# 2 Návrh

V tejto kapitole sa budeme zaoberať návrhom jednotlivých častí aplikácie. Najskôr spomenieme funkcie, ktoré bude naša aplikácia ponúkať. Ďalej popíšeme, ako sme spomedzi mnohých alternatív vybrali vhodný algoritmus pre potrebu našej aplikácie. Pri navrhovaní aplikácie sme sa venovali analýze získaných statických dát a dát o meškaní. Spomíname tiež, ako budeme pristupovať k týmto dátam pri implementácii, teda akým spôsobom ich uložíme do databázy a následne do dátovej štruktúry. V neposlednom rade spomenieme, ako bude fungovať naša aplikácia z pohľadu jej architektúry.

## 2.1 Funkcie aplikácie

Ako sme si mohli všimnúť v 1.9, všetky aplikácie ponúkajú vyhľadávanie z aktuálnej polohy rovnako ako aj možnosť výberu zastávky priamo z mapy. Tieto funkcie bude používateľovi ponúkať aj naša aplikácia. Väčšina spomenutých aplikácií ponúkala zobrazenie histórie vyhľadávania, ktorá odľahčí používateľa od zadávania parametrov v prípade, že vyhľadáva väčšinou tie isté spoje. Túto funkcionalitu nájde používateľ aj v našej aplikácii. História vyhľadávania sa bude ukladať do pamäte zariadenia.

Okrem aplikácie UBIAN a CP si používatelia vedia zobraziť všetky linky v MHD a postupnosti zastávok, ktoré obsluhujú. Túto funkciu bude ponúkať aj naša aplikácia. Čo sa týka nastavenia prídavných preferencií pri vyhľadávaní, aplikácie ponúkajú rôzne preferencie. Najčastejšími z nich sú: maximálny počet prestupov, minimálny čas na prestup, limit pre peší presun a zobrazenie len nízkopodlažých vozidiel. Naša aplikácia bude mať možnosť nastavenia všetkých týchto preferencií.

Jediná aplikácia, ktorá ponúka informácie o reálnom pohybe vozidiel vo forme meškania, je aplikácia UBIAN. Predpokladáme však, že aj táto aplikácia vyhľadáva v statických cestovných poriadkoch a pri vyhľadanom spoji len pripíše informáciu vo forme meškania. Uvažujeme tak na základe toho, že po vyhľadaní spojov zo zastávky po zadaní aktuálneho času aplikácia ponúkne také spoje, ktoré podľa statických cestovných poriadkov majú na túto zastávku v blízkej budúcnosti príchod. Ak existuje taký

spoj, ktorý mal odchod z danej zastávky v minulosti, ale má meškanie a na zastávke ešte nebol, aplikácia ho nezobrazí. Naša aplikácia ponúkne aj tie spojenia, ktoré kvôli meškaniu na zastávku ešte nedorazili. Pre vyhľadaný spoj zobrazí čas odchodu zo statických dát cestovného poriadku a pripíše k nemu informáciu o meškaní.

Medzi ďalšie funkcionality patrí zakúpenie lístka priamo cez aplikáciu. Táto funkcionalita však nesúvisí priamo so zadaním našej práce a navyše je potrebná zmluva s dopravcami. Preto naša aplikácia túto možnosť ponúkať nebude.

Len aplikácia CG Tranzit funguje aj v offine režime. Hoci je užitočné ponúknuť vyhľadávanie bez možnosti prístupu na internet, znamenalo by to že náš algoritmus by bežal na klientskej strane. Keďže hlavnou úlohou našej aplikácie je vyhľadávanie spojov z reálnych dát, aplikácia bude fungovať online s tým, že zaručí používateľom, vždy aktuálne spojenia.

V tabuľke 2.1 je zobrazený prehľad funkcionalít našej navrhovanej aplikácie a iných

existujúcich aplikácii na vyhľadávanie spojov v MHD Bratislava.

------------------------------tabuľka-------------------------------------------------------------------------

## 2.2 Algoritmus

Pri hľadaní algoritmu na nájdenie optimálnej cesty sme najskôr siahli po najznámejšom vyhľadávacom grafovom algoritme. Dijkstrov algoritmus 1.2.1 sa zdal vhodný, avšak jeho vylepšená verzia A\* algoritmus 1.2.2 je pri správne zvolenej heuristike efektívnejšia.

Keďže zastávky, ktoré predstavujú vrcholy v grafe majú dané súradnice, môžeme pri vyhľadávaní optimalizovať prehľadávaný priestor. Pri štúdiu článkov sme narazili na rôzne optimalizácie prehľadávaného priestoru. Minimalizácia v okolí virtuálnej cesty a minimalizácia bounding boxom sú spomenuté v 1.5.

V prípade cestovných poriadkov je náročné správne namodelovať graf, ktorý dokáže efektívne spracovať časovo závislé dáta. V 1.4 boli spomenuté dva overené prístupy Time dependent a Time expanded model, ktoré tento problém riešia.

Ďalšou výzvou je prispôsobiť grafový vyhľadávací algoritmus, aby dokázal vypočítať najoptimálnejšiu cestu, prihliadal na prestupy medzi rôznymi módmi a popri tom počítal s ďalšími pridanými kritériami. V 1.3 boli spomenuté návrhy časovo závislých algoritmov. Algoritmus 1.3.1 sa dokáže vysporiadať aj viacerými módmi. Vráti však len jednu cestu. V 1.7 je popísaný algoritmus, ktorý efektívne vyhľadáva rôzne alternatívne cesty.

Kvôli dynamickej povahe verejnej dopravy, grafový prístup v kombinácii s vyhľadávacím grafovým algoritmom vyžaduje veľa pre-processingu a to sa odráža na výpočtových časoch.

Výhodou vo verejnej doprave je, že vozidlá sa pohybujú po vyznačených linkách, ktorých trasy poznáme. Schéma verejnej dopravy sa preto dá zachytiť do pomerne jednoduchých dátových štruktúr. Tento fakt si všimli aj autori algoritmu RAPTOR, ktorý sme opísali v 1.8.

V našej aplikácii sme sa rozhodli použiť tento negrafový algoritmus. Jeho výhodou je, že nie je potrebné vytvárať model a nie je potrebné riešiť multimodalitu hromadnej dopravy. Ľahšie zvláda dynamickosť dát ako meškanie linky, zrušenie linky alebo zmenu trasy. Využijeme základnú verziu RAPTOR algoritmu popísanú v 1.8.2 a súčasne využijeme aj jeho optimalizáciu opísanú v 1.8.3, kedy označujeme zastávky, aby sme nemuseli prechádzať tie linky, ktorým sa nevylepšil čas Tauk-1(p). Optimalizáciu local-prunning nevyužijeme, keďže chceme použiť rozšírenú verziu RAPTOR algoritmu a to je rRAPTOR algoritmus popísaný v sekcii 1.8.4. Algoritmus rRAPTOR nám nevráti len jednu najkratšiu cestu, ktorá začína najskôr od zadaného času, ale vráti nám množinu najkratších ciest,

začínajúcich v zadanom časovom úseku.

Cesta, ktorú nám vráti základná verzia RAPTOR algoritmu je najkratšia a nezáleží koľko prestupov bude obsahovať. Našou úlohou je používateľovi poskytnúť optimálnu cestu a keďže optimálna cesta môže byť pre každého používateľa iná, ponúkneme mu viacero alternatív. Túto funkciu ponúka vylepšený RAPTOR algoritmus popísaný v 1.8.7. Algoritmus nám vráti viacero optimálnych ciest, pričom prihliada na to, aby jednotlivé cesty neboli rovnaké na veľkej časti úseku.

Našim cieľom je zlúčiť rRAPTOR algoritmus s vylepšeným RAPTOR algoritmom na hľadanie viacerých ciest a zakomponovať mechanizmus, schopný zohľadniť zadané používateľské preferencie, ktoré bude algoritmus prijímať ako vstupné parametre. Preferencie, ktoré bude vedieť algoritmus zohľadniť: minimálny čas na prestup, maximálny počet prestupov, maximálna dĺžka pešieho prestupu a vyhľadanie len nízkopodlažných spojov.

Na obrázku (algorithm.png) sú zachytené jednotlivé podalgoritmy, ktoré budú tvoriť náš výsledný algoritmus.

Výstupom z nášho algoritmu bude množina ciest, začínajúcich na zastávke ps, po čase Tau a končiacich v zastávke pt. Jednotlivé cesty sú optimálne, vyhovujú prípadným používateľským preferenciám a ich časy odchodov sú z časového intervalu <Tau; Tau +delta Tau>.

### 2.2.1 Vyhľadávanie z aktuálnej polohy

Ak používateľ zadá ako začiatočnú zastávku aktuálnu polohu, nájdeme zastávky v okolí radiálnym

vyhľadávaním, ako bolo spomenuté v 1.6. Najbližšia vyhľadaná zastávka k aktuálnemu bodu nemusí znamenať optimálne riešenie a preto budeme považovať za začiatočnú zastávku každú z nich. Pre každú zastávku zvlášť spustíme algoritmus, ktorý nám pre každú zo zastávok vráti množinu optimálnych ciest a na záver z nich vyberieme n najoptimálnejších ciest. Do finálneho riešenia zakomponujeme peší presun od aktuálnej polohy po začiatočnú zastávku vybraných optimálnych ciest.

## 2.3 Dáta

Pri vývoji aplikácie a pre jej testovanie sú nevyhnutné dáta. Pre účely aplikácie budeme potrebovať dáta zo statických cestovných poriadkov a dáta omeškaní jednotlivých jázd.

### 2.3.1 Dáta statických cestovných poriadkov

Od Dopravného podniku Bratislava sme získali statické cestovné poriadky, ktoré mali platnosť od 5.2.2018 – 31.12.2018. Dáta sú vo formáte GTFS.

#### 2.3.1.1 GTFS

General Transit Feed Specification (GTFS) je dohodnutý formát dát, ktorý používajú tisíce poskytovateľov verejnej dopravy a mnohé softvérové aplikácie. Špecifikácia definuje súbory, v ktorých sú reprezentované entity v tabuľke. V stĺpcoch sú popísané vlastnosti entity a v každom riadku je nový záznam.

Súbor **agency.txt obsahuje vlastnosti:** agency\_id, agency\_name, agency\_url, agency\_timezone,agency\_lang, agency\_phone. V našich dátach sa nechádza len jedna spoločnosť, ktorá prevádzkuje spoje a preto súbor s touto entitou nebudeme spracovávať.

Zástavky sú zachytené v súbore **stops.txt. Vlastnosti sú: s**top\_id, stop\_name, stop\_lat, stop\_lon, zone\_id. V zozname sa zastávka s rovnakým názvom (stop\_name) nachádza viac krát. Jedná sa skôr o názov platformy (napr. Račianske mýto), pretože jednotlivé zastávky sú umiestnené v rámci platformy na rôznych súradniciach.

Súbor **routes.txt** obashuje zoznam liniek. Pre každú linku ponúka vlasnotsti**:** route\_id, agency\_id, route\_short\_name, route\_long\_name, route\_type, route\_text\_color. Vlastnosti agency\_id a route\_text\_color budeme ignorovať pretože sú stále rovnaké. Vlasnosť route\_long\_name nie je určená. Vlastnosť route\_type predstavuje typ módu. Podľa štandardu GTFS, 0 definuje električku, 3 autobus a 11 trolejbus.

Jazdy jednotlivých liniek sú zaznamenané v súbore **trips.txt,** ktorý obsahuje vlastnosti: route\_id, service\_id, trip\_id, trip\_headsign, trip\_short\_name, direction\_id. Podľa GTFS špecifikácie by service\_id mala byť množina service\_day oddelená pomlčkou. V našich dátach existuje pre jeden záznam jazdy len jeden service\_day. V prípade, že jazda patrí viacerým service\_day, je vytvorený nový záznam s rovnakými vlastnosťami jazdy pre každý service\_day. Vlastnosť trip\_headsign predstavuje konečnú zastávku jazdy. Túto vlastnosť nevyužijeme pretože ju vieme získať iným spôsobom. V žiadnom zázname jazdy nie je vyplnené pole trip\_shortname, takže túto vlastnosť budeme ignorovať. Direction\_id má hodnoty 0 a 1. V dátach chýba informácia o nízkopodlažných spojoch.

Súbor **stop\_times.txt** má vlastnostitrip\_id, arrival\_time, departure\_time, stop\_id, stop\_sequence, stop\_headsign, pickup\_type, drop\_off\_type. Arrival\_time a departure\_time sú vždy rovnaké, preto jeden z nich budeme ignorovať. Vlastnosť stop\_sequence je poradie zastávky vrámci linky. Poradnie nie je iterované od 1, ale začína rôznymi číslami. Položka stop\_headsign nie je v žiadnom zázname vyplnená. Vlastnosť pickup\_type a drop\_off\_type nadobúdajú hodnoty 0 a 3. V GTFS špecifikácii znamená číslo 0, že jazda na tejto zastávke stojí vždy a 3 určuje, že na zastávke stojí len na znamenie. Hodnoty sú pre obe vlastnosti vždy rovnaké, takže jednu z nich môžeme opäť ignorovať.

**Súbor celandar.txt obsahuje stĺpce** service\_id, monday, tuesday, wednesday, thursday, friday, saturday, sunday, start\_date, end\_date. Vlasntosť service\_id predstavuje unikátny názov typu služby (pracovný deň, víkend,...). Vlasnosti ,monday,...sunday môžu nadobúdať hodnoty 0 a 1. Ak je napr. hodnota monday = 0, znamená to, že všetky pondelky medzi start\_date a end\_date patria do služby service\_id. Naopak, ak je hodnota 1, tak nejazdí.

V súbore **calendar-dates.txt sú vlastnosti:** service\_id, date, exception\_type. V prípade, že vlastnosť exception\_type má hodnotu 1, znamená to, že service\_id navyše jazdí v deň date. Ak je exception\_type = 2, znamená to, že v deň date service\_id nejazdí.

### 2.3.2 Dáta o meškaní

Podarilo sa nám získať aj informácie o meškaní za rok 2018, pre mesiac február, marec a apríl. Pre každý deň v mesiaci existuje súbor vo formáte .csv. Každý záznam obsahuje údaje:

Id – identifikačné číslo záznamu

Datum a cas – dátum a čas, kedy bol záznam o meškaní zaevidovaný

Vozidlo – identifikačné číslo vozidla

Linka – identifikačné číslo linky

Poradie – poradie jazdy v linke

Cislo zastavky – identifikačné číslo zastávky

Nazov zastavky – názov zastávky

Meskanie – hodnota meškania v minútach.

Z pozorovania dát vyplýva, že hodnoty meškania majú hodnotu 0, záporné celé čísla alebo hodnotu n/a. V prípade, že narazíme na hodnotu 0 alebo n/a, tento záznam neberieme do úvahy. Môže ale platiť, že hodnota 0 znamená, že meškanie je z intervalu <0 min, 1 min) minúta, hodnota 1 z intervalu <1 min, 2 min). Mení to nejako náš pohľad na vec? Na prelome dní sa tam vyskytuje hodnota -1229. Kto vie čo znamená?

Takže naša aplikácia bude môcť ponúknuť správne cesty s prihliadnutím na meškanie spojov od 5.2.2018 – 30.4.2018. Statické vyhľadávanie bez meškania bude správne fungovať od 5.2.2018 do konca roka 2018.

### 2.3.3 Pešie presuny

V dátach sa nenachádzajú informácie o peších presunoch. Google API ponúka možnosť vyhľadania peších vzdialeností medzi 2 bodmi. Počet bezplatných dopytov na Distance Matrix API je obmedzený. Nebudeme vyhľadávať pešie vzdialenosti medzi každou dvojicou zastávok, nakoľko je to nepotrebné.

Budeme postupovať nasledovne: pre každú zastávku p nájdeme zastávky v okolí 800 metrov radiálnym vyhľadávaním. Pomocou Distance Matrix API zistím vzdialenosti nájdených zastávok od zastávky p. Každú dvojicu zastávok uložíme do súboru foot\_paths.txt spolu so zistenými vzdialenosťami určených v minútach.

## 2.4 Databáza

Na serveri budú uložené dáta aplikácie v PostgreSQL databáze. Databáza sa naplní pri prvotnom spustení aplikácie na serveri. Schéma databázy je popísaná entitno-relačným diagramom na obrázku 2.2.

---------------------------------Obr. 2.2: Entitno-relačný diagram -------------------------------------------------------

Entita stop\_areas obsahuje zoskupenie zastávok, ktoré majú rovnaké názvy (name).

V entite stops evidujeme zónu mesta (zone), do ktorej zastávka patrí. Hodnota on\_request určuje, či sa na zastávke nastupuje a vystupuje na znamenie. Každá zastávka má priradené súradnice, ktoré sa udržujú v entite coords. V entite coords sú súradnice určené atribútmi zemepisná šírka (latitude) a zemepisná výška (longitude).

Entita foot\_paths obsahuje atribút time, ktorý určuje čas v sekundách potrebný na peší presun zo zastávky from na zastávku to.

V entite routes sa udržuje zoznam liniek, jazdiacich aktuálne v bratislavskej hromadnej doprave. Linka je určená názvom (name) a módom (mode). V Bratislave jazdia 3 rôzne módy: električka, trolejbus a autobus. Každá linka má počas dňa viaceré jazdy (trips).

Entita (trips) uchováva informáciu o tom, či je vozidlo, ktoré bolo pridelené konkrétnej jazde nízkopodlažné, ktorým smerom ide (direction) a zároveň počas akých typov dní (service\_day) jazda premáva. Každá jazda linky je tvorená postupnosťou zastávok, ktoré linka obsluhuje.

V entite stop\_times je zachytený čas (time), kedy jazda stojí na zastávke a v akom poradí sú zastávky v rámci jazdy (sequence\_order).

V entite service\_days sú názvy rôznych typov dní, v ktorých jazdia jazdy liniek.

Entita calendar\_dates obsahuje zoznam všetkých dátumov (date), v rozsahu platnosti aktuálneho cestovného poriadku. Ku každému dátumu je priradený typ dňa (service\_day). Jeden dátum môže prislúchať k viacerým typom dňa a naopak jeden typ dňa môže prislúchať viacerým dátumom.

## 2.5 Dátová štruktúra

Rovnako dôležitá je dobre navrhnutá dátová štruktúra, s ktorou bude algoritmus vedieť rýchlo a efektívne pracovať. Budeme sa inšpirovať dátovou štruktúrou z 1.8.6, ktorá bola navrhnutá pre základnú verziu RAPTOR algoritmu. Dátová štruktúra bude obsahovať aktuálne cestovné poriadky s prispôsobenými príchodmi a odchodmi vozidiel podľa prípadných meškaní.

Dátová štruktúra uvedená v článku ráta s tým, že jednotlivé jazdy v rámci linky majú rovnakú postupnosť zastávok. V našich dátach to tak nie je. Linka obsahuje jazdy, ktoré idú jedným aj druhým smerom. Zástavky cez ktoré prechádza linka majú síce rovnaký názov, ale majú iné identifikačné čísla, ktorými sú definované. Okrem rôznych smerov obsahuje linka aj také jazdy, ktorých postupnosť zastávok je iná ako pri väčšine. Najmä v ranných a večerných hodinách prechádzajú niektoré jazdy len cez určitú podpostupnosť zastávok.

RAPTOR algoritmus potrebuje, aby všetky jazdy, ktoré patria konkrétnej linke mali rovnakú postupnosť zastávok. Rozhodli sme sa preto zoskupiť jazdy s rovnakou postupnosťou zastávok do úsekov. Teraz platí, že 1 linka má viacero úsekov a jednému úseku linky prislúcha viac jázd linky.

Často sa potrebujeme dopytovať na všetky linky, ktoré stoja na konkrétnej zastávke a rovnako potrebujem vedieť postupnosť zastávok, ktoré patria konkrétnej linke.

StopTimes bude objekt alebo pole, ktoré obsahuje nielen informáciu o čase, kedy podľa statického poriadku má stáť jazda linky na zastávke, ale aj informáciu o predpokladanom časovom meškaní danej jazdy na zastávku

VYMYSLIEŤ a Popísať čo všetko sme tam zmenili.

## 2.6 Architektúra aplikácie

Klient sa bude dopytovať na server pre vyhľadanie spojenia. Server spustí výpočet nad dátovou štruktúrou, ktorá má aktuálne cestovné poriadky s informáciou o prípadnom meškaní spojov a vráti odpoveď klientovi.

Algoritmus bude pracovať nad dátovou štruktúrou, ktorá bude obsahovať stále aktuálne dáta. Dátová štruktúra bude rovnako ako algoritmus uložená na serverovej strane.

### 2.6.1 Serverová strana

Na serverovej strane bude bežať aplikačný server Tomcat. Na uchovanie dát použijeme relačnú databázu. Na komunikáciu s klientom budeme používať REST API.

### 2.6.2 Klientská strana

Na klientskej strane sme sa rozhodli pre progresívnu webová aplikácia PWA. Je to webová aplikácia, ktorá sa dokáže správať ako mobilná aplikácia, ktorá sa neustále aktualizuje, pričom nie je potrebná jej inštalácia. Po návšteve webovej stránky na mobilnom zariadení používateľ dostane upozornenie od stránky, či si ju chce uložiť do zariadenia ako mobilnú aplikáciu. Progresívna webová aplikácia zaberá minimum miesta v pamäti a má svoj vlastný úložný priestor, kde sa budú ukladať preferencie a história vyhľadávania.

### 2.6.3 Spracovanie dát

Pri spustení aplikácie alebo po aktualizácii cestovných poriadkov sa spustí služba, ktorá nám z úložiska, kde sú aktuálne cestovné poriadky namapuje dáta do našej databázy. Po tom ako budú dáta uložené v databáze sa spustí ďalšia služba, ktorá obnoví dátovú štruktúru podľa nových cestovných poriadkov.

Ďalšia služba bude vytvorená na spracovanie údajov o meškaní. Hoci máme v súbore pre konkrétny deň údaje o meškaní jázd na celý deň, chceme sa čo najviac priblížiť reálnemu nasadeniu. Budeme teda rátať s tým, že nové údaje o meškaní pribúdajú po minúte. Služba bude spúšťaná každú minútu. Bude čítať súbor pre aktuálny deň, získa záznamy, ktoré pribudli v poslednej minúte a aktualizuje meškanie pre konkrétnu jazdu. Údaje o meškaní sú evidované pre zastávku s, na ktorej meškanie vzniklo. Aktualizácia meškania jazdy bude prebiehať tak, že pre všetky zastávky jazdy od zastávky s až po konečnú zastávku jazdy zapíše do dátovej štruktúry hodnotu získaného meškania.

Spôsob akým bude aplikácia nasadená je znázornená na obrázku 2.1.

---------------------------------------------------Obr. 2.1: Diagram nasadenia---------------------------------------------

# 3 Implementácia

V tejto kapitole popíšeme technológie, ktoré sme vybrali na implementáciu aplikácie. Ďalej popíšeme základné problémy, ktoré vznikli pri spracovaní dát, pri implementácii dátovej štruktúry a navrhovaného algoritmu. Ukážkami kódov priblížime, ako sme tieto problémy riešili.

## 3.0 Technológie

Serverovú časť aplikácie sme implementovali v jazyku Java, konkrétne vo verzii Java11 s využitím frameworku SpringBoot.

Na mapovanie Java objektov na databázové entity sme použili ORM framework Hibernate. Počas implementácie sme však zistili, že Hibernate framework nebol veľmi dobrou voľbou. Hibernate sme si pôvodne vybrali, pretože sa nám zdal postačujúci pre jednorazové vloženie údajov do tabuliek pri spracovaní GTFS dát a následne len čítanie tabulie, bez nutnosti komplikovaných SQL príklazov. Vzťahy medzi tabuľkami sú však komplexné a údajov je veľké množstvo. Najväčšiu záťaž sme pocítili pri čítaní všetkých údajov z tabuľky trips, ktoré sú prepojené so všetkými ostatnými tabuľkami. Databázu preto budeme používať len na posielanie údajov pre frontend (stops, stop\_areas, routes).

Pre komunikáciu s front-endom sme si vybrali REST architektúru. Táto architektúra je dokumentovaná pomocou technológie Swagger.

Na front-endovej strane sme si zvolili progresívnu webovú aplikáciu a na implementáciu sme použili moderný JavaScriptový framework VUE.js.

## 3.1 Dáta

Dátová sada vo formáte GTFS obsahuje veľa záznamov. Už pri implementácii bolo potrebné myslieť na to, ako budeme overovať správnosť mapovania dát do databázy, korektnosť vytvorenej dátovej štruktúry a v neposlednom rade vedieť ohodnotiť a upraviť algoritmus tak, aby počítal čo najoptimálnejšie cesty. Pri veľkej dátovej sade sa ťažko testuje správnosť implementácie a preto sme pred samotnou implementáciou navrhli zmenšenú testovaciu sadu dát.

### 3.1.1 Parsovanie dát zo súboru

Dáta zo súboru parsujeme, ak chceme vytovriť databázu (ImportService) alebo dátovú štruktúru pre algoritmus (LoadService).

Dáta zo súboru stops.txt sme zachytili v 3 rôznych objektoch. Polohu zastávky sme vyčlenili do zvlášť objektu Coords. Zastávky s rovnakým názvom sme zoskupili do objektu StopArea. Samotný objekt Stop zachytáva zónu a či je zastávka na znamenie.

Keďže v dátach nemáme informáciu o peších presunoch medzi jednotlivými zastávkami, pomocou Google Distance Matrix API sme zistili vzdialenosti medzi jednotlivými zastávkami a obohatili sme dáta o súbor foot\_paths.txt. Dáta z tohto súboru sú v namapované na objekty FootPath.

Každý dátum z rozmedzia dátumov platnosti aktuálneho cestovného poriadku potrebujeme zaradiť do pod nejaký typ dňa (pracovné dni, víkendy, školské prázdniny, štátne sviatky,...). Z dát uložených v  GTFS štandarde sa komplikovaných spôsobom zisťuje, ktoré typy dní prislúchajú konkrétnemu dátumu. Objekty CalendarDate budú uchovávať dátumy z rozmedzia dátumov platnosti cestovného poriadku. V objektoch ServiceDay budú uložené jednotlivé typy dní.

Dáta zo súborov trips.txt, routes.txt a stop\_times.txt len mapujeme na objekty Trip, Route a StopTime. Zmenou je, že informáciu o tom, či je zastávka na znamenie si neudržujeme v objekte StopTime, ale priamo v objekte Stop. V dátach chýbala informácia o tom, či je jazda linky vedená nízkopodlažným vozidlom. Parameter lowFloor náhodne generujeme pre jednotlivé jazdy do objektu Trip.

## 3.2 Dátová štruktúra

### 3.2.1 Vytvorenie dátovej štruktúry

Dátová štruktúra sa vytvára pri spustení serverovej aplikácie. Najskôr sme začali implementovať vytvorenie dátovej štruktúry podľa návrhu, teda ťahaním údajov z databázy. Pri testovacích dátach tento úkon trval pár sekúnd. S dátami od dopravného podniku trval tento proces niekoľko hodín. Spomalenie spôsobovalo ťahanie údajov z databázy. Tu sme zistili, že Hibernate framework funguje veľmi pomaly pri veľkom množstve dát a najmä pri komplikovaných schémach. Rozhodli sme sa teda s tomto prípade obísť databázu a dátovú štruktúru vytvárame hneď pri parsovaní súboru. Tento proces trvá približne 20 minút, čo je oveľa prijateľnejšie pre produkciu. Pri implementovaní bolo však 20 minútové čakanie veľmi obmedzujúce pri každom spustení aplikácie. Vytvorený a naplnený model dátovej štruktúry sme serializovali do súboru. Proces deserializácia však trval stále dlho, tak sme sa rozhodli využiť paralelizáciu na 6 vláknach pri deserializácii modelu dátovej štruktúry.

Ako sme spomínali na produkcii nie je problematické, že vytvorenie dátovej štruktúry trvá dlho, keďže dátová štruktúra sa vytvára len raz pri prvotnom spustení aplikácie (alebo po aktualizácii cestovných poriadkov) na serverovej strane. Ak by sme premýšľali o držaní dátovej štruktúry na klientskej strane, vytvorenie dátovej štruktúry by sa to mohlo vykonať pri prvotnej inštalácii ale dátová štruktúra by zaberala veľa pamäte. Serializovaný model dátovej štruktúry má skoro 1GB. Druhá možnosť je načítavať dátovú štruktúru pri každom spustení aplikácie, čo by trvalo dlho.

Síce frontendové jazyky sú rýchlejšie a objekty sú menšie, stále by mohol byť problém s vytvorením dátovej štruktúry a pamäťou, ktorá aplikácia zaberá.

### 3.2.2 Implementácia dátovej štruktúry

Pre celý beh aplikácie bude potrebná len jedna inštancia dátovej štruktúry. Pri spustení aplikácie sa vytvorí aplikačný kontext, ktorý reprezentuje množinu prepojených komponentov. Tieto komponenty manažuje IoC kontainer. Objekt DataStructure je označený anotáciou @Component, ktorý pri autowirovaní vytvorí DataStructureModel načítaním z GTFS dát v produkčnom prostredí alebo deserializovaním zo súboru v implementáčnej fáze. @Component je vlastne @Bean a ten sa v Springu správa ako Singleton.

Trieda DataStrctureModel definuje dátovú štruktúru pre RAPTOR algoritmus, ktorá je tvorená šiestimi dátovými štruktúrami.

public class DataStructureModel implements Serializable {

private Map<Route, List<Subroute>> routeSubroutes = new HashMap<>();

private Map<Long, List<Transfer>> stopTransfers = new HashMap<>();

private Map<Stop, List<Subroute>> stopSubroutes = new HashMap<>();

private Map<String, Map<Long, Integer>> stopIndexInSubroute = new HashMap<>();

private List<CalendarDate> calendarDates = new ArrayList<>();

...

}

Štruktúra routeSubroutes je definovaná mapou, kde kľúčom je linka r a hodnota je pole úsekov, ktoré patria linke r. Kľúčom v stopTransfers je identifikačné číslo zastávky p a hodnotou je pole peších prestupov, pričom začiatočná zastávka je zastávka p. Mapa stopSubroutes ma zastávku p ako kľúč a hodnotou je pole úsekov linky, ktoré stoja na zastávke p. Štruktúra stopIndexInSubroute poskytuje poradové číslo zastávky v rámci úseku linky. Posledné je pole objektov CalendarDate, ktoré poskytuje všetky dátumy z rozsahu platnosti cestovných poriadkov a ku každému dátumu d je priradená množina typov dní.

## 3.3 Algoritmus

### 3.3.1 Označovanie zastávok medzi kolami

Algoritmus nepočíta s tým, že existuje možnosť na zo začiatočnej zastávky riešiť pešie presuny, keďže marked stops vymaže už pri inicializácii množiny Q a pridávajú ich do množiny len ak zlepšíme nejaký čas. Ten chalan to má dobre. Iniciálne marked stops načíta zo serchmarams – nulté kolo. Potom v prvom kole marked stops nevymazáva pri vytváraní Q. A tým pádom sa pri trasovaní peších presunov hľadajú a tie z pôvodných zastávok. Množinu marked stops meníme až po prvom kole a sú to tie zastávky, ktoré majú v tom kole iný čas ako nekonečno. Toto môže byť ale oveľa viac zastávok ako keď sa išlo pôvodným štýlom. Bolo by dobré držať si nejakú množinu markedStopsInNewRound a do nej pridávať tie zastávky pri každom zlepšení a na záver tieto zastávky vložiť do markedStops. Tak budem mať tie pôvodné markedStops aj pri traverzovaní peších presunov a zároveň markedStops budú len tie, ktoré boli v predchádzajúcom kole zlepšené. Tomu šuhajkovi to funguje asi preto lebo on nekopíruje pri inicializácii nového kola všetko z predchádzajúceho kola. Čo je možno aj dobre lebo vie v ktorom kole čo vzniklo.

### 3.3.2 Zapracovanie používateľských preferencií.

#### 3.3.2.1 Maximálny počet prestupov

RAPTOR algoritmus končí vtedy, ak už neexistujú žiadne ďalšie označené zastávky, teda ak sa v predchádzajúcom kole nevylepšil žiadny z časov. RAPTOR algoritmus beží v kolách. 1. kolo znamená 0 prestupov, 2. kolo 1 prestup, atď. Aby bol algoritmus obmedzený počtom prestupov, stačí pridať podmienku a obmedziť algoritmus navyše maximálnym počtom kôl.

public RaptorResults search(SearchParams searchParams) {

initializeMarkedStops(searchParams);

RaptorResults raptorResults = initializeRaptorResults(searchParams);

while(markedStops.size() > 0 && **raptorResults.getRound() <= searchParams.getMaxNumberOfTransfers()**){

...

}

return raptorResults;

}

#### 3.3.2.2 Maximálny čas pešieho presunu

Pri prechádzaní zastávok, ktoré boli v predchádzajúcom kole vylepšené hľadáme pešie presuny do okolitých zastávok. Okolité zastávky boli pre každú zastávku radiálne vyhľadané v okolí 800 metrov pri importe dát do databázy. V algoritme hľadáme pre zastávku len také pešie presuny do iných zastávok, ktorých trvanie je menšie alebo rovné ako používateľom definovaný maximálny čas pešieho presunu.

private void traverseTransfers(RaptorResults raptorResults, Set<Stop> improvedStops, int maxTimeOfWalking){

for(Stop stop: markedStops){

for(Transfer transfer: dataStructure.getDataStructureModel().getStopTransfers().get(stop)){

if(**transfer.getDuration() <= maxTimeOfWalking**) {

...

}

}

}

}

V algoritme však môže nastať prípad, vo viacerých kolách po sebe bude vybratý peší presun a v súčte trvanie týchto peších presunov môže trvať dlhšie ako používateľom definovaný maximálny čas pešieho presunu.

Takúto cestu neponúkneme používateľovi a vyfiltrujeme ju pri tvorení zoznamu výsledných ciest.

#### 3.3.2.3 Minimálny čas na prestup

Nech nastane prípad, že na zastávke A vystúpime v čase 6:00 a hodnota minimálneho času na prestup je 2 (minúty). Hľadáme najskoršiu jazdu linky r, ktorú vieme stihnúť na zastávke A. Nevyberieme jazdu t2 ale jazdu t3, keďže minimálny čas na prestup sú 2 minúty.

Môže nastať ale prípad, že sme sa na zastávku A dostali v čase 6:00 peším presunom. V tomto prípade by sme nechceli prirátavať ešte minimálny čas na prestup, ten chceme prirátavať len v prípade, že prestupujeme medzi dvoma jazdami. Tento prípad však budeme ignorovať, pretože najrýchlejší čas, ktorým sa dostaneme na nejakú zastávku môžeme dosiahnuť viacerými spôsobmi a my si uchovávame všetky. Teda ak sa na zastávku A vieme dostať v čase 6:00 peším presunom aj nejakou jazdou, nebudeme vedieť či máme brať minimálny čas na prestup do úvahy alebo nie. Preto ho budeme brať do úvahy vždy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linka r | t1 | t2 | t3 | t4 |
| A | 5:55 | 6:00 | 6:05 | 6:10 |
| B | ... | ... | ... | ... |

Tu sa zároveň vieme ukázať ako funguje mechanizmus hľadania vhodnej jazdy. V dátovej štruktúre sú jazdy linky zoradené podľa času príchodu na zastávku pre každý typ dňa kedy linka premáva. Pri hľadaní jazdy hľadáme smerom od konca poľa. Ak jazda vyhovuje podmienke, označíme ju ako vyhovujúcu a pokračujeme ďalej v prehľadávaní skorších jázd. Ak nájdeme jazdu, ktorá nevyhovuje podmienke ukončíme cyklus. Nájdená jazda zostane tá naposledy vyhovujúca. Index tejto jazdy si uložíme do poľa lastFoundTripsIndices v prípade, že by sme pre túto linku potrebovali nájsť ešte nejakú skoršiu možnú jazdu. Jazdy, ktoré prídu na zastávku neskôr ako označená jazda nás už nezaujímajú.

private SubrouteTrip getEarliestTripOperatingInServiceDay(Subroute subroute, int stopIndex, Time time, ServiceDay serviceDay, int minTimeForTransfer, boolean onlyLowFloor) {

…

for(int i = this.lastFoundTripsIndices.get(subrouteServiceDay); i >= 0; i--){

SubrouteTrip trip = subrouteTrips.get(i);

Time timePlusMinTransferTime = new Time(time).addMinutes(minTimeForTransfer);

Time tripStopTime = trip.getStopTimeObjects()[stopIndex].getTime();

if**(!onlyLowFloor || trip.isLowFloor()**) {

if (**tripStopTime.isBefore(timePlusMinTransferTime)**) { break; }

else { lastFound = trip; }

lastFoundTripsIndices.put(subrouteServiceDay, i);

}

}

return lastFound;

}

#### 3.3.2.4 Vyfiltrovanie len nízkopodlažných vozidiel

Informáciu o nízkopodlažných vozidlách sme nedostali v dátach statických cestovných poriadkoch ale náhodne sme si ich vygenerovali pri importovaní dát a teda každá jazda linky má priradenú informáciu o type vozidla.

V predchádzajúcom kóde môžeme vidieť podmienku, ktorá zabezpečí, aby v prípade nastavenia vyfiltrovania len nízkopodlažných spojov, algoritmus hľadal len jazdy s nízkopodlažnými vozidlami.

## 3.4 Frontend

-----------------------------------------------------unofficial---------------------------------------------------------------------------------

**AKO FUNGUJE A AKO SPÚŠŤAME**

Existujú dva profily: default a test.

Default profil pracuje s reálnymi dátami od Dopravného podniku Bratislava. Profil test používa testovacie dáta vytvorené na účely implementácie a testovania. Každý profil sa spája s inou databázou. Databázu je potrebné najskôr lokálne vytvoriť. Pre defaultný profil je potrebné vytvoriť databázu s názvom mhdBa a pre testovací profil mhdBaTest.

Ak chcem vytvoriť a naplniť databázu nastavíme spring.jpa.hibernate.ddl-auto=create-drop. A zavoláme request api/import. Ak je databáza už vytvorená nastavíme túto vlastnosť na validate.

V application.properties máme podľa profilu nastavenú cestu k GTFS dátam.

Môže sa stať, že algoritmus vyhľadá takú cestu, ktorá obsahuje dvakrát po sebe vyhľadá peší presun. Pešie presuny slúžia na prestup medzi dvoma linkami, preto nechceme umožniť, aby ponúklo dva pešie presuny po sebe. Pri hľadaní peších presunov, prechádzame označené zastávky (tie, ktoré boli v predchádzajúcom kole vylepšené). Ak nájdeme zastávku, do ktorej sa dá dostať v najlepšom čase len peším presunom, z takejto zastávky nehľadáme ďalšie pešie presuny. Čo znamená, že pešie presuny hľadáme len z takých označených zastávok, do ktorých sme sa v predchádzajúcom kole dostali v najlepšom čase nejakou jazdou.

public boolean previousRoundBestArrivalTimeReachedBySomeTrip(Stop stop){

if(!roundActions.containsKey(stop)){

return true;

}

TimeRound bestTimeRound = bestArrivals.get(stop);

return roundActions.get(stop).stream()

.anyMatch(ra -> ra.getRound().equals(ROUND -1)

&& ra.getArrivalTime().equals(bestTimeRound.getTime())

&& ra.getAction().getActionType().equals(ActionType.TRIP));

}

private void traverseTransfers(RaptorResults raptorResults, Set<Stop> improvedStops, int maxTimeOfWalking){

for(Stop stop: markedStops) {

if (**raptorResults.previousRoundBestArrivalTimeReachedBySomeTrip(stop)**) {

for (Transfer transfer :dataStructure.getDataStructureModel().getStopTransfers().get(stop)){

...

}

}

}s

}

Ak používateľ zadá začiatočný bod ako aktuálnu lokalitu pravá akcia, ktorú vykoná je peší presun. Tu je tiež potrebné ošetriť, aby sme v 1. kole nevyhľadávali pešie presuny.

...

**HĽADANIE CESTY**

1. predvyplníme prázdne results – popísať ako sme navrhli dátové štruktúry, často sa stáva, že porovnávané časy sú rovnaké. Asi by sme mali vytvoriť pre každý akciu. Aj sme vytvorili.

2 traverzujeme cez cesty a pešie presuny, naplníme results

3. vyfiltrujeme raptorResults- v roundActions zostávajú aj tie z prechádzajúcich kôl, ktorých časy už boli vylepšené ale tak ich budem ignorovať pri filtrovaní výsledných ciest. Filtrovať chcem aj také, ktoré síce majú rovnako dobrý čas ako najlepší, ale vznikli až v ďalšom kole. Takže tie:

- ktoré majú horší čas ako najlepší čas

- ktoré majú rovnaký čas ale vo vyššom kole

4. pathBuilder

Čo je prestup?

Ako sme impementovali RAPTOR (Zmeny)

Filtre

Ako funguje môj rRaptor? (results) návrh

Prečo nepoužijeme rRaptor ani k-najoptimálnejšich ciest?

Trip vyberáme aj podľa ServiceDay?

Ak neexistuje null ZMENA“!

1. Pod prestupom v našom prípade rozumieme zmenu medzi jednotlivými akciami. Pod akciou rozumieme peší presun FP, alebo nejaký úsek jazdy. TS tvorí postupnosť akcií. FP, TS je cesta, ktorá obsahuje jeden prestup. TS FP TS TS je cesta, ktorá obsahuje 3 prestupy. V algoritme je obmedzenie, aby mohli vzniknúť 2 pešie prestupy bezprostredne za sebou. Cesta, ktorá je vyhľadávaná z aktuálnej lokality bude vždy začínať peším presunom.

2. Najskôr si vytvoríme vyhľadávacie parametre pre algoritmus. Používateľské preferencie sú len celé čísla s ktorými neurobíme nič. Je potrebné spracovať čas a dátum začiatočné a konečné zastávky. Kedže vyhľadávame v časovom rozmedzí, ako vstup príde začiatočný dátum a čas a konečný dátum a čas, ktoré ohraničujú časové rozmedzie. Čas, ktorý získame ako reťazec prekonvertujeme do našej vlastnej triedy Time. Vlastná čas sme si vytvorili, pretože často potrebujeme mať maximálny čas, alebo polnoc, ktorý sa neviaže na dátum.Treba porovnávať len časy ako tak. Čo sa týka konečnej zastávky, je to jednoduché.Vstupom je pre mňa identifikačné číslo skupiny zastávok (stopArea), ktorá zoskupuje zastávkys rovnakým názvom. Po vytvorení tabuľky výsledkov raptor algoritmom pole zastávok danej stopArea bude slúžiť ako parameter pre vytvorenie ciest. Pri začiatočných zastávkach nastáva komplikovanejšia sitácia, kedže aplikácia ponúka ako možnosť zadanie štartovnej pozície aj aktuálnu polohu, je potrebné radiálnym vyhľadávaním nájsť všetky zastávky v okolí tejto zastávky. Ako polomer vyhľadávania sme si určili vzdialenosť 1,5km. S využitím DISTANCE MATRIX GOOGLE API nájdeme okolité zastávky. Parametrom pre raptor algoritmus nebude poloe začiatočných zastávok, ale mapa, pretože v prípade vyhľadávania z aktuálnej lokality si potrebujeme pamätať aj počet minút potrebných na presun z aktuálnej polohy po začiatočnú zastávku. V prípade vyhľadávania zo zastávky bude táto hodnota nastavená na 0min. Ak už máme vstupné parametre pre algoritmus, môžeme sa pustiť do samotného vyhľadávania. RAPTOR algoritmus nám vráti tabuľku, kde je pre každú zastávku zaznamenaný najlepší možný čas príchodu na túto zastávku. My si navyše

Problém: Zoradenie jázd v rámci linky

Najskôr sme sa rozhodli využiť atribút sequence order, ktorý je priradený ku každému objektu StopTime. StopTime okrem iného definuje v akom čase stojí jazda linky na zastávke. Atribút sequence order by mal určiť poradie zastávky v rámci jazdy a zároveň aj v rámci celého dňa. Prvá jazda linky na prvej zastávke má sequence order 1 a toto číslo sa navyšuje až po poslednú zastávku poslednej jazdy linky. Vieme teda vypočítať poradie jazdy v linke. Po hlbšom skúmaní sme zistili, že atribút sequence order nezoraďuje jazdy linky tak, ako by sme to potrebovali pre náš algoritmus. Zároveň sme zistili, že časy príchodu jazdy na zastávku sú v niektorých prípadoch väčšie ako 23:59:59 viď obrázok. Dokonca existujú také jazdy, ktoré prichádzajú na všetky zastávky linky až na druhý deň. Tieto výnimky v dátach nám po menšej analýze nedávali zmysel a kedže sme boli už v pokročilej fáze vývoja algoritmu, rozhodli sme sa že takéto dáta ignorovať. Z toho dôvodu vyhľadávanie na prelome dní nemusí správne fungovať.

V každom prípade potrebujeme mať zoradené jazdy linky tak, aby vyhovovali nášmu algoritmu a teda podľa času príchody na jednotlivé zastávky jazdy linky. Jazdy sa udržujú v štruktúre stopSubroutes, kde pre každú zastávku p existuje pole subroutes, ktoré na zastávke p stoja. Subroute je zoskupenie jázd s rovnakou postupnosťou zastávok v rámci linky. Subroute si udržuje pole jázd, ktoré sú indexované pomocou kľúča, ktorým je identifikačné číslo ServiceDay, v ktorý táto jazda operuje.

Toto pole jázd sme zoradili podľa času príchodu jazdy na zastávku.

TIME

S časom sme potrebovali pracovať úplne nezávisle od dátumu. Vytvorili sme si preto vlastnú triedu Time, ktorá má parametre: hours, minutes, seconds a nextDay. Trieda umožňuje skonštruovať čas z textového reťazca alebo definovať maximálny čas: 23:59:59. Časy vieme medzi sebou porovnávať, odpočítavať alebo pripočítavať k nim minúty. Pri konštruovaní času overujeme, či čas nepresiahol maximálny možný čas a ak áno priradíme parametru nextDay hodnotu true a od času odpočítame 24 hodín. Pri zoraďovaní jázd používame metódy triedy Time na porovnanie časov príchodu jazdy na prvú zastávku s prihliadnutím na parameter nextDay. Prvú zastávku má každá z jázd rovnakú, keďže patria do tej istej subroute.

TRIP FINDER

RAPTOR algoritmus funguje tak, že hľadáme najskoršiu vyhovujúcu jazdu po čase T, ktorá stojí na zastávke p a jazdí v niektorý z prívetivých typov dní. Hľadanie jazdy funguje tak, že začíname od poslednej jazdy v utriedenom poli. Spoliehame sa, že jazdy sú utriedené podľa času príchodu na prvú zastávku vzostupne. Kým nie je čas menší ako čas T, iterujeme jazdy smerom na začiatok poľa a do premennej lastTrip si ukladáme vždy aktuálnu jazdu. V momente keď narazíme na jazdu, ktorá na zastávke p stojí skôr ako je čas T, máme v premennej lastTrip vyhovujúcu jazdu. Môže nastať prípad, že všetky jazdy stoja na zastávke p skôr ako v čase T, preto sme museli pridať podmienku, ktorá overí, či príchod poslednej jazdy na zastávku p je neskôr ako čas T.

SPRACOVANIE MEŠKANIA

Dostávame sa k jadru našej práce a prispôsobenie algoritmu, aby prihliadal na dáta o meškaní. Ako sme už spomínali, na pozadí beží služba, ktorá zabezpečuje, že v dátovej štruktúre je pre každý StopTimeObject správna hodnota parametra delay pre aktuálny deň a aktuálny čas.

Táto služba sa spúšťa každú minútu. Zisťuje aký je dnes deň a podľa toho vyberá súbor s informáciami o meškaní. Hľadá v ňom také riadky, ktoré vznikli za predchádzajúcu minútu. Každý riadok nám poskytne informácie o meškaní: kedy meškanie vzniklo, na akej zastávke, na akej linke a aká je jeho hodnota. Po získaní zoznamu meškaní, zapracovávame takéto dáta do dátovej štruktúry. Konkrétne dáta vkladáme do štruktúry stopSubroutes, obsahujúcej aj StopTimeObject-y, ktoré si pre každú zastávku na každej jazde držia čas, kedy jazdy linky stojí na zastávke (time) a zároveň, s akým predpokladaným omeškaním príde jazdy linky na zastávku (delay).

Ako vyhľadáme jazdu?

Získanú hodnotu meškania v minútach sme priradili k získanej zastávke a ku všetkým zastávkam nasledujúcim za ňou.

Tento postup sa vykonáva každú minútu. Dátová štruktúra je teda každú minútu aktuálna.

Pri počítaní optimálnej cesty sme prihliadali na dve rôzne situácie:

- kedy používateľ zadal, pre vyhľadanie spoja dnešný deň

- kedy používateľ zadal pre vyhľadanie iný ako dnešný deň.

V prvom prípade prihliadame aj na dáta o meškaní v druhom prípade nás zaujímali len časy zo statických cestovných poriadkoch.

Keďže máme historické dáta z roku 2018, potrebovali sme nasimulovať situáciu, kedy používateľ zadá v parametroch vyhľadávanie pre dnešný deň. Vyhľadávanie funguje v dvoch režimoch:

1. kedy neprihliadame na dáta o meškaní

2. keď prihliadame na dáta o meškaní.

Prvý režim nastáva vtedy, keď používateľ vyhľadáva spoje pre