hodnota.
- Podľa nezávislých testov [5] je v niektorých testoch až desaťkrát rýchlejšie ako Neo4J.

Nevýhodami OrientDB sú:

- Má menšiu podporu pre populárne frameworky. Konkrétne OGM pre Laravel nepodporuje ukladanie a čítanie atribútov z hrán.
- Menej intuitívny dopytovací jazyk v kontexte grafových databáz.

1.4.3 Titan

Projekt Titan bol od roku 2012 vyvíjaný skupinou ThinkAurelius. Avšak v roku 2017 bol odkúpený firmou Datstax a projekt bol zastavený. Projekt je ďalej udržiavaný, ako open source verzia pod menom JanusGraph.
Titan bol projekt určený na veľké distribuované enterprise riešenia. Nasadzovaný na cloudové platformy ako napr. Apache Hadoop a Apache Spark. Podporuje rôzne distribuované úložné priestory, ako napr. Apache HBase a Oracle Berkeley DB. Ďalej podporuje rôzne vyhľadávacie prostriedky, ako napr. Elasticsearch a Solr [1] [**?**].

Medzi hlavné výhody databázy Titan patrí:

- Podpora rôznych zásuvných modulov vyžívaných v enterprise riešeniach.

Nevýhodami databázy Titan sú:

- Ukončenie vývoja po akvizícii firmou DataStax.
- Jej určenie na veľké distribuované systémy.

1.4.4 Vyhodnotenie

V procese výberu databázy sme prihliadali na rôzne faktory. Posudzovali sme hlavne kompatibilitu, jednoduchosť použitia a rýchlosť. Nakoniec sme sa rozhodli pre Neo4J práve pre jeho rozšírenosť a použitie jazyka Cypher. Aj keď podľa nezávislých testov je OrientDB rýchlejšie ako Neo4J v kontexte porovnávania relačných a grafových databáz, tento fakt nezohral veľkú rolu, lebo obe databázy sú pri navigovaní grafu rýchlejšie ako relačné. Databázu Titan sme vylúčili z výberu z dôvodu ukončenia jej vývoja a určenia primárne na distribuované systémy.

1.5 Dopytovací jazyk Cypher

Dopytovací jazyk Cypher bol vyvinutý firmou NeoTechnology pre ich databázu Neo4J a prípadnú štandardizáciu tohoto jazyka v ostatných grafových databázach. Cypher bol navrhnutý tak, aby sa ľahko čítal a chápal. Preto používa intuitívnu *"ASCII Art"* notáciu. Cypher používa oblé zátvorky na reprezentáciu vrcholov, hranaté zátvorky a pomlčky na reprezentáciu hrán, a znaky väčší, menší na upresnenie smerovania hrany. Medzi hlavné dopytovacie príkazy v Cypher patrí príkaz MATCH. Týmto príkazom podobne ako SELECT v SQL začína každý dopyt.

V algoritme č.1 vidíme niekoľko príkladov dopytov. V prvom príklade vidíme dopyt, ktorý vyhľadá vrchol n s *id = 42* (*id* nie je atribút vrchola, preto ho musíme získať pomocou funkcie *id()*), pričom z tohoto vrchola vychádzajú (všimnime si smerovanie "ASCII ART šípky") hrany so značkou VZTAH, ktoré vedú k cieľovým vrcholom so značkou *znA*. Všimnime si, že v zátvorkách reprezentujúcich vrcholy a hrany priraďujeme premenným *n*, *m* a *r* kolekcie vrcholov a hrán. Keďže je v dopyte *RETURN r, m*, vrátia sa vrcholy v týchto kolekciách, čiže nie pôvodný vrchol s *id = 42*.

V druhom príklade vidíme dopyt, ktorý začne z vrcholov, ktorých atribút *nazov* má hodnotu *názov*. Ďalej vyhľadá a vráti všetky vrcholy, ktoré sú od neho v grafe vzdialené tri hrany typu *VZTAH* a ich atribút *text* vyhovuje regulárnemu výrazu */M.*/*.

V poslednom príklade vidíme príkaz, ktorý najprv vyhľadá vrcholy s atribútom *meno* a hodnotami *Adam* a *Bernard*, a pridá hranu smerujúcu od prvého smerom k druhému. Vytvorená hrana bude mať atribút *vaha* s hodnotou *7*.

Algoritmus 1 Príklady dopytov v jazyku Cypher

```
1
2  1)
3  MATCH (n)-[r:VZTAH]->(m:znA)
4  WHERE id(n) = 42 RETURN r, m
5
6  2)
7  MATCH (n:typ)-[r:VZTAH*3]->(m:znB)
8  WHERE n.nazov = 'názov' AND m.text=~ "M.*" RETURN m
9
10 5)
11 MATCH (a:typ),(b:znA) WHERE a.meno = 'Adam' AND b.meno = 'Bernard'
12 CREATE (a)-[r:VZTAH { vaha: 7 }]->(b)
13 RETURN r
```

2 Špecifikácia

V tejto kapitole najprv stručne opíšeme hlavnú funkcionalitu navrhovanej aplikácie, potom zadefinujeme funkcionálne a nefunkcionálne požiadavky na aplikáciu.

Aplikácia, ktorú sme sa rozhodli implementovať bude slúžiť na plánovanie a optimalizáciu trasy cestovateľa po svete. Bude umožňovať používateľovi sledovať ceny cesty z miest, ktoré si zadá, do destinácií, ktoré si pridá na zoznam obľúbených destinácií. Ďalej mu bude ukazovať ostatných používateľov, ktorí si zvolili rovnaké destinácie a bude vedieť používateľovi odporučiť ďalšie destinácie na základe zhody s ostatnými používateľmi.

2.1 Funkcionálne požiadavky

1. Aplikácia bude umožňovať registráciu a prihlásenie používateľa.
2. Pri registrácií sa budú vyžadovať prihlasovacie údaje: e-mail a heslo. Okrem toho sa bude vyžadovať zadanie domovskej destinácie.
3. Po prihlásení používateľa sa zobrazí obrazovka s mapou, zoznamom obľúbených destinácií, ktoré chce navštíviť a zoznam odporúčaných destinácií.
4. Na mape bude vyobrazená používateľova domovská destinácia a všetky destinácie, ktoré má v zozname obľúbených.

5. V zozname obľúbených budú všetky destinácie, ktoré si používateľ pridal. Zoznam bude vo forme tabuľky, ktorej riadok bude obsahovať meno destinácie a cenu najlacnejšej trasy z domovskej destinácie do cieľovej destinácie.

6. Budú sa dať zobraziť informácie o všetkých dostupných trasách ku konkrétnej destinácii s ich cenami a spôsobmi dopravy.

7. V detailoch obľúbenej destinácie bude zoznam ostatných používateľov, ktorí danú destináciu majú tiež na zozname obľúbených.

8. V zozname odporúčaných destinácií budú destinácie, ktoré majú na zozname ľudia, s ktorými má prihlásený používateľ najviac spoločných destinácií.

9. V aplikácií bude obrazovka, kde si bude používateľ vyberať niekoľko zo svojich obľúbených destinácií a nechá si vyrátať optimálnu trasu z domovskej destinácie cez všetky zvolené miesta a potom spať (TSP).

2.2 Nefunkcionálne požiadavky

1. Systém bude zrealizovaný na webovej platforme.

2. Aplikácia bude využívať natívnu grafovú databázu.

3. Aplikácia bude kompatibilná s poslednými verziami webových prehliadačov Google Chrome, Mozilla Firefox, Microsoft Edge.

4. Užívateľské rozhranie systému musí byť plne funkčné aj na mobilných telefónoch s operačným systémom Android a IOS.

5. Aplikácia bude implementovaná s použitím jazyka PHP a PHP frameworku.

6. Systém bude nasadený na virtuálnom serveri s operačným systémom Ubuntu 16.04.2 LTS poskytnutom Ústavom informatiky a matematiky FEI STU.

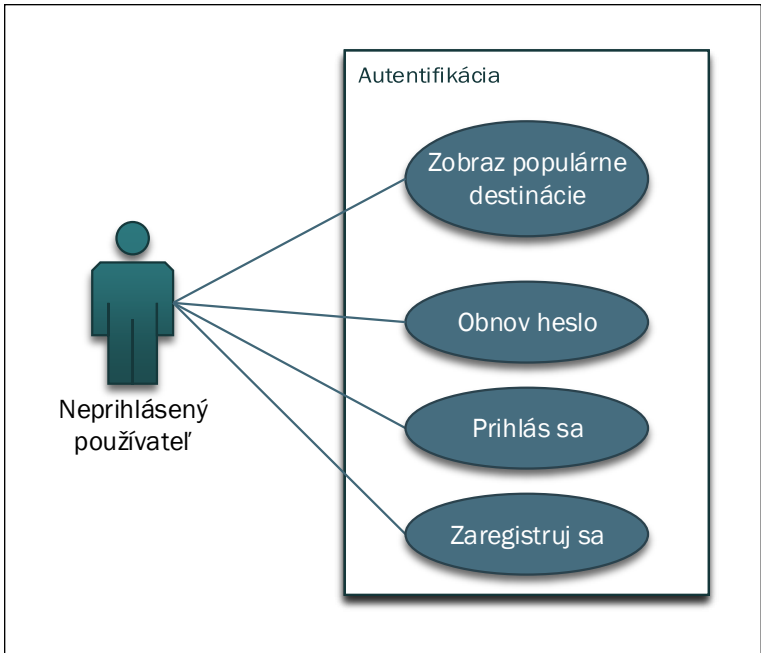
3 Návrh

V tejto kapitole si priblížime proces návrhu našej aplikácie. Popíšeme prípady použitia, ktoré pre našu aplikáciu chceme pokryť. Ďalej si priblížime návrh štruktúry databázy a nakoniec si priblížime štruktúru samotnej aplikácie, ktorú budeme implementovať.

3.1 Prípady použitia

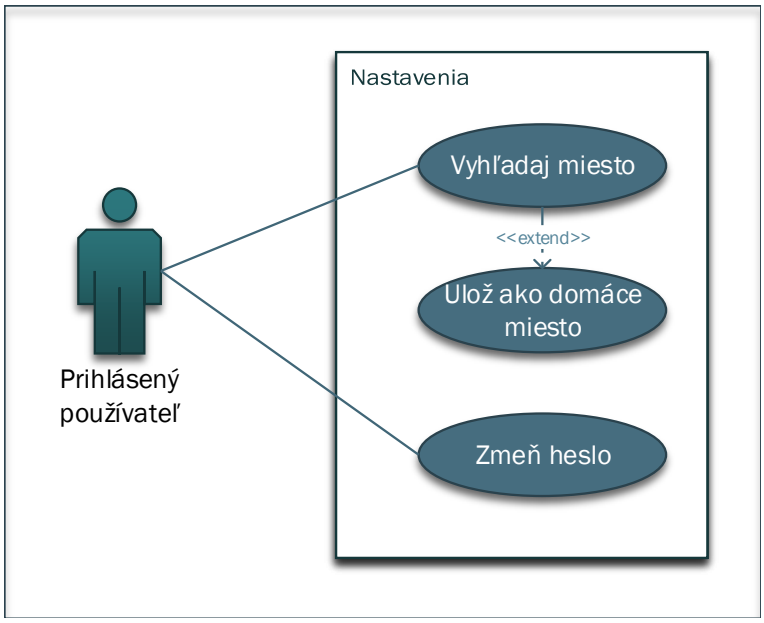
Prípady použitia popisujú interakcie medzi aplikáciou a používateľom. Popisujú roly, ktoré rôzni aktéri majú v týchto interakciách.

1. V prvom prípade použitia (obrázok č.3) si bude môcť neprihlásený používateľ prezerať mapu a zoznam najpopulárnejších destinácií v našej aplikácii. Ďalej sa môže prihlásiť, zaregistrovať a požiadať o obnovu hesla.



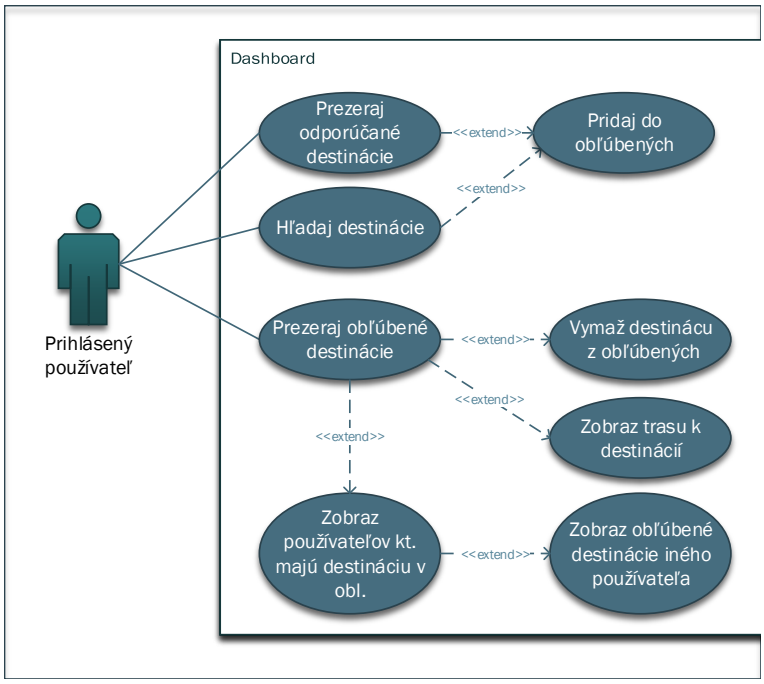
Obrázok 3: Prípad použitia - Registrácia, autentifikácia, obnova hesla a prezeranie informácií

2. V druhom prípade použitia (obrázok č.4) opisujeme nastavenia. Používateľ si bude môcť vyhľadať destináciu a zvoliť si ju, ako svoju domovskú destináciu alebo si zmeniť heslo.



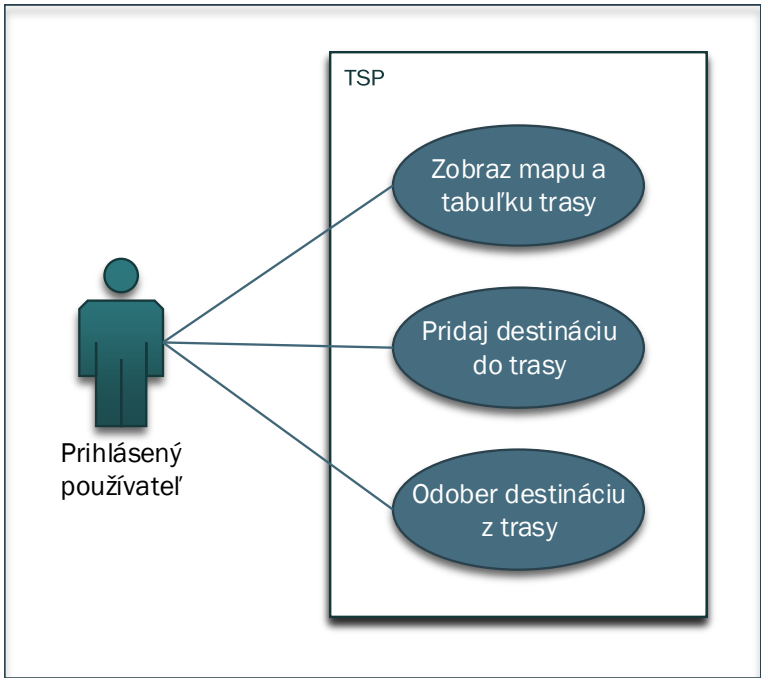
Obrázok 4: Prípad použitia - Nastavenia

3. Tretí prípad použitia (obrázok č.5) opisuje akcie, ktoré bude môcť používateľ vykonávať na hlavnej stránke. Bude môcť vyhľadávať destinácie a pridávať si ich do obľúbených. Destinácie, ktoré má v obľúbených bude môcť mazať a zobraziť detaily trasy, ktorá k destinácii vedie. Ďalej si bude môcť prezerať ostatných používateľov, ktorí majú rovnaké miesto v obľúbených a prejsť na zoznam obľúbených destinácií jedného z týchto používateľov.



Obrázok 5: Prípad použitia - Dashboard

4. Posledný prípad použitia (obrázok č.6) popisuje obrazovku TSP. Na tejto obrazovke bude môcť používateľ prezerať svoju mapu optimálnej cestovateľskej trasy, pridávať a odoberať z trasy destinácie zo zoznamu obľúbených.



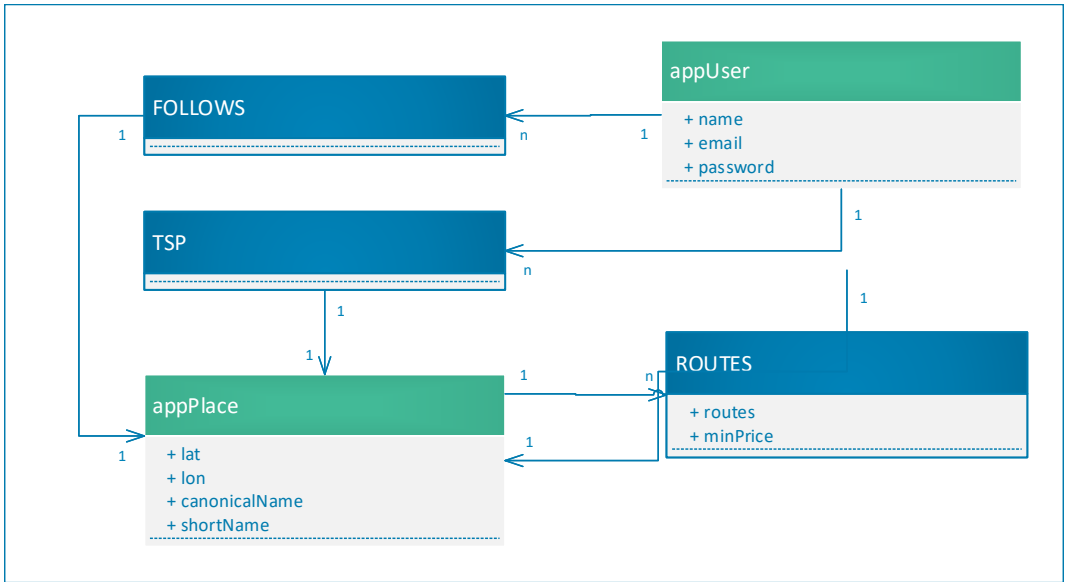
Obrázok 6: Prípad použitia - TSP

3.2 Štruktúra databázy

V našej databáze sa budú nachádzať dve hlavné entity: entita používateľa *appUser* a entita destinácie *appPlace*. Entita používateľa bude obsahovať iba základné atribúty, ako meno a atribúty potrebné pre autentifikáciu používateľa, čiže email a heslo. Entita destinácie bude obsahovať atribúty potrebné na identifikáciu tejto destinácie: unikátny názov a krátky názov vhodný na zobrazovanie, a atribúty potrebné na lokalizáciu destinácie: zemepisná šírka a zemepisná dĺžka.

Ďalej bude naša databáza obsahovať hrany: trasa *ROUTES*, obľúbené *FOLLOWS* a zoznam pre optimalizáciu trasy *TSP*. Hrana *ROUTES* bude vždy viesť od jedného miesta k druhému. Bude reprezentovať zoznam trás, ktoré budú viesť od jednej destinácie k druhej. Bude obsahovať atribút *minPrice* hovoriaci o cene najlacnejšej trasy medzi spomínanými destináciami a atribút *routes* obsahujúci serializované detaily všetkých trás. Hrany *FOLLOWS* a *TSP* budú vždy smerovať od používateľa k destinácii a nebudú obsahovať žiadne atribúty.

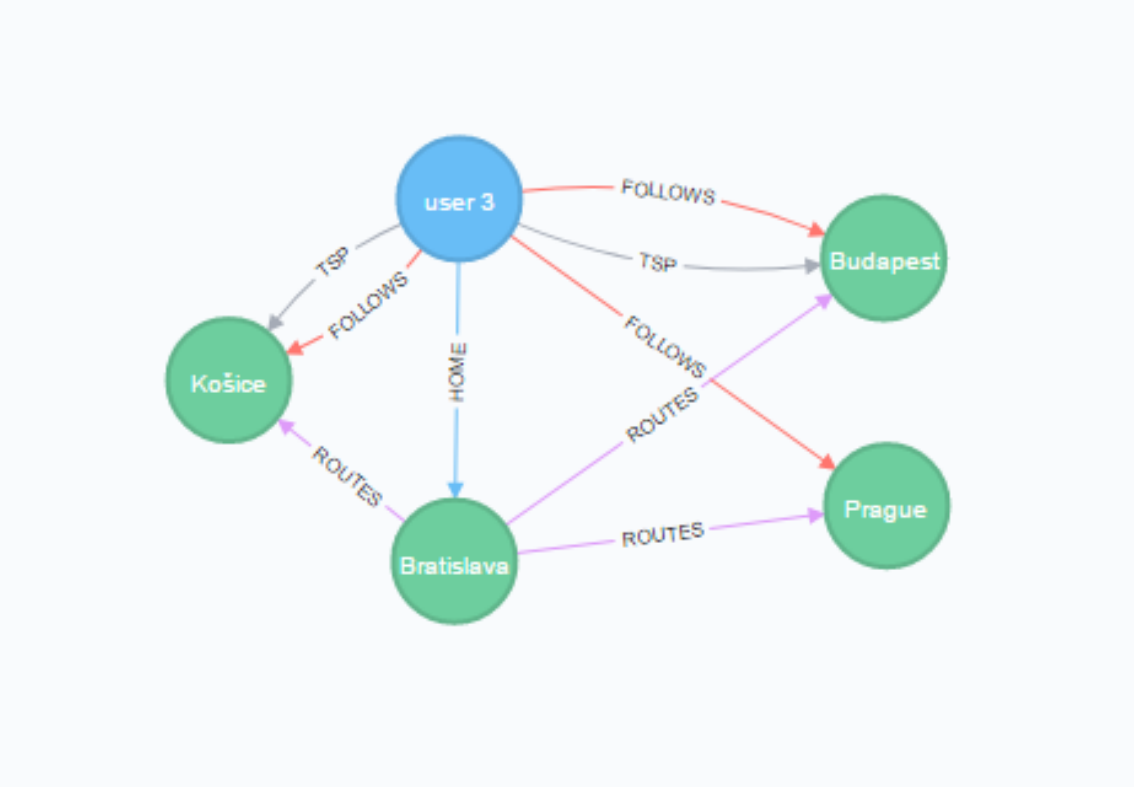
Na obrázku č. 7 vidíme UML diagram štruktúry našej databázy.



Obrázok 7: Štruktúra databázy

Na obrázku č. 8 vidíme reprezentáciu dát uložených v našej databáze. Používateľ *user 3* má ako domácu destináciu (HOME) nastavené mesto *Bratislava*. V obľúbených destináciách (FOLLOWS) má mestá: *Praha*, *Košice* a *Budapešť*. Systém pridal hranu *ROUTES* smerom od domácej destinácie používateľa k cieľovým destináciam. Používateľ

má tiež pridané mestá *Budapešť* a *Košice* na zozname optimalizácie, takže sme pridali hranu *TSP*.



Obrázok 8: Príklad dát v databáze

3.3 Štruktúra aplikácie

Aplikácia bude implementovaná podľa vývojového vzoru MVC. Bude obsahovať dve modelové triedy. Jednu pre používateľov a jednu pre destinácie. Časť kontroléra bude obsahovať tri triedy. Každá trieda bude obsluhovať dopyty jednej podstránky aplikácie. Podstránky sú nasledovné: domovská obrazovka, obrazovka nastavení, a obrazovka pre optimalizáciu trasy. Zobrazovanie dynamického obsahu bude implementované pomocou JavaScript šablón a knižnice na zobrazenie máp. Dáta na zobrazenie budú poskytované pomocou API našej aplikácie.

4 Implementácia

V tejto kapitole si priblížime proces implementácie aplikácie. Najprv si opíšeme niektoré z použitých technológií, následne opíšeme proces inštalácie a konfigurácie frameworku Laravel, a databázy Neo4J. Ďalej opíšeme proces našej implementácie aplikácie s pomocou NeoEloquent OGM a pomocou Neo4J PHP Client. Ďalej opíšeme postup našej implementácie optimalizácie trasy. Potom opíšeme implementáciu získavania informácií z API Rome2Rio a našu implementáciu používateľského rozhrania pomocou Mapbox.js a Handlebars.js. Nakoniec si predstavíme používateľské rozhranie našej aplikácie.

4.1 Použité technológie

v tejto kapitole si priblížime niektoré z technológií, ktoré sme využili na implementáciu našej aplikácie. V skratke opíšeme, na čo konkrétna technológia slúži, jej históriu a licenciu, pod ktorou je vydávaná.

4.1.1 Laravel framework

Laravel je open source framework pre aplikácie v jazyku PHP. Je od roku 2011 vyvíjaný Taylorom Otwellom, ako framework určený na stavbu webových aplikácií s vývojovým vzorom MVC. Niektoré funkcie sú implementované, ako zásuvné moduly, ktoré sa inštalujú pomocou vlastného správcu závislostí. Laravel je orientovaný na to, aby vývoj v ňom bol syntakticky, čo najprehľadnejší a najjednoduchší. Toto je jeden z dôvodov, prečo patrí od roku 2015 k najpopulárnejším PHP frameworkom. Kód frameworku je voľne dostupný pod licenciou MIT.

4.1.2 NeoEloquent OGM

NeoEloquent OGM je voľne šíriteľná knižnica, ktorá umožňuje využívať grafovú databázu Neo4J spolu s existujúcim dátovým modelom vo frameworku Laravel. Štruktúra NeoEloquentu je modelovaná podľa natívneho Eloquent Modelu Laravel. Knižnica je vydávaná od roku 2014 pod licenciou MIT spoločnosťou Vinelab.

4.1.3 GraphAware Neo4j PHP Client

Neo4j PHP Client je klientská knižnica pre databázu Neo4J. Na rozdiel od OGM je táto knižnica schopná vykonávať programátorsky zadané dopyty. Poskytuje tým pádom prístup k plnému rozsahu Cypher API databázy Neo4j. Knižnica je vyvíjaná partnerskou firmou Neo Technology a firmou GraphAware od roku 2014. Je taktiež vydávaná pod licenciou MIT.

4.1.4 Mapbox.js

Mapbox.js je knižnica na vytváranie projektov s interaktívnymi mapami. Je založená na voľne šíriteľnej knižnici Leaflet vyvíjanou Vladimírom Agafonkinom. Rozširuje túto knižnicu o funkcie, ako automatické zoskupovanie bodov do skupín a poskytuje prehľadnú dokumentáciu. Knižnica samotná je voľne šíriteľná pod BSD licenciou. Avšak obrázky s mapami sú poskytované firmou Mapbox a v zadarmo dostupnej verzii je knižnica obmedzená na päťdesiattisíc zobrazení na mesiac.

Túto knižnicu postupne nahrádza Mapbox-gl.js vyvíjaná tou istou firmou, ktorá využíva technológiu WebGL na lepšiu akceleráciu vykresľovania vektorových máp. Ale v čase začiatku projektu táto knižnica ešte nepodporovala vyššie spomenuté zoskupovanie bodov.

4.1.5 Handlebars.js

Handlebars.js je jednoduchá silná open source knižnica na prácu so šablónami. Vyvíjaná je Yehudom Katzom a komunitou na GitHube od roku 2010 a vydávaná pod licenciou MIT.

4.1.6 Rome2Rio Api

Rome2Rio je portál, ktorý zbiera údaje o cenách dopravy po celom svete a umožňuje vyhľadávať cenu cesty medzi dvoma ľubovoľnými destináciami. Rome2Rio taktiež poskytuje niekoľko otvorených a platených API. My využívame dve z nich: Search API a Autocomplete API. Autocomplete API slúži na vyhľadávanie destinácií z databázy Rome2Rio a nie je obmedzené na počet volaní. Search API slúži na vyhľadávanie trás medzi jednotlivými destináciami, pričom je potrebné sa identifikovať API kľúčom a je obmedzené na 100 000 volaní za mesiac.

4.2 Inštalácia a konfigurácia Laravel Frameworku

Na inštaláciu frameworku Laravel sme použili nástroj pre správu PHP balíkov *Composer*. Pomocou tohoto nástroja sme nainštalovali balík *laravel/installer* [7] a následne sme použitím príkazu *laravel new projekt* vytvorili priečinok so základnou inštaláciou frameworku.

Jediné nastavanie, ktoré bolo potom potrebné, bolo už len nastavenie názvu aplikácie a databázy. O tom ďalej v sekcii 4.3.

4.3 Inštalácia a konfigurácia databázy Neo4J

Aby sme mohli nainštalovať databázu Neo4j, musíme si najprv do systému pridať repozitár Neo Technology. Potom je nám k dispozícii na inštaláciu balík Neo4J. Po inštalácii je nám

ihneď dostupné administračné rozhranie databázy na adrese: *localhost:7474*. Pri prvom prihlásení sme vyzvaní na zmenu hesla.

Keďže základná inštalácia frameworku Laravel neobsahuje ovládač pre databázu Neo4J, museli sme použiť ovládač integrovaný v balíku NeoEloquent a to pomocou registrácie poskytovateľa služby. Po zaregistrovaní služby NeoEloquentServiceProvider sa automaticky zaregistruje ovládač pre databázu a pridajú sa nové možnosti pre konfiguráciu databázy. Následne stačí vykonať štandardnú konfiguráciu mena hostiteľa, portu a prístupových údajov.

Na získanie vzdialeného prístupu k administračnému rozhraniu databázy bez otvorenia portu 7474 verejnosti, využívame SSH tunel. Administračné rozhranie používa okrem portu 7474 ešte port 7687, lebo na komunikáciu s databázou využíva technológiu WebSocket.

4.4 Implementácia OGM

Keďže framework Laravel natívne obsahuje len ovládače pre relačné databázy a nástroj na objektovo relačné mapovanie Eloquent, použili sme open source knižnicu NeoEloquent, ktorá obsahuje ovládač pre databázu Neo4J, a zároveň rozširuje dátový model o prvky grafovej databázy. NeoEloquent umožňuje manipuláciu s vrcholmi aj hranami v Neo4J. Manipulácia s vrcholmi je rovnaká, ako s entitami v relačnej databáze. NeoEloquent ďalej umožňuje vytvárať perzistentné objekty, upravovať ich a vyhľadávať v nich. Práca s hranami je mierne odlišná. Najprv treba zadefinovať, ktorý objekt môže mať, aké vzťahy. Tieto vzťahy treba jednoznačne identifikovať ich značkou a kardinalitou. Tento vzťah vraciame, ako návratovú hodnotu funkcie daného objektu. Vrátený objekt sa správa podobne, ako štandardná entitná trieda Eloquent.

Knižnicu sme nainštalovali pomocou správcu balíkov *composer* pomocou príkazu *composer require vinelab/neoeloquent*.

V nasledujúcom príklade 3 vidíme implementáciu dvoch rôznych entitných tried. Trieda používateľa dedí od triedy NeoEloquent, a teda sa stáva naviazanou na vrchol v našej databáze. Názov tejto entity v databáze je spojením menného priestoru, v ktorom bol vytvorený a názvu triedy, takže v našom prípade AppUser. Trieda obsahuje verejné funkcie, ktoré vracajú objekty hrán. Objekty hrán dostávame volaním zdedených funkcií hasMany, hasOne a belongToMany. Ako prvý argument funkcie berú názov triedy, ktorej vzťah chceme vrátiť, a ako druhý argument berie typ hrany. Pomocou tohoto typu je identifikovaná značka hrany v databáze.

Po spustení aplikácie a zadaní prvých dát sa v databáze vytvorili vrcholy a hrany

podľa modelu, ktorý sme si zadefinovali v kapitole 3.2. Vizualizáciu dát môžeme vidieť na obrázku č.8.

Trieda NeoEloquent funguje vo väčšine prípadov presne ako Eloquent. Avšak v jednom prípade sme mali problém z CSRF tokenmi. CSRF token je bezpečnostný prvok, ktorý ochraňuje webovú stránku pred útokom falšovania požiadaviek z inej adresy. Laravel má tento bezpečnostný prvok vstavaný v sebe. Je to 40-znakový reťazec, ktorý sa generuje každému používateľovi pri zobrazení formulára. Tento istý reťazec sa zároveň uloží do databázy, a keď Laravel príjme formulár, overí či sa jeho token nachádza v databáze. Táto funkcionalita však po prejdení na grafovú databázu nefungovala. Pri každom odoslaní formulára vyhlasovalo nezhodu CSRF tokenu a v databáze sa neobjavila entita, ktorá by tieto tokeny mohla obsahovať. Tento problém sme zatiaľ obišli deaktiváciou tohoto bezpečnostného prvku.

Ďalší problém, na ktorý sme narazili počas implementácie, bol problém kompatibility knižnice NeoEloquent verzie 1.4 s databázou Neo4J vo verzii 5.2. Pri ukladaní nového vrcholu vždy nastala chyba. Bolo potrebné znížiť verziu databázy.

Algoritmus 2 Ukážka triedy NeoEloquent

```
1  namespace App;
2
3  class User extends \NeoEloquent implements Authenticatable {
4
5          // Jeden používateľ môže mať v obľúbených viac miest
6          public function follows() {
7                  return $this->hasMany('App\Place', 'FOLLOWS');
8          }
9
10         // Jeden používateľ má jedno miesto ako domov
11         public function home() {
12                 return $this->hasOne('App\Place', 'HOME');
13         }
14         ...
15 }
16
17 ...
18 class Place extends \NeoEloquent {
19         // Inverzný vzťah - jedno miesto má v obľúbených viac
20         // používateľov
21         public function followers(){
22                 return $this->belongsToMany('App\User','FOLLOWS');
23         }
24         ...
25 }
```

4.5 Autentifikácia

Jednou zo silných stránok frameworku Laravel je práve autentifikácia. Pre vytvorenie základnej funkcionality registrácie, prihlasovania a obnovenia zabudnutého hesla stačí použiť Artisan - v konzole frameworku príkaz *php artisan make:auth*, ktorá vytvorí URL cesty, obrazovky, triedu používateľa a triedy obsluhujúce túto funkcionalitu. My sme potrebovali použiť vlastnú triedu používateľa, ktorá dedí od nášho balíka NeoEloquent. Na implementáciu autentifikácie stačilo implementovať rozhranie Authenticatable, pridať do triedy používateľa, pole skrytých a verejných atribútov, a použiť charakteristiku 'AuthenticableTrait'. A autentifikácia fungovala rovnako, ako s relačnou databázou vďaka tomu,

že NeoEloquent implementuje podobnú funkcionalitu, ako natívne ORM Eloquent.

Algoritmus 3 Ukážka autentifikovateľnej triedy

```
1
2  namespace App;
3
4  use Illuminate\Contracts\Auth\Authenticatable;
5  use Illuminate\Auth\Authenticatable as AuthenticableTrait;
6
7  class User extends \NeoEloquent implements Authenticatable {
8
9          use AuthenticableTrait;
10
11         // pole verejných atribútov
12         protected $fillable = [
13                 'name', 'email', 'password', 'tspCache'
14         ];
15
16         // pole skrytých atribútov
17         protected $hidden = [
18                 'password', 'remember_token',
19         ];
20         ...
21 }
```

4.6 Databázové dopyty pomocou OGM

API pre získanie dát na zobrazenie zoznamu obľúbených destinácií sme v prvej verzii implementovali pomocou OGM. Na príklade algoritmu č.4 vidíme pseudokód tohoto prístupu. Najprv naša PHP funkcia urobila dopyt na všetky miesta, ktoré má používateľ v obľúbených. Potom v cykle prešla všetky výsledky a pre každú destináciu poslala dopyt na trasu medzi domovskou destináciou a touto destináciou. Ďalej poslala dopyt na získanie všetkých používateľov, ktorí majú destináciu tiež v obľúbených.

Pri testovaní rýchlosti sme zistili, že pri takomto prístupe priemerná latencia nášho API vzrástla až na 1,857 sekundy pri 200 položkách na zozname. Tento nárast latencie pripisujeme réžii, ktorá si vyžaduje diskrétne volanie funkcií OGM v každom cykle. Ďalej sa testovaniu rýchlosti venujeme v kapitole 5.

Algoritmus 4 Pseudokód algoritmu na získavanie dát pre zoznam obľúbených destinácií pomocou OGM

```
1  používateľ = dopytPoužívateľOGM()
2  destinácie = používateľ->dopytOblubeneOGM()
3  domov = používateľ->dopytDomovOGM()
4
5  výsledok = []
6  FOR každá destinácia z destinácie
7          trasa = domov->dopytTrasaOGM(destinácia)
8          ľudia = destinácia->dopytLudiaOGM()
9
10         výsledok << destinácia, trasa, ľudia
11 ENDFOR
```

4.7 Databázové dopyty pomocou Cypher klienta

Aby sme sa vyhli problému z kapitoly 4.6, rozhodli sme sa implementovať dopyty na väčšie sady dát pomocou jedného dopytu v jazyku Cypher. Na vykonanie priameho dopytu sme použili knižnicu *PHP Neo4J Client*. Táto knižnica vykonáva dopyty v jazyku Cypher a teda nám poskytuje priamy prístup k Cypher API databázy. V príklade 5 je dopyt, ktorý nahradzuje celú funkcionalitu kódu z príkladu 4. Dopyt pozostáva z nasledovných úkonov (vykonávajú sa po riadkoch):

1. Získaj vrchol typu AppUser s ID = 1 a ulož ho do premennej user. Ďalej získaj všetky vrcholy, ku ktorým z vrcholu user vedie hrana typu FOLLOWS a ulož ich do premennej dest.

2. Získaj vrchol, ku ktorému vedie hrana typu home od vrchola user a ulož ho do premennej home.

3. Získaj všetky hrany, ktoré vedú od vrchola home do jednotlivých vrcholov v premennej dest.

4. Získaj všetky vrcholy, od ktorých vedie hrana typu FOLLOWS ku všetkým vrcholom v premennej dest. Keďže aj vrchol user má hranu FOLLOWS do každého vrcholu v dest, vynechaj vrchol user z výberu.

5. Definuj formát výstupu.

Po vykonaní dopytu sa nám vráti výstup vo forme tabuľky obdobnej, ako pri vykonaní JOIN príkazu v relačnej databáze. Keďže sme na jeden dopyt získali všetky informácie, ktoré potrebujeme na zobrazenie tabuľky, nepotrebujeme vykonávať už žiadne ďalšie dopyty. Tento prístup znížil priemernú latenciu nášho API na 37% (695,7ms) oproti prístupu pomocou OGM z kapitoly 4.6

Algoritmus 5 Dopyt na dáta pre zoznam obľúbených destinácií v jazyku Cypher

```
1 MATCH (user:AppUser)-[:FOLLOWS]->(dest) WHERE id(user)=1
2 MATCH (user)-[:HOME]->(home)
3 MATCH (home)-[route:ROUTES]->(dest)
4 MATCH (follower:AppUser)-[:FOLLOWS]->(dest) WHERE follower <> user
5 RETURN dest, route, follower ;
```

4.8 Implementácia optimalizácie trasy

V aplikácií sa nachádza obrazovka, na ktorej si môže používateľ vybrať destinácie, pre ktoré by chcel vyrátať optimálnu trasu vzhľadom na cenu. Táto trasa bude začínať v domovskej destinácii používateľa, prejde všetkými vybranými destináciami práve raz a vráti sa späť do domovskej destinácie. Toto je klasický prípad NP-úplného problému pocestného obchodníka (ďalej TSP).

Pre jednoduchosť a spoľahlivosť sme implementovali prístup rekurzívneho prehľadávania všetkých možnosti trás. Implementovali sme rekurzívnu funkciu v PHP, ktorá prejde všetkými možnými trasami medzi našimi destináciami a vyberie prechod grafom, ktorý je optimálny vzhľadom na cenu celkovej trasy. Do argumentu berie naša funkcia domovskú destináciu, zoznam destinácií, ktoré ešte neprehľadala a domovskú destináciu, do ktorej sa má nakoniec vrátiť. V každom vnorení sa zoznam neprehľadaných vrcholov zmenšuje. Vyššie spomenuté argumenty sú perzistentné objekty resp. polia inštancií perzistentnej triedy *Place*.

Keď sa táto funkcia dopytuje na cenu trasy vedúcej zo začiatočného vrchola do cieľového, zavolá funkciu, ktorá dá dopyt na našu databázu. Ak sa táto trasa nenachádza v databáze, vypýta si údaje o konkrétnej trase od API Rome2Rio. Keďže každý dopyt na API trvá cca 200ms, prvé volanie pri väčšej trase trvá niekoľkonásobne dlhšie ako každé nasledujúce volanie.

Aj po načítaní všetkých potrebných trás bolo naše riešenie príliš pomalé. Už pri siedmych miestach výpočet TSP trval až 7 minút. Toto bolo spôsobené tým, že naša funkcia, ktorá počíta s výpočtovou zložitosťou $O(n!)$, beží na vrstve PHP a pri každom

volaní funkcie invokuje knižnicu NeoEloquent, a dopytuje sa na hranu medzi dvoma destináciami. Keď funkcia v takomto režime bežala, videli sme, že proces Neo4J konzistentne bral cca 30% z procesorového času. Z tohoto dôvodu sme naimplementovali vyrovnávaciu pamäť na dopyt k databáze a priemerný čas sa znížil na 80 sekúnd.

Naša implementácia front-endu vyžaduje dva dopyty na náš server. Jeden na zobrazenie mapy a druhý na zobrazenie tabuľky s výsledkami. Z tohoto dôvodu sme pridali ešte jeden level vyrovnávacej pamäte na celé volanie funkcie pre rátanie TSP. Taktiež sme upravili implementáciu front-endu tak, aby sa tieto dva dopyty vykonali vždy synchrónne za sebou. Týmto sme znížili nároky na výpočtové kapacity nášho servera.

Dlhý čas výpočtu pri dopytovaní sa na databázu pripisujeme tomu, že naše dopyty na databázu boli diskrétne pri každom vnorení funkcie. Aj keď sme používali perzistentné objekty, implementácia každého dopytu NeoEloquent pozostáva z jedného dopytu v jazyku Cypher na databázu. Toto volanie obsahuje vyhľadanie začiatočného a konečného bodu podľa id, so zložitosťou $O(log(n))$, a následne vyhľadanie hrany, so zložitosťou $O(1)$. Tento prístup nevyužíva hlavnú výhodu grafovej databázy, no s použitím NeoEloquent nie je možné postaviť komplexný dopyt, ktorý by vedel vyriešiť TSP za jeden beh. Takýto dopyt je v Neo4J možné implementovať pomocou dopytovacieho jazyka Cypher. Príklad takéhoto dopytu môžeme vidieť v algoritme č.6. Tento dopyt sme však neimplementovali, kvôli jeho komplexnosti a faktu, že oproti našej funkcii v PHP neposkytuje lepšiu teoretickú výpočtovú zložitosť.

Algoritmus 6 Ukážka riešenia TSP pomocou dopytovacieho jazyka Cypher

```
1  MATCH (from:Node {name: "Source node" })
2  MATCH path = (from)-[:CONNECTED_TO*6]->()
3          WHERE ALL(n in nodes(path) WHERE 1 = length(filter(m in
4          nodes(path) WHERE m = n)))
5  AND length(nodes(path)) = 7
6  RETURN path,
7          reduce(distance = 0, edge in relationships(path) |
8                  distance + edge.distance)
9  AS totalDistance
10 ORDER BY totalDistance ASC
11 LIMIT 1
```

4.9 Rome2rio API

Všetky údaje o destináciách a trasách berieme z API Rome2Rio. Pomocou Autocomplete API umožňujeme používateľovi pridávať destinácie. Používateľ zadáva písmená do autocomplete textového poľa na stránke. Toto pole posiela dopyty na Autocomplete API a ono vracia pole objektov s miestami. Tieto objekty obsahujú informácie o type destinácie(obec, mesto, región, štát, letisko,...), geografickú polohu, názov v dlhom a krátkom tvare, a kanonický názov. Používateľ si potom vyberie jednu z destinácií a príslušný objekt sa zašle na náš server. Na jednoznačnú identifikáciu objektu používame kanonický názov, ktorý je podľa dokumentácie unikátnym identifikátorom miesta.

Ak destináciu ešte nemáme v databáze, pridáme tento objekt do databázy. Toto riešenie nie je úplne ideálne z bezpečnostného hľadiska, lebo umožňuje zaslanie falošného miesta do našej databázy. Ak by útočník vyrobil objekt s reálnym kanonickým názvom no falošnými údajmi napr. o zemepisnej šírke a dĺžke, toto miesto by sa potom nesprávne zobrazovalo všetkým používateľom. Na vyriešenie tohoto problému by postačilo urobiť ešte jeden dopyt z nášho servera na autocomplete API, ktorým by sme si len vypýtali údaje k miestu za pomoci kanonického názvu.

Ďalšie údaje berieme z Rome2Rio Search API. Sú to údaje o možných trasách a ich cenách. Toto API je obmedzené na počet volaní. Preto sme na volanie tohoto API implementovali vyrovnávaciu pamäť. Vždy keď voláme Search API, voláme ho na dve miesta ,ktoré už máme uložené v našej databáze ako vrcholy. Tento fakt sme využili a vytvorili sme ďalší typ hrany - CACHE. Keďže pri mestách, ktoré sú dopravné uzly vystúpila veľkosť odpovede API až na rádovo 500kb a tento typ dopytu sa nerobí veľmi často, rozhodli sme sa odpoveď servera neukladať priamo do databázy, ale v nezmenenej podobe na disk, a do databázy uložiť len vek súboru, a referenciu na súbor na disku. Keďže v aktuálnej podobe nevyužívame celú odpoveď tohoto API, mohli by sme optimalizovať využitie miesta na disku tým, že by sme najprv údaje zapracovali a uložili len tie, ktoré využívame.

4.10 Vykresľovanie máp

Na vizualizáciu destinácií a trás sme na rôznych miestach a aplikácií použili javascriptovú knižnicu Mapbox.js.

Keďže Mapbox.js rozširuje knižnicu Leaflet, všetky funkcie knižnice sa volajú z globálneho objektu *L*. Táto funkcia sa zavolá po načítaní stránky. Keďže sa mapy Mapbox.js sťahujú zo serverov Mapbox, musíme najprv aplikáciu identifikovať API kľúčom, ktorý sme si vygenerovali po registrácii. Následne inicializujeme samotnú mapu volaním funkcie

L.mapbox.map. Táto funkcia berie, ako prvý parameter id HTML elementu, do ktorého sa má mapa zobraziť, a ako druhý parameter berie textový identifikátor typu mapy, ktorý chceme zobraziť. Po inicializácii sa vytvorí vrstva pre mapu, ktorá bude obsahovať označenia destinácií. Dáta, ktoré obsahujú geografickú polohu destinácií sa vyžiadajú vo formáte GeoJson z API našej aplikácie. Keďže pri malom priblížení mapy by sa nedali zreteľne rozlíšiť destinácie, ktoré sú blízko seba, po načítaní dát vytvoríme vrstvu zoskupení pomocou funkcie *L.MarkerClusterGroup*. Táto vrstva spojí blízke body do zoskupení a následne ukryje značky týchto bodov, a nahradí ich značkou zoskupenia. Pri pridávaní bodov do zoskupení sa iteruje cez všetky body, ktoré sú po načítaní v mape a každý sa pridá do vrstvy zoskupení. Následne sa vrstva zoskupení pridá do objektu mapy. Výsledok môžeme vidieť na obrázku 9. Ako zdrojový formát dát pre všetky mapy v našej aplikácii používame štandardný formát GeoJson. GeoJson je dátová štruktúra zakódovaná do formátu JSON obsahujúca informácie potrebné na vykresľovanie kartografických veličín. V našej implementácii používame nasledovnú štruktúru GeoJson objektu. Hlavný objekt GeoJson obsahuje typ, v našom prípade *FeatureCollection* a pole objektov typu *Feature*. Každý z týchto objektov obsahuje pole súradníc a objekt dodatočných atribútov slúžiacich na upresnenie vizuálu vykresleného objektu. Dané pole súradníc nám spôsobovalo mierne nedorozumenia, lebo v číslovanom poli je, ako prvý prvok uložená zemepisná dĺžka (longitude), a ako druhý prvok zemepisná šírka (latitude) [4], čo je presne opačne, ako všetky ostatné API, ktoré sme používali.

4.11 Vykresľovanie tabuliek

Aby sme zlepšili UX stránky rozhodli sme sa na vykresľovanie dynamického obsahu využiť namiesto HTML obsahu renderovaného na serveri, javascriptovú knižnicu na prácu so šablónami a dynamický obsah načítavame z API našej aplikácie vo formáte JSON. Na vykresľovanie dynamického obsahu sme si zvolili knižnicu Handlebars.js.

Najprv je zadeklarovaná premenná v *template*, ktorá bude neskôr obsahovať funkciu renderujúcu šablónu. Potom je zadeklarovaná funkcia *refreshPage*, ktorá bude volaná vždy, keď nejaká funkcia vyvolá udalosť s menom *appRefresh*. Táto funkcia využije AJAX API knižnice jQuery a stiahne potrebné dáta. Potom využije funkciu *template*, ktorá do argumentu zoberie dáta vo forme poľa objektov a vráti vygenerované HTML, ktoré sa následne vkladá na stránku.

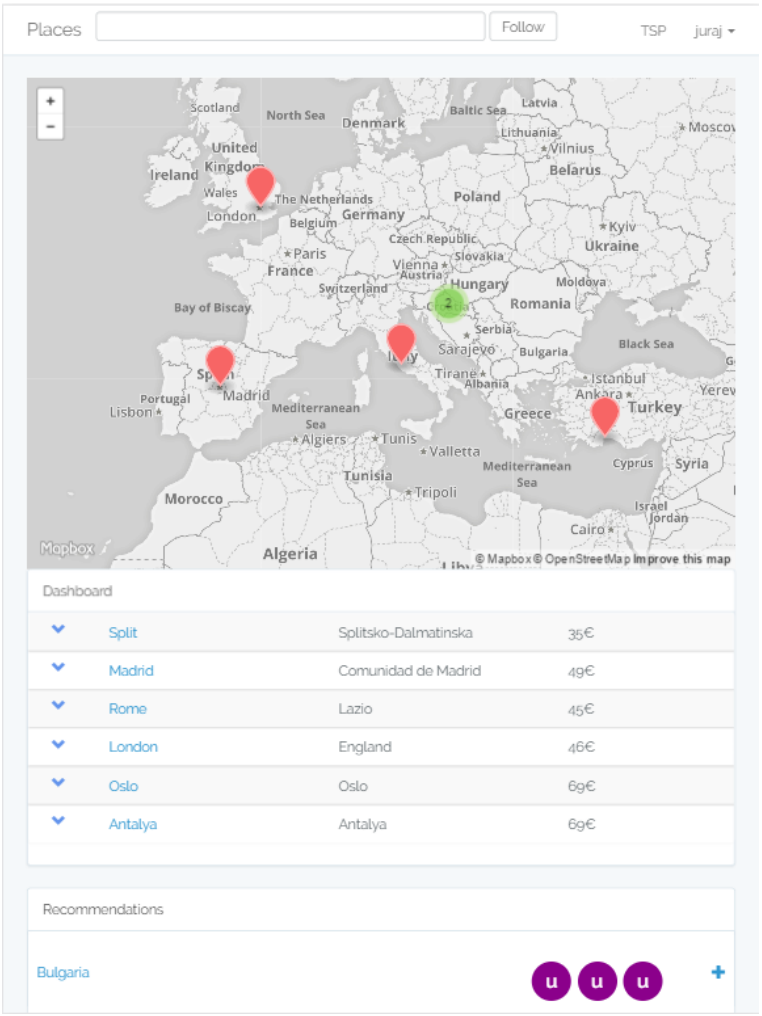
V druhej časti kódu sa najprv volá funkcia *Handlebars.compile* a do argumentu zoberie šablónu, ktorá je uložená v elemente s id *places-template*. Ako návratovú hodnotu vráti funkciu, ktorú uložíme pod menom *template*. Následne sa na udalosť *appRefresh*

naviaže volanie funkcie *refreshPage* a prvýkrát sa zavolá táto udalosť.

Naša šablóna *places-template* obsahuje aj aktívne linky a v nasledujúcom bloku je príklad implementácie vymazania miesta z obľúbených. Na všeobecnú udalosť *click* je naviazaná filtrovaná udalosť, ktorá spustí AJAX dopyt na vymazanie miesta z obľúbených, ak element, ktorý udalosť vyvolal obsahuje triedu *delete* a atribút *data-id*.

4.12 Používateľské rozhranie

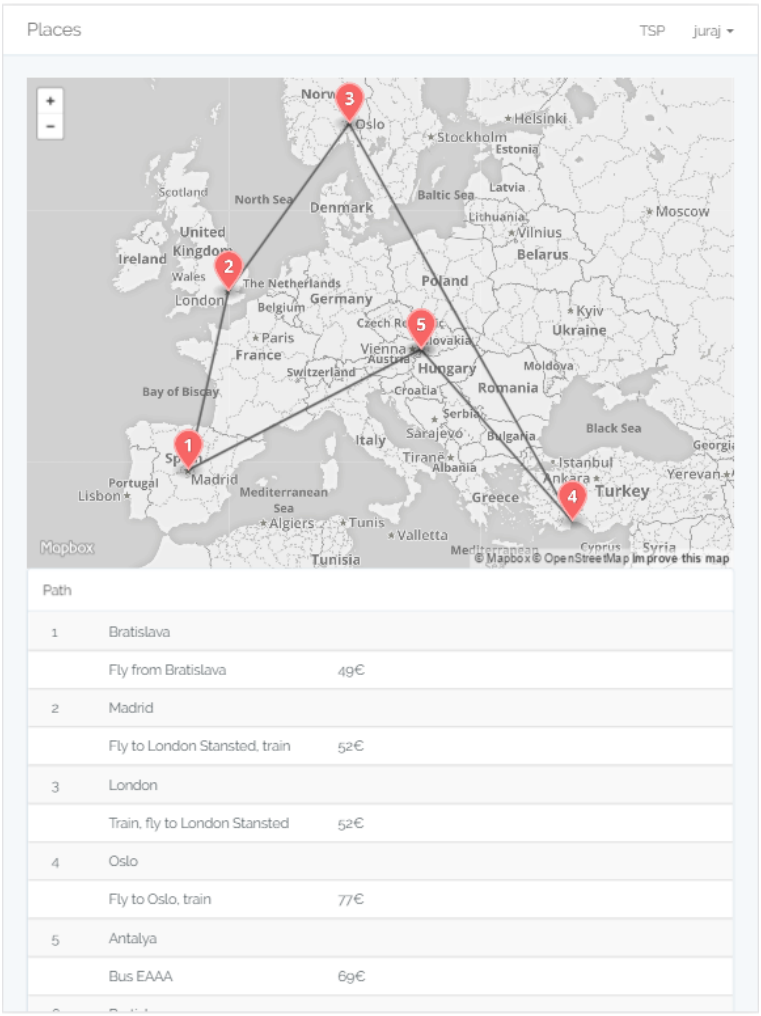
Používateľské rozhranie je implementované pomocou HTML, CSS a JavaScript. Použili sme CSS framework Bootstrap pre zabezpečenie responzivity a konzistentného výzoru, a UX aplikácie. Vo vrchnej časti rozhrania sa nachádza navigačný panel.



Obrázok 9: Ukážka hlavnej obrazovky používateľa

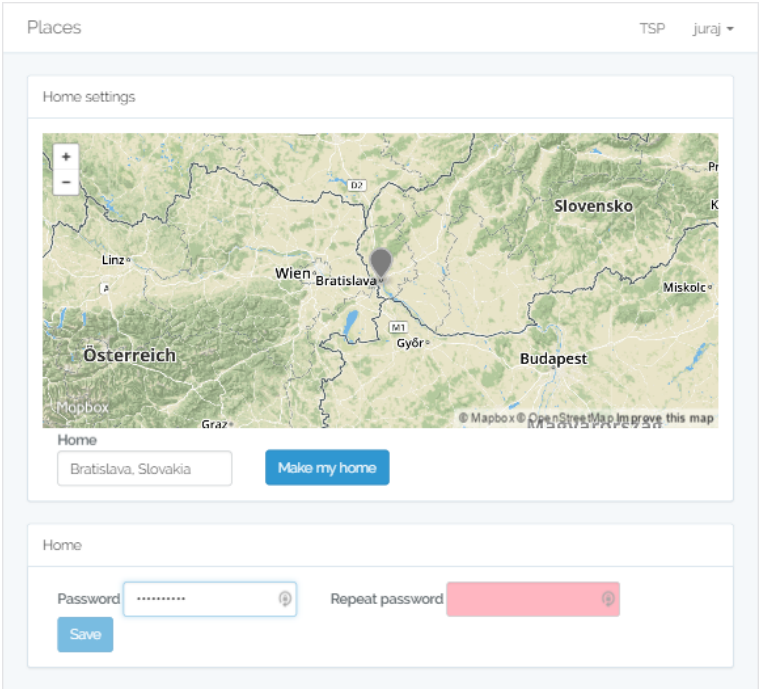
Na hlavnej obrazovke (obrázok č.9) je v navigačnom paneli pridané textové pole na pridávanie destinácií na zoznam obľúbených. V hlavnej časti aplikácie sa nachádza mapa s

vyobrazenými destináciami zo zoznamu, ďalej tabuľka s destináciami, kde si môže používateľ rozbaliť detaily konkrétnej destinácie alebo ju zmazať a tabuľka s odporúčanými destináciami, ktoré si môže používateľ pridať do svojho zoznamu.



Obrázok 10: Ukážka obrazovky TSP

Na obrazovke TSP (obrázok č.10) sa nachádza mapa s reprezentáciou optimálnej trasy. Na mape sú značky pospájané priamkami, očíslované podľa poradia, v akom ich má používateľ navštíviť na dosiahnutie optimálnych nákladov. Nižšie je tabuľka s podrobnejším opisom trasy s vymenovanými spôsobmi dopravy. V spodnej časti stránky je zoznam obľúbených destinácií, z ktorých môže destinácie pridávať a odoberať zo zoznamu na optimalizáciu.



Obrázok 11: Ukážka obrazovky Nastavenia

Na obrazovke nastavení sa nachádza mapa, ktorá vyobrazuje aktuálne domáce miesto používateľa. Pod ňou sa nachádza textové pole, pomocou ktorého môže používateľ vyhľadať a nastaviť domáce miesto. V spodnej časti obrazovky nastavení je sekcia, kde si môže používateľ zmeniť heslo.

5 Testovanie

Všetky testy sme vykonávali na našom serveri s Ubuntu 16.04.2 LTS s jedným virtuálnym jadrom a 4GB RAM. Testovali sme na databáze Neo4j Community verzia 3.1.4 a na databáze MYSQL verzia 5.7.18. Tabuľky boli uložené pomocou InnoDB. Pri testovaní rýchlosti naša aplikácia bežala na serveri Apache verzia 2.4.18 s interpreterom PHP verzia 7.0.15.

Ako sadu dát sme použili reálne dáta o destináciách a trasách z Rome2Rio API. Aby sme zabezpečili zvyšovanie, ako veľkosti, tak aj hustoty grafu, začali sme s jedným používateľom s desiatimi položkami. V druhom kroku sme pridali používateľa s päťdesiatimi destináciami, ktorý mal zároveň pridané všetky destinácie prvého používateľa a takto sme pokračovali až po počet destinácií 1400. Dáta sme generovali našou aplikáciou a následne migrovali do MYSQL databázy štruktúrovanou, ako v našom návrhu opísaného v kapitole 3.2.

5.1 Latencia databázy Neo4j a MySQL

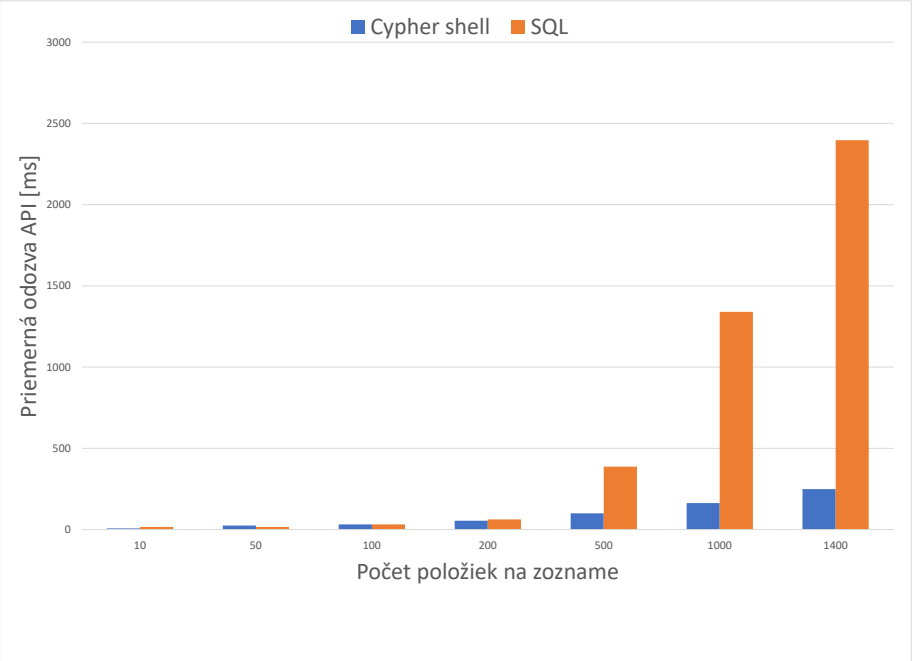
Prvý test, ktorý sme robili bol syntetický test rýchlosti komplexného dopytu na databázu. Cieľom testu bolo porovnať škálovateľnosť relačnej a grafovej databázy v prípadoch podobných nášmu. Dopyt, ktorého latenciu sme testovali bol na zozname obľúbených destinácií. Ku každej destinácii sme vyhľadali ešte trasu, ktorá k nej vedie z domovskej destinácie používateľa a zoznam ostatných používateľov, ktorý tiež majú túto destináciu na zozname. Dopyt v databáze Neo4J môžeme vidieť na algoritme 5. Dopyt v databáze MYSQL môžeme vidieť na algoritme 7.

Algoritmus 7 SQL dopyt pre testovanie rýchlosti

```
1 SELECT * FROM `follows` f
2 LEFT JOIN `place` p ON f.`id_place` = p.`id`
3 LEFT JOIN `routes` r ON r.`id_from` = u.`id_home`
4 AND r.`id_to` = p.`id`
5 LEFT JOIN `follows` fo ON fo.`id_place` = p.`id`
6 LEFT JOIN `users` u ON u.id = fo.`id_user`
7 LEFT JOIN `tsp` t ON t.`id_user` = f.`id_user`
8 AND t.`id_place` = f.`id_place`
9 WHERE f.`id_user` = 231
```

Tabuľka 1: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

Počet položiek	10	50	100	200	500	1000	1400
Cypher shell	6,7	25,2	32	54	99,8	162,7	248,9
SQL	15,5	16,1	31,6	62,5	387,3	1339,8	2396,6



Obrázok 12: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

Na výsledkoch testov v tabuľke vidíme, že pri zväčšovaní a zahusťovaní grafu latencia databázy MySQL rastie oveľa rýchlejšie ako Neo4j. Pri 1400 záznamoch vystúpala priemerná latencia až na 2,3966 sekundy. Tento rýchly nárast latencie MySQL prisudzujeme rastu veľkosti spájacích tabuliek, ktorých prehľadávanie je hlavnou príčinou spomalenia relačnej databázy pri prehľadávaní sady dát so zložitou štruktúrou s mnohými vzťahmi. Na rozdiel od toho databáza Neo4J dosahovala pri rovnakej dátovej sade priemernú latenciu 248,9ms. Tieto výsledky ukazujú vhodnosť použitia grafovej databázy oproti relačnej pre prípad použitia, kde má dátová sada štruktúru grafu.

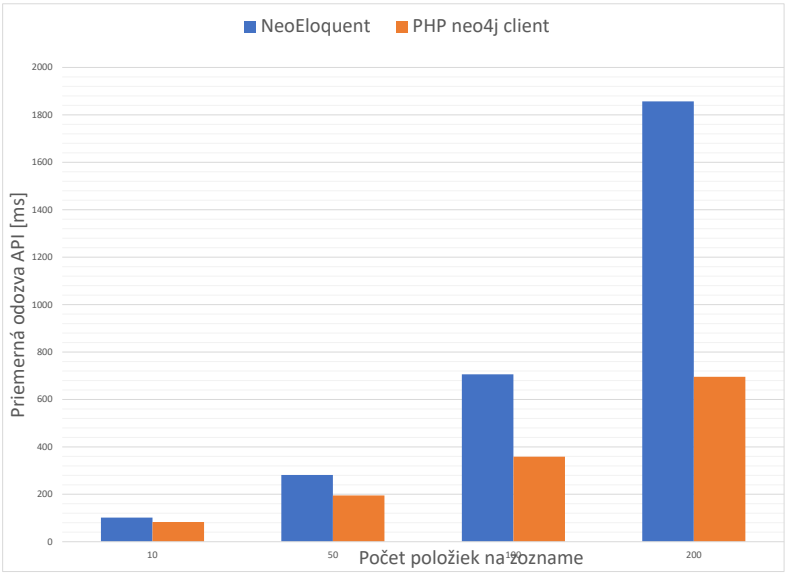
5.2 Latencia rôznych prístupov k databáze Neo4J

Druhý test, ktorý sme robili bol reálny test latencie nami implementovaného API v dvoch verziách. V prvej verzii s použitím OGM a v druhej s použitím PHP Neo4J Client. Tento test sme vykonali, ako súčasť implementácie, aby sme vedeli posúdiť, ktoré riešenie použijeme vo finálnej verzii aplikácie. Pri teste sme používali rovnakú dátovú sadu, ako v prvom teste. Latenciu sme merali tak, že sme posielali dopyty na naše API pre vykreslenie zoznamu obľúbených destinácií a zaznamenávali sme čas od doručenia dopytu na server do začiatku sťahovania dát. Test sme opakovali desať krát.

V tabuľke č.2 vidíme priemernú latenciu nášho API v závislosti od počtu položiek na zozname. Dáta sú ďalej vizualizované v grafe na obrázku č.13. Z výsledku nášho testu sme usúdili, že prístup pomocou OGM nebol schopný využiť plný potenciál našej grafovej databázy a rozhodli sme sa vo finálnej verzii zmeniť implementáciu na prístup pomocou PHP Neo4j Client.

Tabuľka 2: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

Počet položiek	10	50	100	200
NeoEloquent	101,6	281,7	705,7	1857
PHP neo4j client	83,2	195	358,5	695,7



Obrázok 13: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

6 Zhodnotenie

V tejto kapitole sa budeme venovať zhodnoteniu výsledkov našej implementácie a celkovému zhrnutiu výsledkov nášho testovania.

6.1 Zhodnotenie implementácie

Ako súčasť práce sme implementovali aplikáciu na plánovanie a optimalizáciu trasy cestovateľa. Aplikácia obsahuje registráciu a prihlásenie používateľa. Po prihlásení si používateľ môže vyhľadať a pridať destinácie na svoj zoznam obľúbených. Pri zobrazovaní zoznamu aplikácia poskytuje informácie o trasách vedúcich z konkrétnych destinácií do konkrétnych destinácií vrátane informácií o cene a spôsobe dopravy, a informáciu o tom, akí ďalší používatelia majú tiež destináciu na zozname. Ďalej aplikácia na základe už pridaných destinácií zobrazuje používateľovi zoznam odporúčaných destinácií. Všetky tieto údaje sú dopytované pomocou rýchlejšieho prístupu k databáze pomocou PHP Neo4J Client.

Ďalej je v aplikácii obrazovka na optimalizáciu trasy. Na tejto obrazovke si používateľ zo svojho zoznamu môže vybrať niektoré destinácie a aplikácia mu vráti poradie, v akom by mal používateľ navštíviť zvolené destinácie pre optimalizovanie nákladov na cestu. Pre nedostatok času a zložitosť tohoto výpočtu sme implementáciu optimalizácie nechali v staršej verzii – s použitím prístupu k databáze pomocou OGM. Aby sme zlepšili UX používateľa, implementovali sme niekoľko vrstiev vyrovnávacej pamäte, aby sa výsledky zobrazovali, čo najskôr.

6.2 Zhodnotenie testovania

Po fáze implementácie sme testovali latenciu našich dopytov na databázu. Testy sme porovnávali s obdobnými testami na rovnakých dátach v databáze MySQL. Pri testoch sme dokázali, že grafová databáza má aj v našom reálnom príklade merateľne vyšší výkon pri práci s grafovou sadou dát veľkosti niekoľko sto vrcholov a hrán. Potvrdili sme teda lepšiu škálovateľnosť grafovej databázy.

6.3 Zhodnotenie skúseností s vývojom pomocou grafovej databázy

Natívna štruktúra použitej grafovej databázy prirodzene poňala našu sadu dát bez potreby dáta trasnformovať či rozdelovať do tabuliek. Flexibilita bezshémovej grafovej databázy nám oproti relačným databázam poskytla možnosť rýchlo iterovať, testovať variácie v štruktúre našej databázy.

Grafová databáza Neo4J ďalej poskytuje jednoduché a pritom silné nástroje na prácu s grafmi. Nástroje, ako jednoduchý a silný dopytovací jazyk Cypher, a používateľské rozhranie databázy s prehľadnými interaktívnymi vizualizáciami dát pomohli zmierniť krivku učenia spojenú s novým typom databázy.

Záver

V našej práci sme sa venovali problematike využitia grafových databáz, a aké výhody prináša ich použitie v praxi.

Najprv sme priblížili problematiku grafových databáz a popísali ich odlišnosti oproti iným typom z hľadiska práce so vzťahmi. Následne sme sa venovali predstaveniu troch grafových databáz, zhodnotili sme ich výhody a nevýhody, a pre ďalšiu prácu sme vybrali databázu Neo4J.

Následne sme špecifikovali a navrhli aplikáciu, pre plánovanie a optimalizáciu trasy cestovateľa. Potom sme pristúpili k implementácii. Aplikáciu sme implementovali na platforme PHP frameworku Laravel s využitím vybranej grafovej databázy Neo4J. V priebehu implementácie sme zistili, že sme použili neefektívny spôsob prístupu ku grafovej databáze, a preto sme aplikáciu čiastočne prerobili tak, aby sa najnáročnejšie operácie vykonávali efektívnejšie. Aplikácia obsahuje celú špecifikovanú funkcionalitu, avšak funkciu optimalizácie trasy sme už kvôli zložitosti tohoto problému nestihli prerobiť, aby používala efektívnejší prístup do databázy.

Po implementácii sme vykonali testy latencie ekvivalentných dopytov v grafovej a relačnej databáze na našich reálnych dátach. Z výsledkov sme potvrdili prevahu vo výkone databázy Neo4J oproti relačnej databáze MySQL.

Nakoniec sme, ako hlavné výhody použitia grafovej databázy Neo4J vybrali: Rýchlosť v prehľadávaní grafu oproti relačnej databáze, vhodnosť dátového modelu grafovej databázy pre našu sadu dát a vhodnosť nástrojov databázy Neo4J na využitie pri implementácii, a testovaní našej aplikácie.

Zoznam použitej literatúry

[1] AT, S. I. Db-engines ranking - popularity ranking of graph dbms, 2017. https://db-engines.com/en/ranking/graph+dbms [Online; prístupné dňa 14. 05. 2017].

[2] Codd, E. F. A relational model of data for large shared data banks, 1970. http://dl.acm.org/citation.cfm?id=362685 [Online; prístupné dňa 14. 05. 2017].

[3] Ian Robinson, Jim Webber, Emil Eifrem. *Graph Databases*. 2. O'Reilly Media, Inc. USA, 2015. s. 224. ISBN: 9781491932001.

[4] (IETF), I. E. T. F. The geojson format, 2016. https://tools.ietf.org/html/rfc7946 [Online; prístupné dňa 14. 05. 2017].

[5] Myiuru Dayarathana, T. S. A relational model of data for large shared data banks, 2013. https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnx0b2t5b3RlY2hzdXp1bXVyYWxhYmVuZ3xneDoyMGRiOGFlM2Y2OGY5Mzhj [Online; prístupné dňa 14. 05. 2017].

[6] neotechnology. Neo4j debian packages, 2017. https://debian.neo4j.org [Online; prístupné dňa 14. 05. 2017].

[7] Otwell, T. Laravel - the php framework for web artisans, 2017. https://laravel.com/docs/5.0 [Online; prístupné dňa 14. 05. 2017].

Prílohy

A Štruktúra elektronického nosiča

\
\Bakalarska_praca.pdf
\zdrojovy_kod
\zaloha_databazy