

**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



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

SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
Fakulta elektrotechniky a informatiky
Ústav informatiky a matematiky
Iľkovičova 3, 812 19 Bratislava



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

Práca sa zaoberá problematikou grafových databáz a výhod jej využitia v praxi. V prvej časti práce sa nachádza analýza

Kľúčové slová: grafová databáza, OGM, Cypher, Neo4j, performance, škálovateľnosť

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

abstractEN

Keywords: graph database, OGM, Cypher, Neo4j, performance, scalability

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 17.5.2017

.....
podpis autora

Podakovanie

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

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

Obsah

Úvod	1
1 Analýza problému	2
1.1 Grafové databázy	2
1.2 Vzťahy v relačných databázach	3
1.3 Vzťahy v dokumentových databázach	3
1.4 Výber grafovej databázy	4
1.4.1 Neo4J	5
1.4.2 OrientDB	5
1.4.3 Titan	6
1.4.4 Vyhodnotenie	6
1.5 Dopytovací jazyk Cypher	6
2 Špecifikácia	8
2.1 Funkcionálne požiadavky	8
2.2 Nefunkcionálne požiadavky	9
3 Návrh	10
3.1 Prípady použitia	10
3.2 Štruktúra databázy	11
3.3 Štruktúra aplikácie	12
4 Implementácia	13
4.1 Použité technológie	13
4.1.1 Laravel framework	13
4.1.2 NeoEloquent OGM	14
4.1.3 GraphAware Neo4j PHP Client	14
4.1.4 Mapbox.js	14
4.1.5 Handlebars.js	15
4.1.6 Rome2Rio Api	15
4.2 Inštalácia a konfigurácia Laravel Framework-u	16
4.3 Inštalácia a konfigurácia databázy Neo4J	16
4.4 Implementácia OGM	16
4.5 Autentifikácia	17
4.6 Databázové dopyty pomocou OGM	19

4.7	Databázové dopyty pomocou Cypher klienta	20
4.8	Implementácia optimalizácie trasy	21
4.9	Rome2rio API	23
4.10	Vykresľovanie máp	23
4.11	Vykresľovanie tabuliek	24
4.12	Používateľské rozhranie	25
5	Testovanie	27
5.1	Latencia databázy Neo4j a MySQL	27
5.2	Latencia rôznych prístupov k databáze Neo4J	28
6	Zhodnotenie	30
6.1	Zhodnotenie implementácie	30
6.2	Zhodnotenie testovania	30
6.3	Zhodnotenie skúseností s vývojom pomocou grafovej databázy	30
	Záver	32
	Zoznam použitej literatúry	33
	Prílohy	I

Zoznam obrázkov a tabuliek

Obrázok 1	Ukážka riešenia vzťahov v relačnej databáze [6, 12]	4
Obrázok 2	Ukážka riešenia vzťahov v dokumentovej databáze [6, 15]	4
Obrázok 3	Prípad použitia - Registrácia, autentifikácia a nastavenia	10
Obrázok 4	Prípad použitia - Registrácia, autentifikácia a nastavenia	11
Obrázok 5	Prípad použitia - dashboard	12
Obrázok 6	Prípad použitia - tsp	13
Obrázok 7	Štruktúra databázy	14
Obrázok 8	Príklad dát v databáze	15
Obrázok 9	Ukážka hlavnej obrazovky používateľa	25
Obrázok 10	Ukážka obrazovky TSP	26
Obrázok 11	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií	28
Obrázok 12	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií	29
Tabuľka 1	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií	28
Tabuľka 2	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií	29

Zoznam skratiek a značiek

TSP - problém pocestného predajcu (Travelling Salesman Problem)

UML - Unified Modeling Language verzie 2.5

ORM - Mapovanie objektov na relácie (Object to Relational Mapping)

OGM - Mapovanie objektov na graf (Object to Graph Mapping)

UX - (User Experience)

AJAX - Asynchrónny JavaScript dopyt (Asynchronous Javascript and XML)

MVC - Model, Zobazenie, Ovládač (Model View Controller)

Zoznam algoritmov

1	Príklady dopytov v jazyku Cypher	7
2	Ukážka triedy neoEoquent	18
3	Ukážka autentifikovateľnej triedy	19
4	Pseudokód algoritmu na získavanie dát pre zoznam obľúbených destinácií pomocou OGM	20
5	Dopyt na dáta pre zoznam obľúbených destinácií v jazyku Cypher	21
6	Ukážka riešenia TSP pomocou dopytovacieho jazyka Cypher	22
7	SQL dotaz pre testovanie rýchlosti	27

Úvod

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

Grafové databázy sú natívne prispôsobené práci s grafmi, neukladajú údaje v tabuľkách ale vo vrcholoch a hranách. Tento spôsob uloženia dát implicitne poskytuje výhody pri práci sa dátovými sadami podobnými grafu.

V našej práci sa oboznámime s hlavnými predstaviteľmi grafových databáz. Vyberieme jedného z predstaviteľov vyberieme, navrhujeme a budeme implementovať aplikáciu, ktorá bude vybrať databázu používať. Na koniec 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 rieši vzťahy medzi dátami. Ďalej si priblížime ako riešia vzťahy relačné a dokumentové databázy. Následne si predstavíme troch najpopulárnejších predstaviteľov grafovej databázy a vyberieme z nich jednu s ktorou budem ďalej pracovať. Na koniec 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 vhodná na takúto reprezentáciu, lebo to, čo namodelujeme na papier vieme priamo implementovať v našej grafovej databáze. [6, 25–27]

Grafové databázy využívajú model ktorý pozostáva z vrcholov, hrán, 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, východzí vrchol, cieľový vrchol a cieľový vrchol. Fakt, že hrany musia mať smer a názov pridáva to sémantickú prehľadnosť do grafu, ak zvolíme správne názov vieme rýchlo identifikovať štruktúru grafu a identifikovať význam vzťahov. Hrany môžu rovnako ako vrcholy obsahovať aj atribúty. Atribúty v hranách môžu byť praktické na pridanie kvalitatívnych dát (napr. váha, vzdialenosť) ku vzťahom, tieto dáta sa potom môžu použiť pri prehľadávaní grafu.

Natívne grafové databázy používajú bez indexovú priliehlosť, toto znamená, že prepojenia sa riešia pomocou ukazovateľov a nie cudzích kľúčov ako v relačných databázach. Absencia cudzích kľúčov praxi znamená, že pri prehľadávaní vzťahov v 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. Takisto vo vrcholoch sú uložené priame ukazovatele na všetky hrany vychádzajúce z a mieriace do dotyčného vrcholu. Takáto štruktúra poskytuje už spomínanú výpočtovú zložitosť $O(1)$ v oboch smeroch hrany, takže nielen v smere z východzieho bodu do cieľového ale aj opačným smerom.[6, 149–158] Pri relačnej databáze by toto muselo byť riešené reverzným vyhľadávaním v cudzích kľúčoch.

1.2 Vzťahy v relačných databázach

Relačná databáza je databáza, v ktorej sú údaje uložené podľa relačného databázového modelu podľa E. F. Codd z roku 1970. Relačný model je založený na matematickom aparáte relačných množí na predikátovej logiky. Databázová relácia sa od matematickej líši tým, že využíva pomocný aparát nazvaný schéma. V databáze schéma definuje názov relácie (tabuľky) a koľko obsahuje stĺpcová aký dátový typ obsahujú.[5]

Vývojári databáz sa už desaťročia snažia relačné databázy prispôbiť na to aby boli schopné efektívne pracovať s husto prepojenými dátovými sadami. Keďže však štruktúra relačných databáz bola pôvodne navrhnutá na ukladanie formulárov a tabuľkových štruktúr, majú problém s keď sa snažia poňať ad-hoc vzťahy, ktoré sa vyskytujú v dátach v reálnom svete. [6, 11–12]

Vzťahy sa vyskytujú v relačných databázach no iba vo fáze návrhu, v reálnej implementácii tieto vzťahy nahrádzajú spájacie tabuľky a cudzie kľúče. Týmto prístupom rýchlo narastá komplexita a znižuje sa prehľadnosť dátového modelu a pridáva sa záťaž na databázu vznikom veľkých spájacích tabuliek a stĺpcov s cudzími kľúčmi, ktoré môžu byť riedko zaplnené, no zaberajú diskový priestor. [6, 11–12]

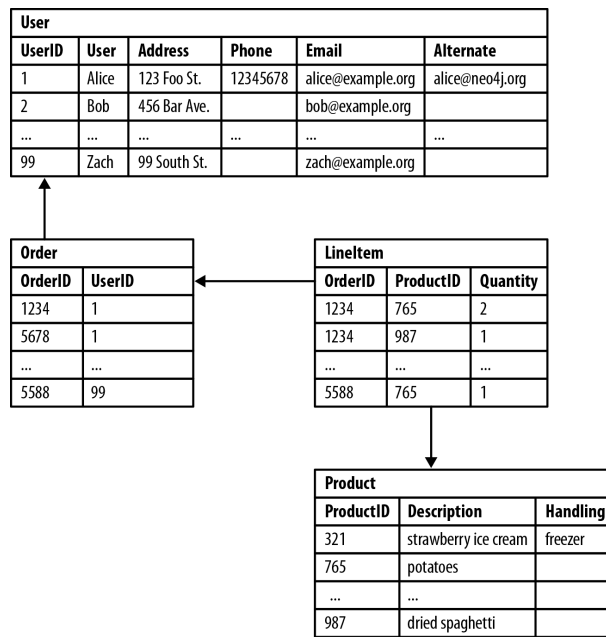
Pri prehľadávaní vzťahov v relačnej databáze, každé prepojenie pridáva výpočtovú komplexitu. V každej ďalšej prepojenej tabuľke treba vyhľadať záznam s požadovaným kľúčom $O(\log(n))$. Používané relačné databázové systémy riešia tento výpočtový problém použitím indexov a inými optimalizáciami, no pokiaľ je dátová sada štruktúrovaná s veľkým množstvom vzťahov systém sa spravidla spomalí.

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

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 kolekcí podľa ich významu v databáze. Štruktúra grafovej databázy je primárne určená na dopyty podľa kľúča a neposkytuje výhodu pri prehľadávaní cez vzťahy oproti relačným databázam. Rôzne implementácie dokumentových databáz implementujú odlišné prístupy k riešeniu vzťahov.

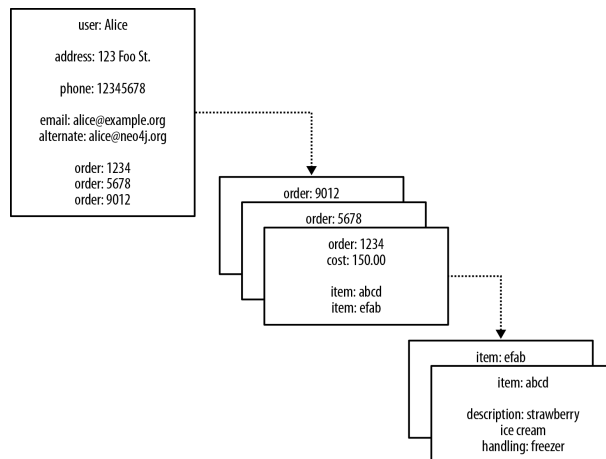
Bežný spôsob na pridanie vzťahov do štruktúry dokumentovej databázy je pridanie atribútu do dokumentu, ktorý bude odkazovať na kľúč a kolekciu dokumentu, s ktorým chceme vytvoriť vzťah. Týmto spôsobom vieme použiť koncept cudzích kľúčov z relačných databáz. Tento prístup sa však stretá s rovnakými problémami v rýchlosti ako



Obrázok 1: Ukážka riešenia vzťahov v relačnej databáze [6, 12]

pri relačných databázach s pridanou komplexitou v tom, že tento dopyt musíme často implementovať aplikačne.

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



Obrázok 2: Ukážka riešenia vzťahov v dokumentovej databáze [6, 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 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 shardovanie cache pamäte, rozšírené monitorovanie a zálohovanie za behu.[4] [1]

Medzi hlavné výhody Neo4J patrí:

- Je dlhodobo najpopulárnejšia grafová databáza, podľa rebríčka db-rank[4]. Z toho vyplýva dobrá dokumentácia a podpora na internete a väčšia podpora pre integráciu s populárnymi frameworkami.
- Dopytovací jazyk Cypher, ktorý táto databáza podporuje. Tento jazyk je vizuálne zrozumiteľný a navrhnutý primárne na prácu s 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 nabralo 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 edíciou s podporou migrácie a synchronizácie na Neo4J a pridanými analytickými nástrojmi. [4] [2]

Medzi hlavné výhody OrientDB patrí:

- Podporuje okrem grafového modelu ukladania dát aj dokumentové a key-value úložisko
- Podľa nezávislých testov [8] je v niektorých testoch až desať krát rýchlejšie ako Neo4J.

Nevýhodami OrientDB sú:

- Menšia podpora 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 2012 vyvíjaný skupinou ThinkAurelius, no v roku 2017 bol odkúpený firmou Datstax a projekt Titan bol zastavený. Projekt je ďalej udržiavaný ako open source verzia pod menom JanusGraph. Titan je projekt určený na veľké distribuovane enterprise riešenia, je nasadzovaný na cloudové platformy ako napr. Apache Hadoop a Apache Spark podporuje rôzne distribuované úložné priestory ako napr. Apache HBase a Oracle BerkeleyDB. Ďalej podporuje rôzne vyhľadávacie enginy ako napr. Elasticsearch a Solr. [4] [3]

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. Na koniec 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 tohto 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 a menší na upresnenie smerovania hrany. Medzi hlavný

dopytovací príkaz v Cypher je MATCH, týmto príkazom podobne ako SELECT v SQL začína každý dopyt.

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

Na 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.* /*.

Na poslednom príklade vidíme príkaz, ktorý najprv vyhľadá vrcholy s atribútom *meno* *Patrik* a *Adam* 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 funkcionálnu 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 domáceho miesta, ktoré si používateľ zadá. Do destinácií, ktoré si pridá na zoznam obľúbených destinácií. Bude ďalej používateľovi ukazovať ostatných používateľov, ktorý 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 registrácii sa budú vyžadovať prihlasovacie údaje: e-mail, heslo. Okrem toho sa bude vyžadovať zdanie domácej destinácie.
3. Po prihlásení používateľa sa mu zobrazí obrazovka s mapou, zoznamom obľúbených destinácií, ktoré chce navštíviť a zoznam odporúčaných destinácií.
4. Na mape bude vyobrazená používateľova domáca destinácia a všetky destinácie ktoré má v zozname obľúbených destinácií.
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 domáceho miesta do destinácie nachádza.
6. Budú sa dať zobrazíť 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 aplikácii bude obrazovka kde si bude používateľ vybrať niekoľko zo svojich obľúbených destinácií a nechať si vyrátať optimálnu trasu z domáceho miesta 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 byť kompatibilná s webovými prehliadačmi Google Chrome, Mozilla Firefox, Microsoft Edge.
4. Užívateľské rozhranie systém musí byť plne použiteľné aj na mobilných telefónoch s OS Android a IOS.
5. Aplikácia bude implementovaný 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.

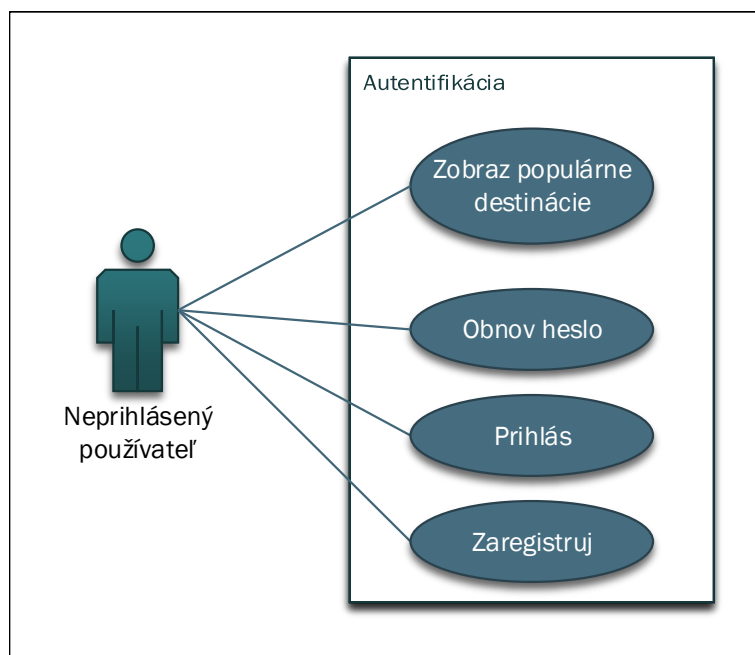
3 Návrh

V tejto kapitole si priblížime proces návrhu našej aplikácie, popíšeme prípady použitia ktoré pre našu aplikáciu očakávame. Ďalej si priblížime návrh štruktúry databázy a na koniec si priblížime štruktúru samotnej aplikácie, ktorú budeme implementovať.

3.1 Prípady použitia

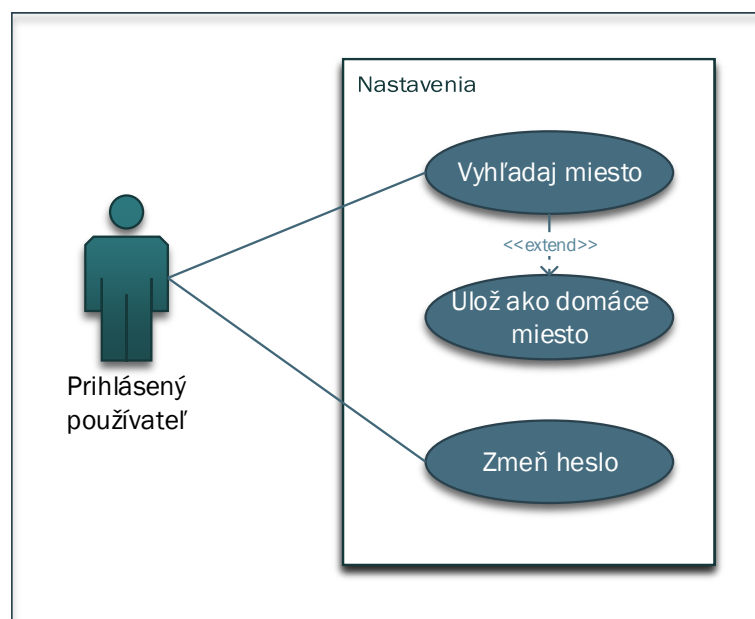
Prípady použitia popisujú interakcie medzi aplikáciou a používateľom, popisujú roly, ktoré rôzni aktéri hrajú v týchto interakciách.

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: Prípád použitia - Registrácia, autentifikácia a nastavenia

2. V druhom prípade použitia (obrázok č. 4) opisujeme nastavenia. Používateľ si bude môcť vyhľadať destináciu a zvoliť si ju ako svoje domáce miesto alebo si zmeniť heslo.
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 zobrazíť detaily



Obrázok 4: Prípád použitia - Registrácia, autentifikácia a nastavenia

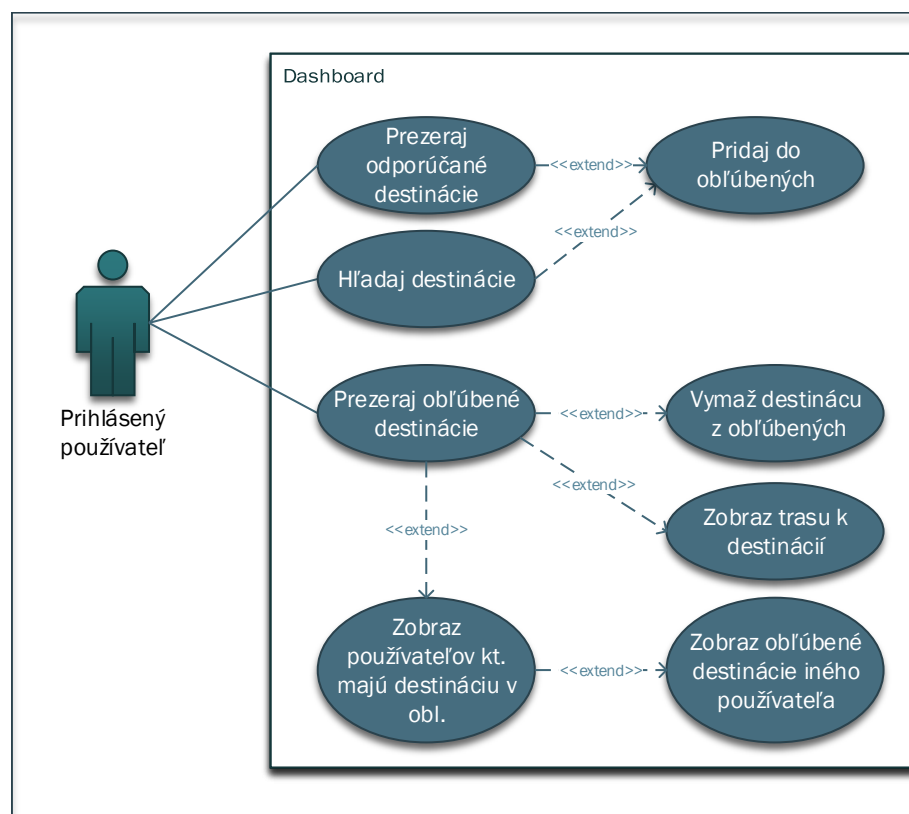
trasy, ktorá k destinácií vedie. Ďalej si bude môcť prezerať ostatných používateľov ktorý majú rovnaké miesto v obľúbených a prejsť na zoznam obľúbených destinácií jedného s týchto používateľov.

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.

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 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 trasy bude vždy viesť od jedného miesta k druhému reprezentuje 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 destinácií a nebudú obsahovať žiadne



Obrázok 5: Prípad použitia - dashboard

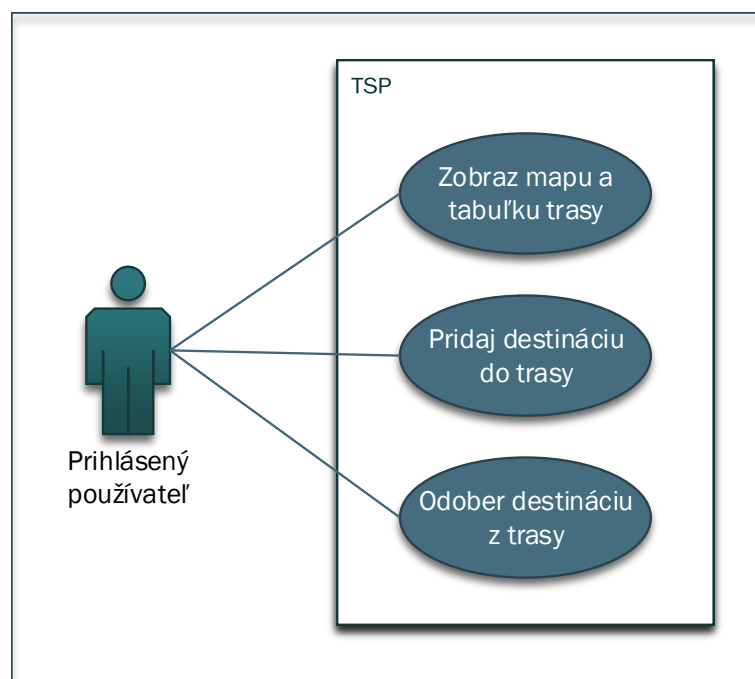
atribúty.

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

Na obrázku č. 8 vidíme reprezentáciu dát uložených v našej databáze. Používateľ *user 3* má ako domáce miesto (HOME) nastavené mesto *Bratislava*, v obľúbených destináciách (FOLLOWS) má mestá *Praha*, *Košice* a *Budapešť*, systém pridal hranu *ROUTES* smerom od domáceho miesta používateľa k cieľovým miestam. Používateľ má tiež pridané mestá *Budapešť* a *Košice* na zozname optimalizácie takže sme pridalí hranu *TSP*.

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: domáca 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.



Obrázok 6: Prípad použitia - tsp

4 Implementácia

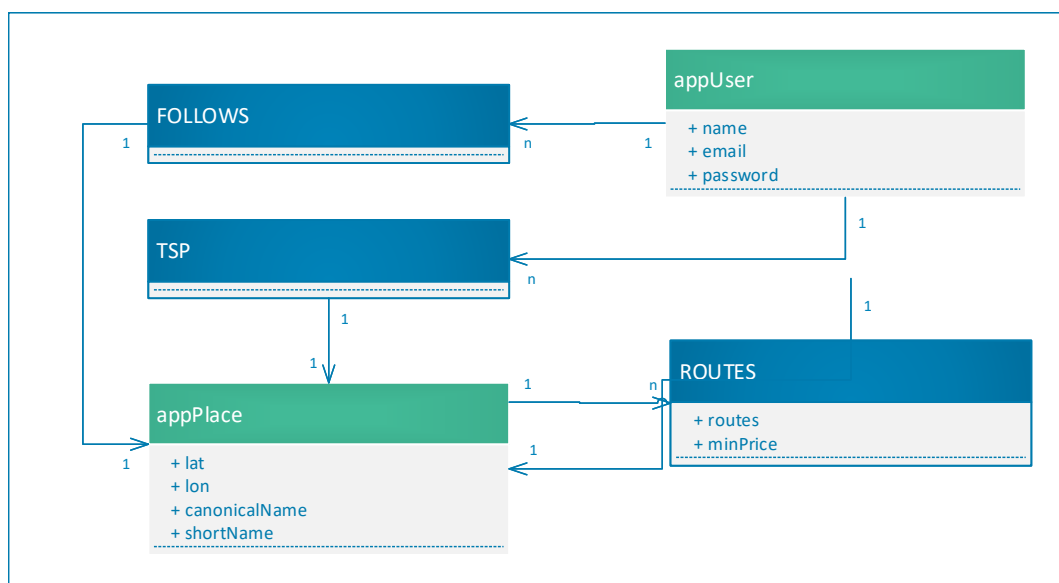
V tejto kapitole si priblížime proces implementácie aplikácie. Najprv si opíšeme niektoré z použitých technológií, následne opíšeme proces inštalácie a konfigurácie frameworku Laravel a databázy Neo4J. Ďalej opíšeme proces našej implementácie aplikácie s pomocou NeoEloquent OGM a pomocou Neo4J PHP Client. 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. Na koniec 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. Laravel 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 s vlastného správcu závislostí. Laravel je orientovaný na to, aby vývoj



Obrázok 7: Štruktúra databázy

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

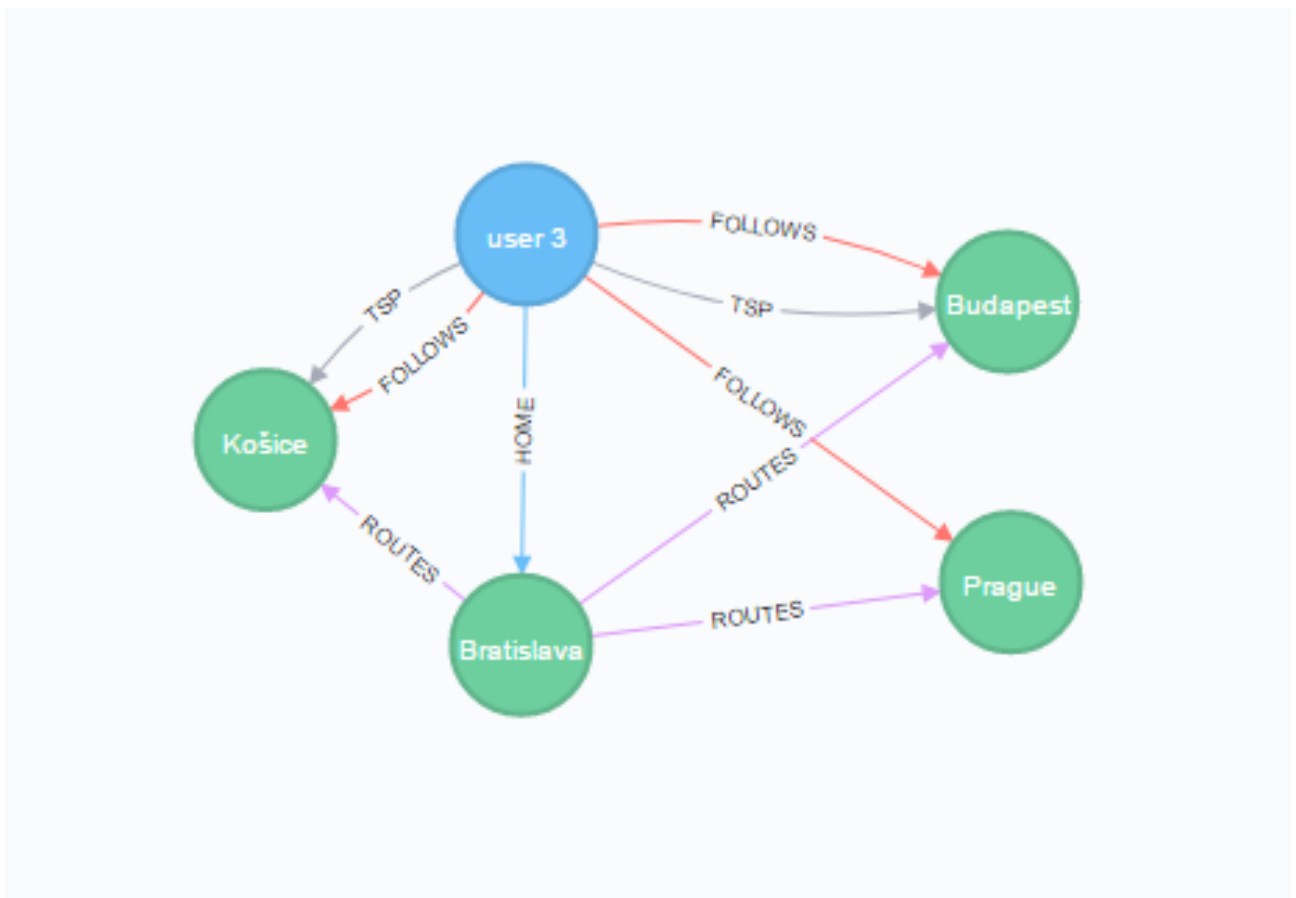
NeoEloquent OGM je voľne šíriteľná knižnica, ktorá umožňuje využívať grafovú databázu Neo4J spolu s existujúcim dátovým modelom vo frameworku Laravel. Štruktúra NeoEloquent je modelovaná podľa natívneho Eloquent Modelu Laravel. Knižnica je vydávaná od roku 2014 pod licenciou MIT spoločnosťou Vinelab

4.1.3 GraphAware Neo4j PHP Client

Neo4j PHP Client je klientská knižnica pre databázu Neo4J. Na rozdiel od OGM je táto schopná vykonávať programátorky zadané dopyty, poskytuje tým pádom prístup k plnému rozsahu Cypher API databázy Neo4j. Knižnica je vyvíjané partnerskou firmou Neo Technology, firmou GraphAware od roku 2014. Je 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 vo verzií zadarmo je knižnica obmedzená na



Obrázok 8: Príklad dát v databáze

päťdesiatisíc zobrazení na mesiac.

Túto knižnicu postupne nahrádza Mapbox-gl.js vyvíjaná tou istou firmou, ktorá využíva technológiu WebGL na lepšiu akceleráciu vykresľovania vektorových máp, no 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 z šablónami, vyvíjaná je Yehudom Katzom a komunitou na GitHubu 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 vyhľadať cenu cesty medzi dvomi 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, je potrebné sa identifikovať API kľúčom a je obmedzené na 100 000 volaní za mesiac.

4.2 Inštalácia a konfigurácia Laravel Framework-u

Na inštaláciu frameworku Laravel sme použili nástroj pre správu PHP balíkov *Composer*. Pomocou tohto nástroja sme nainštalovali balík *laravel/installer* [9], následne sme požitím príkazu *laravel new projekt* vytvorili priečinok so základnou inštaláciou frameworku.

Jediné nastavenie ktoré bolo po tom 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 prihlasení sme vyzvaní na zmenu hesla.

Keďže základná inštalácia frameworku Laravel neobsahuje ovládač pre databázu Neo4J museli sme použiť ovládač integrovaný v balíku NeoEloquent, to sa registráciou 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, port 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 Web-Socket.

4.4 Implementácia OGM

Keďže framework Laravel natívne obsahuje len ovládače pre relačné databázové ovládače a nástroj na objektovo relačné mapovanie Eloquent Použili sme open source knižnicu NeoEloquent, ktorý 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 umožňuje vytvárať perzistentné objekty, upravovať ich a vyhľadávať v nich. Pri práci s hranami je mierne odlišná. Najprv treba zadať ktorý objekt môže mať aké vzťahy, tieto vzťahy treba unikátne 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 menového 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 *belongsToMany*. Ako prvý argument funkcie berú názov triedy ktorou vzťah chceme vrátiť, ako druhý argument berie typ hrany, pomocou tohto 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 8.

Trieda NeoEloquent funguje vo väčšine prípadov presne ako Eloquent no 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 prijme formulár overí či sa jeho token nachádza v databáze. Táto funkcionality 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 tohto 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.

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 - konzolu frameworku príkaz *php artisan make:auth*, ktorá vytvorí URL cesty, obrazovky, triedu používateľa a triedy obsluhujúce túto funkcionality. My sme potrebovali použiť vlastnú triedu používateľa, ktorá dedí od nášho balíka NeoEloquent,

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 domáce
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 používateľov
20     public function followers(){
21         return $this->belongsToMany( 'App\User', 'FOLLOWS' );
22     }
23     ...
24 }
```

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 funkcionality 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 tohto 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 domácim miestom na touto destináciu. Ďalej poslala dopyt na získanie všetkých používateľov ktorý 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 réžií ktorú si vyžaduje diskkrétne volaní funkcií OGM v každom cykle. Ďalej sa testovaniu rýchlosti venujeme v kapitole ??.

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 každú 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 ?? rozhodli sme sa implementovať dopyty na väčšie dátové sady 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. Na príklade 5 je dotaz ktorý nahradzuje celú funkcionalitu kódu z príkladu 4. Dopyt pozostáva vykoná nasledovné úkony (po riadkoch):

1. Získa vrchol typu AppUser s ID = 1 a uložení ho do premennej user, ďalej získa všetkých vrcholov ku ktorým z vrcholu user vedie hrana typu FOLLOWS a uloží ich do premennej dest.
2. Získa vrchol ku ktorému vedie hrana typu home od vrcholu user a uloží ho do premennej home.
3. Získa všetky hrany, ktoré vedú od vrcholu home do jednotlivých vrcholov v premennej dest.
4. Získa všetky vrcholy od ktorých vedie hrana typu FOLLOWS k všetkým vrcholom

v premennej `dest`, keďže aj vrchol `user` má hranu `FOLLOWS` do každého vrcholu v `dest`, vynechá vrchol `user` z výberu.

5. Definuje 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 ??

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

Pre jednoduchosť a spoľahlivosť sme implementovali prístup rekurzívneho prehľadania všetkých možností 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 naša funkcia berie východzie miesto, zoznam destinácií ktoré ešte neprehľadala a domáce miesto do ktorého sa má nakoniec vrátiť. V každom vnorení sa zoznam neprehľadaných vrcholov znižuje. Vyššie spomenuté argumenty sú perzistentné objekty resp. polia inštancií perzistentnej triedy *Place*.

Keď sa táto funkcia dopytuje na cenu trasy vedúcej z východzieho vrcholu do cieľového zavolá 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 niekoľko násobne dlhšie ako každé nasledujúce.

Aj po načítaní všetkých potrebných trás však bolo naše riešenie príliš pomalé, pri siedmych miestach už trvalo výpočet TSP až 7 minút. Toto bolo spôsobené tým, že naša funkcia, ktorá počíta s výpočtovou komplexitou $O(n!)$ beží na vrstve PHP a pri každom volaní funkcie invokes 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 tohto dôvodu sme nainplementovali cache 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 tohto dôvodu sme pridali ešte jeden level cache na celé volanie funkcie pre rátanie TSP. Taktiež sme upravili implementáciu front-endu tak, aby sa tieto dva dopyty vykonali vždy synchronne za sebou. Týmto sme znížili nároky na výpočtové kapacity nášho serveru.

Dlhý čas výpočtu pri dopytovaní sa na databázu pripisujeme tomu, že naše dopyty na databázu boli diskkrétne 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 $O(\log(n))$ a následne vyhľadanie hrany $O(1)$. 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 na algoritme 6. Tento dopyt sme však neimplementovali, kvôli jeho komplexite 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 nodes(path)
4         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 dotazy 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 jenu z destinácií a príslušný objekt sa zašle na náš server. Na unikátnu identifikáciu objektu používame kanonický názov, ktoré 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, dĺžke toto miesto by sa potom nesprávne zobrazovalo všetkým používateľom. Na vyriešenie tohto problému by postačilo urobiť ešte jeden dotaz 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 toho API implementovali pamäť cache. 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 cache súboru a referenciu na súbor na disku. Keďže v momentálnej podobe nevyužívame celú odpoveď tohto API mohli by sme optimalizovať využitie miesta na disku tým, že by sme najprv údaje zapracovali a uložili len tie, ktoré využívame.

Pri testovaní sme zistili, že pri niektorých trasách API vrátilo niektoré z trás s nulovou cenou. Tieto trasy sme ignorovali.

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 sa sťahujú z 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 zobrazíť, ako druhý parameter berie textový identifikátor typu mapy, ktorý chceme zobrazíť. Po inicializácii sa vytvorí vrstva pre mapu ktorá bude obsahovať markery destinácií. Dáta ktoré obsahujú geografickú polohu destinácií sa vyžadajú vo formáte GeoJson z API našej aplikácie. Keďže pri malom priblížení mapy by sa nedali zreteľne rozlíšiť destinácie ktoré sú blízko pri sebe, po načítaní dát vytvoríme vrstvu klastrov pomocou funkcie *L.MarkerClusterGroup*. Táto vrstva spojí blízke body do klastrov, následne skryje značky týchto bodov a nahradí ich značkou klastru. Na pridanie bodov do klastrov sa iteruje cez všetky body, ktoré sú po načítaní v mape a každý sa pridá do vrstvy klastrov. Následne sa vrstva klastrov 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, u nás *FeatureCollection*, a pole objektov typu *Feature*. Každý z týchto objektov obsahuje pole koordinátov, a objekt dodatočných atribútov slúžiacich na upresnenie vizuálu vykresleného objektu. Dané pole koordinátov nám spôsobovalo mierne nedorozumenia, lebo v číslovanom poli je ako prvý prvok je uložená zemepisná dĺžka (longitude) a ako druhý zemepisná šírka (latitude) [7], 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 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 spustí event s menom *appRefresh*. Táto funkcia využije AJAX API knižnice jQuery a stiahne potrebné dáta, potom využije funkciu *template*, ktorá do argumentu berie dáta vo forme poľa objektov a vracia vygenerované HTML ktoré sa následne vkladá na stránku.

V druhej časti kódu sa najprv zavolá funkcia *Handlebars.compile* do argumentu zoberie šablónu ktorá je uložená v elemente s id *places-template*. Ako návratovú hod-

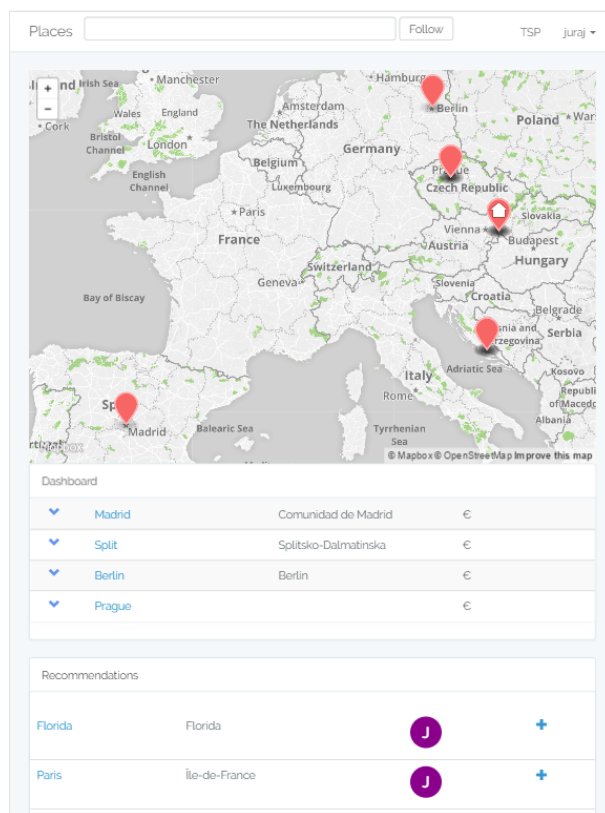
notu vráti funkciu, ktorú uložíme pod menom *template*. Následne sa na event *appRefresh* naviaže volanie funkcie *refreshPage* a prvý krát sa spustí tento event.

Naša šablóna *places-template* obsahuje aj aktívne linky a v nasledujúcom bloku je príklad implementácie vymazania miesta z obľúbených. Na generálny event *click* je naviazaný filtrovaný event, ktorý spustí AJAX dotaz na vymazanie miesta z obľúbených ak element ktorý event spustil obsahuje triedu *delete* a atribút *data-id*.

4.12 Používateľské rozhranie

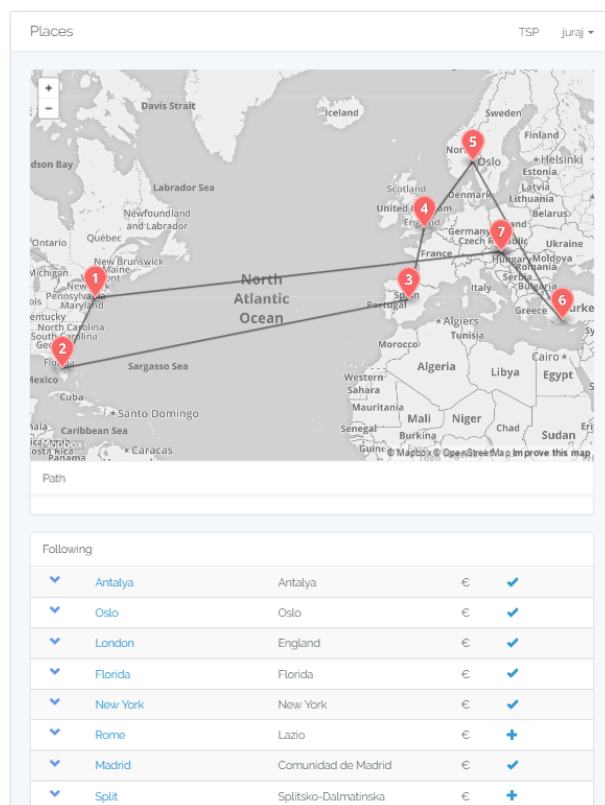
Používateľské rozhranie je implementované pomocou HTML, CSS a JavaScript. Použili sme CSS framework Bootstrap pre zabezpečenie responzivity a konzistentného výzoru a UX aplikácie. Vo vrchnej časti rozhrania sa vždy nachádza navigačný panel.

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



Obrázok 9: Ukážka hlavnej obrazovky používateľa

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 oblúbených destinácií, z ktorých môže destinácie pridávať a odoberať zo zoznamu na optimalizáciu.



Obrázok 10: Ukážka obrazovky TSP

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 verzie 3.1.4 a na databáze MYSQL verzie 5.7.18, tabuľky boli uložené pomocou InnoDB. Pri testovaní rýchlosti naša aplikácia bežala na serveri Apache verzie 2.4.18 s interpreterom PHP verzie 7.0.15.

Ako dátovú sadu 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, takto sme pokračovali až po počet destinácií 1400. Dáta sme generovali našou aplikáciou a následne migrovali do MYSQL databázy štruktúrovanou ako v našom návrhu 7

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 zoznam obľúbených destinácií, ku každej destinácii sme vyhľadali ešte trasu ktorá k nej vedie z domáceho miesta 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.

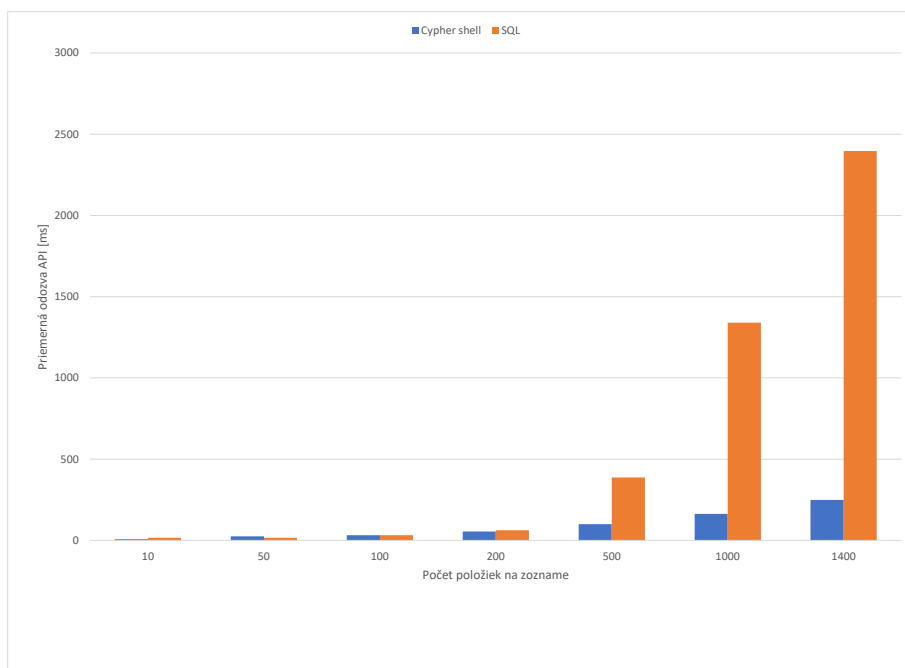
Algoritmus 7 SQL dotaz pre testovanie rýchlosti

```
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
```

Na výsledkoch testu v tabuľke vidíme, pri zväčšovaní a zahusťovaní grafu latencia databázy MySQL rastie omnoho rýchlejšie ako Neo4j. Pri 1400 záznamoch vyšúpala

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 11: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

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 dátovej sady 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

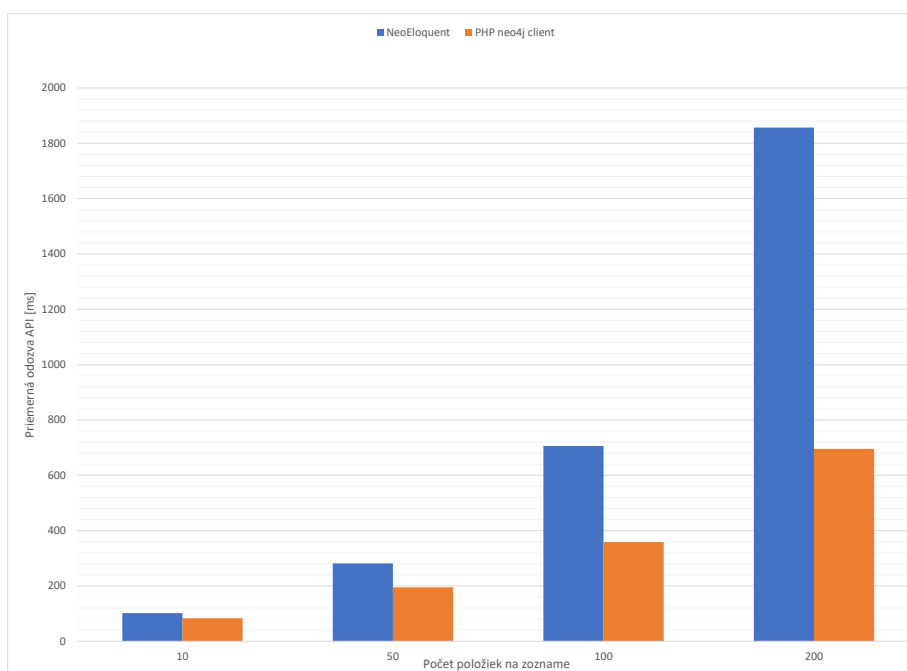
Druhý test ktorý sme robili bol reálny test latencie nami implementovaného API v dvoch verziách. V prvej verzii s použitím OGM a v druhej s použitím PHP Neo4J Client. Tento test sme vykonali ako súčasť implementácie, aby sme vedeli posúdiť ktoré riešenie

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

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. 12. Zo 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.



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

6 Zhodnotenie

V tejto, poslednej kapitole sa budeme venovať zhodnoteniu výsledkov našej implementácie a 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 do konkrétnych destinácií v rátane informácií o cene a spôsobe dopravy a informáciu o tom, aký ďalší používatelia majú tiež aplikáciu 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 aplikáciou obrazovka na optimalizáciu trasy. Na tejto obrazovke si používateľ zo svojho zoznamu môže vybrať niektoré destinácie a aplikácia po vráti poradie v akom by mal používateľ navštíviť zvolené destinácie pre optimalizovanie nákladov na cestu. Pre nedostatok času a komplexitu tohto výpočtu sme implementáciu optimalizácie nechali v staršej verzii – s použitím prístupu k databáze pomocou OGM. Aby sme zlepšili UX používateľa implementovali sme niekoľko vrstiev cache 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 prípade merateľne vyšší výkon pri práci s grafovým dátovým setom veľkosti niekoľko sto vrcholov a hrán. Potvrdili sme teda lepšiu škálovateľnosť grafovej databázy v

6.3 Zhodnotenie skúseností s vývojom pomocou grafovej databázy

Natívna štruktúra použitej grafovej databázy prirodzene poňala našu dátovú sadu bez potreby dáta transformovať, či rozdeľovať 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, pri tom 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 databáz z hľadiska práce so vzťahmi. Následne sme sa venovali predstaveniu troch predstaviteľov 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. Po tom 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, preto sme aplikáciu čiastočne prerobili aby sa v najnáročnejšie operácie vykonávali efektívnejšie. Aplikácia obsahuje celú špecifikovanú funkcionálnu, avšak funkciu optimalizácie trasy sme už kvôli komplexite tohto 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.

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

Zoznam použitej literatúry

- [1] Neo4j graph database: Unlock the value of data relationships, 2017. <https://neo4j.com/product/> [Online; prístupné dňa 15. 05. 2017].
- [2] Orientdb - distributed multi-model and graph database, 2017. <http://orientdb.com/orientdb/> [Online; prístupné dňa 15. 05. 2017].
- [3] Titan: Distributed graph database, 2017. <http://titan.thinkaurelius.com/> [Online; prístupné dňa 15. 05. 2017].
- [4] 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].
- [5] 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].
- [6] Ian Robinson, Jim Webber, Emil Eifrem. *Graph Databases*. 2. O'Reilly Media, Inc. USA, 2015. s. 224. ISBN: 9781491932001.
- [7] (IETF), I. E. T. F. The gejson format, 2016. <https://tools.ietf.org/html/rfc7946> [Online; prístupné dňa 14. 05. 2017].
- [8] MYIURU DAYARATHANA, T. S. A relational model of data for large shared data banks, 2013. http://users.dcc.uchile.cl/~cgutierr/cursos/BD/A_Relational_Model_of_Data_for_Large_Shared_Data_Banks.pdf [Online; prístupné dňa 14. 05. 2017].
- [9] OTWELL, T. Laravel - the php framework for web artisans, 2017. <https://laravel.com/docs/5.0> [Online; prístupné dňa 14. 05. 2017].

Prílohy