



УНИВЕРЗИТЕТ У НОВОМ САДУ
**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ**



Дејан Грубишић

**Систем за проналажење најкраће путање
између локација са тежинама путања
променљивим у реалном времену
заснован на архитектури Ламбда**

МАСТЕР РАД

Нови Сад, 2019



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6


КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР:			
Идентификациони број, ИБР:			
Тип документације, ТД:	Монографска документација		
Тип записа, ТЗ:	Текстуални штампани материјал		
Врста рада, ВР:	Мастер рад		
Аутор, АУ:	Дејан Грубишић		
Ментор, МН:	др Владимир Димитриески, доцент		
Наслов рада, НР:	Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда		
Језик публикације, ЈП:	Српски / ћирилица		
Језик извода, ЈИ:	Српски		
Земља публикавања, ЗП:	Република Србија		
Уже географско подручје, УГП:	Војводина		
Година, ГО:	2019		
Издавач, ИЗ:	Факултет техничких наука		
Место и адреса, МА:	21000 Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6		
Физички опис рада, ФО: (поглавља/страна/ цитата/табела/слика/графика/прилога)	6/45/0/21/19/0/0		
Научна област, НО:	Електротехничко и рачунарско инжењерство		
Научна дисциплина, НД:	Примењене рачунарске науке и информатика		
Предметна одредница/Кључне речи, ПО:	најкраћа путања, динамички граф, архитектура Ламбда, Апачи Спарк, велики скупови података		
УДК			
Чува се, ЧУ:	У библиотеци Факултета техничких наука, Нови Сад		
Важна напомена, ВН:			
Извод, ИЗ:	У овом раду представљен је систем за проналажење најкраћег пута између више чворова у графу са променљивим тежинама грана, заснован на архитектури Ламбда. Модули за пакетну обраду и обраду у реалном времену имплементирани су у технологији Апачи Спарк. Реализовани су такође и модул за визуелизацију, уз употребу Пајтон библиотеке Даш, и модул за генерисање нових тежина у графу. Складиште података се заснива на дистрибуираном фајл систему Хадуп, а комуникација између модула остварена је коришћењем система за размену порука Кафка. Све компоненте имплементиране су у програмском језику Пајтон и извршавају се унутар контејнера технологије Докер.		
Датум прихватања теме, ДП:			
Датум одбране, ДО:			
Чланови комисије, КО:	Председник:	др Соња Ристић, редовни професор	Потпис ментора
	Члан:	др Владимир Иванчевић, доцент	
	Члан, ментор:	др Владимир Димитриески, доцент	



KEY WORDS DOCUMENTATION

Accession number, ANO :			
Identification number, INO :			
Document type, DT :	Monographic publication		
Type of record, TR :	Textual printed material		
Contents code, CC :	Graduate-master Thesis		
Author, AU :	Dejan Grubišić		
Mentor, MN :	Vladimir Dimitrieski, PhD, docent		
Title, TI :	System for detecting the shortest path between multiple nodes in real-time based on Lambda Architecture		
Language of text, LT :	Serbian / Cyrillic		
Language of abstract, LA :	Serbian		
Country of publication, CP :	Republic of Serbia		
Locality of publication, LP :	Vojvodina		
Publication year, PY :	2019		
Publisher, PB :	Faculty of Technical Sciences		
Publication place, PP :	21000 Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovica 6		
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	6/45/0/21/19/0/0		
Scientific field, SF :	Electrical and Computer Engineering		
Scientific discipline, SD :	Applied Computer Science and Informatics		
Subject/Key words, S/KW :	shortest path, dynamic graph, Lambda Architecture, Apache Spark, big data		
UC			
Holding data, HD :	The Library of Faculty of Technical Sciences, Novi Sad, Serbia		
Note, N :			
Abstract, AB :	<p>In this paper, we present the system for finding the shortest path between multiple nodes in a dynamic graph. The system is based on the Lambda Architecture. Modules for batch and real-time processing are implemented in Apache Spark. Besides this, we implemented a module for vizualization by using Python Dash and a module for generating new weights on edges. The storage is built on top of the Hadoop Distributed File System and communication is based on the Kafka system for exchanging messages. All components are implemented in Python and executed inside Docker containers.</p>		
Accepted by the Scientific Board on, ASB :			
Defended on, DE :			
Defended Board, DB :	President:	Sonja Ristić, Full Professor, Ph.D.	
	Member:	Vladimir Ivančević, Assistant Professor, Ph.D.	Menthor's sign
	Member, Mentor:	Vladimir Dimitrieski, Assistant Professor, Ph.D.	

	УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Датум:
	ЗАДАТАК ЗА ИЗРАДУ МАСТЕР РАДА	Лист/Листова:

(Податке уноси предметни наставник - ментор)

Врста студија:	<input checked="" type="checkbox"/> Мастер академске студије <input type="checkbox"/> Основне струковне студије
Студијски програм:	Рачунарство и аутоматика
Руководилац студијског програма:	др Милан Видаковић, ред. проф.

Студент:	Дејан Грубишић	Број индекса:	E2 83/2018
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	др Владимир Димитриески, доц.		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА МАСТЕР РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;
- литература

НАСЛОВ МАСТЕР РАДА:

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

ТЕКСТ ЗАДАТКА:

- Проучити основне концепте пакетне обраде и обраде у реалном времену.
- Упознати се са архитектуром Ламбда и описати област примене, предности и мане.
- Идентификовати технологије за имплементацију архитектуре Ламбда и алгоритме за претрагу графа.
- Развити систем погодан за складиштење и анализу велике количине података, који омогућава проналажење најкраће путање између локација на географској мапи, са тежинама путања ажурираним у реалном времену.
- Систем имплементирати коришћењем софтверских контејнера и применом концепата архитектуре Ламбда.
- Реализовати приказ мапе и израчунатих путања између одабраних локација.

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

Садржај

1	Увод.....	3
1.1	Мотивација.....	4
1.2	Циљ и задаци	6
1.3	Структура рада	6
2	Преглед постојећег стања у области	7
3	Архитектура система	13
3.1	Архитектура Ламбда	13
3.2	Парадигма Мапирање-Редукција.....	16
3.3	Архитектура пројектованог система	17
4	Опис коришћених технологија	21
4.1	Контејнери у технологији Докер	21
4.2	Дистрибуирани фајл систем Хадуп	22
4.3	Апачи Спарк	24
4.3.1	Пакетна обрада у технологији Спарк.....	24
4.3.2	Обрада у реалном времену у технологији Спарк	26
4.4	Кафка	28
4.5	Зоо-кипер.....	29
4.6	Даш	30
5	Имплементација система за налажење најкраће путање	32
5.1	Покретање апликације	32
5.2	HDFS.....	32
5.3	Кафка	33
5.4	Уписивач у HDFS	34
5.5	Генератор	35
5.6	Модул за приказ	36
5.7	Пакетна Обрада	39
5.8	Обрада у реалном времену	42
6	Закључак	44
7	Литература.....	46
8	Биографија	48

Слике

Слика 1.1 Употреба претраге „више-један“ чвор код такси возила.....	5
Слика 1.2 Рутирање интернет саобраћаја	5
Слика 2.1 Врсте графова	7
Слика 2.2 Таксономија графовских алгоритама претраге	8
Слика 3.1 Модули архитектуре Ламбда.....	15
Слика 3.2 Ажурирање података у архитектури Ламбда	16
Слика 3.3 Пример Мапирање-Редукција: Налажење броја понављања речи у тексту ...	17
Слика 3.4 Шема пројектованог система	19
Слика 4.1 Архитектура HDFS система	23
Слика 4.2 Шема Трансформација и Акција над RDD	25
Слика 4.3 Операције над RDD.....	25
Слика 4.4 Архитектура система за извршавање Спарк.....	26
Слика 4.5 Структура обраде токова у технологији Спарк.....	27
Слика 4.6 Операције над интервалиима у технологији SparkStreaming.....	27
Слика 4.7 Кафка систем за размену порука	28
Слика 4.8 Архитектура технологије Кафка.....	29
Слика 4.9 Повезаност Зоо-кипера са Кафком	30
Слика 5.1 Задавање чворова за претрагу	37
Слика 5.2 Приказ пронађене путање.....	38

Листинзи

Листинг 5.1 Покретање апликације	32
Листинг 5.2 Повезивање са складиштем <i>HDFS</i>	32
Листинг 5.3 Ажурирање фајла из складишта <i>HDFS</i>	33
Листинг 5.4 Читање фајла из складишта <i>HDFS</i>	33
Листинг 5.5 Читање из складишта <i>HDFS</i> у делу за рачунање у реалном времену	33
Листинг 5.6 Инстанцирање читаоца система Кафка	33
Листинг 5.7 Инстанцирање издавача система Кафка	34
Листинг 5.8 Упис издавача на тему система Кафка	34
Листинг 5.9 Дефинисање контејнера модула <i>hdfs_writer</i>	34
Листинг 5.10 Псеудокод функције <i>main</i> и <i>consume</i> модула <i>hdfs_writer</i>	35
Листинг 5.11 Дефинисање контејнера модула генератор	35
Листинг 5.12 Покретање контејнера модула генератор	36
Листинг 5.13 Покретање модула генератор	36
Листинг 5.14 Дефинисање компоненти корисничког интерфејса	37
Листинг 5.15 Ажурирање приказа графа	39
Листинг 5.16 Скрипта за покретање модула за пакетну обраду и обраду у реалном времену	40
Листинг 5.17 Алгоритам за налажење свих путањи између задатих чворова модула за пакетну обраду	41
Листинг 5.18 Упис података у складиште <i>HDFS</i>	42
Листинг 5.19 Функција за обраду сигнала у делу за обраду у реалном времену	42
Листинг 5.20 Повезивање модула за обраду у реалном времену на тему система Кафка	42
Листинг 5.21 Обрада новокреираних ивица и ажурирање путања модула за обраду у реалном времену	43

1 Увод

Обрада велике количине података представља један од најважнијих захтева савременог рачунарства. Према директору компаније Гугл (енг. *Google*), свака два дана креира се приближно око пет егза-бајта података (2^{60} бајта), што је еквивалентно количини података сакупљеној од почетка цивилизације до 2003. године. Тренд стварања података непрестано се повећава и превазилази тренутне могућности њихове обраде. Како се обрадом података долази до корисних информација, постоји значајна потреба за стварањем алгоритама и система способних за обраду велике количине података.

Како би се обрадила велика количина података неопходно је имати умрежену хардверску инфраструктуру. У многим применама користи се високо специјализовани хардвер интегрисан у кластере рачунара. Овакви системи користе наменски пројектоване операције управљања задацима и обрадом података, чиме се остварују високе перформансе у погледу брзине извршавања. Развој оваквих система значајно је утицао на остваривање напретка и у осталим природним наукама, јер је захваљујући великој моћи обраде постало могуће моделовати и вршити експерименте над високо комплексним системима, као на пример онима који се проучавају у биологији и хемији.

Овакви рачунарски системи су по правилу изузетно скупи, због чега је њихова употреба значајно ограничена. Као алтернатива специјализованим кластерима настала је идеја о умрежавању хардвера опште намене (енг. *commodity hardware*), чија је јединична цена по правилу повољнија. Предност овог приступа је лако проширивање система, које се постиже једноставним додавањем нових рачунара у мрежу, при чему рачунари могу имати чак и различите конфигурације и оперативне системе. Умрежавањем рачунара створила се могућност да организације, у периоду ван радног времена, услужно изнајмљују рачунарске ресурсе за обраду велике количине података, без икаквих улагања.

Ипак, с порастом броја компоненти у систему значајно се повећава његова комплексност. Како би се омогућило лако управљање обрадом и обезбедио правилан рад сваке компоненте, користе се системи за модуларизацију система и изолацију компоненти у виду контејнера. Докер (енг. *Docker*) је најпознатија технологија за контејнеризацију и кориштена је у овом раду. Поред изолације и правилног рада компоненти, потребно је ефикасно распоређивати задатке по рачунарима и скалирати њихов број у зависности од потреба. Овакве функционалности пружају алати за оркестрацију као што су Кубернетес (енг. *Kubernetes*) и Мезос (енг. *Mesos*) који значајно убрзавају и поједностављују развој дистрибуираних апликација.

Услед велике комплексности обраде велике количине података неминовно долази до људских грешака и потребно је заштити податке од брисања. У случајевима када се чувају само агрегиране вредности података, може доћи до проблема када постоји грешка у коду, јер је тешко, а понекад и немогуће, реконструисати полазне податке. Начин да се ово превазиђе је чување података у изворном облику и покретање обраде над свим подацима. Овај приступ гарантује прецизност и поузданост, али најчешће

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

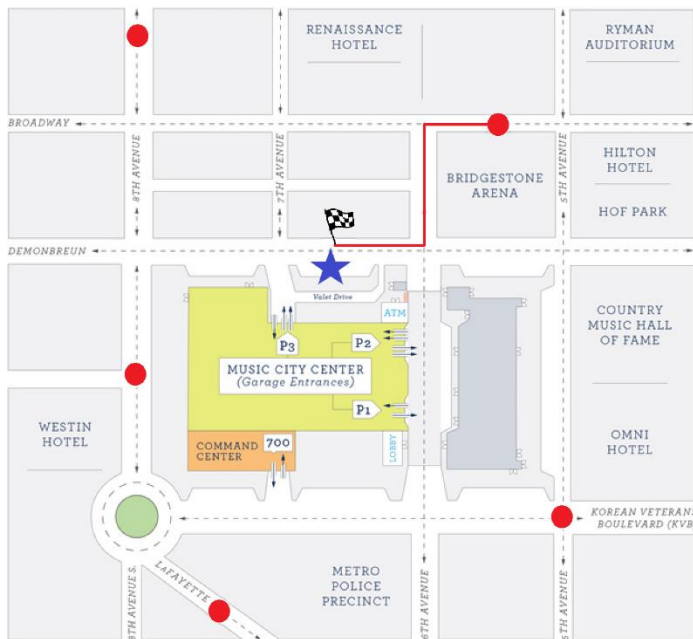
захтева много времена, што у применама са одзивом у реалном времену није прихватљиво. Да би се добио резултат у жељеном року често се користе апроксимативни алгоритми, који омогућавају брз одзив али на рачун прецизности. Ова два захтева била су основни мотив за стварање архитектуре Ламбда, која ће бити детаљно описана у следећим поглављима.

1.1 Мотивација

Многи проблеми, као што су анализа социјалних мрежа, аутоматизовање транспортних система и начин функционисања самог интернета, заснивају се на теорији графова. Структура типа графа омогућава дефинисање појмова и одређивање веза између њих. Компаније као што су Гугл, Фејсбук (енг. *Facebook*) и Јаху (енг. *Yahoo*), имају у својим системима стотине милиона корисника који свакодневно размењују поруке и генеришу податке. Ове компаније често на основу тих података спроводе анализе над графовима. Пример једне овакве обраде може бити груписање корисника према интересовању, како би им се пружале услуге препоручивања нових производа или налажење минималног броја корисника који су повезани са свим осталим корисницима у мрежи.

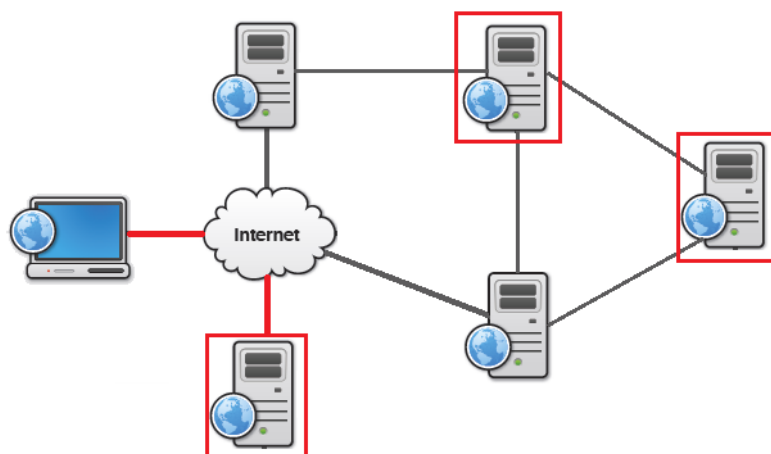
Теорија графова може бити примењена такође и у домену регулисања саобраћаја. Готово сви транспортни системи садрже навигацију засновану на проналажењу најкраћег пута на мапи. Као и у претходном случају, овакви графови су по својој природи динамички (промењиви у времену) и често је потребно ажурирати граф и давати одговоре на упите над графом у реалном времену. У овом случају, тежине грана графа могу се дефинисати као време потребно да возило стигне са једног места на друго. Како ово време зависи од гужве у саобраћају које је промењиво у времену, потребно је ажурирати граф и пронаћи нову најбржу путању. Налажење нове најбрже путање потребно је извршити у ограниченом временском интервалу, чиме је описани задатак још комплекснији.

Некада је потребно наћи место на мапи које задовољава одређени критеријум. Уколико би задатак био да се пронађе најближи ресторан, било би потребно наћи најкраћи пут између тачке на којој се корисник налази и локација свих објеката које се могу сврстати у ресторане. Са друге стране, постоје многе примене за које је потребно наћи најкраћи пут од више дефинисаних тачака до једне тачке. На пример, у случају информационог система за подршку такси служби, потребно је одредити које возило може најбрже да стигне до корисника. У једном тренутку може више такси возила бити разматрано за вожњу, али само најближи би требало да добије право на вожњу. Сличан проблем јавља се и код обезбеђивања објеката или пружању прве помоћи, када су на више различитих места постављене екипе, од којих најближа треба да дође на задато место.



Слика 1.1 Употреба претраге „више-један“ чвор код такси возила

Налажење најкраћег пута у графу интернет комуникација и рутирање података према задатим чворовима данас представља важан проблем преноса података на интернету. Да би се обезбедила што већа пропусност мреже, подаци се рутирају према најближим серверима. Како се густина саобраћаја врло брзо мења, било би неопходно пратити и ажурирати путању у току преноса порука. Поред овога, претрага би морала да даје одзив у реалном времену како би овај приступ имао употребну вредност. Овај проблем је идејно веома сличан претходним проблемима, међутим, разлику представља огроман број чворова и грана у графу интернета, због чега је за његову обраду потребно много процесорског времена, а како је потребно дати и одговор у реалном времену, овај проблем представља велики изазов.



Слика 1.2 Рутирање интернет саобраћаја

Постоје још многе примене обраде динамичких графова великих димензија као што су научна израчунавања интеракције молекула и биолошких система. У свим наведеним

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

применама налажење најкраће путање у графу представља један од основних алгоритама који се често користи и као основа за решавање осталих графовских проблема. Управо је ово била мотивација за развој система за налажење најкраће путање и писање овог рада.

1.2 Циљ и задаци

Циљ овог рада био је пројектовање система за проналажење и ажурирање најкраће путање у реалном времену у графу великих димензија са динамички променљивим тежинама грана. Да би се ово постигло, пројектован је систем заснован на архитектури Ламбда. За складиштење података користи се дистрибуирани фајл систем Хадуп (енг. *Hadoop Distributed File System, HDFS*), док се као основна технологија за обраду података користи Апачи Спарк (енг. *Apache Spark*). Компоненте система су развијене коришћењем контејнера технологије Докер (енг. *Docker*), чиме је омогућен независан развој сваке компоненте. За визуелизацију је коришћена библиотека Пајтон Даш (енг. *Python Dash*) која омогућава приказ графа и ажурира га у зависности од генерисања нових тежина.

1.3 Структура рада

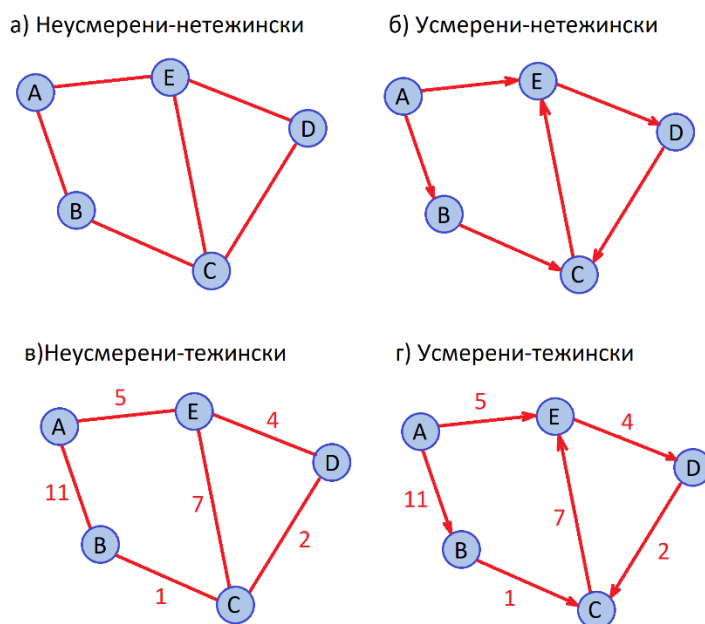
Овај рад састоји се из укупно шест поглавља. У првом, уводном поглављу дата је мотивација за пројектовање система за тражење најкраће путање и мотивација за коришћење архитектуре Ламбда. У другом поглављу дато је стање у области алгоритама за претрагу графова. Треће поглавље посвећено је опису архитектуре Ламбда и парадигми Мапирање-Редукција на којој је заснована, као и архитектура пројектованог система. У четвртом поглављу може се наћи опис коришћених технологија. Пето поглавље посвећено је имплементацији система. Шесто поглавље представља закључак у коме су елаборирани добијени резултати.

2 Преглед постојећег стања у области

У овом поглављу дат је кратак преглед појмова теорије графова и биће приказани алгоритми за налажење најкраће путање.

Граф у општем случају представља структуру чворова који су међусобно повезани гранама. Чворови и гране графа могу имати додељене особине. Са становишта тежина, графови могу бити нетежински и тежински. У нетежинским графовима свака ивица и чвор имају јединичну вредност. Код тежинских графова, гране могу имати различите тежине које најчешће означавају удаљеност у неком простору. Сами чворови такође могу имати тежине. Вредности тежина могу бити или само позитивне, или и позитивне и негативне, што може утицати на избор алгоритма за најкраћу путању.

Гране могу бити унидирекционе или бидирекционе, а код хиперграфова, једној грани може бити придружено више од два чвора. Граф може бити статички, што значи да се тежине и гране графа не мењају, односно динамички, у ком случају је могућа промена тежина путања унутар графа, као и додавање и брисање грана. Алгоритми се такође разликују према дозвољеном времену извршавања. Уколико је време ограничено често се користе хеуристике и апроксимативни алгоритми, док се у другом случају могу користити тачни алгоритми.



Слика 2.1 Врсте графова

Како графови имају велику примену у моделовању мапа, просторни графови имају додељене координате за сваки чвор, што се може користити као хеуристика за одређивање правца претраге. Непросторни графови немају интерпретацију чворова у вези са простором, док планарни графови представљају графове који могу да се напишу у равни тако да се ниједна грана не сече. Поред овога, графови се према густини деле на густе (јакно повезане) и ретке (слабо повезане). Просторни и планарни

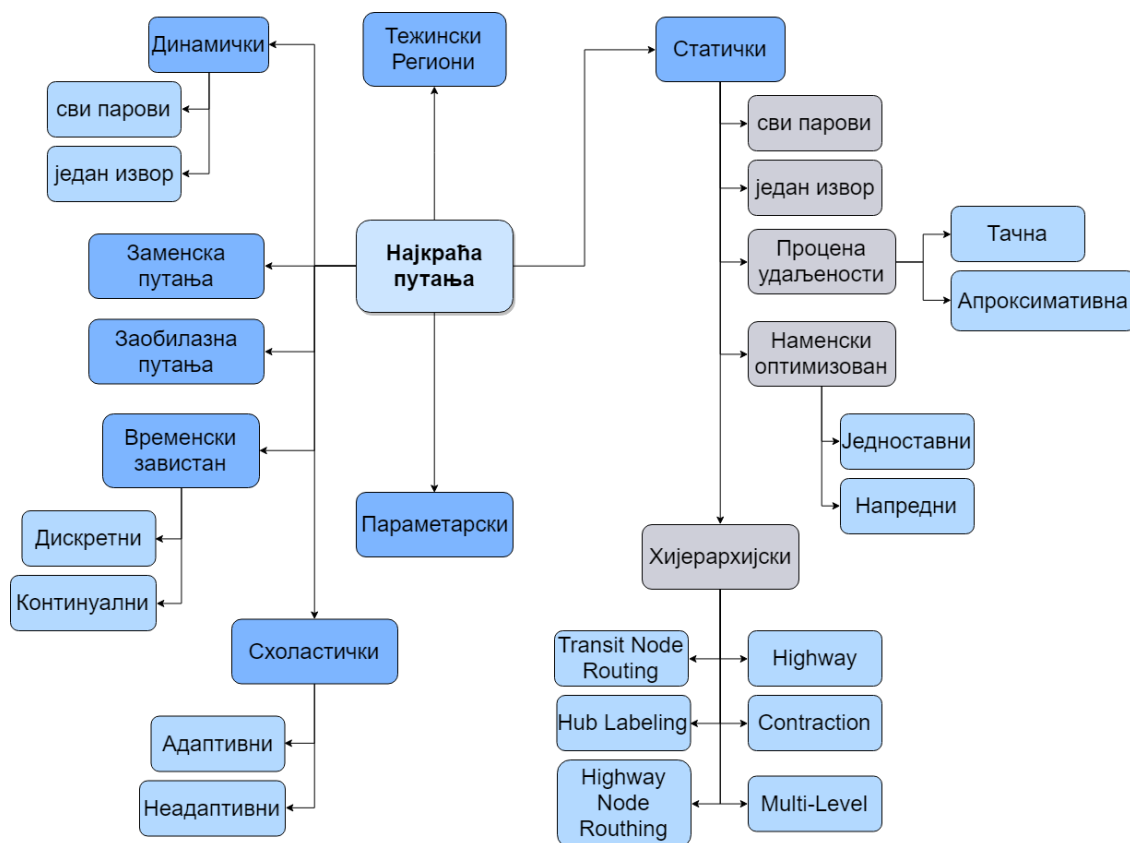
Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

графови, који означавају мапе обично су ретки и локално повезани, док графови који означавају социјалне мреже и интернет конекције могу имати значајно већи број грана у односу на чворове.

Проблем претраге графа представља један од основних и највише изучаваних проблема теорије графова. Овај проблем јавља се у многим областима, због чега су пројектована решења оптимизована према различитим врстама графова и према времену претраге.

Један од првих алгоритама за налажење најкраћег пута у графу описан је 1956. године од стране Дијкстре [1]. Овај алгоритам односи се на графове са ненегативним тежинама грана и његова временска комплексност је $O(E + V \cdot \log V)$, при чему је E број грана и V број чворова у графу. У случају да граф садржи негативне тежине грана, алгоритам Белман-Форд [2] [3] даје најкраћу путању. Овај алгоритам има временску комплексност $O(E \cdot V)$. Уколико је потребно пронаћи најкраће путање између свака два чвора у графу, користи се алгоритам Флојд-Варшал [4] [5].

Различити проблеми захтевали су наменске типове графова, према којима су оптимизовани алгоритми за њихову претрагу. Због велике потребе за брзом манипулацијом над графовима, развијене су многе групе алгоритама и њихова таксономија [6] је дата на Слици 2.2.



Слика 2.2 Таксономија графовских алгоритама претраге

Алгоритми за налажење најкраће путање разликују се према циљу претраге на оне који за циљ имају налажење путање између једног и свих осталих чворова (један извор) или

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

налажење путања између свих чворова (сви парови). Поред овога, у статичким алгоритмима се често ради предпроцесирање, како би се обезбедили брзи одговори на упите. У многим применама је од значаја проценити удаљеност у задатом времену са одговарајућом прецизношћу. У ту сврху се користе алгоритми за процену удаљености чвора, који могу бити тачни или апроксимативни.

Наменски оптимизовани алгоритми у фази предпроцесирања израчунавају додатне информације које се придружују чворовима, како би се приликом обраде користиле хеуристике засноване на тим информацијама. Ови алгоритми деле се на једноставне, који укључују информације о процењеној удаљености од сваког чвора до циља [7] и на напредне који укључују информације унутар ивица и поделу графа на једнаке целине.

Хијерархијски статички алгоритми односе се на груписање чворова и грана према значајности у хијерархију. Пример овога може да буде издвајање путева по величини на некој мапи (енг. *Highway*) [8]. Ауто-путеви, булевари, асфалтиране улице и земљани путеви припадали би посебним нивоима хијерархије и приликом претраге графа полазило би се од највишег нивоа хијерархије, прелазећи на ниже нивое када се постигне одређена близина према циљу.

Редукована слика графа (енг. *Contraction*) се односи на креирање грана које замењују делове графа између два чвора, са истом тежином пута између њих. На овај начин итеративни алгоритми пролазе кроз мање грана и по проналажењу најкраће путање, оригиналне гране се рекреирају из изведених.

Граф са више слојева (енг. *Multi-layer graph*) добија се спајањем суседних чворова како би се повећала грануларност графа. По проналажењу путање, оригинални делови графа се поново конструишу.

Рутирање транзитног чвора (енг. *Transit Node Routing*) односи се на израчунавање најкраће путање између одређених тачака, чиме је приликом претраге довољно доћи до две транзитне тачке, најближе почетном и најближе крајњем чвору.

Обележавање чворова (енг. *Hub Labelling*) заснива се на креирању табеле директне и повратне повезаности. Табеле директне повезаности за сваки чвор дефинишу удаљености од одређеног скупа чворова. У случају повратне повезаности за неке скупове чворова, дефинише се удаљеност до крајњег чвора. Комбиновањем ова два приступа и минимизовањем удаљености добија се најкраћа путања између два чвора [9].

Рутирање чворова аутопута (енг. *Highway Node Routing*) односи се на бирање грана које повезују различита суседства чворова [10]. Суседства представљају скуп чворова удаљених k корака од неког чвора. Уколико постоји грана између два чвора која припадају различитим суседствима, та грана се проглашава аутопутем, а чворови мање кардиналности се редукују. На овај начин се рекурзивно добијају нивои хијерархије.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Тежински региони (енг. *Weight Regions*) представљају области у равни у којој је потребно наћи најкраћи пут између две тачке. За разлику од претходно описаних графова, у овом случају нема ограничења како ће путања бити постављена. За сваки од ових региона дефинисана је ненегативна вредност тежине по јединичној удаљености. Овај проблем дефинисан је у раду [11].

Параметарски алгоритми за проналажење најкраћег пута су они у којима је тежина грана или нека друга особина графа зависна од неке променљиве која се мења. Најкраћа путања је у овом случају зависна од параметра и потребно је израчунати најкраћу путању за сваку могућу вредност параметра.

Динамичка претрага најкраће путање израчунава најкраћу путању у графу у коме се тежине грана креирају, бришу и мењају у времену. Као и у случају претраге статичког графа, налажење путања може бити између једног и више чворова или између свих чворова. За налажење најкраће путање између свих чворова неусмереног бестежинског графа најбржи су апроксимативни алгоритми су Родити-Цвик [12] и алгоритам предложен од Хензингерове, Кринингера и Нанонгкаија [13] са временским комплексностима ажурирања путање респективно $O(m \cdot n / \epsilon)$ и $O(n^{5/2})$, где је n -број чворова, m -број грана и ϵ -фактор апроксимације. Време упита за оба алгоритма је константно. Најбржи детерминистички алгоритам предложен је такође од аутора Хензингер, Кринингер и Нанонгкаи [13], са временском комплексношћу $O(n \cdot m)$, и одзивом упита $O(\log \log n)$.

Што се тиче динамичких алгоритама од једног до осталих чворова, издвајају се алгоритми Факаренпол-Рао [14] над планарним графовима са реалним тежинама грана са временском комплексношћу $O(n \cdot \log^3 n)$ и одзивом упита $O(n^{4/5} \cdot \log^{13/5} n)$, Бернштајн-Родити [15], који остварују временску комплексност ажурирања $O(n^{2+O(1)})$ на ретким графовима. Алгоритам Херзингерове, Кринкера и Нанонгкаиа [16] додатно још унапређује алгоритам Бернштајна и Родитија на временску комплексност $O(n^{1.8+O(1)} + m^{1+O(1)})$.

Заменска путања (енг. *Replacement Path*) односи се на рачунање најкраћег пута који заобилази задату грану за сваку грану која припада најкраћем путу. У планарним усмереним графовима алгоритам Емека, Пелега и Родитија [17] остварује временску комплексност од $O(n \log^3 n)$ за предпроцесирање и $O(h \cdot \log \log n)$ за одзив упита, при чему је h -број грана у новој путањи. Додатно, овај алгоритам може да налази и заменске путање које не садрже задате чворове. Бернштајн је предложио апроксимативни алгоритам [18] за заменске путање над тежинским графовима има временску комплексност $O(m \cdot \log(nC/c)/\epsilon)$ при чему C/c представља однос највеће и најмање тежине у графу, а ϵ је фактор апроксимације. У случају одређивања k путања овај алгоритам остварује временску комплексност од $O(k \cdot m \cdot \sqrt{n})$.

Проблем проналажења заобилазне путање (енг. *Alternative Path*) врло је сличан налажењу заменских путања, с том разликом што корисник мора да специфицира нежељену грану или чвор на почетку. У овој области алгоритам Ксиа и осталих [19]

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

остварује временску комплексност од $O(m \cdot n(\gamma, L)n^{1.5})$, са одзивом на упит од $O(K)$, при чему је K је број чворова у израчунатој путањи, L је пречник графа и γ је коефицијент структуре мреже.

Временски зависни графови (енг. *Time-Dependant Graphs*) су графови у којима је тежина грана дата као функција од времена. У овој таксономији временски зависни графови деле се на континуалне и дискретне.

У континуалним временски зависним графовима алгоритам Динга и осталих [20] решава проблем најмањег времена путовања у динамичком временски зависном графу великих димензија. Просторна комплексност овог алгоритма је $O((n + m)\alpha(T))$, док је временска комплексност $O(\alpha(T) \cdot (n \cdot \log n + m))$, при чему је $\alpha(T)$ функција тежине грана у зависности од времена.

У дискретним временски зависним просторним графовима Демириурек и остали у предложили алгоритам [21] базиран на бидирекционој претрази временски зависног графа уопштавајући алгоритам A^* . Овај алгоритам у првом кораку дели граф на непреклапајуће целине и у другом кораку израчунава најмање време чворова према крајевима целине. У поређењу са алгоритмом ALT заснованом на коришћењу хеуристика са путоказима (енг. *landmarks*), овај алгоритам има значајно боље просторне карактеристике и време извршавања.

Стохастички (енг. *Stochastic*) алгоритми се односе на употребу дистрибуција вероватноће и насумичних променљивих у процени најкраћег пута. Два правца истраживања у овој области су адаптивни и неадаптивни алгоритми.

Код адаптивних алгоритама претпоставља се која би следећа грана требала да буде разматрана за најкраћу путању. Алгоритам Милер-Хоксове и Махмасанија [22] разматра најмање очекивано време пута између једног чвора и свих осталих чворова. Николова и остали су предложили алгоритам [23] у ком свака грана има додељену насумичну дистрибуцију тежина и циљ је повећати вероватноћу да дужина путање не прелази задат границу. За нормалну дистрибуцију тежина добијена је временска комплексност од $n^{\Theta(\log n)}$.

Неадаптивни алгоритми имају за циљ смањење дужине путање. Николова и остали су алгоритам [24] за оптимално планирање путање у графу са насумичном дистрибуцијом тежина. Путања и време поласка су узајамно оптимизујући јер се уводе пенали за путање са прераним и касним временом. Неки случајеви оптимизације могу се редуковати на класичне алгоритме претраге најкраћег пута, док је налажење оптималне путање у функцији почетног времена NP тежак проблем. За фиксно време поласка развијени су полиномијални алгоритми у функцији броја чворова.

Циљ овог рада је имплементација алгоритма који налази све путање између задатих чворова графа у којем се путање ажурирају променом тежина грана у реалном времену. Налажење свих путања спада у класу статичких алгоритама између свих парова за који

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

нису кориштене хеуристике засноване на семантици чворова, јер је замишљено да алгоритам ради за сваки граф.

Алгоритам за налажење свих путања заснива се на двостраној итеративној претрази, која подразумева пролазак кроз све чворове у пречнику N од почетних и крајњих чворова у колико је N број итерација. Иако овај алгоритам није оптималан јер пролази кроз све чворове овај приступ одабран је јер се лако скалира и могуће је ограничити претрагу на регион у ком се налазе почетни и крајњи чворови. Када су једном израчунате све путање у региону просто ажурирање тих путања са новим тежинама грана доводи до налажења најкраће путање, што може бити урађено у времену $O(NewEdges \cdot Paths / p)$, где $NewEdges$ представљају број ажурираних грана, $Paths$ означава број путања и p је број процесора.

3 Архитектура система

У овом поглављу су најпре описани основни концепти архитектуре Ламбда, која је служила као референтна архитектура за пројектовани систем за налажење најкраћих путања. Потом је дат и сам опис компоненти пројектованог система, њихова међусобна повезаност, као и начин рада. У наредним поглављима биће детаљније описана свака компонента система, као и технологија коришћена за имплементацију дате компоненте.

3.1 Архитектура Ламбда

Архитектура Ламбда представља архитектуру система за обраду великих скупова података при чему су загарантовани робусност система и одговор у реалном времену. Ова архитектура настала је као одговор на изазове обраде велике количине података, при чијој обради традиционалне технике, попут трансакционе обраде, нису успеле да дају задовољавајуће резултате.

Захтеви да систем буде скалабилан и робустан у исто време, представљају велики проблем из разлога што се повећавањем броја процесних јединица и меморије повећава и шанса да поједини делови система откажу. Поред овога, повећавањем ресурса повећава се и комуникација између компоненти система, чиме читав систем постаје значајно комплекснији за пројектовање и одржавање. У оваквим системима неминовно долази до људских грешака и неопходно је да пројектовани систем поседује сигурносне механизме, како не би дошло до губитка података и трајне штете.

Ово су управо били проблеми који су довели до увођења архитектуре Ламбда. Архитектура Ламбда је предложена 2015. године од стране Џејсона Марта и Џејмса Варена, који су у књизи „Принципи и употреба скалабилних система за обраду велике количине података у реалном времену“ [25]. Основна идеја заснива се на чињеници да је упит функција над свим подацима у изворном облику, односно подацима који се не могу извести из других података. Чување и обрада над изворним подацима омогућавају робусност система, јер спречавају губитак података услед грешке при агрегирању података, што се често користи код инкременталних система у циљу смањењеног коришћења меморије.

С друге стране, савремени системи често захтевају одговор на упите над великом количином података у реалном времену. У оваквим условима, обрада над свим подацима, захтевала би неприхватљиво много ресурса и не би се испоштовао временски критеријум. Да би се одговор добио у задатом року, подаци се често унапред прерачунавају, како би се добиле сумиране информације, које би могле да се искористе за одговор на упит уместо проласка кроз цео слуп података.

Пример који је дат у поменутој књизи је израчунавање броја посетилаца веб странице у одређеном временском интервалу. Ако било потребно да се добије одговор на питање „Колики број посетилаца је посетио веб сајт у последњих 6 месеци?“, при чему се број посетилаца евидентира на сваки сат, морале би да се сумирају вредности сваког сата за

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

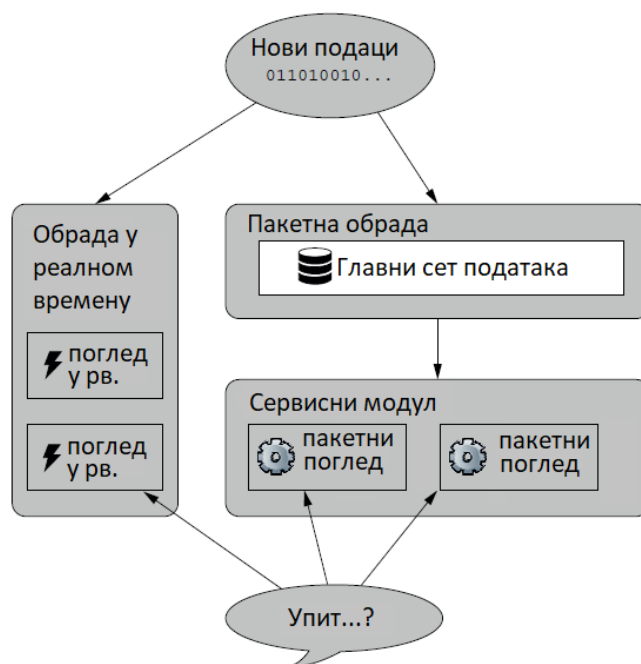
6 месеци. Бржи начин да се добије одговор био би да се поред броја посетилаца за сваки сат прерачуна и њихова сума по дану и месецу, чиме би се комбиновањем сумираних и основних података у неколико корака дошло од решења за било који задат временски интервал.

Проблем код сумирања информација може бити величина потребних меморијских ресурса [25]. У случајевима када је потребно пронаћи број јединствених корисника у току задатог времена, потребно је поред броја јединствених корисника на нивоу сата, чувати и вредности које идентификују кориснике. Број јединствених корисника на нивоу дана, не може се добити простим сумирањем броја јединствених корисника по сату, већ је потребно пронаћи унију идентификатора јединствених корисника за сваки сат. На овај начин меморија за складиштење сумираних података може превазићи почетни скуп изворних података.

Решење за овакав проблем може бити коришћење апроксимативних алгоритама, који захтевају знатно мање додатних меморијских ресурса и могу у кратком времену да предвиде резултат са одређеном тачношћу. У случају броја јединствених посетилаца, алгоритам Хипер-лог-лог (енг. *HyperLogLog*) [26], може да предвиђа са 98% тачности и око 1KB додатне меморије по идентификатору корисника и интервалу од 1 сата. Овакви резултати изузетно су погодни за велики број примена обраде у реалном времену, јер је често циљ да се добије процена у што краћем року а не потпуно тачна вредност.

Архитектура Ламбда комбинује технике сумирања података и апроксимативних алгоритама како би обезбедила брз одзив, док са друге стране обезбеђује робусност и поновљивост израчунавања. Архитектура Ламбда се састоји из модула за пакетну обраду (енг. *Batch layer*), модула за обраду у реалном времену (енг. *Real-time layer*) као и модула за постављање и одговарање на упите (енг. *Serving layer*) (Слика 3.1).

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда



Слика 3.1 Модули архитектуре Ламбда

Пакетна обрада изводи се над читавом колекцијом података и помоћу ње се генеришу сумирани подаци у виду пакетних погледа. Како је реч о великим количинама података, потребно је обезбедити дистрибуирану обраду, са могућношћу лаког скалирања и отпорности на грешке. Оваква парадигма назива се Мапирање-Редукција (енг. *map-reduce*) и биће објашњена у наставку. Подаци који се користе приликом обраде, складиште се у блоковима, чиме се лако управља великим скуповима података. Употреба блокова, са друге стране доводи до немогућности за брз приступ и ажурирање појединачних података. Ово такође значи да коришћењем пакетне обраде неће бити могуће обрадити податке пристигле од њеног покретања, чиме ће њен резултат увек бити застарео. Из тог разлога уведена је обрада у реалном времену.

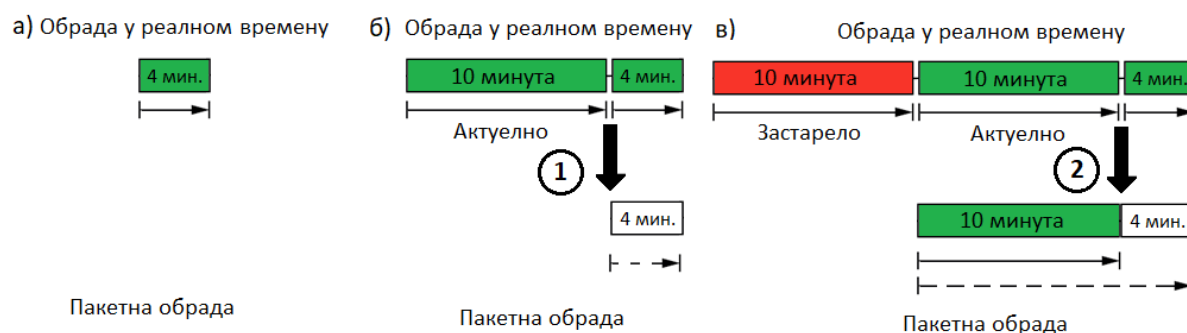
Обрада у реалном времену омогућава брз упис и читање појединачних података и користи се за обраду токова података који улазе у систем. Захваљујући брзом приступу меморији, могуће је да се у кратком временском интервалу дође до одговора са задовољавајућом прецизношћу, коришћењем апроксимативних и инкременталних алгоритама. Као резултат обраде генеришу се погледи у реалном времену, који омогућавају упите над тек пристиглим подацима. Ипак, величина података, која на овај начин може да се обрађује знатно је мања од података пакетне обраде.

Да би се добио систем који обрађује велику количину података, који непрестано пристижу у реалном времену, ова два принципа користе се заједно у архитектури Ламбда. Пакетна обрада покреће се изнова над свим подацима, док се обрадом у реалном времену обрађују подаци пристигли од последње пакетне обраде. По завршетку пакетне обраде ажурирају се пакетни погледи, док обрада у реалном времену почиње да прикупља нове податке од тог тренутка.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Сервисни модул служи за индексирање и приступ погледима израчунатим у пакетној и обради у реалном времену. Захваљујући сталном покретању пакетне обраде, апроксимативни резултати, добијени у модулу за обраду у реалном времену, бивају замењени резултатима пакетне обраде, чиме се гарантује коначна тачност система. Ова чињеница омогућава коришћење грубих апроксимативних алгоритама који се брзо израчунавају, без опасности да грешке апроксимације остану у систему.

Ако се узме у обзир да на почетку обраде нема података, тада се покреће обрада у реалном времену која обрађује пристигле податке (Слика 3.2-а). У исто време, сервисни слој има приступ једино погледу реалног времена који је до тада генерисан. Када се скупи задата количина података, покреће се пакетна обрада над подацима који су стигли у претходном кораку, што је илустровано на Слици 3.2-б. Обрада у реалном времену наставља да обрађује пристигле податке и генерише погледе, све док се не заврши пакетна обрада и не добију пакетни погледи.



Слика 3.2 Ажурирање података у архитектури Ламбда

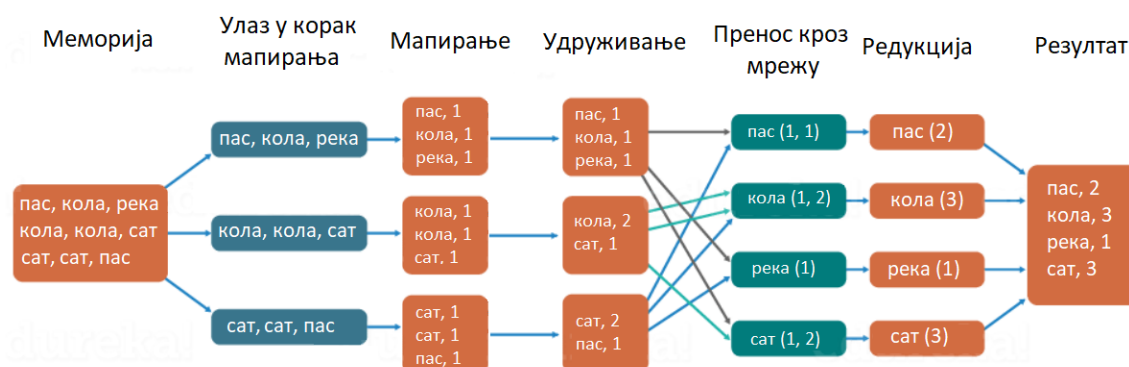
По завршетку прве пакетне обраде, погледи у реалном времену за првих 10 минута замењују се израчунатим пакетним погледом. Свака следећа пакетна обрада покреће увек над читавим скупом података у систему, а не само над пристиглим подацима. Обрада у реалном времену такође наставља са радом и након 4 минута, сервисни слој располаже са првим пакетним погледом и других 14 минута погледа у реалном времену (Слика 3.2-в). Следећи овај принцип, поглед у реалном времену садржи валидне податке добијене од последње пакетне обраде до тренутка упита.

3.2 Парадигма Мапирање-Редукција

Мапирање-Редукција представља парадигму пакетне обраде, која се састоји из корака мапирања и редукције. У првом кораку, сваком податку из скупа података се додељује вредност, након чега се подаци групишу, како би се на њима извршио корак редукције у коме се ради коначна евалуација. У оваквим системима подаци се чувају у дистрибуираним фајл системима, при чему се операција мапирања врши на рачунарима са подацима, како би се смањио број размењених порука. При решавању проблемима који су по својој природи асоцијативни, могуће је након корака мапирања извршити корак удруживања (енг. *Combine*) чиме се врши агрегација података пре њиховог пре слања у корак редукције. Овај корак значајан је за смањивање

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

комуникације у систему. Типичан пример парадигме мапирање-редукција је налажење броја понављања речи у тексту (Слика 3.3).



Слика 3.3 Пример Мапирање-Редукција: Налажење броја понављања речи у тексту

У првом кораку овог примера, делови текста се прослеђују корацима за мапирање. Свакој се речи потом у кораку за мапирање придружује број 1. У кораку удруживања се ови бројеви сумирају за сваку реч, како би се добио број понављања речи у добијеном делу текста. Након овога се свака реч заједно са бројем понављања прослеђује кроз мрежу, чворовима за израчунавање редукције. У кораку редукције се поново сумирају бројеви понављања речи у прослеђеним деловима текста, чиме се добија број укупног појављивања.

Уколико би требало да се обради већа количина текста, било би креирано више корака за мапирање и редукцију, који би се извршавали у паралели. Из овога се може видети велика могућност скалирања, потребна за обраду велике количине података.

Парадигма Мапирање-Редукција, изворно је настала у компанији Гугл (енг. *Google*). Најпознатија имплементација ове парадигме је у технологији Хадуп, фондације Апачи (енг. *Apache Hadoop*), док је она усвојена и у другим технологијама за обраду велике количине података. У овом раду коришћена је технологија Спарк фондације Апачи (енг. *Apache Spark*), због низа предности које ће бити дискутоване у наредном поглављу.

3.3 Архитектура пројектованог система

Како би се читалац што боље упознао са архитектуром система, дата је још једном спецификација проблема.

Потребно је пронаћи и ажурирати најкраћу путању у реалном времену у графу великих димензија са динамички промењивим тежинама грана.

Овакав задатак указује на потребу за дистрибуираном обрадом, која ће у исто време омогућити одзив у реалном времену. Управо из ових разлога, пројектована

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

архитектура заснована је на архитектури Ламбда, која омогућава обраду велике количина података уз одзив у реалном времену.

Основна идеја за решавање задатог проблема била је раздвајање функционалности на два дела:

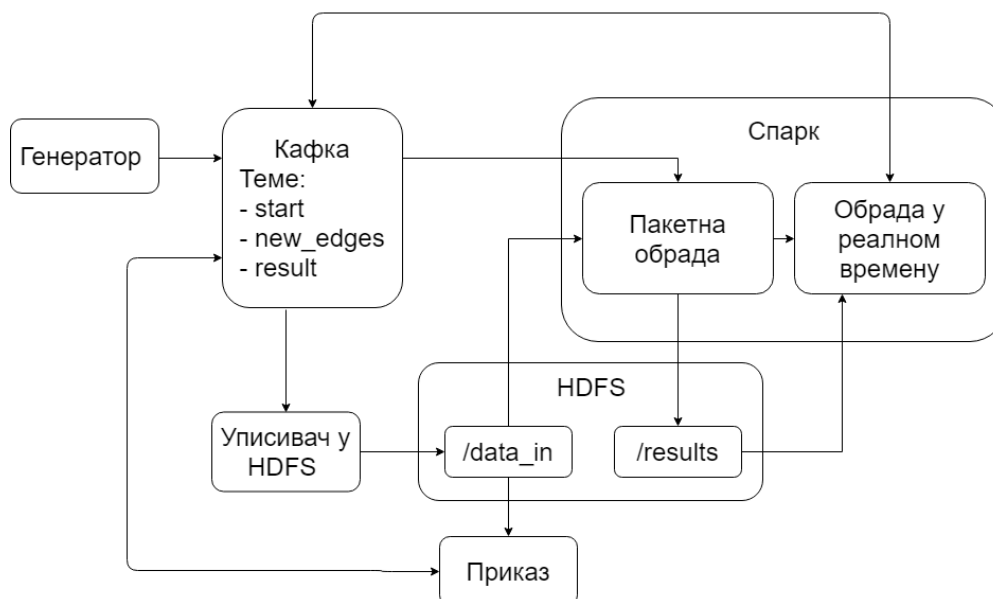
- 1) налажење свих путања између задатих чворова
- 2) ажурирање најкраћих путања у односу на нове ивице у графу

Овакав приступ омогућава да проблем обраде велике количине података у реалном времену, буде подељен на два мања проблема, где ће сваки од два задатка имати само један од ових захтева. Први задатак односи се на налажење свих путања између два или више чворова и њихово сортирање. Оваква обрада, захтева интензивно рачунање и дистрибуирану обраду, што одговара ономе што извршава модул за пакетну обраду архитектуре Ламбда.

Са друге стране, задатак који се односи на ажурирање најкраћих путања, одговара обради токова података и може се свести на ажурирање свих постојећих путања и њихово сортирање. Било која промена тежине ивице, која би могла да утиче на најкраћу путању, биће садржана у скупу путањи. Чак и уколико се генерише сасвим нова грана или чвор у графу који могу да утичу на најкраћу путању, њихов утицај биће урачунат приликом следеће пакетне обраде. У многим применама, операција додавања сасвим нове гране у граф је ретка операција, због чега ажурирање са следећом пакетном обрадом не представља проблем.

За складиштење почетног графа, као и резултата пакетне обраде користи се дистрибуирани фајл систем Хадуп (енг. *Hadoop Distributed File System, HDFS*), чије ће карактеристике бити описане у наредном поглављу. На почетку обраде изворни подаци графа уписују се у директоријум */data_in*, након чега се учитавају у блокове за пакетну обраду и приказ графа (Слика 3.4 Шема пројектованог система) Блок за упис такође има задатак да у току целе обраде преузима нове информације о тежинама ивица из генератора и ажурира њихове вредности у *HDFS* складишту.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда



Слика 3.4 Шема пројектованог система

Након учитавања података у директоријум `/data_in`, блок за приказ генерише слику графа на локалном серверу и пружа кориснику могућност да зада између којих чворова је потребно израчунати најкраћу путању. Апликација за приказ потом шаље листу задатих чворова брокеру модула Кафка на тему: `start`, из које блок за пакетну обраду преузима команде и покреће итеративну претрагу путања између задатих чворова.

Пакетна обрада заснована на парадигми мапирање-редукција имплементираној у технологији Апачи Спарк (енг. *Apache Spark*). Овај модул креира списак најкраћих путања и добијене резултате смешта у датотеку `/results` складишта HDFS. Након овога, шаље се сигнал модулу за обраду у реалном времену да преузме резултате пакетне обраде. По завршетку, пакетна обрада покреће се поново и цео процес почиње испочетка. Пакетна обрада такође поседује механизме да прекине тренутну обраду и крене испочетка у случају да је корисник задао нове путање.

Обрада у реалном времену заснована је на проточној верзији технологије Спарк (енг. *Spark Streaming*). Сваки пут када добије сигнал од пакетне обраде, учитавају се нове путање из датотеке `/results`. Добијене путање се ажурирају у зависности од тока података нових тежина ивица, почевши од тренутка покретања пакетне обраде. Преузимање нових тежина имплементирано је у микро-пакетном маниру (енг. *Micro-batching*) чиме се омогућава паралелно ажурирање путања. Путање се потом сортирају и шаљу модулу за приказ, преко Кафка брокера на теми `results`.

Систем за приказ преузима и приказује најкраћу путању и локално ажурира вредности тежина грана у графу добијених преко Кафка брокера на теми `new_edges`, такође у пакетном маниру из истог разлога. Овакав поступак омогућава да вредности нових путања одмах буду приказане на графику, док добијање најкраће путање долази са закашњењем од модула за обраду у реалном времену.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Кафка брокери омогућавају да цео систем буде конзистентан, зато што поседује механизам за складиштење порука по темама, које гарантује да ће свака порука бити прочитана и обрађена.

На основу описа архитектуре пројектованог система може се закључити да модул за пакетну обраду генерише пакетне погледе који представљају израчунате путање, које потом користи модул за обраду у реалном времену за ажурирање најкраће путање. Сервисни слој је реализован модулом за приказ и омогућава интеракцију са корисником.

4 Опис коришћених технологија

У овом поглављу биће описане технологије кориштене у изради овог мастер рада. На почетку ће бити описана техника контејнеризације и њене предности током развоја и одржавања апликације. Компоненте система реализоване су у оквиру контејнера Докер у оквиру Линукс (енг. *Linux*) оперативног система. Модули за налажење и ажурирање најкраће путање развијени су у технологији Апачи Спарк (енг. *Apache Spark*), која омогућава обраду велике количине података и ажурирање у реалном времену. Складиштење података реализовано је коришћењем дистрибуираног фајл система Хадуп, док је технологија кориштена за размену података Кафка. Визуелни приказ реализован је коришћењем библиотеке Даш уграђене на програмски језик Пајтон.

4.1 Контејнери у технологији Докер

Контејнеризација је техника која омогућава раздвајање апликације на функционалне целине, које могу независно да се развијају и покрећу. У односу на једну монолитну апликацију, контејнери омогућавају бржи развој и већу контролу приликом извршавања програма. За сваку компоненту система креира се контејнер унутар кога се инсталирају сви потребни алати и окружење за њено покретање. На овај начин добија се велика преносивост кода, јер покретање апликације не зависи од програма инсталираних на систему на ком се она покреће, већ сваки део апликације има све потребне инсталације унутар свог контејнера. Из тог разлога, могуће је систем у потпуности развити локално на једном рачунару, док ће се у продукцији систем извршавати на кластеру рачунара. Овај приступ омогућава још и конзистентност софтвера, јер се и развој и продукција реализују у оквиру истог окружења контејнера.

Чест домен примене контејнера је микросервисна архитектура. У оквиру система, унутар сваког контејнера се извршава одређени сервис, који представља целину за себе. У случају грешке код неког сервиса, није потребно поново покретати целу апликацију, већ је довољно покренути дати контејнер. На исти начин могуће је вршити постепено ажурирање контејнера новом верзијом, при чему апликација глобално не престаје да ради и нема ризика од пада читавог система у случају грешке код нове верзије.

Микросервисна архитектура омогућава да се сваки контејнер скалира у зависности од потреба апликације. У случају да дође до повећаног коришћења неког сервиса, апликација ће бити у стању да одговори захтевима реплицирањем контејнера коришћењем оркестратора. Ово је такође начин да се апликација заштити од евентуалних софтверских напада и јединствене тачке отказа (енг. *single point of failure*).

За разлику од виртуелних машина, које омогућавају изолацију и подешавање окружења, контејнеризација се обавља на нивоу оперативног система уместо на нивоу хардвера. Због овога слике контејнера заузимају знатно мање меморије (реда ~10Mb) у односу на виртуелне машине (реда ~10Gb). У њима су садржане команде за додавање фајлова, инсталирање алата и конфигурацију окружења за одређеног контејнера. Сlike

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

је могуће преузимати и надограђивати са регистра слика (енг. *Registry Server*), чиме се брзо може доћи до жељене спецификације. Покретањем контејнера креира се именски простор у корисничком простору у ком се чувају датотеке и програми контејнера. Именски простор унутар кернела система домаћина за сваки контејнер креира идентификатор процеса и мрежне интерфејсе преко којих процеси могу да комуницирају. Поред овога контејнери могу бити повезани на трајна складишта података, која чувају податке и када је контејнер неактиван.

Докер (енг. *Docker*) је технологија која омогућава креирање, управљање и дистрибуцију контејнера. Докер извршно окружење (енг. *Docker Engine*) садржи све потребне методе за рад и покретање контејнера. Спецификација контејнера је изведена или из докер фајла (енг. *Dockerfile*) или се преузима директно из регистра слика, чиме се конфигурише потребно окружење.

За покретање више контејнера у оквиру докер алата користи се Докер-Састављач (енг. *Docker-Compose*), који је кориштен у изради овог мастер рада. Овај алат омогућава једноставно креирање и покретање више контејнера Докер, успостављање међузависности и креирање перзистентних складишта у оквиру докер радног окружења (енг. *Docker-Volumes*). Спецификација система дата је у докер-састављач фајлу (енг. *Docker-compose.yml*), на основу ког су креирани сви контејнери. Овај алат коришћен је у овом раду, због једноставности и лаког коришћења.

Сваки *docker-compose.yml* фајл започиње дефинисањем верзије технологије Докер, након чега следи специфицирање контејнера после кључне речи *services*. Контејнери започињу именом контејнера након чега може бити дефинисано много параметара. Параметри коришћени у овом раду су:

- *image* , *build* – на основу слике или из специфицираног докер-фајла се генерише контејнер,
- *port* – дефинише порт за комуникацију са спољним светом,
- *environment* – дефинише променљиве окружења контејнера,
- *container_name* – поставља име контејнера,
- *depends_on* – дефинише који контејнери морају да буду активни пре датог контејнера,
- *volumes* – креира се перзистентно складиште повезано са контејнером,
- *env_file* – креира систем на основу конфигурационог фајла.

Алати за оркестрацију у пуној мери користе предности микросервисне архитектуре. Ови алати аутоматизују покретање и интеракцију више контејнера. Најпознатији представници алата за оркестрацију су Кубернетес (грч. *κυβερνήτης*), Апачи Сворм (енг. *Apache Swarm*) и Апачи Мезос (енг. *Apache Mesos*)

4.2 Дистрибуирани фајл систем Хадуп

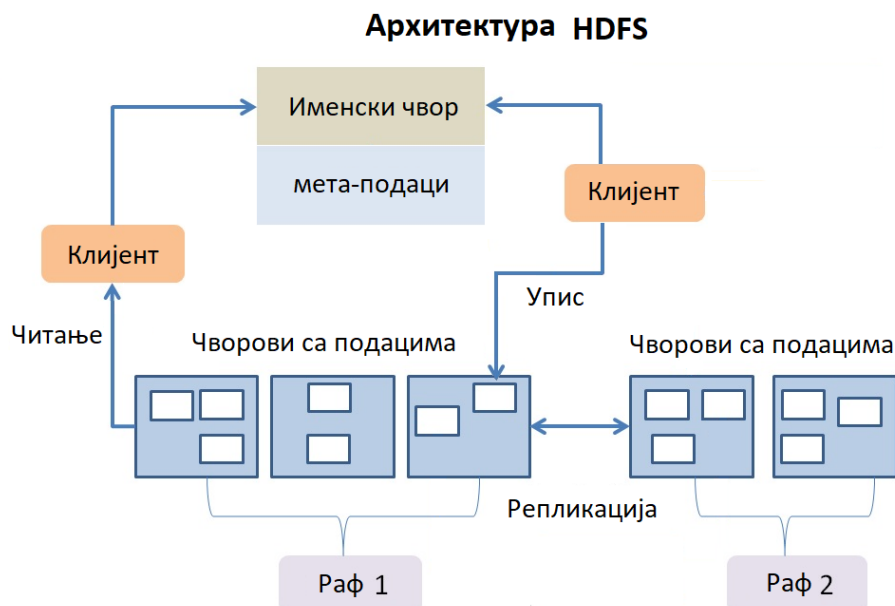
Дистрибуирани фајл систем Хадуп (енг. *Hadoop Distributed File System, HDFS*) је дистрибуиран систем за складиштење велике количине података, који користи обичан-

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

потрошачки хардвер. Овакви системи се обично користе за складиштење од неколико стотина мегабајта до неколико терабајта. Програми који обрађују велике количине података обично захтевају велику пропусну моћ читања података, како би били ефикасни. Да би овај захтев био испуњен *HDFS* поседује оптимизоване методе за читање података у блоковима. Подразумевана величина блока је 64 мегабајта.

Како је у питању велика количина података и дистрибуирано складиште података, *HDFS* има архитектуру која је у значајној мери отпорна на отказе појединачних компоненти. Састоји се из именских чворова (енг. *Namenodes*) и чворова са подацима (енг. *Datanodes*). Именски чворови служе за чување локација и кординисање датотекама у систему. Локације података чувају се као слике именског простора (енг. *Namespace image*), док постоји и дневник измена (енг. *Edit log*), који чува податке о свакој трансакцији. Чворови са подацима садрже блокове података које сачињавају датотеке.

Клијентске апликације приступају складишту преко *HTTP* протокола директно или преко заступника (енг. *Proxy*). Приликом операције читања клијентска апликација шаље захтев за добијање локације датотека од именског чвора. Именски чвор као повратну вредност враћа локацију најближег чвора са подацима који поседује тражену датотеку, након чега следи читање по блоковима. Уколико је постојећа датотека оштећена, прелази се на приступ најближој реплици, док се оштећена датотека замењује адекватном копијом. Упис у *HDFS* реализује се на сличан начин. Именски чвор овог пута, враћа слободне адресе на којима ће бити уписана жељена датотека и њене репликације. Потом почиње процес уписивања у чворове са подацима.



Слика 4.1 Архитектура HDFS система

Уколико би дошло до отказа код именског чвора, изгубиле би се и локације свих датотека. Да би се ово предупредило, обично се уводи постојање секундарног чвора,

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

који ажурира своју слику именског простора. У случају отказа примарног чвора, секундарни чвор преузима посао активног. Да би ово било могуће чворови са подацима су од почетка конфигурисани да комуницирају са оба чвора. Приликом отказа примарни чвор се одваја од мреже и укидају му се права на приступ подацима, јер се мора спречити ситуација у којој примарни чвор постаје активан, док је секундарни већ преузео посао. Овај поступак назива се оградавање (енг. *Fencing*).

Чворови са подацима такође поседују механизме заштите од отказа. На првом месту интегритет података се обезбеђује коришћењем контролне суме (енг. *Check-sum*) и репликација. Контролна сума добија се као резултат функције, која врши униформну расподелу улазних података у опсегу $(0, 2^{32})$ коришћењем алгорита *CRC-32* при чему се за мале промене улаза добија значајно различит резултат. На овај начин се може се проверити да ли је дошло до промене датотеке. Ова операција врши се сваки пут када се подаци учитавају у систем и сваки пут када се подаци читају. Репликацијом се подаци уписују на више места и уколико дође до оштећења неке датотеке, датотека се одбацује и прелази се на читање њене реплике. Уместо одбачене датотеке се покреће креирање нове реплике, како би број реплика у систему остао исти.

4.3 Апачи Спарк

Спарк је технологија за обраду велике количине података опште намене, оптимизирана за дистрибуирану обраду на кластеру. Спарк се заснива на коришћењу дистрибуираних скупова података отпорних на отказе (енг. *Resilient Distributed Dataset, RDD*), над којима су реализовани оператори високог нивоа апстракције. Ова структура посебно је погодна за пакетну обраду, због чега је узета за имплементацију пакетног модула и биће детаљније објашњена у наставку. Поред овога Спарк пружа могућност обраде података у реалном времену засновану на технологији *Spark Streaming*, па се користи у модулу за ажурирање у реалном времену. Реализован је у програмском језику *Scala*, али обезбеђује програмски интерфејс и може бити коришћен у програмским језицима *Java*, *Python* и *R*. Како је циљ овог мастер рада било пројектовање система у програмском језику *Python*, он је изабран и у овом случају.

4.3.1 Пакетна обрада у технологији Спарк

Главна предност у односу на остале технологије базиране на парадигми Мапирање-Редукција је складиштење помоћних резултата унутар меморије уместо на диску, као у случају алата Апачи Хадуп. Када користи кеширање унутар меморије, Спарк је за око 100 пута бржи од алата Апачи Хадуп, док је у случају када се подаци складиште на диск бржи више од 10 пута [27]. Ово је особина која је посебно битна за итеративне алгоритме, који кроз сваку итерацију креирају привремене резултате.

Спарк такође поједностављује писање кода, јер поред корака мапирања и редукције омогућава флексибилне операторе над структуром *RDD*. Спарк поседује и библиотеке за машинско учење *MLib* и обраду графова *GraphX*, који у многим применама олакшавају рад. У време писања рада библиотека *GraphX* није била имплементирана у

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

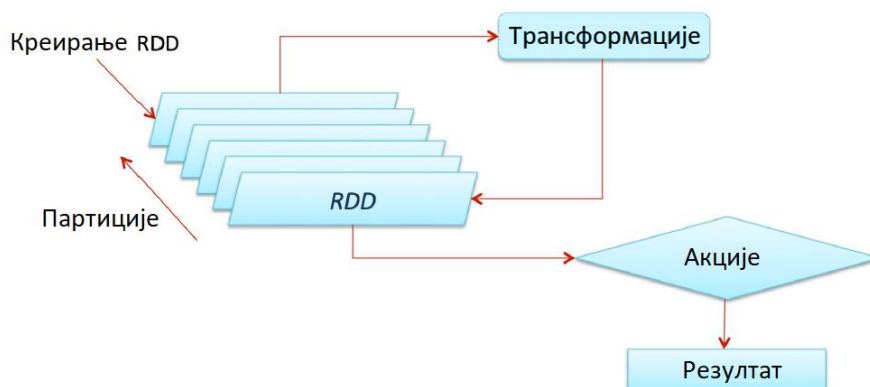
језику *Python*, па је алгоритам за рачунање најкраће путање ручно имплементиран и биће описан у следећем поглављу.

RDD представља основну јединицу обраде у Спарку. Састоји се из листе података у формату кључ-вредност, који су реплицирани на више извршних чворова. Уколико дође до отказа неког чвора, подаци ће и даље бити доступни на другом чвору и обрада неће стати. Како се обрада врши у дистрибуираном маниру, Спарк аутоматски дели *RDD* на партиције између више чворова и покреће обраду. Сваки нов задатак над *RDD* додаје се у граф израчунавања и не покреће се све док не постоји захтев за резултатом обраде. Овај начин обраде назива се обрада са одложеном евалуацијом (енг. *Lazy evaluation*) и представља још један механизам који доприноси брзини.

У Спарку постоје два типа операција над *RDD*:

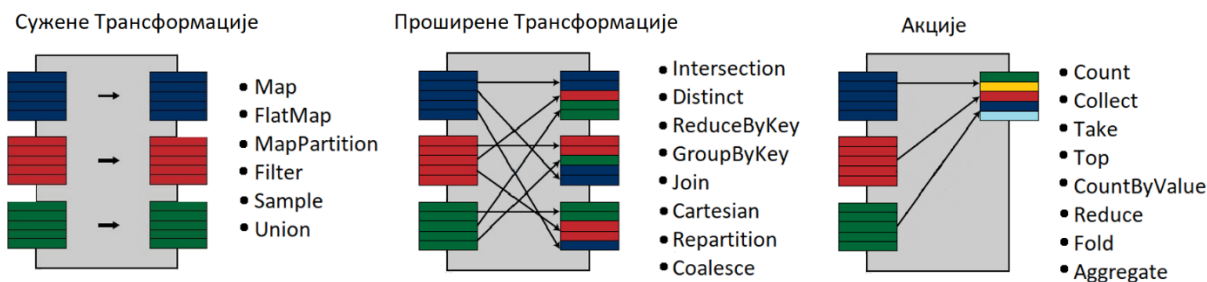
1. трансформације,
2. акције.

Трансформације се односе на креирање и модификовање *RDD*, док се акцијама добија резултат заснован на подацима унутар структуре *RDD*.



Слика 4.2 Шема Трансформација и Акција над *RDD*

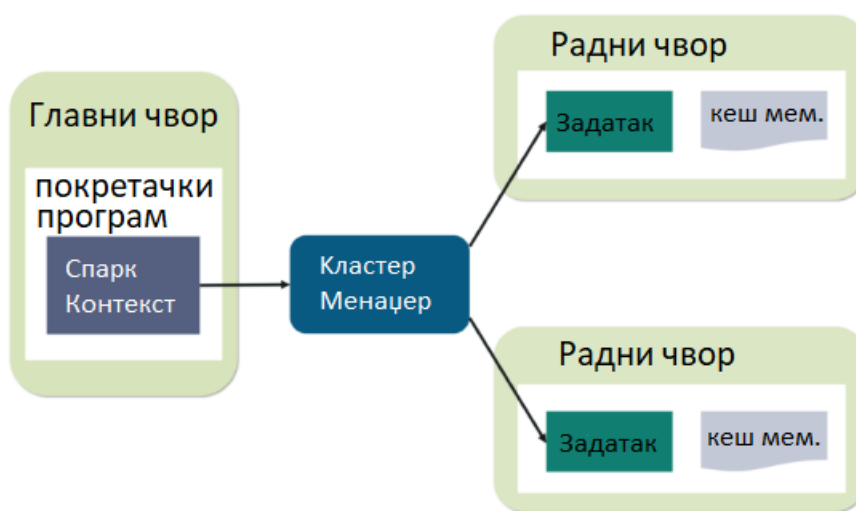
У Спарку постоје две групе трансформација и оне могу да буду сужене и проширене. Уколико свака резултујућа партиција зависи само од једне улазне партиције у питању је сужена трансформација, док је у случају различитих партиција у питању проширена трансформација.



Слика 4.3 Операције над *RDD*

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Спарк систем се састоји од главног чвора и радних чворова. У главном чвору налази се покретачки програм (енг. *Driver program*). На почетку апликације креира се Спарк Контекст (енг. *Spark Context*) преко кога се позивају описане трансформације и акције. Потом се из ових метода креира директни ациклични усмерени граф, који описује редослед извршавања операција. У следећем кораку се он преводи у план извршавања који се састоји из серије физичких корака. Потом покретачки програм заузима ресурсе од кластер менаџера и ови кораци прослеђују кластер менаџеру који распоређује задатке по радним чворовима. Коначно, кораци се покрећу на радним чворовима, док је покретачки програм задужен за њихово надгледање. [28]



Слика 4.4 Архитектура система за извршавање Спарк

Између задатака који се извршавају на различитим чворовима комуникација може бити остварена на два начина. Први начин је преко емитованих променљивих (енг. *Broadcast variable*). Ове променљиве се шаљу сваком чвору и могу се само читати. Други начин су акумулатори, чија се вредност може једино увећавати. Акумулатори се могу користити за бројање или синхронизацију задатака.

4.3.2 Обрада у реалном времену у технологији Спарк

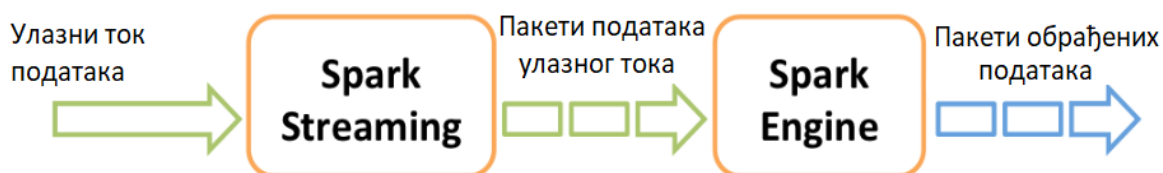
Обрада у реалном времену заснива се на технологији *Spark Streaming*. Ова технологија омогућава обраду токова података, скалабилност и отпорност на отказе. За реализацију токова података подржане су технологије *Kafka*, *Flume*, *Kinesis* и *TCP socket* [29]. *Kafka* представља стандард за дистрибуцију и управљање токовима података, због чега је коришћена у овом раду и биће описана у наставку.

На почетку апликације дефинише се *Spark Context*, који као и у претходном случају повезује апликацију са извршним окружењем технологије Спарк. На основу њега се креира *Streaming Context* и дефинише се период захватања података. *Streaming Context* креира структуру *DStream* са улаза и омогућава трансформације над њим. Прикупљање података почиње позивом методе *streamingContext.start()*, након чега следи метода *streamingContext.awaitTermination()* уколико је потребно зауставити обраду у случају

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

грешке или корисничког захтева. Уколико је потребно да се заустави обрада из кода, може се користити позив методе `streamingContext.stop()`.

Токови података засновани су на структури *DStream*, скраћено од дискретизованих токова (енг, *discretized stream*). Ова структура интерно је имплементирана као низ структуре *RDD*, при чему се сваки члан низа добија као пакет улазних података за дефинисано време. На овај начин се користе добре особине структуре *RDD* и остварује се скалабилност и отпорност на отказе. Позивом методе `foreach(process)` над током података могуће је дефинисати функцију `process`, која је задужена за обраду сваког пакета.

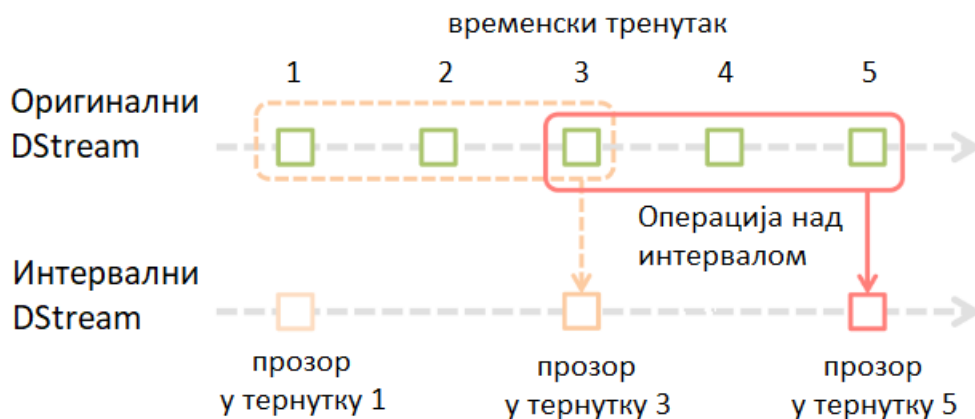


Слика 4.5 Структура обраде токова у технологији Спарк

Трансформације доступне на структури *DStream* су:

- `map`
- `flatMap`
- `filter`
- `repartition`
- `union`
- `count`
- `reduce`
- `countByValue`
- `reduceByKey`
- `join`
- `cogrup`
- `transform`
- `updateStateByKey`

Поред овога, Спарк омогућава обраду токова на нивоу интервала (енг. *Windowed computation*). У овом случају интервали времена се групишу у односу на задату дужину интервала, чиме се добија прозор за задати период. Овако нешто може бити корисно када је потребно повећати грануларност израчунавања.



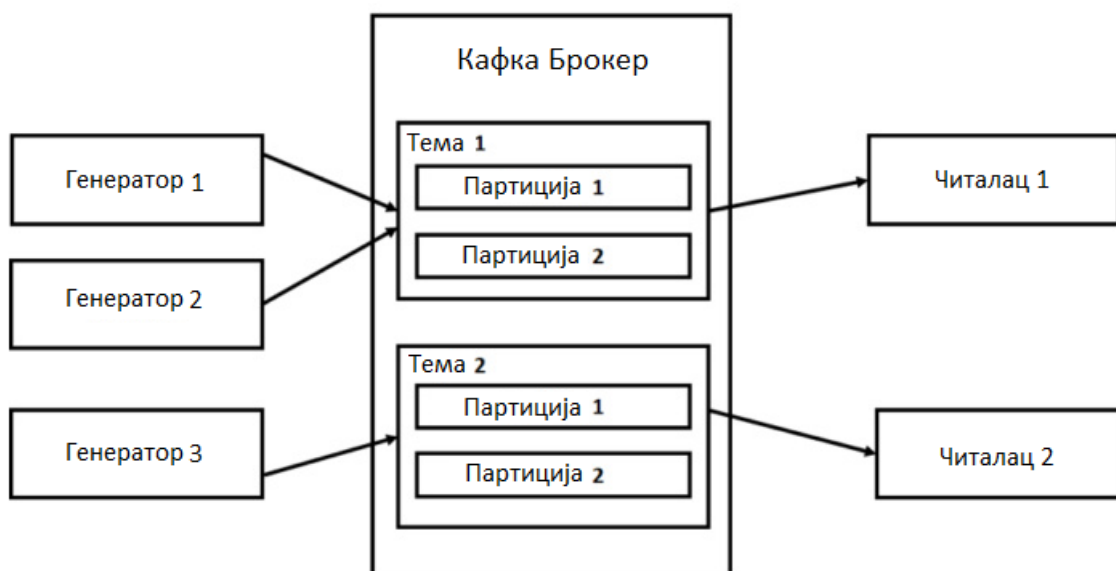
Слика 4.6 Операције над интервалима у технологији SparkStreaming

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Многи системи морају да раде без заустављања и због тога морају бити отпорни на отказе независних од саме апликације. Да би се то обезбедило, Спарк складишти информације о стању система у виду контролних тачака (енг. *checkpoints*), како би био у могућности да рекреира стање у коме је био приликом заустављања. Подаци који се чувају унутар контролних тачака су подаци о конфигурацији система, подаци о скупу операција које се извршава над током података, пакети података које нису још обрађени, као и нови подаци који су генерисани обрадом.

4.4 Кафка

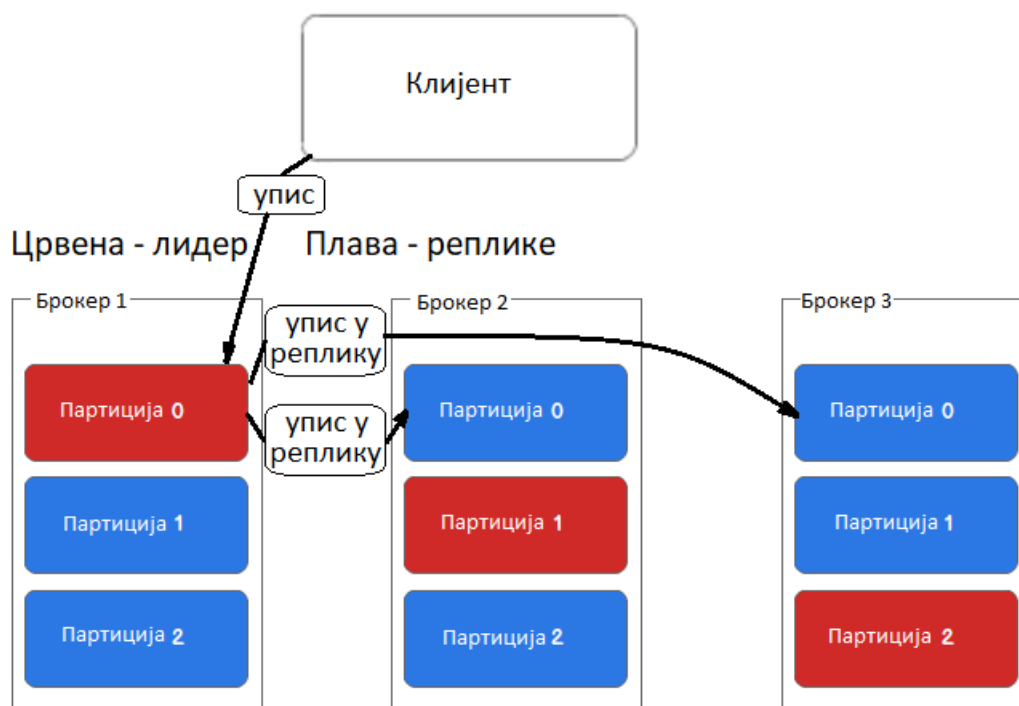
Кафка представља систем за размену података заснован на механизму издавач-читалац (енг. *publish-subscribe, pub-sub*). Овај механизам реализован је посредством брокера, који прихвата податке од генератора података и смешта их по темама које читају модули за обраду. Овај механизам пружа велику флексибилност, јер се компоненте могу лако додавати за читање или упис на неку тему и сва комуникација у систему може бити остварена преко једне компоненте.



Слика 4.7 Кафка систем за размену порука

Подаци се унутар једне теме дистрибуирају на више партиција, чиме се гарантује отпорност на отказе. За сваку тему постоји главна партиција преко које се подаци уписују у додатне реплике. Када дође до пада главне партиције, друга ажурна копија постаје главна. Уколико дође до пада свих реплика, може се чекати или пређашња главна реплика да се поново подигне или се прва подигнута реплика проглашава за главну одакле се извршавање наставља. На овај начин се постиже брзина али, долази до губитка неких података уколико подигнута реплика није била ажурна у тренутку пада.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда



Слика 4.8 Архитектура технологије Кафка

У Кафка кластеру један од брокера врши улогу контролера, који је задужен за праћење стања осталих брокера, креирање нових партиција и тема и осталих административних послова као што је прерасподела партиција по брокерима. Када откаже неки брокер, контролер добија информације за које је партиције тај брокер био лидер и за те партиције покреће гласање на основу ког бира следеће лидере.

Поред овога Кафка чува записе по темама одређени временски период, који је могуће подесити, након чега се подаци или компримују или бришу. Због овога не долази до губитка података уколико фреквенције уписа и читања нису једнаке у сваком тренутку, јер записани подаци могу да се читају са закашњењем. Поред тога подаци остају сачувани и у ситуацијама када долази до раздвајања система услед мрежних проблема или отказа компоненте.

Постојање партиција такође повећава пропусност система. У случајима када је креирано више читалаца за једну тему, они могу бити организовани тако да свако чита по једну партицију. Тада је могуће постићи паралелно читање, тако што се подаци насумично шаљу партицијама једне теме, након чега ће свака порука бити прихваћена од једног обрађивача. Ова техника међутим не обезбеђује читање података у реду у ком су послати, јер може доћи до отказа компоненте, након чега се порука поново чита са закашњењем у односу на остале податке.

4.5 Зоо-кипер

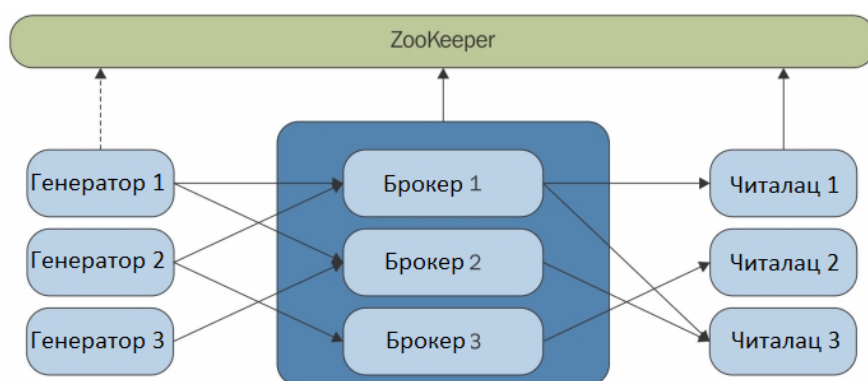
Зоо-кипер (енг. *Zoo-keeper*) је технологија, која служи за одржавање и конфигурацију система. Неопходан за покретање Кафка кластера, а често се користи у позадини код

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

многих технологија за обраду велике количине података, као што су Хадуп, Сторм и Спарк. Како се у овом раду зоо-кипер користи као подршка Кафки, описана је његова улога у одржавању Кафка кластера [30].

Зоо-кипер чува стање система и прати које су компоненте активне. Он је задужен за бирање Кафка контролера периодично шаље сигнал брокерима и у случају да неко од њих не одговори, покреће се процедура бирања. Уколико се контролер нађе у отказу, сви остали брокери аплицирају за место контролера. Први од њих постаје нов контролер, креира се привремено складиште и обавештавају се остали брокери ко је нови контролер.

Зоо-кипер чува све информације у вези са читањем. Ово подразумева податке о лидеру за дату тему, листу читаоца, офсет који може да се чита или уписује, место до ког су дошли са читањем партиције, ко има права шта да чита итд. Уколико постоје групе читаоца, записано је ко су њени чланови и докле се стигло са читањем.



Слика 4.9 Повезаност Зоо-кипера са Кафком

4.6 Даш

Даш (енг. *Dash*) представља радни оквир за креирање веб апликација и реализован је у програмском језику *Python*. У својој основи садржи радне оквире *Flask*, *Plotly.js* и *React.js*, који омогућавају једноставно креирање интерфејса према кориснику, као и визуелизацију потребних података [31]. Овај алат је посебно погодан када је потребно да брзо реализовати визуелизацију уз коришћење контејнера, јер Докер не подржава графички интерфејс, па је визуелизацију могуће једино добити коришћењем веб претраживача.

Даш се састоји од компоненте *layout* која је задужена за изглед апликације и компоненте *callback* која омогућава интеракцију између компоненти. Основни скуп визуелних компоненти, дефинисане су у библиотеци *dash_core_components* и *dash_html_components*, а поред тога остављена је могућност да корисник сам специфицира своје компоненте коришћењем технологије *JavaScript* и *React.js*. У оквиру поменутих библиотека реализоване су компоненте као што су падајуће листе,

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

клизач, дугме, оквир за упис текста и многе друге, а од посебног значаја за овај рад је компонента графикана *Graph*, која омогућава приказ графа. Приступ конфигурисања параметара компоненти је декларативан и дефинише се при дефинисању компоненте *layout*. Параметре компоненти приказа, могуће је динамички мењати у току рада апликације коришћењем позива *callback*.

Сваки пут када корисник кликне на компоненту са веб странице, као што је дугме или падајућа листа, у позадини може да се дефинише позив *callback* који може да да одговор на кориснички захтев. Позив *callback* садржи параметар *Input* у који се наводи идентификатор компоненте за коју се он позива и параметар *Output* којом се наводи која се компонента ажурира након позива. У случају да је за параметар *Input* задата листа, позив ће се догодити при клику сваке компоненте из листе или истеком дефинисаног времена уколико је у питању компонента *Graph*. Такође, за параметар *Output* може да се проследи листа и у том случају ће се ажурирати више компоненти. У овом раду кориштене су компоненте падајућа листа, дугме и графикон, док је за сваку од ових компоненти реализован и одговарајући позив *callback*, чија ће имплементација бити дата у следећем поглављу.

5 Имплементација система за налажење најкраће путање

У овом поглављу дат је опис реализованих модула и начин покретања апликације. У првом делу описано је коришћење фајла *docker-compose.yml* на основу ког се покрећу контејнери и започиње извршавање апликације. Потом је описано повезивање са складиштем *HDFS* и системом за комуникацију Кафка. На крају је дата реализација компоненти уписивача у *HDFS* - *hdfs_writer*, генератора – *gen*, модула за приказ - *plot*, пакетне обраде - *batch* и обраде у реалном времену - *real_time*.

Имплементација овог система налази се на веб сајту:

<https://github.com/dejangrubisic/Shortest-Path-in-Dynamic-Large-scale-Graph>

5.1 Покретање апликације

Основни фајл за покретање је *docker-compose.yml*. У њему су специфициране конфигурације контејнера у којима се покрећу модули. Апликација се покреће командом:

```
docker-compose up
```

Листинг 5.1 Покретање апликације

Контејнери који се покрећу су:

- *gen*
- *plot*
- *hdfs_writer*
- *spark-master*
- *spark-worker1*
- *namenode*
- *datanode*
- *kafka*
- *zookeeper*

Контејнери су реализовани у оквиру оперативног система *Linux* и сваки од њих ће бити разматран у наставку.

5.2 HDFS

Подаци се складиште у технологији *HDFS* реализовану модулима *namenode* и *datanode*. Коришћење овог система доступно је на порту <http://namenode:50070> и користи се из свих кориснички дефинисаних модула. Повезивање се остварује наредбом:

```
hdfs = InsecureClient(os.environ['HDFS_HOST'], user='root')
```

Листинг 5.2 Повезивање са складиштем *HDFS*

при чему *HDFS_HOST* означава поменути порт.

Ажурирање података у складишту *HDFS* било је потребно у случају изворних података графа и резултата пакетне обраде у виду листе најкраћих путања. У оба случаја ажурирање је остварено командом:

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

```
hdfs.upload(hdfs_file, local_file, overwrite=True )
```

Листинг 5.3 Ажурирање фајла из складишта *HDFS*

где се задаје путања *hdfs* фајла на коју се записује локални фајл са путање *local_file*. Параметар *overwrite* је постављен на тачно, чиме се уписивањем нових података стара вредност брише. Читање изворних података графа у пакетној обради је остварено командом:

```
readHDFS(spark, path="hdfs://namenode:8020/data_in")
```

Листинг 5.4 Читање фајла из складишта *HDFS*

где се за наводи пуна путања фајла са подразумеваним портом 8020 за *HDFS*. У обради у реалном времену је за читање резултата обраде кориштена команда:

```
sparkContext.textFile("hdfs://results")
```

Листинг 5.5 Читање из складишта *HDFS* у делу за рачунање у реалном времену

5.3 Кафка

Модули *kafka* и *zookeeper* задужени су за преношење података између компоненти. У овом раду кориштено је три теме преко које компоненте комуницирају:

- Тема *start*– преко ове теме шаљу се идентификатори почетних и крајњих чворова између којих се тражи најкраћи пут, између модула приказ и пакетне обраде модула Спарк. Порука је дата у формату: $s_1 s_2 \dots s_n \backslash n e_1 e_2 \dots e_n$, при чему s и e представљају чворове графа док $\backslash n$ служи за раздвајање почетне и крајње чворове
- Тема *new_edges* – ова тема служи за пренос нових тежина ивица. Модул генератор уписује на ову тему поруку у два формата. Уколико ивица постоји у графу довољно је послати само идентификаторе почетног и крајњег чвора. Ако се ивица креира први пут, поред овога потребно је послати још и координате (географска дужина и ширина) почетног и крајњег чвора.
- Тема *result* – преко ове теме се шаће резултат обраде модула Спарк, најкраћа путања, модулу за приказ. Најкраћа путања дата је у форми
укупна тежина | почетни чвор | тежина | чвор | ... | тежина | крајњи чвор

Повезивање са Кафком за читање добија се са командама:

```
consumer = KafkaConsumer(    bootstrap_servers=KAFKA_HOST,  
                             auto_offset_reset='earliest',  
                             enable_auto_commit=True)  
consumer.subscribe( [TOPIC_EDGES, TOPIC_RESULT] )
```

Листинг 5.6 Инстанцирање читаоца система Кафка

Теме могу специфицирати или у позиву функције *KafkaConsumer* или командом *subscribe*. *KAFKA_HOST* представља локалну променљиву и једнака је *kafka:9092*. Читање се добија у микро-пакетном маниру командом *consumer.poll()*, чиме се добијају сви пристигли подаци. Друга могућност је била да се користи читање коришћењем

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

конструкције *for message in consumer*: чиме се извршавање блокира све до приспећа нове поруке, која се потом добија наредбом *message.value* и даље обрађује.

Повезивање са Кафком за упис се добија командом:

```
producer = KafkaProducer(bootstrap_servers=KAFKA_HOST)
```

Листинг 5.7 Инстанцирање издавача система Кафка

Упис на Кафка тему остварује се коришћењем команде

```
producer.send( TOPIC, value, key)
```

Листинг 5.8 Упис издавача на тему система Кафка

где се на место *TOPIC* уписује име теме, док се за параметре *key* и *value* прослеђује порука и њен идентификатор.

5.4 Уписивач у HDFS

Овај модул има задатак за на почетку иницијализује *HDFS* и да га потом ажурира, при наиласку нових ивица. Блок за иницијализацију контејнера у фајлу *docker-compose.yml* и одговарајући докер фајл дати су у наставку:

Модул <i>hdfs_writer</i>	<i>hdfs_writer</i> / Dockerfile
<pre>hdfs_writer: build: ./hdfs_writer environment: HDFS_HOST: http://namenode:50070 KAFKA_HOST: kafka:9092 K_TOPIC_EDGES: new_edges container_name: hdfs_writer depends_on: - namenode - datanode - kafka</pre>	<pre>FROM python:2.7-slim WORKDIR /hdfs_writer COPY . . RUN pip install hdfs RUN pip install kafka-python CMD ["python", "hdfs_writer.py"]</pre>

Листинг 5.9 Дефинисање контејнера модула *hdfs_writer*

У *docker-compose* фајлу су дефинисане променљиве окружења, име контејнера, зависност у односу на контејнере који чине *HDFS* и дата је путања докер фајла на основу које се конфигурише контејнер. У контејнеру се инсталира програмски интерпретер *python*, креира се радни директоријум у који се копирају датотеке за покретање и подаци графа. Потом се инсталирају пакети *hdfs* и *kafka-python* и коначно покреће се програм за упис.

Програм за упис повезује програм са *HDFS* складиштем и Кафком у форми читаоца, отвара фајл са подацима графа и шаље у *HDFS*.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

<pre>def main(): hdfs = connect_to_hdfs() consumer = connect_to_kafka() with open("data.json") as json_f: data = json.load(json_f) send_to_hdfs(data, hdfs) consume(data, consumer, hdfs)</pre>	<pre>def consume(data, consumer, hdfs): while True: edges = consumer.poll().values() if len(edges) != 0: Foreach edge in edges: data = storelocally(edge) changes +=len(edges) If changes > max or time_elapsed: writeLocalToHDFS(data, hdfs) sleep(5)</pre>
---	---

Листинг 5.10 Псеудокод функције *main* и *consume* модула *hdfs_writer*

Функција *consume* даље ажурира пристигле податке преко Кафка читаоца. Она прво смешта сваку ивицу у локални фајл и ажурира вредност уколико претходно постоји и када број нових ивица пређе задату вредност *max* локални фајл се смешта у дистрибуирану меморију. Ово се такође дешава уколико прође довољно времена од последњег уписа у дистрибуирану меморију, што је интерно дефинисано у константом *time_elapsed*.

5.5 Генератор

Генератор је компонента која има задатак да симулира креирање нових ивица. У некој примени ово ће бити конкретан ток података, као што је на пример време потребно да се стигне између два чвора. Блок наредби којима се конфигурише контејнер *gen* у *docker-compose.yml* и одговарајући докер фајл дати су у наставку:

Модул <i>gen</i>	<i>gen</i> / Dockerfile
<pre>gen: build: ./gen environment: HDFS_HOST: http://namenode:50070 KAFKA_HOST: kafka:9092 K_TOPIC_EDGES: new_edges container_name: gen depends_on: - kafka - hdfs_writer</pre>	<pre>FROM python:2.7-slim WORKDIR /gen COPY . . RUN pip install hdfs RUN pip install kafka-python CMD ["python", "pr.py"]</pre>

Листинг 5.11 Дефинисање контејнера модула генератор

Као и у претходном случају дефинисани су параметри на основу којих се креира контејнер, који је сада завистан од модула *hdfs_writer*. Програм који се покреће у овом случају *pr.py* служи само за одржавање контејнера активним, док се генерисање ивица

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

остварује коришћењем програма *gen.py* који се покреће из интерактивног режима контејнера. Да би се ушло у овај режим користи се команда у терминалу:

```
docker exec -it gen bash
```

Листинг 5.12 Покретање контејнера модула генератор

Програм *gen.py* састоји се из два режима *random* и *user*. Режим *random* служи за насумично генерисање нових ивица, док режим *user* служи за кориснички дефинисано ажурирање ивица. На почетку програма учита се тренутно стање графа из *HDFS* меморије и, у случају насумичног генерисања ивица, подаци се спајају са листом нових ивица, након чега се периодично бира случајна ивица из датог скупа. Нова тежина ивице добија се сабирањем претходне и насумичног броја између -250 и 250 при чему се узима позитивна вредност збира. У случају кориснички дефинисане ивице, проверава се прво да ли таква ивица постоји у графу и ако је тај услов испуњен ивица се периодично увећава за задату вредност.

Генератор се позива командом:

```
python gen.py start_id end_id increment
```

Листинг 5.13 Покретање модула генератор

у случају кориснички дефинисане ивице, при чему се *start_id* и *end_id* односе на идентификаторе ивица док је *increment* задати инкремент.

5.6 Модул за приказ

Модул за приказ се користи као кориснички интерфејс и има задатак да преузме почетне и крајње чворове од корисника, као и да обезбеди приказ графа. Подешавање контејнера обавља се на сличан начин као и у претходним случајевима и по стартовању контејнера покреће се програм *plot_dash.py*.

На почетку апликације учитавају се подаци о графу из *HDFS* меморије у променљиву *data*, креирају се Кафка објекти *producer* и *consumer* за упис на тему *start* односно, читање са тема *edges* и *result*. Претходно описане компоненте приказа дефинисане су променљивом *app.layout*.

Падајућа листа

```
html.Div([
    dcc.Dropdown(
        id='input-drop1', # 'input-drop2' - у случају листе 2
        options = [ {'label': x, 'value': x} for x in data.keys()],
        placeholder="Select Start Node",
        multi=True)
], style={'width': '25%', 'display': 'inline-block', 'vertical-align': 'middle'}),
```

Дугме

```
html.Div([
    html.Button('Submit', id='button')
```

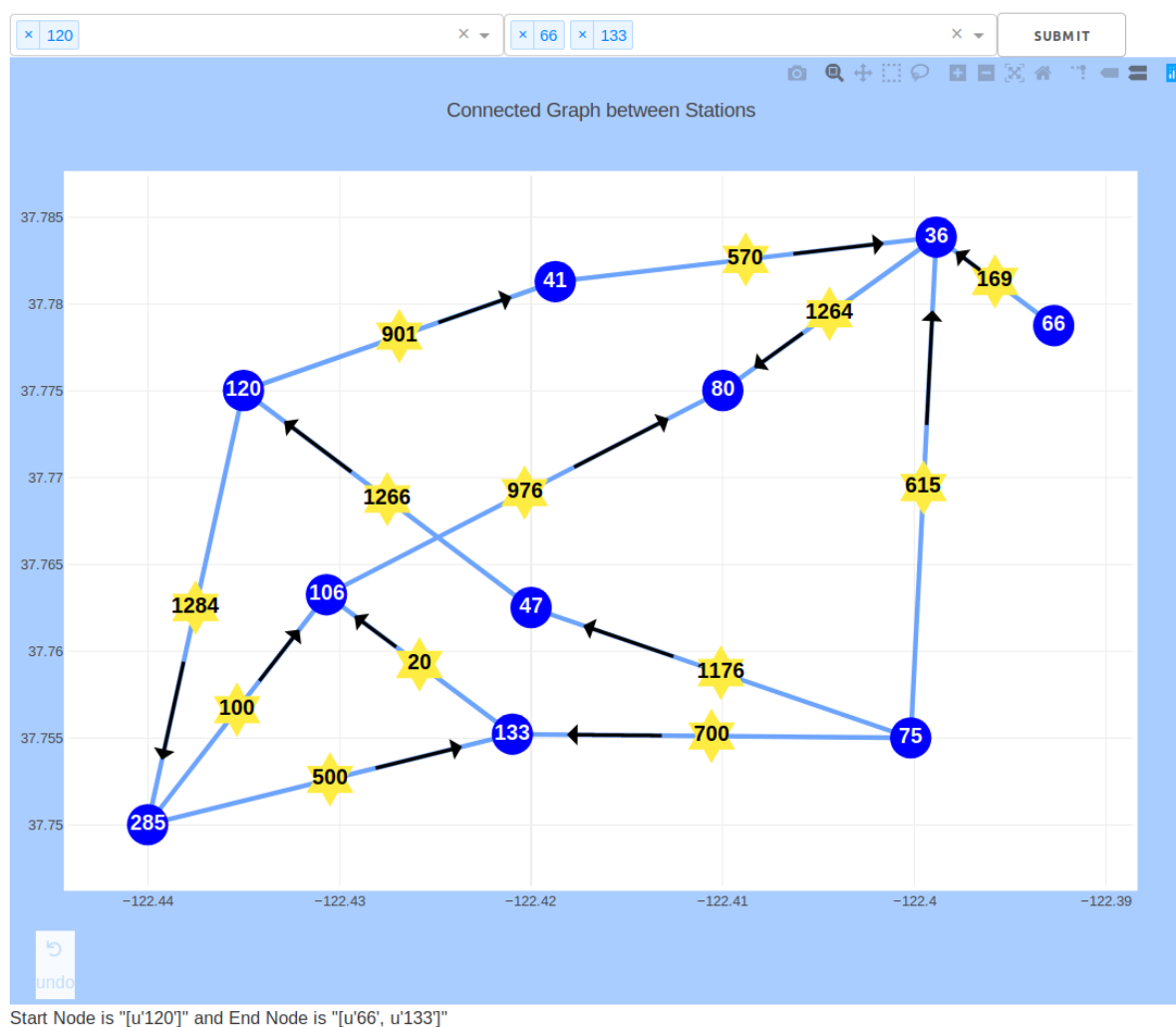
Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

```
],style={'display': 'inline-block', 'vertical-align': 'middle'}),
```

Граф	Интервал
<pre>dcc.Graph(id='live-update-graph', figure=fig, animate=True)</pre>	<pre>dcc.Interval(id='interval-component', interval=1000, n_intervals=0)</pre>

Листинг 5.14 Дефинисање компоненти корисничког интерфејса

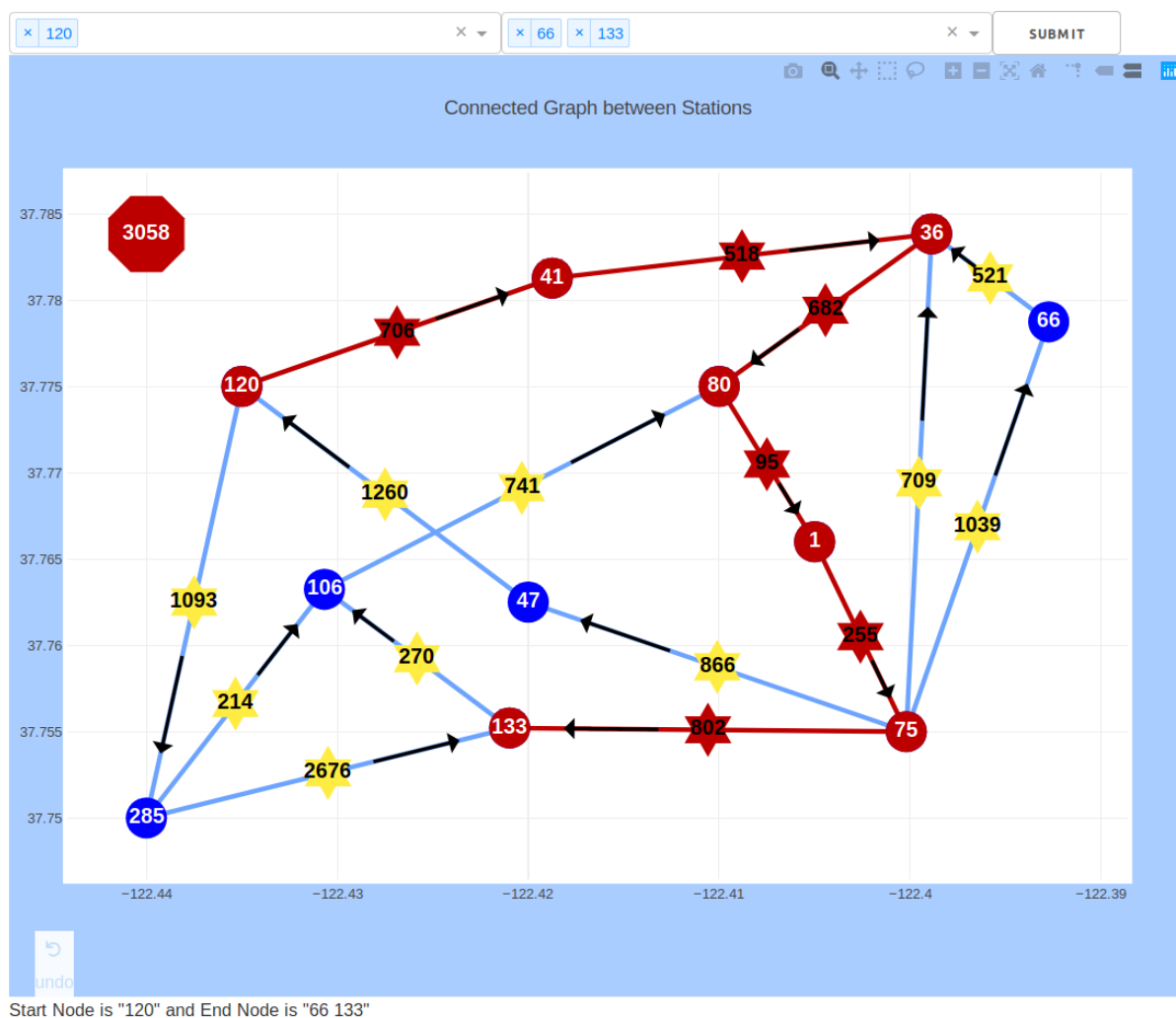
У овом раду коришћене су компоненте *Dropdown*, *Button*, *Graph* и *Interval*. Компонента *Dropdown* служи за избор почетних и крајњих чворова за избор најкраћег пута. Када корисник изабере чворове потребно је да кликне на компоненту *Button* чиме отпочиње процес налажења путање. Када путања буде нађена резултат се приказује, коришћењем компоненте *Graph*. Поред овога, компонента *Graph* је задужена за приказивање ажурираних грана у графу. Ажурирање приказа врши се на 1 секунду, што је подешено као параметар компоненте *Interval*.



Слика 5.1 Задавање чворова за претрагу

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Параметар *fig* компоненте *dcc.Graph* у потпуности одређује изглед графа. Састоји се из компоненте *data* и *layout*. Компонента *data* сачињена је од листе које сачињавају линије, чворови (кругови), тежине (звездице) и резервисана су још два места за приказ најкраће путање и њене укупне тежине (Слика 5.1). Посебну компоненту чине и стрелице, компоненте *layout*, које показују могућ смер кретања у једној грани. Сваки пут када се ажурира путања последња два места компоненте *data* се бришу и прерачунавају се нове вредности. Резултат претраге су путање обележене црвеном бојом, као и укупна тежина, што се може видети на Слици 5.2. Вредности тежина грана се могу мењати у току израчунавања путање и због тога су вредности у звездицама другачије од полазног графа. Поред тога могуће је генерисање и нових грана, као што су гране 85-1 и 1-75 чиме се и те гране узимају у обзир за налажење најкраће путање.



Слика 5.2 Приказ пронађене путање

Интеракција компоненти засновано је на позивима *callback* и у овом раду кориштено је 3 позива *callback*. У првом случају, при клику корисника на дугме, прослеђују се идентификатори изабрани у падајућем менију. Уколико је изабран макар један чвор из листе почетних и крајњих чворова, апликација покреће обраду и шаље идентификаторе чворова модулу Спарк. Други *callback* служи за ажурирање падајућих менија, у случају

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

да је креиран чвор који до тада није постојао. Трећи *callback* користи компоненту *Interval* и активира се на сваке 2 секунде, при чему се ажурира граф.

```
@app.callback( Output('live-update-graph', 'figure'),
               [Input('interval-component', 'n_intervals')])
def update_graph(n_intervals):
    global fig_data
    global fig_arrows

    total_cost, path, new_edges = generator(consumer)
    if len(new_edges) > 0:

        Fig_data, fig_arrows = newEdges(fig_data=fig_data, fig_arrows=fig_arrows,
                                         edges=new_edges)

    if total_cost != -1:
        fig_data = deleteRedTrace(fig_data)
        fig_data = newPath(fig_data=fig_data, path=path)
        fig_data.append( [ createRedNodes(fig_data=fig_data, node_ids=path[:,2]) ] )
        fig_data.append( [ putTotalWeight(fig_data=fig_data, total_cost=total_cost) ] )

    fig_data_flat = list(itertools.chain.from_iterable(fig_data))
    fig["data"] = fig_data_flat
    fig["layout"] = drawArrow(fig_arrows)

    return fig
```

Листинг 5.15 Ажурирање приказа графа

Промењиве *fig_data* и *fig_arrows* служе за чување приказа графа. Функција *generator* задужена је за читање нових ивица и резултата преко Кафке. Уколико постоје нове ивице позива се функција *newEdges*. Унутар ове функције се ажурирају ивице у случају да су оне већ у графу или се генеришу нове, па се као повратна вредност добија и вредност нових стрелица.

Ажурирање најкраће путање започиње брисањем последња 2 елемента приказа графа, претходна најкраћа путања и њена тежина. Функција *newPath* ажурира вредности параметара приказа, чиме се исцртава нова путања. Затим се креира слој чворова које учествују у најкраћој путањи и поставља се укупна тежина функцијама *createRedNodes* и *putTotalWeight*. У последњем кораку се добијене вредности приказа спајају и враћа се коначан резултат, чиме се ажурира слика.

5.7 Пакетна Обрада

У изузетно повезаним графовима, процес проналажења свих путања може представљати изузетно дуг процес, при чему се већина времена губи на тражење заобилазних путања, које су мало вероватно и најкраће. Из овог разлога одабрано је да алгоритам за проналажење путања буде заснован на итеративном креирању путања у односу на задате чворове. У овом алгоритму се у сваком кораку придодаје суседни

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

чвор у листу путања. Процес додавања елемената у путању обавља се у оба смера, од почетних до крајњих чворова и обрнуто.

Путања између неког од почетних и крајњих чворова је пронађена када неки чвор буде био повезан са оба краја. На овај начин се у $N/2$ корака стиже до путање са најмањим бројем грана чији је број N . Ово значи да алгоритам прво налази путање са мањим бројем грана што повећава шансу да сума тежина буде најкраћа. Трајање претраге могуће је ограничити на број корака који алгоритам пролази чиме се може предвидети време израчунавања. Као резултат добијају се све путање у опсегу $2N$ чвора у односу на почетни чвор, где је N број корака, што представља хеуристику у односу на задати проблем.

Апликација започиње програмом *main.py* из ког се позивају програми *real-time.py* и *batch.py*. Програм *real-time.py* се позива први како би се пронашао идентификатор процеса у коме је покренут и проследио програму *batch.py*. Идентификатор процеса се користи на крају пакетне обраде како би се послао сигнал програму *real-time.py* да преузме добијене резултате и настави са радом.

```
#!/usr/local/bin/python
import subprocess

rt_proc = subprocess.Popen(["/spark/bin/spark-submit",
    "--jars", "spark-streaming-kafka-0-8-assembly_2.11-2.4.0.jar", "real_time.py"])

b_proc = subprocess.Popen(["/spark/bin/spark-submit", "batch.py", str(rt_proc.pid) ])

b_proc.wait()
rt_proc.wait()
```

Листинг 5.16 Скрипта за покретање модула за пакетну обраду и обраду у реалном времену

Пакетна обрада започиње проналажењем адекватног идентификатора обраде у реалном времену, као потпроцес процеса, чији је идентификатор прослеђен. Затим се повезује са *HDFS* меморијом и Кафком као читалац и креира се Спарк Контекст. Након овога следи блокирајући позив читања са теме *start* чиме се добијају чворови задати за претрагу најкраће путање.

Претрага започиње учитавањем графа из *HDFS* меморије и проверава се да ли задати чворови постоје у графу. Уколико је услов испуњен, за датим почетним и крајњим чворовима се придодаје листа торки $(0, "")$ која означава дужину најкраћег пута према почетним и крајњим чворовима и саму путању, док се осталим придружује $(inf, "")$, где је *inf* бесконачно, односно највећи цео број дефинисан у систему *sys.maxint*.

Алгоритам за налажење се састоји из првих пет корака *for* петље. У првом кораку метода *flatMap* прослеђује идентификатор чвора и вредност његових параметара функцији *Map*. Међу параметрима се налазе географска дужина и ширина чвора, листа улазних и излазних суседних чворова и придодате путање до почетних и крајњих

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

чворова. Сваки суседни чвор је описан идентификатором, координатама и тежином до задатог чвора.

```
for i in range(steps):

    neighbors_list = rdd.flatMap(lambda x: Map( x[0], x[1] ) )
    neighbors_list = neighbors_list.reduceByKey(lambda x, y: ReduceNeighbors(x, y) )
    rdd = rdd.leftOuterJoin(neighbors_list)
    rdd = rdd.map(lambda x: ( x[0], UpdateGraph(x[1][0], x[1][1]) ) )
    rdd = rdd.map(lambda x: sortPaths( x[0], x[1] ) )

results = rdd.filter(lambda node: connected( node=node ) ) \
    .map(lambda x: saveResults( x[0], x[1] ) ) \
    .flatMap(lambda x: x) \
    .distinct() \
    .sortByKey(ascending=True) \
    .map(lambda x: x[1] )
```

Листинг 5.17 Алгоритам за налажење свих путањи између задатих чворова модула за пакетну обраду

Функција *Map* за сваки чвор који има вредност путање до почетних или крајњих чворова различиту од бесконачно одређује листу суседних чворова. У случају да постоји путања од почетних чворова, гледају се суседни чворови према којима су ивице излазне, док су у случају путање према крају гледају чворови чије ивице увиру у чвор. За сваки суседни чвор се потом креира путања, заснована на путањи задатог чвора и тежини према суседном чвор. Формат је исти као и код торки додатих сваком чвору и за излазне суседне чворове она је (*тежина путање, тренутна путања + ид. чвора + тежина суседног чвора*), док је за улазне путања у обрнутом редоследу.

У следећем кораку се врши редукција суседних чворова по кључу и путање тих чворова се додају у листу путања према почетку и крају. Ово је урађено функцијом *ReduceNeighbors(x, y)* где су *x* и *y* путање истог суседног чвора добијене преко различитих чворова. Како се не би понављале исте путање кориштена је метода *set* над унијом две листе путања.

Потом се добијене путање спајају са почетним подацима графа методом *leftOuterJoin*. Да би се подаци из графа ажурирали кориштена је метода *UpdateGraph*, која прима податке чвора и путања за ажурирање, уколико постоје. Као и у претходном случају метода *set* обезбеђује јединствене путање у сваком чвору, након спајања листа путањи.

Последњи корак обраде представља сортирање путањи према укупној тежини грана. Након проласка кроз све итерације, траже се сви чворови који имају бар једну путању према почетним и крајњим чворовима. Потом се за сваки такав чвор комбинују све његове путање од почетка према крају, чиме се добија листа свих путањи. Како је могуће да различити чворови имају исте путање, узимају се само јединствене путање.

```
res = results.collect()
```

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

```
with open('results.txt', 'w') as outfile:
    outfile.write("\n".join( res ))
hdfs.upload("/results", "./results.txt", overwrite=True )
os.kill(rt_process_id, signal.SIGUSR1)
```

Листинг 5.18 Упис података у складиште *HDFS*

Добијене путање се сакупљају методом *collect()* и уписују у *HDFS*. Након овога шаље се сигнал процесу за обраду у реалном времену, како би могао да ажурира путање пристиглим новим тежинама ивица. У свакој итерацији се још проверава да ли је корисник унео нове чворове за претрагу и у том случају се прелази на излазак из петље и покретање нове претраге. Да би ово било могуће искоришћен је неблокирајући позив читања Кафка порука методом *consumer.poll*.

5.8 Обрада у реалном времену

Обрада у реалном времену започиње креирањем функције која се активира по приспећу сигнала из пакетне обраде. Ова функција поставља глобалну променљиву *read_results* на *True* што ће се касније користити за поновно учитавање резултата пакетне обраде.

```
def receiveSignal(signum, frame):
    globals()['read_results'] = True
signal.signal(signal.SIGUSR1, receiveSignal)
```

Листинг 5.19 Функција за обраду сигнала у делу за обраду у реалном времену

Апликација се такође повезује на Кафку као издавач, и креира се Спарк контекст као и *Streaming* контекст. На основу *Streaming* контекста креира се дискретни ток података, чиме се добијају нове ивице уписане из генератора на тему *new_edges*:

```
edges = KafkaUtils.createDirectStream( ssc, [ TOPIC_EDGES ], {"metadata.broker.list":
"kafka:9092"})
```

Листинг 5.20 Повезивање модула за обраду у реалном времену на тему система Кафка

Након овога, програм се блокира до приспећа сигнала пакетне обраде. Када наиђе сигнал, дефинисана је функција за обраду сваке пристигле путање *words.foreachRDD(process)*. Методе *ssc.start()* и *ssc.awaitTermination()* се позивају како би се покренула обрада, која може да се прекине прекидом са тастатуре.

```
def process(time, rdd):
    global msg
    new_results = False

    if globals()['read_results'] == True:
        paths_info = getBestPaths(rdd.context)
        for buffered_edge in globals()['buffer_edges']:
            paths_info = updateEdge(rdd, buffered_edge, paths_info)
            sendBestPath(paths_info, producer)
        globals()['buffer_edges'] = list()
```

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

```
globals()['read_results'] = False
else:
    paths_info = globals()['paths_info']

new_edge = rdd.collect()
if len(new_edge) != 0 :
    new_edge = [ int(new_edge[0]), int(new_edge[1]), int(new_edge[2]) ]
    globals()['buffer_edges'].append(new_edge)
    paths_info = updateEdge(rdd, new_edge, paths_info)
    sendBestPath(paths_info, producer)

words.foreachRDD(process)
ssc.start()
ssc.awaitTermination()
```

Листинг 5.21 Обрада новокреираних ивица и ажурирање путања модула за обраду у реалном времену

Функција *process()* најпре проверава да ли је потребно учитати нове резултате преко глобалне променљиве *read_results*. Уколико је то случај, учитавају се нове путање и ажурирају се са свим добијеним гранама које су креиране након почетка пакетне обраде, како не би дошло до губитка података. Одмах затим се позива функција *sendBestPath* у којој се путање сортирају и путања са најмањом тежином укупног пута се шаље модулу за приказ.

Новокреирана ивица добијена од модула генератор се форматира у погодан облик, чува у листу добијених ивица од последњег преузимања путања из складишта *HDFS* и смешта у листу путања методом *updateEdge*. У овој функцији се ивица преноси свим радним станицама, након чега се траже путање које садрже задату ивицу и уколико такве путање постоје, ажурирају се у складу са новом тежином. Када се заврши ажурирање путања за све пристигле ивице, путање се сортирају по најмањом вредности пута и функцијом *sendBestPath* се шаљу модулу за приказ.

6 Закључак

Савремене апликације често захтевају обраду велике количине података и одзив на упите у реалном времену. Да би се добио одговор у задатом времену често се користе апроксимативни алгоритми, који дају резултат са мањом прецизношћу. У многим применама овај приступ је задовољавајући, али је пожељно да се у дужем времену добију тачни резултати. Архитектура Ламбда испуњава овај услов и омогућава робусност и скалабилност система који могу бити направљени од потрошачког хардвера.

Претрага графа представља један од основних алгоритама теорије графова који има примену у многим областима. Проналазак најкраће путање на мапи, рутирање интернет комуникација, постављање електронских компоненти на чиповима само су неки примери употреба претраге графа. У зависности од дате апликације развијене су многе графовске структуре и алгоритми, који у највећем броју случајева врше поделу графа и стварају хијерархију путања, како би се омогућила паралелна обрада и избегао пролазак кроз непотребне чворове.

Овај рад усмерен је коришћење предности архитектуре Ламбда приликом проналаска путања између више почетних и крајњих чворова у динамичком графу. Према наивном алгоритму, сваки пут када се промени нека тежина гране у графу покретала би се претрага графа. Како су у питању графови великих димензија, овакав приступ захтевао би неоправдано много времена и резултат израчунавања би увек био застарео, јер би се у току обраде вредности тежина грана промениле.

Реализован алгоритам заснива се на подели проблема на два дела. У првом делу се проналазе све путање између наведених чворова у одређеном опсегу, док се у другом делу израчунате путање ажурирају пристиглим тежинама грана од почетка обраде и сортирају како би се пронашла најкраћа. Алгоритам за налажење путања заснива се на бидирекционој итеративној претрази од почетних и крајњих чворова. Како овај алгоритам налази све путање у опсегу, модул за обраду у реалном времену ће бити у стању да ажурира постојеће путање новим тежинама, али неће бити у могућности да разматра новокреиране гране графа за избор најкраће путање. Ипак, овај проблем не представља препреку у многим апликацијама, јер ће новокреирана грана бити обухваћена следећом пакетном обрадом и учествоваће у избору најкраће путање.

Мана овакве имплементације је то што приликом првог покретања претраге пакетне обраде, обрада у реалном времену не може да почне док се пакетна обрада не заврши. Решење овог проблема било би креирање апроксимативног алгоритма приликом првог покретања, како би се добила прва процена најкраћег пута док се не заврши пакетна обрада.

Даљи напредак могао би бити остварен креирањем више слојева у структури графа, које би се добило спајањем суседних чворова. На овај начин, значајно би било могуће редукovali граф и чиме би се могле детектовати путање и на удаљености већој од $2N$.

Систем за проналажење најкраће путање између локација са тежинама путања променљивим у реалном времену заснован на архитектури Ламбда

Претрага би започињала од највишег нивоа хијерархије и прелазила би на нижи ниво, када се установи повезаност графова. У применама претраге графова са семантиком чворова, попут географских мапа, најкраћи пут често има особину да се налази унутар простора који ограничавају локације почетних и крајњих чворова. На основу овога, граф би се могао додатно редуковати елиминисањем чворова, који су значајно удаљени у односу на овај простор.

7 Литература

- [1] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," 1959.
- [2] R. Bellman, "On a routing problem," 1958.
- [3] L. R. Ford, "Network Flow Theory," 1956.
- [4] R. Floyd, "Algorithm 97: Shortest Path," 1962.
- [5] S. Warshall, "A Theorem on Boolean Matrices," 1962.
- [6] e. a. Amgad Madkour, "A Survey of Shortest-Path Algorithms".
- [7] N. N. B. R. P. Hart., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths".
- [8] D. S. P. Sanders, "Highway Hierarchies Hasten Exact Shortest Path Queries".
- [9] I. A. e. al., "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks".
- [10] P. S. D. Schultes, "Dynamic Highway-Node Routing".
- [11] J. M. a. C. Papadimitriou, "The Weighted Region Problem: Finding shortest paths through a weighted planar subdivision".
- [12] U. Z. L. Roditty, "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs".
- [13] S. K. D. N. M. Henzinger, "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization".
- [14] J. F. a. S. Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time".
- [15] A. B. a. L. Roditty, "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions".
- [16] S. K. a. D. N. M. Henzinger, "A Subquadratic-Time Algorithm for Decremental Single-Source Shortest Paths".
- [17] D. P. L. R. Y. Emek, "A Near-Linear Time Algorithm for Computing Replacement Paths in Planar Directed Graphs".

- [18] A. Bernstein, "A Nearly Optimal Algorithm for Approximating Replacement Paths and k Shortest Simple Paths in General Graphs".
- [19] K. X. e. al., "Finding Alternative Shortest Paths in Spatial Networks".
- [20] B. D. e. al., "Finding Time-Dependent Shortest Paths over Large Graphs".
- [21] U. D. e. al., "Online Computation of Fastest Path in Time-Dependent Spatial Networks".
- [22] H. M. E. Miller-Hooks, "Least Expected Time Paths in Stochastic, Time-Varying Transportation Networks".
- [23] E. N. e. al., "Stochastic Shortest Paths via Quasi-Convex Maximization".
- [24] E. N. e. al., "Optimal Route Planning under Uncertainty".
- [25] J. W. N. Martz, "Big Data Principles and Best Practices of Scalable Realtime Data Systems," 2015.
- [26] E. F. O. G. F. M. P. Flajolet, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," 2007.
- [27] "Difference Between MapReduce and Spark," EDUCBA, [Online]. Available: www.educba.com/mapreduce-vs-spark/.
- [28] "Apache Spark Architecture – Spark Cluster Architecture Explained," [Online]. Available: <https://www.edureka.co/blog/spark-architecture/>.
- [29] A. Spark, "Spark Streaming Programming Guide," [Online]. Available: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>.
- [30] "Role of Apache ZooKeeper in Kafka," [Online]. Available: <https://data-flair.training/blogs/zookeeper-in-kafka/>.
- [31] "Dash User Guide," [Online]. Available: <https://dash.plot.ly/>.

8 Биографија

Дејан Грубишић рођен је 6. јануара 1996. године у Новом Саду. Основну школу завршио је 2010. године када је уписао гимназију „Јован Јовановић Змај“. Током средњошколског образовања посебно се занимао за физику. Учествовао је у многим такмичењима са хором и одржао је неколико говора као члан реторичке секције. Факултет Техничких Наука уписао је 2014. године на смеру за Енергетику, Електронику и Телекомуникације, да би 2018. године завршио усмерење Ембедед Системи и Алгоритми. Током факултета радио је на истраживачким пројектима у Немачкој и Сједињеним Америчким Државама у области ембедед система и програмских преводиоца за паралелну оптимизацију. Мастер студије на студијском програму Рачунарство и Аутоматика усмерење Рачунарство Високих Перформанси уписао је 2018. године и положио све испите предвиђене планом и програмом.