УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Лука Радујевић

ИМПЛЕМЕНТАЦИЈА МИКРОСЕРВИСА ЗА ВИСОКОПЕРФОРМАНТНО БРОЈАЊЕ УЧЕСТАЛОСТИ ПРИДРУЖИВАЊА КЉУЧЕВА

ДИПЛОМСКИ РАД

Основне академске студије

Нови Сад, 2021.

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**:	Завршни (Bachelor) рад
Аутор, **АУ**:	Лука Радујевић
Ментор, **МН**:	др Владимир Мандић
Наслов рада, **НР**:	Имплементација микросервиса за високоперформантно бројање учесталости придруживања кључева
Језик публикације, **ЈП**:	Српски / ћирилица
Језик извода, **ЈИ**:	Српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	2021
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	8/71/24/9/21/7/5
Научна област, **НО**:	ИМТ – Информационе технологије
Научна дисциплина, **НД**:	Инжењерство информационих система
Предметна одредница/Кqучне речи, **ПО**:	Алгоритми, структуре података, микросервис
УДК	
Чува се, **ЧУ**:	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	Дизајн и имплементација микросервиса који омогућава високоперформантно бројање учесталости придруживања кључева. Дизајн и имплементација структуре података која је специјализована за ефикасно читање најучесталијих асоцираних кључева и која представља језгро микросервиса. Упоредна анализа пројектованог решења и микросервиса за бројање учесталости асоцијација који користи базу података.
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:			
	Председник:	др Ђорђе Пржуљ, ванредни професор	
	Члан:	др Соња Ристић, редовни професор	Потпис ментора
	Члан, ментор:	др Владимир Мандић, доцент	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Bachelor Thesis
Author, **AU**:	Luka Radujević
Mentor, **MN**:	Vladimir Mandić, PhD
Title, **TI**:	An implementation of a high-performance microservice for the association frequency counting
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**:	2021
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	8/71/24/9/21/7/5
Scientific field, **SF**:	IMT – Information Technologies
Scientific discipline, **SD**:	Information Systems Engineering
Subject/Key words, **S**/**KW**:	Algorithm, data structure, microservice
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	Design and implementation of a high-performance microservice for the association frequency counting. Design and implementation of a data structure which is specialized in association frequency counting and represents core of the microservice. Evaluation of the implemented solution in local environment. Comparative analysis of the high-performance microservice for the association frequency counting and microservice for association frequency couning based on using database.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	

Defended Board, **DB**:			
	President:	Đorđe Pržulj, PhD, associate professor	
	Member:	Sonja Ristić, PhD, full professor	Menthor's sign
	Member, Mentor:	Vladimir Mandić, PhD, assistant professor	

Садржај

1. Увод 1
   1.1. Контекст и мотивација 1
   1.2. Предмет истраживања 3
   1.3. Структура рада 3
2. Теоријски концепти решења 5
   2.1. Преглед постојећих имплементација 5
   2.2. Асимптотска анализа алгоритамске сложености 5
   2.3. Основне структуре података 8
      2.3.1. Једноструко повезана листа 9
      2.3.2. Ред 10
      2.3.3. Речник 11
      2.3.4. Имплементација конкурентног речника у ДОТНЕТ радном оквиру 17
3. Захтеви решења 18
   3.1. Појмови 18
   3.2. Интерфејс микросервиса 18
   3.3. Бројач учесталости асоцијација 19
4. Дизајн решења 21
   4.1. Дизајн АФЦ-а 21
      4.1.1. Дизајн функције *упишиНовуАсоцијацију* 23
      4.1.2. Дизајн функције *прочитајКНајфреквентнијихАсоциранихКључева* 28
      4.1.3. Дизајн функције *обришиСвеАсоциранеКључевеСаАсоцијатором* 28
   4.2. Дизајн микросервиса 29
   4.3. Оптимизација коришћењем реда 29
5. Имплементација решења 30
   5.1. ДОТНЕТ 30

5.2. Структура решења 31

5.3. Имплементација АФЦ-а 31

5.4. Имплементација конкурентног реда 34

5.5. Имплементација микросервиса 35

5.6. Перзистенција података 40

5.7. Анализа перформанси 41

6. Ограничења решења 43

6.1. Ограничење бројача фреквенција АФЦ-а 43

6.2. Ограничење оперативне меморије 44

6.3. Ограничење евентуалне конзистентности 44

6.4. Ограничење екстремног конкурентног приступа АФЦ-у 45

7. Евалуација 46

7.1. Резултати тестирања микросервиса за високоперформантно бројање учесталости придруживања кључева у локалном окружењу 46

7.1.1. Резултати тестирања времена извршавања функција микросервиса 47

7.1.2. Резултати тестирања функције *упишиНовуАсоцијацију* АФЦ-а 49

7.1.3. Резултати тестирања заузећа меморијског простора микросервиса за високоперформантно бројање учесталости придруживања кључева 51

7.2. Упоредно решење – микросервис за бројање учесталости придруживања кључева имплементиран коришћењем базе података 53

7.2.1. Имплементација упоредног решења 53

7.2.2. Резултати тестирања времена извршавања операција упоредног решења. 54

7.2.3. Резултати тестирања заузећа меморијског простора упоредног решења ... 56

7.3. Поређење добијених резултата 57

8. Закључак 60

Референце 62

Прилози 64

Слике

Слика 1. Структура рада 3
Слика 2. „Тета“ нотација (слика преузета из (Cormen et al., 2009)) 6
Слика 3. „Велико о" нотација (слика преузета из (Cormen et al., 2009)) 7
Слика 4. „Омега“ нотација (слика преузета из (Cormen et al., 2009)) 7
Слика 5. Једноструко повезана листа 9
Слика 6. Ред 10
Слика 7. Хеш функција (слика преузета из (Cormen et al., 2009)) 12
Слика 8. Хеширање коришћењем технике уланчавања (слика преузета из (Cormen et al., 2009)) 13
Слика 9. Бројач учесталости асоцијација - АФЦ 22
Слика 10. Блок-дијаграм функције *упишиНовуАсоцијацију* подељене на процедуре 23
Слика 11. Алгоритам уписа нове асоцијације у АФЦ - сценарио у ком „глава“ листе није дефинисана 24
Слика 12. Алгоритам уписа асоцијације у АФЦ - итерација кроз елементе листе 25
Слика 13. Алгоритам уписа асоцијације у АФЦ - сценарио када је пронађен елемент који има исти кључ као кључ који треба да буде уписан 26
Слика 14. Приказ стања АФЦ-а у ком ПЕРК не показује ни на један елемент 26
Слика 15. Приказ стања АФЦ-а у ком је учесталост инкрементираног елемента мања од учесталости ПЕРК-а 27
Слика 16. Приказ стања у АФЦ-у у ком ПЕРК постоји а ПЕРК+ не постоји 27
Слика 17. Приказ стања у АФЦ-у у ком и ПЕРК и ПЕРК+ постоје 27
Слика 18. Блок-дијаграм функције *прочитајКНајфреквентнијихАсоцираниxКључева* . 28
Слика 19. Дијаграм класа микросервиса за високоперформантно бројање учесталости придруживања кључева 31
Слика 20. Дијаграм класа упоредног решења 54
Слика 21. Блок-дијаграм функције *упишиНовуАсоцијацију* АФЦ-а 69

Табеле

Табела 1. Преглед временске сложености извршавања операција основних структура података 8
Табела 2. Интерфејс микросервиса за високоперформантно бројање придруживања кључева 19
Табела 3. Функције АФЦ-а 20
Табела 4. Времена извршавања функције *прочитајКНајфреквентнијихАсоцираниxКључева* 48
Табела 5. Заузеће оперативне меморије од стране АФЦ-а 51
Табела 6. Резултати тестирања заузећа простора АФЦ-а на диску 52
Табела 7. Времена извршавања функције *прочитајКНајфреквентнијихАсоцираниxКључева* упоредног решења 55
Табела 8. Времена извршавања функције *обришиСвеАсоциранеКључевеСаАсоцијатором* упоредног решења 56
Табела 9. Заузеће простора на диску упоредног решења 57

Графикони

Графикон 1. Времена извршавања функције *упишиНовуАсоцијацију* 47
Графикон 2. Упис нових асоцијација: 100 различитих асоцијатора, 100.000 различитих асоцираних кључева по асоцијатору са највише 5 понављања асоцијација по асоцијатору 50
Графикон 3. Упис нових асоцијација: 100.000 различитих асоцијатора, 100 различитих асоцираних кључева по асоцијатору са највише 5 понављања асоцијација по асоцијатору 50
Графикон 4. Упис нових асоцијација: 10 различитих асоцијатора, 10.000 различитих асоцираних кључева по асоцијатору са највише 10 понављања асоцијација по асоцијатору 50
Графикон 5. Упис нових асоцијација: 10.000 различитих асоцијатора, 10 различитих асоцираних кључева по асоцијатору са највише 10 понављања асоцијација по асоцијатору 50
Графикон 6. Резултати времена извршавања функције *упишиНовуАсоцијацију* упоредног решења.. 55
Графикон 7. Приказ поређења времена извршавања функције *упишиНовуАсоцијацију* 58

Листинзи

Листинг 1. Класа `AFCList` 32

Листинг 2. Класа `Node` 32

Листинг 3. Имплементација функције *прочитајКНајфреквентнијихАсоциранихКључева* 33

Листинг 4. Класа `ConcurrentQueue` 34

Листинг 5. Операција уписа у конкурентни ред 34

Листинг 6. Операција уклањања елемента из конкурентног реда 35

Листинг 7. Регистрација зависности у `Startup` класи 36

Листинг 8. Интерфејс `IAssociationRepository` 36

Листинг 9. Инстанцирање АФЦ-а у оквиру репозиторијума асоцијација 37

Листинг 10. Интерфејс `IQueueRepository` 38

Листинг 11. Регистрација позадинских процеса 39

Листинг 12. Имплементација позадинског процеса за упис асоцијација из конкурентног реда у АФЦ 40

Листинг 13. Имплементација перзистенције АФЦ-а 41

Листинг 14. Операција уписа асоцијације у АФЦ 65

Листинг 15. Контролер веб АПИ-ја 66

Листинг 16 - Имплементација репозиторијума упоредног решења 68

Листинг 17. Конзолна апликација за симулирање слања ХТТП захтева за упис нових асоцијација 70

1. Увод

Како би се кориснику олакшало претраживање током коришћења апликација, прибегава се креирању одређених пречица за претрагу или препорука садржаја. Неколико истраживања је показало да је за задовољство корисника потребно да време одзива апликације буде до две секунде (Nah, 2004:15; Galletta et al., 2004:26; Hoxmeier et DiCesare, 2000:143), што указује на потребу да поред обезбеђења препорученог садржаја, треба да се обезбеди да он буде приказан довољно брзо. Будући да корисници током времена почињу да стварају навике (Neal et al., 2012:492), неретко се дешава да су нове претраге повезане са садржајем који је прегледан у прошлости. Водећи се претходном претпоставком, циљ овог рада је био креирање микросервиса који ће да омогући ефикасну претрагу најпосећенијег садржаја. Ипак, микросервис за високоперформантно бројање учесталости придруживања кључева, описан у овом раду, успео је да постане много универзалније решење и да употреба у контексту система за препоруку буде само једна од бројних примена.

1.1. Контекст и мотивација

Системи за препоруку постоје већ дуги низ година и развијено је више различитих врста ових система (Felfernig et al., 2014:266). Један од примера система препоруке је Гугл, познати интернет претраживач. На основу унесених карактера или кључних речи, Гуглов систем, разматрајући најпретраживаније теме, излистава неколицину препорука у виду речи за претрагу које би потенцијално могле да заинтересују корисника (Sullivan, 2018). Са друге стране, платформа Нетфликс користи другачију стратегију препоруке. Наиме, систем за препоруку који ова платформа користи је заснован на методама машинског учења и тежи да предвиди који садржај може да буде интересантан кориснику

(Falk, 2019:8). За разлику од претходних техника препоруке, мотивација за овај рад лежи у препоруци садржаја заснованој на претходним посетама и њиховој учесталости.

Са циљем максимизације поновне употребљивости пројектованог решења које треба да омогући високоперформантно бројање учесталости придруживања кључева, архитектурни стил који је коришћен јесте микросервисна архитектура. За разлику од њеног претходника – сервисно-оријентисане архитектуре, која представља монолитну архитектуру чији модули не могу да функционишу једни без других, микросервисна архитектура представља флексибилну архитектуру која се састоји од више микросервиса (Newman, 2019). Како аутор Тенес наводи, микросервис представља „процес који се извршава независно и поседује сопствену перзистентну меморију“ (Thönes et al., 2015:113). Још једна важна одлика микросервиса јесте ограничен скуп функционалности, односно чињеница да микросервис ради тачно оно чему је намењен (Thönes et al., 2015:113; Larrucea et al., 2018:96).

Бројање учесталости придруживања вредности представља технику која може да се примени у контексту бројних информационих система. Ипак, ова операција ретко представља примарни задатак информационих система, стога је веома погодно да се један овакав задатак повери на извршавање микросервису, који не мора да се налази у домену информационог система који га користи. Вредности које се придружују морају да задовоље услов јединствености – обе вредности морају да представљају јединствени идентификатор ентитета који репрезентују. Из овог разлога, термин кључа је преузет из теорије база података и у наставку рада се говори о бројању учесталости придруживања кључева.

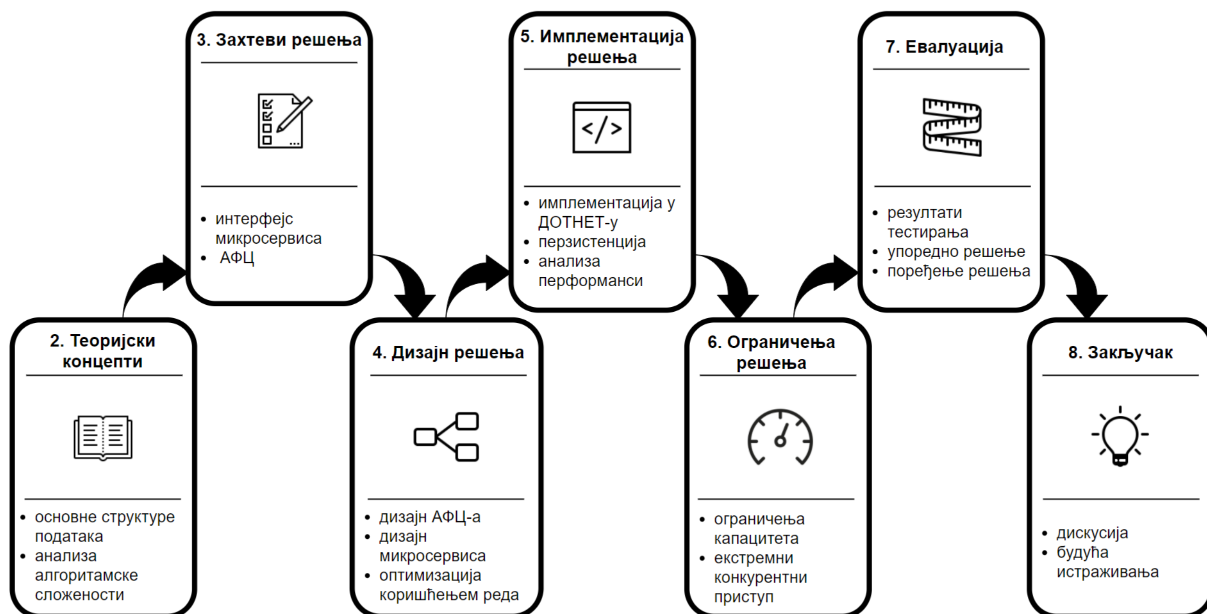
Неки од примера где би бројање учесталости придруживања кључева било веома корисно су, у контексту поменутих система за препоруку најчешће претраживаног садржаја, информациони системи за продају и резервације. Наиме, када је реч о информационим системима за продају, бројање учесталости придруживања кључева корисника и кључева категорија производа које су корисници посетили би омогућило да се кориснику препоруче други производи из најпосећенијих категорија. Аналогно примеру информационих система за продају, информациони системи за путовања би, бројањем учесталости придруживања кључева корисника и кључева држава у које је корисник путовао, могли да препоруче корисницима смештај у другим смештајним објектима у најпосећенијим државама. Поменути информациони системи, као и контекст ком припадају, представљају само неке од примера корисника микросервиса за високоперформантно бројање учесталости придруживања кључева.

1.2. Предмет истраживања

Микросервис за високоперформантно бројање учесталости придруживања кључева треба буде потпуно независан од контекста у ком се користи, као и да буде независан од типова кључева који сачињавају асоцијације. Бројање учесталости асоцирања кључева мора да се извршава са високом перформантношћу. Да би овај циљ био испуњен, микросервис у основи треба да садржи структуру података која је специјализована за бројање учесталости придруживања кључева.

1.3. Структура рада

Садржај који је изложен у остатку рада је подељен на седам поглавља. Структура поглавља је приказана на слици **Слика 1**.



Слика 1. Структура рада

У поглављу 2 су описане и детаљно анализиране основне структуре података које су од значаја за дизајн решења. Приказане су и технике које се користе за процену сложености алгоритама аналитичким путем.

У оквиру поглавља 3 су представљени захтеви решења у виду скупа функција које микросервис мора да имплементира, као и перформанси које решење мора да обезбеди. Потом је дефинисана апстрактна структура података која представља језгро

микросервиса и која је специјализована за високоперформантно бројање учесталости придруживања кључева.

Поглавље 4 приказује дизајн апстрактне структуре података, представљене у поглављу 3. Детаљно су објашњени алгоритми који могу да буду употребљени приликом имплементације ове структуре података. Након тога је објашњен дизајн микросервиса за високоперформантно бројање учесталости придруживања кључева коришћењем претходно поменуте структуре података. На крају поглавља, приказан је начин за оптимизацију решења.

Поглавље 5 приказује имплементацију микросервиса коришћењем ДОТНЕТ (енг. *.NET*) радног оквира. Имплементација у потпуности прати дизајн приказан у поглављу 4. У оквиру овог поглавља је приказан начин на који је могуће да се обезбеди перзистенција података. На крају поглавља, приказана је детаљна анализа временске и просторне сложености имплементираног решења, ослањајући се на теоријске концепте објашњене у поглављу 2.

Ограничења примећена током имплементације решења су представљена у оквиру поглавља 6. Објашњени су узроци појаве сваког ограничења и продискутовани начини како представљена ограничења могу да буду превазиђена.

У оквиру поглавља 7 су приказани и анализирани резултати који су добијени тестирањем микросервиса у локалном окружењу. Потом је приказана имплементација упоредног решења – микросервиса за бројање учесталости асоцијација коришћењем базе података. На крају поглавља, приказана је упоредна анализа претходно поменутих микросервиса.

Последње поглавље рада дискутује у којој мери је постављени циљ рада испуњен. Поред тога, у овом поглављу су дискутована унапређења имплементираног решења, као и правци у којима би ово решење могло даље да се развија у циљу популаризације.

2. Теоријски концепти решења

У поглављу које следи су представљене основне структуре података, њихове перформансе и начин на који су имплементиране. Дат је приказ асимптотске анализе, као често примењиване технике за аналитичку процену сложености алгоритама. Наведени појмови су објашњени на основу дефиниција и доказа из књиге „Увод у алгоритме“ (Cormen et al., 2009).

2.1. Преглед постојећих имплементација

Како је дискутовано у уводном делу, системи за препоруку се већ годинама активно користе. Алгоритми за препоруку садржаја који се користе у оквиру различитих система за препоруку су јако разноврсни. На основу прегледа литературе и извора података који су прикупљени током израде овог рада, у контексту система за препоруку није пронађена имплементација чији се алгоритам препоруке заснива на бројању учесталости придруживања кључева. Такође, ни ван овог контекста није пронађено решење које би могло да послужи као упоредни пример.

2.2. Асимптотска анализа алгоритамске сложености

Анализа раста времена извршавања алгоритма даје једноставан увид у ефикасност алгоритма и омогућава поређење са другим алгоритмима. Уколико посматрамо како се време извршавања повећава у односу на пораст броја улазних величина, при чему се број улазних величина повећава без ограничења, говоримо о асимптотској анализи

сложености алгоритма. Алгоритми који се покажу као асимптотски ефикаснији, често не буду ефикаснији када се посматра мали број улаза.

Нотације које се користе за асимптотско описивање времена извршавања се дефинишу у контексту функција чији домен представља скуп природних бројева проширен нулом и користе се за опис најгорег времена извршавања. Поред њихове употребе за анализу времена извршавања алгоритма, поменуте нотације могу да се користе и у контексту израчунавања просторне комплексности алгоритма, односно количине меморије коју алгоритам заузима. Функције које се најчешће користе у асимптотској анализи су:

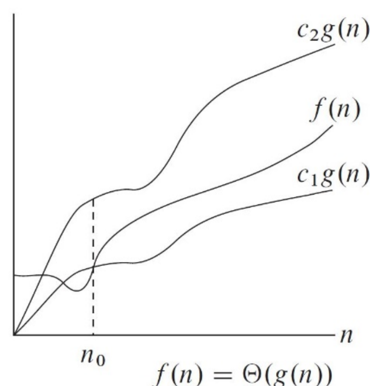
- константна функција $f(n) = 1$,
- линеарна функција $f(n) = n$,
- полиномијална функција $f(n) = n^c$; c $\in$ N $\cup$ {0} ,
- логаритамска функција $f(n) = \log(n)$,
- факторијелска функција $f(n) = n!$

и оне описују колико пута у току извршавања алгоритма треба да се обради свака појединачна улазна величина.

Ако функцију раста времена извршавања са порастом броја улаза означимо са $f(n)$ и ако важи да одређени алгоритам има најгоре време извршавања у ознаци $-\theta(g(n))$, приказана „тета" нотација означава да за дату функцију $g(n)$ постоји такав скуп функција $\theta(g(n))$ да важи:

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),$ за свако $n \geq n_0$.

Дакле, нотација $\theta(g(n))$ представља скуп функција ком функција $f(n)$ припада ако и само ако постоје такве позитивне константе c_1 и c_2, за које важи да је функција $f(n)$ омеђена функцијама $c_1 g(n)$ и $c_2 g(n)$ за довољно велики број улаза.

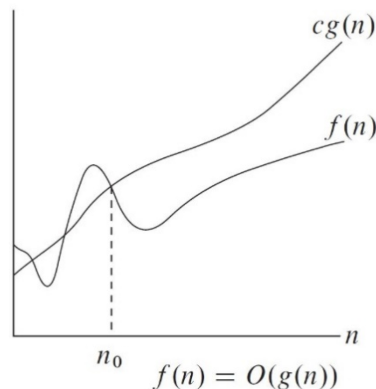


Слика 2. „Тета" нотација (слика преузета из (Cormen et al., 2009))

Нотација „велико о“ (O) представља горњу асимптотску границу функције $f(n)$. За дату функцију $g(n)$ дефинише се такав скуп функција $O(g(n))$ да важи:

$0 \leq f(n) \leq cg(n)$, за свако $n \geq n_0$.

$O(g(n))$ представља скуп функција ком функција $f(n)$ припада уколико постоји таква константа c да важи да је функција $f(n)$ ограничена функцијом $cg(n)$ са горње стране. Између O и θ нотације важи релација подскупа – $O(g(n)) \subseteq \theta(g(n))$. „Велико о“ нотација је приказана на слици **Слика 3**.

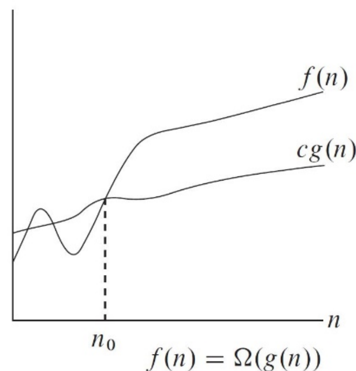


Слика 3. „Велико о" нотација (слика преузета из (Cormen et al., 2009))

Користећи O нотацију, време извршавања алгоритма може да се процени на основу посматрања структуре кода. Извршење појединачне операције је према дефиницији ограничено са $O(1)$ са горње стране. Пролазак кроз све елементе петље се извршава са временском сложеношћу $O(n)$.

Последња нотација која се разматра јесте „омега“ нотација (Ω). Овом нотацијом се означава доња асимптотска граница функције $f(n)$. За посматрану функцију $g(n)$, са $\Omega(g(n))$ означавамо скуп функција за који важи:

$0 \leq cg(n) \leq f(n)$, за свако $n \geq n_0$.



Слика 4. „Омега“ нотација (слика преузета из (Cormen et al., 2009))

Када важи да алгоритам има време извршавања у означи $\Omega(g(n))$, то значи да је за довољно велику величину улаза функција $f(n)$ ограничена функцијом $cg(n)$ са доње стране. Графички приказ тета нотације дат је на слици **Слика 4**.

Практично посматрано, уколико важи да одређени алгоритам има линеарну сложеност, то значи да ће сваки појединачни улаз да буде обрађен макар једанпут. Са друге стране, у случају полиномијалне или факторијелске сложености, свака улазна величина ће да буде обрађена више од једног пута, тако да се алгоритми које карактеришу овакве сложености сматрају лошим. Са друге стране, алгоритми који имају логаритамску или константну комплексност, током извршавања не обрађују сваки улаз појединачно и они се сматрају високоперформантним алгоритмима.

2.3. Основне структуре података

Употреба основних типова података, као што су низови и показивачи, може да послужи за изградњу комплексних структура података. Уколико се структуре података креирају само употребом основних типова података, те структуре података се називају основне структуре података. Основне структуре података које су коришћене у овом раду су једноструко повезана листа, ред и речник. Приказ њихових апстрактних структура је дат у табели **Табела 1**. „Апстрактне структуре података су класе објеката чије је понашање дефинисано скуповима вредности и операцијама“ (Dale et al., 1996:3), што значи да се у оквиру апстрактних структура података не приказује начин на који операције треба да се изврше.

Табела 1. Преглед временске сложености извршавања операција основних структура података

структуре података / **операције**	једноструко повезана листа	ред	речник
претрага задате вредности	$O(n)$	/	$O(1^*)$
упис нове вредности	$O(1)$	$O(1)$	$O(1^*)$
брисање задате вредности[1]	$O(n)$	$O(1)$	$O(1^*)$

Операције уписа, читања и брисања података представљају најважније операције у контексту структура података. У оквиру табеле **Табела 1** су приказане временске сложености извршавања поменутих операција користећи O нотацију. У тексту који следи

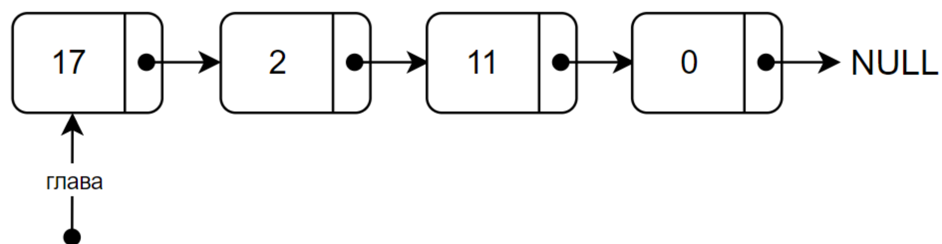
[1] Напомена: у контексту реда, временска сложеност која је приказана се односи на операцију брисања вредности карактеристичну за ред

је анализирана свака од основних структура података појединачно, односно објашњене су технике за имплементацију основних операција и продискутоване су временске сложености тих операција.

2.3.1. Једноструко повезана листа

Према дефиницији, повезана листа је структура података у којој су елементи уређени у линеарном поретку. Особина која одликује сваку листу јесте да је линеарна уређеност елемената одређена показивачима којима елементи показују једни на друге. Поред показивача, сваки елемент у листи има обележје „кључ“, које чува вредност која је уписана у листу. Такође, заједничка особина за сваку листу је да постоји показивач на први елемент који се зове глава. Постоји више облика повезаних листа, при чему су за потребе овог рада представљене само једноструко повезане листе. Код једноструко повезане листе, сваки елемент поседује само један показивач – показивач ка наредном елементу. Последњи елемент листе не показује ни на један елемент.

Будући да се за сваки елемент у листи заузима простор у меморији, једноструко повезана листа има линеарну просторну комплексност – $O(n)$.



Слика 5. Једноструко повезана листа

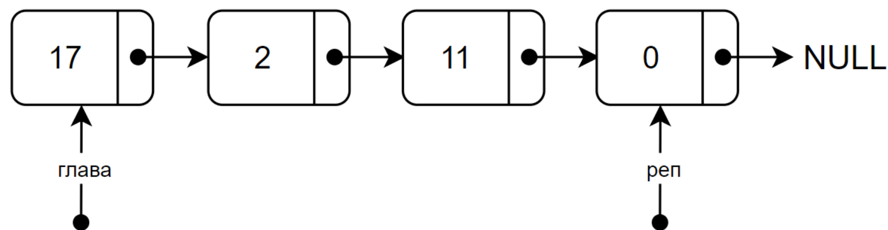
Операција читања задате вредности из листе подразумева пролазак кроз све елементе листе, све док се не пронађе тражена вредност или док се не наиђе на показивач који не показује ни на један елемент, чиме се утврђује да тражена вредност не постоји. Временска комплексност ове операције користећи тета нотацију је линеарна – $\theta(n)$.

Операција уписа нове вредности у првом кораку подразумева креирање новог објекта у меморији, што се извршава у константном времену. Наредни корак представља проналажење последњег елемента листе, чији се показивач на наредни елемент постави тако да показује на новокреирани елемент. Због потребе проласка кроз читаву листу да би се пронашао последњи елемент, временска комплексност ове операције је, користећи θ нотацију, линеарна – $\theta(n)$.

Брисање елемента који садржи одређену вредност у једноструко повезаној листи подразумева проналажење те вредности у листи и преповезивање показивача када је вредност пронађена. Преповезивање је операција која подразумева премештање показивача тако да показује на неки други елемент, чиме се врши реорганизација листе. Ова операција се извршава у константном времену – $O(1)$. Као и код операције додавања нове вредности у листу, због потребе проналажења елемента у листи пре него што се изврши преповезивање, операција брисања одређеног елемента у једноструко повезаној листи има линеарну временску комплексност према тета нотацији – $\theta(n)$.

2.3.2. Ред

Ред представља динамичку структуру података коју карактерише предефинисано брисање елемената. Брисање елемената врши се по принципу да је први елемент који је уписан у ред онај елемент који ће први да буде обрисан из реда. Ред може да се имплементира на више начина, а у наставку је објашњена имплементација путем једноструко повезане листе. Као и код једноструко повезане листе, ред се састоји од елемената који садрже обележје „кључ“ и садрже показивач ка наредном елементу. Такође, ред поседује показивач на први елемент у листи, али поседује и додатни показивач на последњи елемент, који се назива реп.



Слика 6. Ред

Као и у случају једноструко повезане листе, сваки елемент реда заузима одређени меморијски простор. У складу са тим, просторна комплексност реда је линеарна – $O(n)$.

Операција претраживања одређене вредности није карактеристична за употребу реда, стога није предмет разматрања. Операција уписа нове вредности подразумева креирање новог објекта у меморији. Након тога, показивач на наредни елемент елемента на који показује „реп“ се постави тако да показује на новокреирани објекат. Такође, и „реп“ се поставља тако да показује на новокреирани објекат. Ова операција извршава се у константном времену – $O(1)$.

Операција брисања одређене вредности из реда, која представља најзначајнију операцију реда, извршава се тако што се обрише елемент на који показује „глава“, а „глава“ се помери тако да показује на следбеник обрисаног елемента. Ова операција, као и претходна, извршава се у константном времену – $O(1)$.

2.3.3. Речник

Речник као структура података настаје због потребе за постојањем такве структуре података која треба да подржи операције уписа, брисања и провере присуства одређеног елемента у динамичком скупу података. Под динамичким скупом података се мисли на математички појам скупа, уз чињеницу да динамички скупови података могу да расту и да се смањују током времена, за разлику од математичких скупова који су фиксни. Имплементације динамичких скупова су такве да је сваки елемент представљен као објекат чијим атрибутима може да се манипулише само ако постоји показивач на тај објекат. Код неких динамичких скупова важи претпоставка да је једно од обележја које објекти поседују кључ, који служи за идентификацију. Уколико су сви кључеви различити, динамички низ можемо да посматрамо као колекцију кључева и придружених вредности.

Веома ефикасан начин за имплементацију речника су хеш табеле. Хеш табеле настају као решење проблема које са собом носи директно адресирање – још једна техника за имплементацију речника. Наиме, директно адресирање се користи када је скуп свих могућих кључева мали. Под претпоставком да сваки елемент има кључ из скупа свих могућих кључева $U = \{0, 1, 2, \dots, m-1\}$ и да је сваки кључ јединствен, за имплементацију директног адресирања користимо низ, који се још назива табела директног адресирања, и означава се са $T[0, 1, 2, \dots, m-1\}$. Свака позиција у низу се назива слот и одговара једном кључу из скупа свих кључева. Сваком слоту се придружује објекат чији кључ одговара том слоту, а уколико скуп не садржи кључ који одговара постојећем слоту, слоту се придружује вредност *NULL*. Операције претраге, уписа и брисања се реализују у константном времену – $O(1)$.

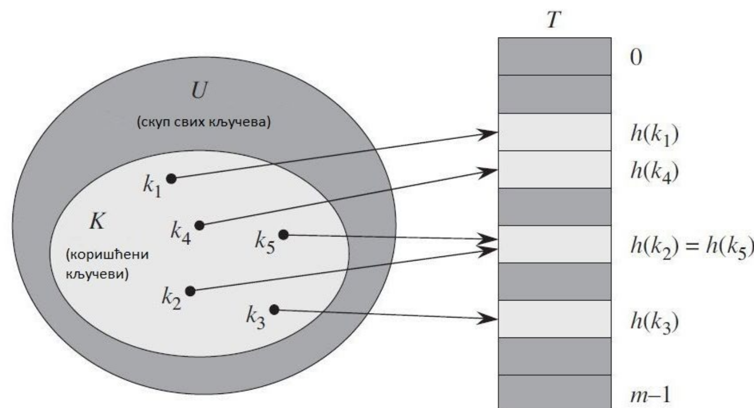
Велики недостатак директног адресирања јесте неефикасност употребе меморије у случају великог скупа свих могућих кључева, која у одређеним случајевима може да резултује немогућношћу креирања довољно велике табеле директног адресирања због ограничења меморије на рачунару. Такође, некада кључеви нису нумерички типови

података и треба да се обезбеди таква имплементација речника, која може да подржи кључеве који нису нумеричког типа.

Ови проблеми су превазиђени коришћењем технике хеширања – код директног адресирања, елемент са кључем k је смештен у слоту k, док је код хеширања елемент смештен у слоту $h(k)$, где h представља хеш функцију која пресликава кључ у слот. Формално дефинисано, хеш функција пресликава скуп свих могућих кључева U у скуп слотова хеш табеле:

$h: U \rightarrow \{0, 1, 2, \dots, m-1\}$,

при чему m представља величину хеш табеле која је много мања од величине скупа свих могућих кључева. Илустрација рада хеш функције приказана је на слици **Слика 7**.

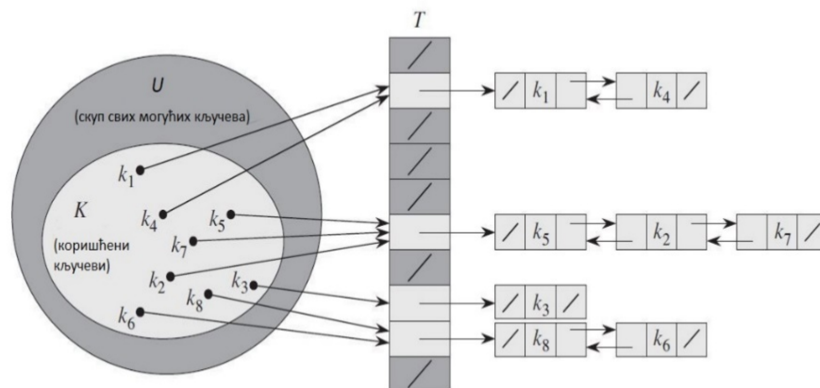


Слика 7. Хеш функција (слика преузета из (Cormen et al., 2009))

У контексту хеш табела се уводи појам фактора оптерећења, у ознаци α. Уколико са m означимо број слотова у које се смешта n елемената, фактор оптерећења представља однос броја елемената који су уписани и броја слотова. У ситуацији када би сви слотови били попуњени, вредност фактора оптерећења би била 1. Да би било могуће да се упише нова вредност у хеш табелу, мора да се повећа укупан број слотова. Када је реч о проширењу величине табеле, најбоље праксе су показале да величина треба да се удвостручи, односно потребно је да се формира нова табела у коју би се пресликале све вредности из старе табеле. Уколико би операција уписа у табелу била једина операција, вредност фактора оптерећења би увек била већа од 0,5. Како се поред операције уписа користи и њој комплементарна операција брисања, вредности фактора оптерећења могу да буду мање од 0,5. Најбоље праксе у случају брисања слотова препоручују смањивање величине табеле за половину када фактор оптерећења спадне испод $1/4$. Контролисањем вредности фактора оптерећења се постиже рационална употреба заузетог простора – са

једне стране се избегава заузимање превише меморијског простора, док се са друге стране избегава превише често реструктурирање табеле.

Када је реч о хеш функцијама, иако важи да хеш функције решавају важне проблеме, оне саме уводе један нови проблем – дешава се да два или више кључева буду пресликана у исти слот. Оваква појава се назива колизија. Пример колизије је илустрован на слици **Слика 7**, тако што се кључеви 2 и 5 пресликавају у исти слот. Због чињенице да је скуп свих могућих кључева U много већи од скупа слотова, појава колизија је неизбежна, па је тежња усмерена ка томе да се број колизија смањи што је више могуће. Постоји више начина како да се реши проблем колизије, али је за потребе овог рада у наставку представљена техника уланчавања. Техника уланчавања подразумева да се сви елементи смештају у повезану листу, при чему сваки слот чува показивач на „главу“ повезане листе или, уколико ниједан елемент није хеширан на дати слот, слот садржи *NULL*. Уколико се више кључева хешира у исти слот, сваки наредни кључ се уписује на почетак листе. Техника уланчавања је приказана на слици **Слика 8**.



Слика 8. Хеширање коришћењем технике уланчавања (слика преузета из (Cormen et al., 2009))

Након представљања структуре хеш табеле која користи принцип уланчавања, важно је да се размотре перформансе најзначајнијих функција. Наиме, претрага да ли елемент са одређеним кључем постоји у хеш табели траје онолико времена колико је потребно да се прође кроз повезану листу на коју показује слот на који се пресликао кључ елемента, будући да се операција хеширања извршава у константном – $O(1)$ времену. Једнако време је потребно да се изврше и операције уписа и брисања вредности, будући да је прво потребно да се утврди да ли се одређени елемент налази у хеш табели.

Остаје да се одговори на питање колико је просечно потребно времена да би се прошло кроз повезану листу на коју слот показује. Време проласка кроз повезану листу

директно зависи од дужине повезане листе, која је условљена начином на који су елементи распоређени на слотове хеш табеле. Најбољи распоред би представљао случај у ком би свака вредност била хеширана на различити слот, док би се у најгорем случају све вредности сместиле у један слот. Квалитет распоређивања вредности на различите слотове је одређен квалитетом употребљене хеш функције. Под претпоставком да сваки елемент има једнаку вероватноћу да се хешира у било који од слотова независно од осталих елемената, очекивана дужина листе на сваком слоту је једнака n/m, односно једнака је фактору оптерећења α. Ова претпоставка се назива једноставно униформно хеширање. Још једна претпоставка која се узима у обзир јесте да се израчунавање хеширане вредности кључа извршава у константном – $O(1)$ времену. Користећи претходно изложене претпоставке, у наставку су анализиране две теореме које тврде да је просечно време претраге постојања елемента са одређеним кључем комплексности $O(1+\alpha)$.

T1: У хеш табели која користи уланчавање за решење проблема колизије, под претпоставком једноставног униформног хеширања, просечно време за неуспешну претрагу постојања елемента са одређеном вредношћу кључа је $O(1+\alpha)$.

Доказ: користећи се чињеницом да је време израчунавања хеширане вредности кључа константне временске комплексности – $O(1)$ и чињеницом да је у случају једноставног униформног хеширања дужина листе на коју слот показује α, важи претпоставка да је просечно време потребно за неуспешно претраживање постојања елемента са одређеним кључем временске сложености $O(1+\alpha)$.

T2: У хеш табели која користи уланчавање за решење проблема колизије, под претпоставком једноставног униформног хеширања, просечно време за успешну претрагу постојања елемента са одређеном вредношћу кључа је $O(1+\alpha)$.

Доказ: узимајући у обзир да се приликом уписа новог елемента у листу на коју слот показује елемент уписује на почетну позицију (он постаје „глава“ листе), број елемената кроз које треба да се прође приликом проналажења циљаног елемента је број елемената који су уписани након траженог елемента у листи плус један. Очекивани број елемената који су уписани након циљаног елемента се добија као просек збира јединице и броја елемената који су уписани након сваког појединачног елемента из скупа n, у ознаци:

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right],$$

где X_{ij} представља случајну променљиву која моделује скуп елемената чије су хеширане вредности једнаке и које су уписане након посматраног елемента. На основу правила алгебре случајних променљивих (Mathai et Haubold, 2017:85), формула за очекивани број елемената који су уписани у исти слот постаје:

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]\right).$$

Под претпоставком једноставног униформног хеширања, вероватноћа да две променљиве буду хеширане у исти слот је $1/m$, тако да формула сада постаје:

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right).$$

Користећи се основним алгебарским операцијама, трансформишемо тренутни облик формуле у:

$$= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{nm}\left(\sum_{i=1}^{n}(n) - \sum_{i=1}^{n}(i)\right)$$

$$= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right)$$

$$= 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.$$

На основу последњег резултата закључујемо да је временска комплексност проласка кроз повезану листу реда $\theta(1+\alpha)$.

Из претходне две теореме може да се закључи да уколико је број слотова приближан броју елемената, фактор оптерећења тежи јединици, односно временска комплексност претраге елемента са траженом вредношћу кључа је константна – $O(1)$.

Током претходног разматрања је као претпоставка коришћено једноставно униформно хеширање. Поставља се питање како треба да изгледа хеш функција која ће да обезбеди једноставно униформно хеширање. Постоји више метода које могу да се употребе за дизајнирање добре хеш функције. Будући да је за имплементацију микросервиса за високоперформантно бројање придруживања кључева коришћена

Мајкрософтова имплементација речника у ДОТНЕТ радном оквиру, дат је опис технике дељења, која је коришћена у тој имплементацији.

Метода дељења користи остатак при дељењу кључа k бројем слотова m за добијање хеширане вредности и формално се дефинише као:

$h(k) = k \bmod m.$

Избор броја слотова представља врло важан корак, будући да код ове технике управо тај број диктира број колизија које ће да настану. Бројеви који треба да се избегавају јесу сви степени броја два. Најбоље праксе су показале да за број слотова треба да се користе прости бројеви који су што више удаљенији од најближих бројева који представљају степен броја два.

Последња особина посматрана у контексту хеш табела јесте просторна комплексност. Наиме, како наводи аутор Џош Триплет, уколико је хеш табела дизајнирана тако да буде мала, десиће се ситуација да ће дугачки ланци да буду креирани и перформасе ће да буду јако лоше (Triplett, 2011:1). Исти аутор такође наводи да уколико је хеш табела превише велика, заузеће се превише меморије, што такође неће добро да се одрази на перформансе. Будући да не постоји оптимална величина хеш табеле која може да се предефинише приликом креирања саме структуре, хеш табела мора динамички да мења величину током времена. Стратегије за одабир тренутка када да се промени величина хеш табеле су различите. Будући да се за потребе овог рада користи Мајкрософтова имплементација речника, у наредном одељку су приказане стратегије које се користе у тој имплементацији.

Када је реч о асимптотској анализи просторне комплексности, посматра се претпоставка да повећање броја уписаних елемената у речник, услед реструктурирања хеш табеле, доводи до дуплирања броја заузетих меморијских локација. Како важи да дизајн хеш табела настоји да број уписаних елемената у речнику буде сличан броју слотова хеш табеле, повећање броја слотова услед реструктурирања може да се посматра као $m = cn,\ c \in \mathbb{R}$. Будући да је c константа, просторна комплексност речника је линеарна – $O(n)$.

2.3.4. Имплементација конкурентног речника у ДОТНЕТ радном оквиру

У претходном одељку су дефинисани теоријски концепти на којима почива изградња речника као структуре података и уједно су представљени начини за имплементацију тих концепата. За потребе овог рада није креиран речник из почетка, већ је коришћена имплементација конкурентног речника у ДОТНЕТ радном оквиру. У овом одељку је објашњено коришћењем којих техника је имплементиран конкурентни речник у ДОТНЕТ радном оквиру.

За разлику од технике одржавања фактора оптерећења, која је дискутована у претходном одељку, тренутак када се врши повећање величине хеш табеле у имплементацији речника у ДОТНЕТ-у јесте тренутак када број колизија пређе унапред дефинисани број (Microsoft, n.d.). Након сваке колизије која настане, бројач се повећава до тренутка када достигне предефинисану границу. Када број колизија пређе предефинисану границу, величина хеш табеле се повећава на следећи прост број из предефинисане листе простих бројева (Microsoft, n.d.). Када је реч о смањивању величине хеш табеле, у понуђеном изворном коду пројекта није пронађена информација о тренутку када би се величина хеш табеле смањила и на колико се величина смањује.

Конкурентни речник имплементиран у ДОТНЕТ-у као хеш функцију користи дупло хеширање (Microsoft, n.d.). Прва хеш функција служи за рачунање хеш кода вредности која се уписује, док друга функција користи методу дељења за добијање нове вредности. Израчунате вредности се сабирају и резултат тог сабирања се користи као кључ за смештање нове вредности у речник.

3. Захтеви решења

У првом делу поглавља које следи су дефинисани захтеви у виду функција које микросервис за високоперформантно бројање учесталости придруживања кључева мора да имплементира и њихових перформанси. У другом делу поглавља је дефинисана апстрактна структура података која служи за складиштење асоцијација кључева у оквиру микросервиса и која је специјализована за претрагу најучесталијих асоцираних кључева.

3.1. Појмови

Како би се у тексту који следи избегла двосмисленост и непрецизност, важно је да се прецизно дефинише појам асоцијације кључева и да се усвоји конвенција о именовању кључева у оквиру асоцијације кључева. Асоцијација двају кључева представља бинарну релацију између првог и другог кључа, такву да важи да је други кључ асоциран од стране првог кључа. Могуће је да један кључ асоцира више различитих кључева. За први кључ се у наставку рада користи назив асоцијатор, док ће други кључ да се назива асоцирани кључ.

3.2. Интерфејс микросервиса

Микросервис за високоперформантно бројање придруживања кључева своје функционалности излаже клијентским апликацијама преко крајњих тачака (ендпоинта), којима клијентске апликације приступају коришћењем ХТТП захтева. Функције које микросервис треба да обезбеди су приказане у табели **Табела 2**.

Табела 2. Интерфејс микросервиса за високоперформантно бројање придруживања кључева

функција	повратна вредност	први параметар	други параметар
прочитајКНајфреквентнијихАсоциранихКључева	колекција кључева	асоцијатор	/
упишиНовуАсоцијацију	/	асоцијатор	асоцираниКључ
обришиСвеАсоциранеКључевеСаАсоцијатором	/	асоцијатор	/

Функција *прочитајКНајфреквентнијихАсоциранихКључева* омогућава читање предефинисаног броја најучесталијих асоцираних кључева са асоцијатором, који се прослеђује као улазни параметар. Број најучесталијих асоцираних кључева који ће да буде прочитан се експлицитно дефинише и представља фиксну вредност. Повратна вредност ове функције је колекција асоцираних кључева, сортирана у опадајућем поретку према учесталости асоцирања. Функција *упишиНовуАсоцијацију* омогућава да се нова асоцијација сачува у оквиру микросервиса. Ова функција прихвата два параметра – асоцијатор и асоцирани кључ. Прихваћени кључеви представљају придружене кључеве, односно асоцијацију кључева. Функција *обришиСвеАсоциранеКључеве* врши брисање свих кључева који су асоцирани са асоцијатором, који се прослеђује као улазни параметар. Важно је да се напомене да типови података за први и други параметар нису експлицитно дефинисани, будући да различити системи користе различите типове података за јединствене идентификаторе.

Захтев који свака функција микросервиса мора да испуни јесте постизање високих перформанси у контексту времена извршавања, односно да свака од поменутих функција мора да се извршава у константном – $O(1)$ времену.

3.3. Бројач учесталости асоцијација

Бројач учесталости асоцијација (енг. *Association frequency counter*), у даљем тексту АФЦ, представља структуру података која треба да омогући складиштење, брисање и изузетно ефикасну претрагу најучесталијих придружених кључева. Функције које АФЦ треба да обезбеди су приказане у табели **Табела 3**.

Табела 3. Функције АФЦ-а

функција	повратна вредност	први параметар	други параметар
прочитајКНајфреквентнијихАсоциранихКључева	колекција кључева	асоцијатор	бројКључева
упишиНовуАсоцијацију	/	асоцијатор	асоцираниКључ
обришиСвеАсоциранеКључевеСаАсоцијатором	/	асоцијатор	/

Функција *прочитајКНајфреквентнијихАсоциранихКључева* треба да омогући брзо читање најучесталијих кључева који су асоцирани са асоцијатором који се прослеђује као улазни параметар. Број асоцираних кључева који треба да се прочитају је прослеђен као други параметар. Повратна вредност ове функције је колекција асоцираних кључева, сортирана у опадајућем поретку према учесталости асоцирања. Временска сложеност ове функције треба да буде константна – $O(1)$. Функција *упишиНовуАсоцијацију* треба да омогући да се асоцијација кључева, који су прихваћени као улазни параметри, сачува у оквиру АФЦ-а. Висока перформантност ове функције није приоритет, будући да ова функција треба да изврши реструктурирање АФЦ-а у циљу ажурирања стања. На крају, функција *обришиСвеАсоциранеКључевеСаАсоцијатором* треба да изврши брисање свих кључева који су асоцирани са асоцијатором, који се прослеђује као улазни параметар. Као и код функције читања најучесталијих асоцираних кључева, временска сложеност функције брисања свих асоцираних кључева треба да буде $O(1)$.

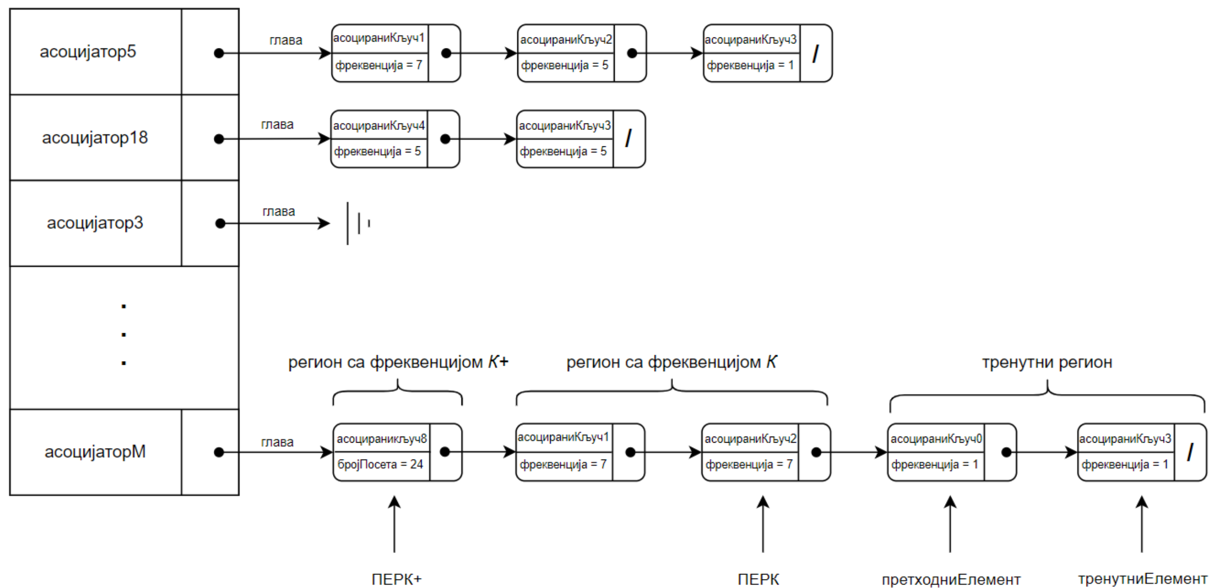
4. Дизајн решења

Након дефинисања захтева које решење мора да испуни, у поглављу које следи је објашњено на који начин би микросервис за високоперформантно бројање учесталости придруживања кључева могао да буде имплементиран. У првом делу поглавља је објашњен дизајн АФЦ-а, а потом је приказано како АФЦ може да буде употребљен у контексту микросервисне архитектуре за постизање захтева из поглавља 3. На крају поглавља, представљен је начин за побољшање перформанси микросервиса коришћењем реда.

4.1. Дизајн АФЦ-а

АФЦ је пројектован тако да његову основу чини речник. Водећи се чињеницом да микросервиси, као и други веб сервери, раде у вишенитном режиму (Silberschatz et al., 2012:161) и да је АФЦ пројектован као централно складиште асоцијација ком више нити приступа у исто време, речник који се користи мора да буде сигуран у условима конкурентног коришћења. Други градивни елемент АФЦ-а представљају једноструко повезане листе. Повезане листе се користе у АФЦ-у тако што су вредности у речнику показивачи на прве елементе повезаних листа. Асоцијације се у АФЦ-у чувају тако што се асоцијатори смештају у речник као кључеви речника. Са друге стране, асоцирани кључеви се смештају у повезане листе, и то као објекти који садрже два обележја: асоцирани кључ и учесталост асоцирања. На овај начин је постигнуто да се уместо чувања више асоцијација двају кључева сачува једна асоцијација између асоцијатора и

више асоцираних кључева, што омогућава лако бројање учесталости придруживања. АФЦ је приказан на слици **Слика 9.**.



Слика 9. Бројач учесталости асоцијација - АФЦ

Коришћење речника омогућава високу перформантност приликом приступа асоцираним кључевима. Са друге стране, елементи повезане листе су сортирани у опадајућем поретку по учесталости асоцирања, а уколико више елемената има једнаку учесталост, они су сортирани према времену уписа. На овај начин је омогућен изузетно брз приступ најучесталијим асоцираним кључевима.

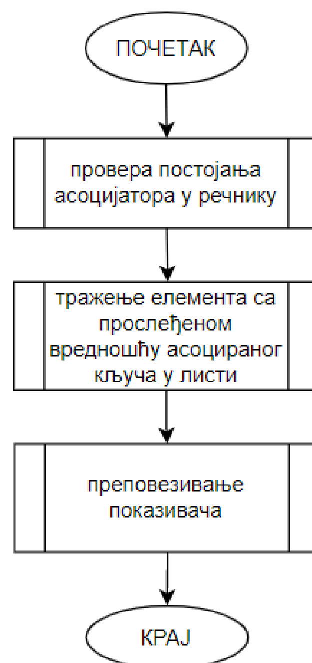
Поред речника и повезаних листа, АФЦ треба да садржи четири показивача на елементе повезане листе. Показивачи се у контексту АФЦ-а користе за повећање ефикасности извршавања функција. Показивач „тренутниЕлемент“ служи да се приликом проласка кроз повезану листу сачува информација кроз који елемент се тренутно пролази. Елемент кроз који се тренутно пролази се назива тренутни елемент. Показивач „претходниЕлемент“ служи за брз приступ претходнику тренутног елемента, будући да једноструко повезана листа не омогућава брз приступ претходнику елемента. Да би друга два показивача била дефинисана, потребно је да се дефинише појам региона у АФЦ-у. Регион представља скуп елемената једноструко повезане листе који имају једнаку фреквенцију. Због опадајуће сортираности једноструко повезаних листа према фреквенцији у оквиру АФЦ-а, региони су такође сортирани у истом поретку. Постоје три значајна региона: тренутни регион, регион са фреквенцијом К и регион са фреквенцијом К+. Тренутни регион чине сви елементи који имају фреквенцију као елемент на који показује показивач „тренутниЕлемент“, укључујући и тај елемент. Регион са фреквенцијом К представља суседни регион тренутног региона, односно први следећи

регион са већом фреквенцијом у сортираном поретку региона. Аналогно, регион са фреквенцијом К+ се дефинише као суседни регион региона са фреквенцијом К, односно први следећи регион са већом фреквенцијом у сортираном поретку региона. Када су дефинисани региони од значаја у оквиру АФЦ-а, могуће је да се дефинишу преостала два показивача. Показивач „последњиЕлементРегионаК“ (у даљем тексту ПЕРК) представља референцу на најстарије уписани елемент који се налази у региону са фреквенцијом К. Аналогно, показивач „последњиЕлементРегионаК+“ (у даљем тексту ПЕРК+) представља најстарије уписани елемент у региону са фреквенцијом К+.

У наставку поглавља је дат детаљан опис начина на који се извршавају функције АФЦ-а.

4.1.1. Дизајн функције *упишиНовуАсоцијацију*

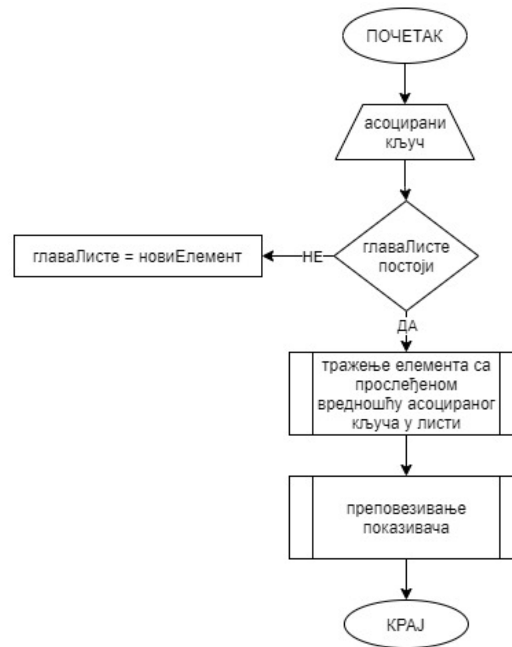
Операција уписа асоцијације представља најкомплекснију операцију у АФЦ-у. Алогритам уписа асоцијације у АФЦ је приказан као блок-дијаграм и дат је у прилогу на слици **Слика 21**. Због комплексности функција и постојања више путања извршавања алгоритма, алгоритам је подељен у три процедуре. Подела операције *упишиНовуАсоцијацију* на процедуре је приказана на слици **Слика 10**.



Слика 10. Блок-дијаграм функције *упишиНовуАсоцијацију* подељене на процедуре

Уколико вредност кључа која представља асоцијатор није претходно уписана у АФЦ, асоцијатор се уписује у речник АФЦ-а као кључ речника. Након тога се креира

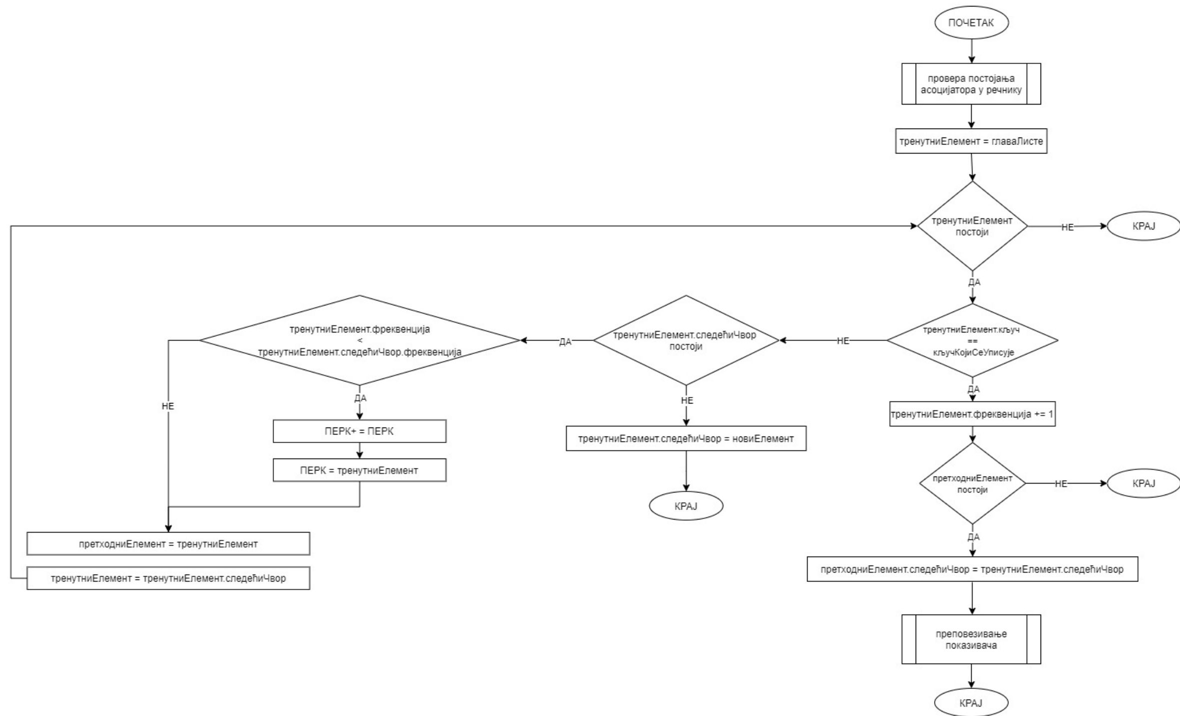
први елемент повезане листе, чија фреквенција добија вредност 1. Као вредност речника се уписује показивач на „главу“ повезане листе. Овај корак је приказан на слици **Слика 11**.



Слика 11. Алгоритам уписа нове асоцијације у АФЦ - сценарио у ком „глава“ листе није дефинисана

У случајевима када асоцијатор нове асоцијације већ постоји у речнику, потребно је да се на основу асоцијатора приступи повезаној листи АФЦ-а. Након приступа листи, потребно је да се изврши пролазак кроз повезану листу у циљу проналаска асоцираног кључа. На слици **Слика 12** је приказано итерирање кроз повезану листу. Током сваке итерације се проверава да ли је кључ елемента на који показује показивач „тренутниЕлемент“ једнак асоцираном кључу који треба да се упише. Уколико није, итерирање се наставља тако што се показивач „тренутниЕлемент“ помери тако да показује на следбеника елемента на који тренутно показује. Важно је да се након сваке итерације обезбеди да остали показивачи остану валидни. Из тог разлога се сваки пут пре померања показивача „тренутниЕлемент“ врши померање показивача „претходниЕлемент“ тако да показује на елемент на који показује показивач „тренутниЕлемент“. Ажурирање показивача ПЕРК и ПЕРК+ се врши тако што када се проласком кроз повезану листу утврди да је тренутни елемент последњи елемент региона, ПЕРК+ се промени тако да показује на елемент на који је показивао ПЕРК, а ПЕРК се промени тако да показује на елемент на који показује показивач „тренутниЕлемент“. Показивач „претходниЕлемент“ након завршене итерације показује

на елемент на који је показивао показивач „тренутниЕлемент“, а „тренутниЕлемент“ се постави да показује на свог следбеника.

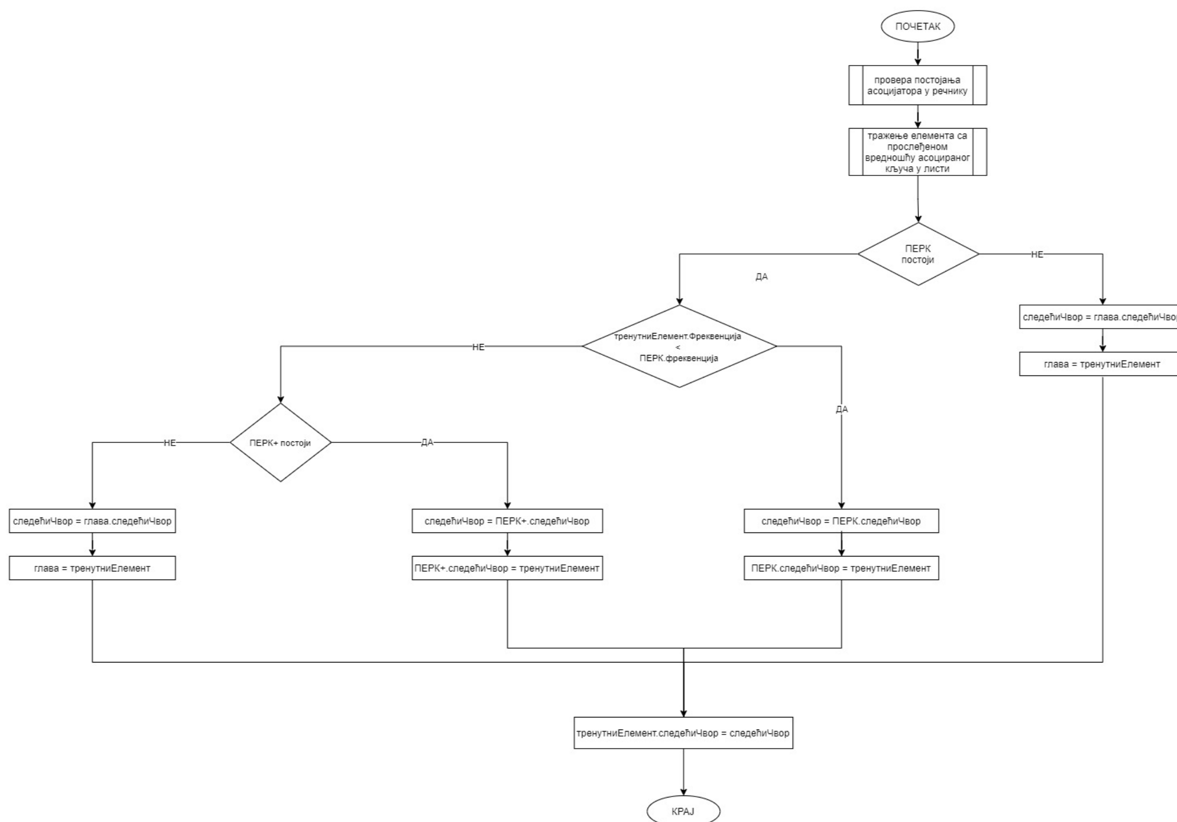


Слика 12. Алгоритам уписа асоцијације у АФЦ - итерација кроз елементе листе

Ако асоцирани кључ који се уписује није пронађен у листи, нови елемент се уписује на крај листе и његовој фреквенцији се додељује вредност један. Важно је да се напомене да уколико постоји више елемената који имају број посета једнак јединици, у том случају се не примењује правило да су ти елементи сортирани у повезаној листи према времену уписа. Наиме, асоцирани кључеви који су први пут уписују у листу се не сматрају најпосећенијим ентитетима и могу да представљају случајну посету. Из тог разлога се овакве асоцијације уписују на крај листе.

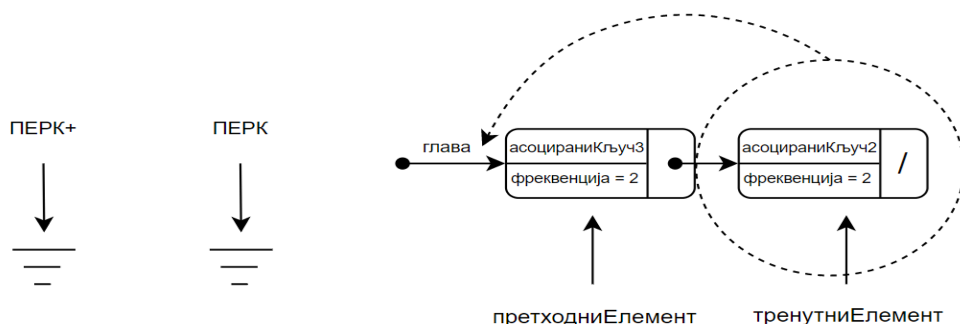
Када се приликом уписа асоцијације у повезану листу утврди да асоцирани кључ који се уписује већ постоји у листи и елемент који садржи тај кључ буде пронађен, вредност фреквенције елемента се инкрементира. На блок-дијаграму на слици **Слика 12** је приказано да је први корак након инкрементирања провера да ли постоји претходни елемент. На овај начин се утврђује да ли је елемент чија се фреквенција инкрементира глава листе и уколико јесте, у том случају се не врши преповезивање елемената.

Након што је утврђено да претходни елемент постоји, врши се ажурирање показивача „претходниЕлемент“. Наставак алгоритма подразумева преповезивање показивача и промену позиције елемента чија фреквенција је инкрементирана. Приказ наставка алгоритма је дат на слици **Слика 13**.



Слика 13. Алгоритам уписа асоцијације у АФЦ - сценарио када је пронађен елемент који има исти кључ као кључ који треба да буде уписан

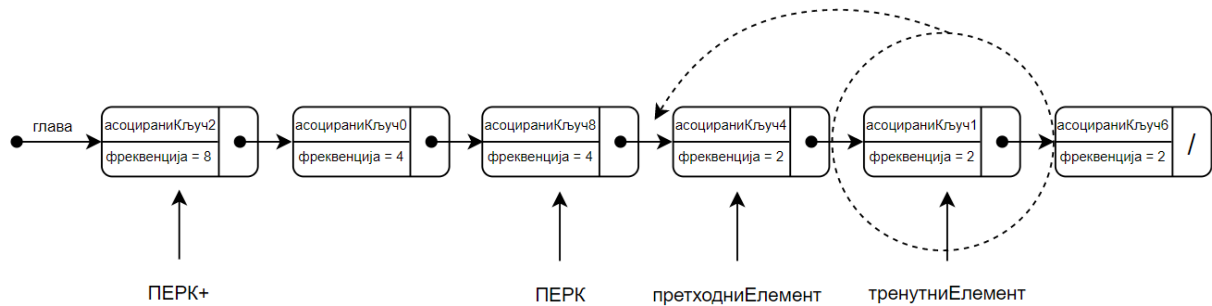
Прва провера треба да утврди да ли ПЕРК показује на неки елемент. Уколико не показује, реч је о проласку кроз први регион у листи. На слици **Слика 14** је приказан сценарио у ком показивач „претходниЕлемент" постоји, а ПЕРК не постоји. Уколико се стање промени тако што се упише асоцијација са асоцираним кључем 2, тај елемент се помера тако да постаје глава листе.



Слика 14. Приказ стања АФЦ-а у ком ПЕРК не показује ни на један елемент

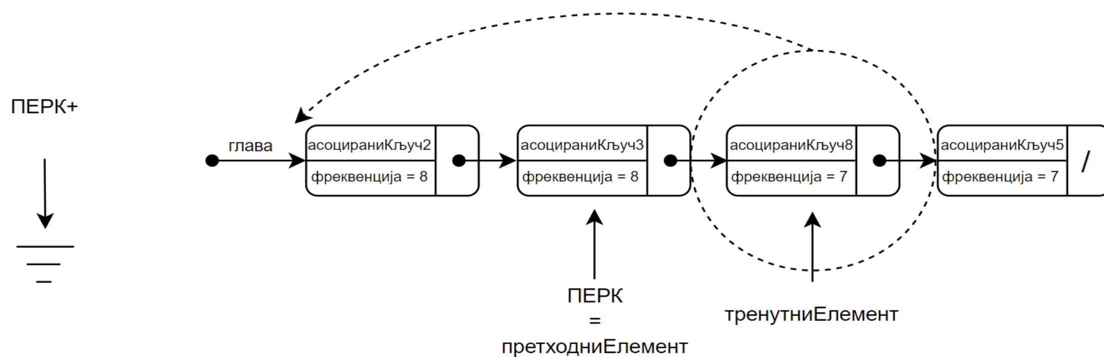
Када је утврђено да ПЕРК постоји, потребно је да се изврши провера да ли је фреквенција инкрементираног елемента мања од фреквенције ПЕРК-а. На овај начин је утврђено да је разлика између фреквенција ПЕРК-а и инкрементираног елемента већа од један и да је ПЕРК довољан за реструктурирање повезане листе. Дискутовани сценарио је приказан

на слици **Слика 15**, тако што се приказано стање мења уписом асоцијације са асоцираним кључем 1.



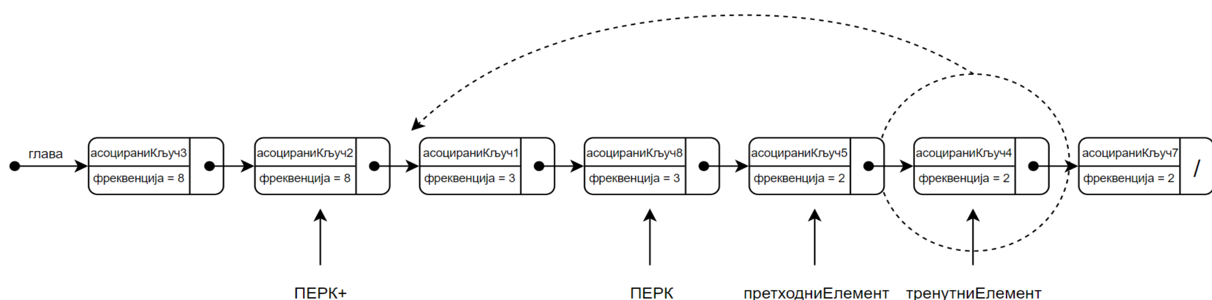
Слика 15. Приказ стања АФЦ-а у ком је учесталост инкрементираног елемента мања од учесталости ПЕРК-а

Када је утврђено да је фреквенција инкрементираног елемента није мања од фреквенције ПЕРК-а, проверава се постојање показивача ПЕРК+, који се користи у преповезивању. Ако ПЕРК+ не постоји, ради се о ситуацији где инкрементирани елемент треба да постане глава листе. Овај сценарио је приказан на слици **Слика 16**.



Слика 16. Приказ стања у АФЦ-у у ком ПЕРК постоји а ПЕРК+ не постоји

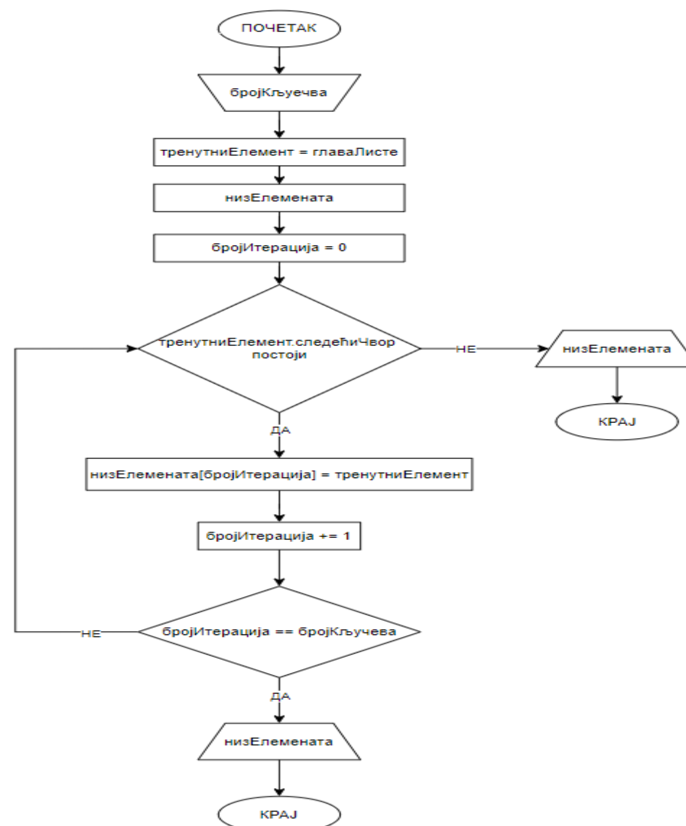
Последњи сценарио је уједно и најучесталији. Наиме, када постоје и ПЕРК и ПЕРК+, инкрементирани елемент се поставља на почетак региона *К*. Ова операција се извршава коришћењем показивача ПЕРК+, тако што се инкрементирани елемент постави тако да буде следбеник елемента на који показује ПЕРК+ и изврши се потребно преповезивање елемената листе помоћу показивача „претходниЕлемент“. Овакав случај је приказан на слици **Слика 17**.



Слика 17. Приказ стања у АФЦ-у у ком и ПЕРК и ПЕРК+ постоје

4.1.2. Дизајн функције *прочитајКНајфреквентнијихАсоциранихКључева*

Операција претраге најучесталијих асоцираних кључева подразумева приступ повезаној листи на основу асоцијатора, који се прихвата као улазни параметар. Након приступа повезаној листи, врши се пролазак кроз елементе листе док се не прочита потребан број кључева, који је такође прослеђен као улазни параметар ове функције. Блок-дијаграм функције *прочитајКНајфреквентнијихАсоциранихКључева* је приказан на слици **Слика 18**.



Слика 18. Блок-дијаграм функције *прочитајКНајфреквентнијихАсоциранихКључева*

4.1.3. Дизајн функције *обришиСвеАсоциранеКључевеСаАсоцијатором*

Функција *обришиСвеАсоциранеКључевеСаАсоцијатором* треба да изврши брисање асоцијатора из речника, чиме се уклања показивач на први елемент повезане листе, будући да се вредност брише из речника када је обрисан кључ. Када структура у

оперативној меморији остане без показивача, сакупљач смећа (енг. *Garbage collector)* уклања дату структуру из меморије (Jones, 2011:3). На овај начин су избрисани сви асоцирани кључеви.

4.2. Дизајн микросервиса

Иницијално, микросервис за високоперформантно бројање учесталости придруживања кључева је дизајниран тако да функције микросервиса само позивају истоимене функције АФЦ-а. Дизајн АФЦ-а, дискутован у претходном одељку, омогућава високоперформантно читање најучесталијих придружених кључева тако што његова унутрашња структура омогућава једноставност извршавања ове операције. Са друге стране, операција уписа нове асоцијације представља уско грло ове структуре података. Наиме, операција уписа нове асоцијације настоји да одржи унутрашњу структуру АФЦ-а погодном за операцију читања најучесталијих придружених кључева. Како би микросервис за високоперформантно бројање учесталости придруживања кључева могао да испуни захтеве дефинисане у поглављу 3, операција уписа нове асоцијације се не ослања у потпуности на АФЦ. Реализација операције уписа нове асоцијације на нивоу микросервиса је описана у одељку који следи.

4.3. Оптимизација коришћењем реда

Зарад постизања високе перформантности операције уписа нове асоцијације на нивоу микросервиса, врши се привремено складиштење асоцијације коришћењем реда. Наиме, када микросервису стигне ХТТП захтев којим се уписује нова асоцијација кључева, уместо директног уписа асоцијације у АФЦ, оно што микросервис треба да изврши јесте упис асоцијације у ред. У блиском тренутку у будућности, позадински процес који се извршава у оквиру микросервиса преузима асоцијације из реда и уписује их у АФЦ. Будући да је ред, као што је дискутовано у случају речника, ресурс ком се приступа конкурентно из различитих нити, ред који се користи за оптимизацију операције уписан мора да буде сигуран са становишта конкурентног коришћења. На овај начин се постиже да клијентска апликација, која шаље ХТТП захтев микросервису, не чека на извршавање „скупе“ операције уписа нове асоцијације у АФЦ пре него што добије одговор.

5. Имплементација решења

У овом поглављу је представљен ДОТНЕТ као технологија која је коришћена за имплементацију решења. Објашњена је имплементација АФЦ-а на основу дизајна из претходног поглавља. Након тога је приказана имплементација конкурентног реда, а потом имплементација микросервиса коришћењем претходно имплементираних структура података. Посебан одељак овог поглавља објашњава начин на који се врши перзистенција података у оквиру микросервиса. На крају поглавља, приказана је аналитичка процена временске комплексности решења.

5.1. ДОТНЕТ

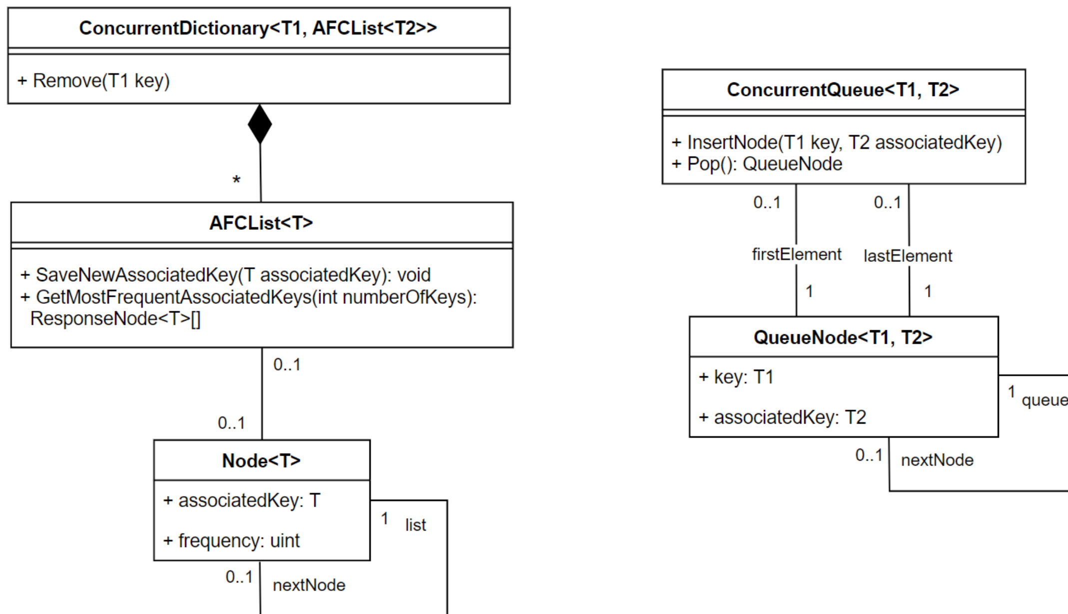
ДОТНЕТ (енг. *.NET*) је платформа за развој апликација која се састоји из алата, програмских језика и библиотека за креирање различитих врста апликација. Постоји више имплементација ДОТНЕТ-а, при чему је у овом раду коришћен ДОТНЕТ кор (енг. *.NET Core*), верзије 3.1. ДОТНЕТ кор представља вишеплатформску имплементацију ДОТНЕТ-а за развој веб сајтова, сервиса, конзолних и других врста апликација. (Troelsen & Japikse, 2017)

ДОТНЕТ омогућава имплементацију коришћењем неколико програмских језика. Микросервис за високоперформантно бројање учесталости придруживања кључева је имплементиран коришћењем програмског језика Ц# (енг. *C#*). Ц# је модеран, објектно -оријентисани програмски језик. Ц# је сигуран са становишта типова података, јер спречава грешке типова током компајлирања. (Troelsen & Japikse, 2017)

5.2. Структура решења

Имплементирано решење се састоји из два пројекта. Први пројекат представља веб АПИ пројекат, који је креиран помоћу истоименог шаблона у ДОТНЕТ-у. Овај пројекат садржи предефинисана подешавања за прихватање ХТТП захтева и њихово рутирање до ендпоинта. Веб АПИ пројекат такође омогућава лаку регистрацију зависности и спецификацију животног циклуса тих зависности.

Други пројекат представља библиотека класа (енг. *Class Library*), која је коришћена за креирање структура података. Коришћењем библиотеке класа се постиже додатна прегледност и омогућена је поновна употребљивост свих компоненти које се налазе унутар ње. Веб АПИ пројекат референцира библиотеку класа и користи структуре података које су сачуване у њој. На слици **Слика 19** је приказан дијаграм класа имплементираног решења.



Слика 19. Дијаграм класа микросервиса за високоперформантно бројање учесталости придруживања кључева

5.3. Имплементација АФЦ-а

На основу дизајна из претходног поглавља, АФЦ је имплементиран тако да његову основу чини речник, чији су кључеви асоцијатори, а вредности су показивачи на посебне врсте сортираних повезаних листа, карактеристичних за АФЦ. У складу са дефиницијом, АФЦ се инстанцира на следећи начин:

```
IDictionary<TKey, AFCList<TAssociatedKey>> afc = new
ConcurrentDictionary<TKey, AFCList<TAssociatedKey>>(); .
```

Како је имплементација конкурентног речника у ДОТНЕТ-у генеричка структура података, омогућено је да тип података који ће да се користи за асоцијаторе буде дефинисан приликом инстанцирања речника. Класа `AFCList` представља генеричку, сортирану повезану листу, карактеристичну за АФЦ. У имплементацији која је дискутована у овом пројекту, оба кључа су типа `GUID`, при чему овај тип података означава 128 битни јединствени идентификатор (Troelsen & Japikse, 2017:313). Код за имплементацију класе `AFCList` је приказан у листингу **Листинг 1**.

```
1      [Serializable()]
2      public class AFCList<T>
3      {
4          private Node<T> firstElement;
5          public AFCList()
6          {
7              firstElement = null;
8          }
9          public void InsertElement(T key) { … }
10         public IEnumerable<ResponseNode<T>> GetFirstKElements(
11                                                                int numberOfElements)
12         { … }
13     }
```

Листинг 1. Класа `AFCList`

Класа `AFCList` поседује само једно приватно обележје, које представља „главу" листе. Ово обележје је типа класе `Node`. Имплементација класе `Node` је приказана у листингу **Листинг 2**.

```
1      [Serializable()]
2      public class Node<T>
3      {
4          public T Key { get; set; }
5          public uint Frequency { get; set; }
6          public Node<T> NextNode { get; set; }
7
8          public Node(T associatedKey)
9          {
10             AssociatedKey = associatedKey;
11             Frequency = 1;
12             NextNode = null;
13         }
14         public bool HasNext()
15         {
16             return this.NextNode == null ? false : true;
17         }
18 }
```

Листинг 2. Класа `Node`

Обележја класе `Node` су асоцирани кључ, фреквенција и показивач на следећи чвор у листи, који је типа класе `Node`. Сваки чвор приликом инстанцирања добија подразумеване вредности за фреквенцију и следећи чвор, како је приказано у листингу **Листинг 2**. На овај начин је омогућено да класа `AFCList` буде листа повезаних чворова, који чувају информацију о асоцираном кључу и броју учесталости асоцирања кључева.

У оквиру класе `AFCList` су имплементиране операције уписа нове асоцијације и читања најфреквентнијих асоцираних кључева, дефинисаних у одељку 3.3. Функција *упишиНовуАсоцијацију* је имплементирана као функција `SaveNewAssociatedKey`, при чему имплементација у потпуности прати алгоритам описан у оквиру дизајна. Код функције `SaveNewAssociatedKey` је дат у прилогу у оквиру листинга **Листинг 14**. Функција *прочитајКНајфреквентнијихАсоциранихКључева* је имплементирана као функција `GetMostFrequentAssociatedKeys`. Коришћењем бројача `counter` се пролазак кроз листу ограничава на број итерација једнак броју асоцираних кључева који овом функцијом треба да буду прочитани. Када бројач достигне максималну вредност, или када је утврђено да тренутни елемент нема следбеника, пролазак кроз листу се завршава и функција враћа сортирану колекцију објеката који садрже асоциране кључеве и њихове фреквенције. Имплементација функције која чита најфреквентније асоциране кључеве је приказана у листингу **Листинг 3**.

```
 1      public IEnumerable<ResponseNode<T>> GetMostFrequentAssociatedKeys(
 2                                                              int numberOfKeys)
 3      {
 4          int counter = 0;
 5          var placeHolder = firstElement;
 6          while (placeHolder != null)
 7          {
 8              yield return new ResponseNode<T> {
 9                  AssociatedKey = placeHolder.AssociatedKey,
10                  Frequency = placeHolder.Frequency
11              };
12              counter++;
13              placeHolder = placeHolder.NextNode;
14              if (counter == numberOfKeys) break;
15          }
16      }
```

Листинг 3. Имплементација функције *прочитајКНајфреквентнијихАсоциранихКључева*

Последња функција, *обришиСвеАсоциранеКључевеСаАсоцијатором* је имплементирана ван класе `AFCList`. Наиме, за имплементацију ове функције је довољно да се позове функција конкурентног речника `Remove`, којој се прослеђује асоцијатор као улазни параметар.

5.4. Имплементација конкурентног реда

Имплементација конкурентног реда који се користи за оптимизацију операције уписа асоцијације на нивоу микросервиса не одступа у великој мери од дефиниције реда која је дата у одељку 2.3.2. Класа `ConcurrentQueue` репрезентује конкуренти ред и садржи два приватна обележја – показивач на први и последњи елемент реда. Код класе је приказан у листингу **Листинг 4**.

```
1     [Serializable()]
2     public class ConcurrentQueue<TKey, TAssociatedKey>
3     {
4         private QueueNode<TKey, TAssociatedKey> _head;
5         private QueueNode<TKey, TAssociatedKey> _tail;
6         private readonly object balanceLock = new object();
7
8         public ConcurrentQueue()
9         {
10            _head = null;
11            _tail = null;
12        }
13                  …
14    }
```

Листинг 4. Класа `ConcurrentQueue`

Чворови који се налазе у основи конкурентног реда су типа класе `QueueNode`. Ова класа садржи три обележја – асоцијатор, асоцирани кључ и показивач на следећи чвор.

Функције за додавање и избацивање елемента из реда су имплементиране тако да буду сигурне са становишта конкурентног коришћења. Наиме, за обезбеђење сигурности приликом конкурентног коришћења, у реду се користи концепт закључавања. Закључавање је механизам који спречава друге нити да уђу у одређени блок кода све док тај блок не буде ослобођен од стране нити која га је заузела (Silberschatz, 2018:270). У листингу **Листинг 5** је приказана операција додавања елемента у ред.

```
1 public void InsertNode(QueueNode<TKey, TAssociatedKey> newNode)
2 {
3     lock(balanceLock)
4     {
5         if (_tail == null) _head = _tail = newNode;
6         else
7         {
8             _tail.NextNode = newNode;
9             _tail = newNode;
10        }
11    }
12 }
```

Листинг 5. Операција уписа у конкурентни ред

Приликом позива функције додавања асоцијације у ред, врши се закључавање читавог блока. Проверава се да ли постоји „реп“, и уколико не постоји, елемент који се додаје у ред постаје „реп“ реда. Ако „реп“ реда постоји, елемент који се додаје у ред постаје нови „реп“ реда.

Позив функције избацивања елемента из реда такође подразумева закључавање блока кода. Након што је блок закључан, проверава се да ли у реду постоје сачувани чворови. Уколико не постоје, функција враћа вредност *NULL*. Са друге стране, врши се провера да ли је елемент на који показује „глава“ једини елемент у низу. Уколико јесте, у том случају се „глава“ и „реп“ постављају на *NULL*, а функција враћа елемент који је избачен из реда. У супротном, „глава“ се поставља да показује на следбеника елемента на који је „глава“ претходно показивала, а функција враћа елемент на који је показивач „глава“ претходно показивао. Приказ ове функције дат је у листингу **Листинг 6**.

```
 1 public QueueNode<TKey, TAssociatedKey> Pop()
 2 {
 3     lock (balanceLock)
 4     {
 5         if (_head == null)
 6         {
 7              return null;
 8         }
 9         if (_head.NextNode == null)
10         {
11             var nodeToReturn = _head;
12             _head = _tail = null;
13             return nodeToReturn;
14         } else
15         {
16              var nodeToReturn = _head;
17              _head = _head.NextNode;
18              return nodeToReturn;
19         }
20     }
21 }
```

Листинг 6. Операција уклањања елемента из конкурентног реда

5.5. Имплементација микросервиса

Док су имплементације АФЦ-а и конкурентног реда реализоване у оквиру пројекта типа библиотеке класа, микросервис је имплементиран као веб АПИ пројекат. Микросервис за високоперформантно бројање учесталости придруживања кључева поседује само један контролер који садржи три ендпоинта – ендпоинт за упис нове

асоцијације, ендпоинт за читање најучесталијих асоцираних кључева и ендпоинт за брисање свих асоцираних кључева. Код за имплементацију контролера приказан је у листингу **Листинг 15**. Логика контролера је изузетно једноставна – све што контролер ради јесте позивање функција репозиторијума. Контролер поседује референцу на репозиторијум асоцијација, који се прослеђује контролеру коришћењем принципа инјекције зависности (енг. *dependency injection*) кроз конструктор. Захваљујући олакшаној регистрацији зависности која је подржана у оквиру веб АПИ пројекта, репозиторијум асоцијација се региструје у оквиру класе `Startup` тако да његов животни циклус буде синглтон (енг. *singleton*). Коришћењем синглтона је омогућено постојање само једне инстанце класе током животног циклуса апликације, без обзира где се инјектује као зависност (Troelsen & Japikse, 2017:1285). Регистрација зависности приказана је у листингу **Листинг 7**.

```
 1    public class Startup
 2    {
 3        …
 4        public void ConfigureServices(IServiceCollection services)
 5        {
 6            services.AddControllers();
 7            services.AddSingleton<IAssociationRepository <Guid, Guid>,
 8                                  AssociationRepository <Guid, Guid>>();
 9            services.AddSingleton<IQueueRepository <Guid, Guid>,
10                                  QueueRepository<Guid, Guid>>();
11        }
12        …
13    }
```

Листинг 7. Регистрација зависности у `Startup` класи

Репозиторијум асоцијација је имплементиран помоћу класе `AssociationRepository`, која имплементира интерфејс `IAssociationRepository`. Код интерфејса `IAssociationRepository` је приказан у листингу **Листинг 8**.

```
1    public interface IAssociationRepository<TKey, TAssociatedKey>
2    {
3        IEnumerable<ResponseNode<TAssociatedKey>>
4        GetMostFrequentAssociatedKeys(TKey key);
5        void SaveNewAssociation(TKey key, TAssociatedKey associatedKey);
6        void DeleteAllAssociatedKeys(TKey key);
7    }
```

Листинг 8. Интерфејс `IAssociationRepository`

Класа `AssociationRepository` представља место одакле треба да се позивају функције АФЦ-а. Из тог разлога је АФЦ имплементиран као обележје репозиторијума и инстанциран је у оквиру конструктора. Инстанцирање АФЦ-а у оквиру репозиторијума се извршава тако што се прво покуша учитавање перзистиране структуре са диска.

Уколико АФЦ није учитан са диска, креира се нова инстанца. Перзистенција АФЦ-а је детаљније дискутована у наредном одељку. Инстанцирање АФЦ-а у оквиру репозиторијума је представљено у листингу **Листинг 9**.

```
 1         private readonly IDictionary<TKey, AFCList<TAssociatedKey>> _afc;
 2         private readonly ILogger<AssociationRepository<TKey, TAssociatedKey>>
 3 _logger;
 4         private readonly IQueueRepository<TKey, TAssociatedKey>
 5 _queueRepository;
 6         private readonly IConfiguration _configuration;
 7         private readonly string _fileName = "afc.bin";
 8
 9         public AssociationRepository(ILogger<AssociationRepository<TKey,
10 TAssociatedKey>> logger, IQueueRepository<TKey, TAssociatedKey> queueRepository,
11 IConfiguration configuration)
12         {
13             _logger = logger;
14             var path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
15 _fileName);
16             if (File.Exists(path))
17             {
18                 try
19                 {
20                     Stream openFileStream = File.OpenRead(path);
21                     BinaryFormatter deserializer = new BinaryFormatter();
22                     Dictionary<TKey, AFCList<TAssociatedKey>> basicDictionary =
23 (Dictionary<TKey,
24 AFCList<TAssociatedKey>>)deserializer.Deserialize(openFileStream);
25                     openFileStream.Close();
26                     _afc = new ConcurrentDictionary<TKey,
27 AFCList<TAssociatedKey>>(basicDictionary);
28                     basicDictionary = null;
29                     GC.Collect();
30                     GC.WaitForPendingFinalizers();
31                     logger.LogInformation($"AFC loaded successfully at
32 {DateTime.Now.ToString(@"MM\/dd\/yyyy h\:mm tt")}");
33                 }
34                 catch (Exception e)
35                 {
36                     logger.LogError(e, "Error loading AFC");
37                 }
38             } else
39             {
40                 _afc = new ConcurrentDictionary<TKey,
41 AFCList<TAssociatedKey>>();
42                 logger.LogWarning("New AFC created");
43             }
44             _queueRepository = queueRepository;
45             _configuration = configuration;
46         }
```

Листинг 9. Инстанцирање АФЦ-а у оквиру репозиторијума асоцијација

Функције `GetMostFrequentAssociatedKeys` и `DeleteAllAssociatedKeys` се потпуно ослањају на АФЦ и оне се извршавају тако што се позову истоимене функције АФЦ-а. Приликом позива функције `GetMostFrequentAssociatedKeys`, истоименој функцији АФЦ-а треба да се проследи број асоцираних кључева који треба да буде прочитан. Овај број је дефинисан у оквиру конфигурационих фајлова и учитава се у оквиру репозиторијума асоцијација. Са друге стране, функција `SaveNewAssociation` се извршава другачије. Наиме, када нова асоцијација треба да буде уписана, она се прво уписује у конкурентни ред. Да би упис у конкурентни ред био могућ, репозиторијум посећених ентитета претраге мора да има референцу на репозиторијум реда. Референца на репозиторијум реда се прослеђује кроз конструктор коришћењем инјекције зависности. Животни циклус ове зависности је такође регистрован као синглтон, што може да се види на листингу **Листинг 7**. Репозиторијум реда се имплементира помоћу класе `QueueRepository`, која имплементира интерфејс `IQueueRepository`. Код овог интерфејса је приказан на листингу **Листинг 10**.

```
1     public interface IQueueRepository
2     {
3         void InsertNode(QueueNode node);
4         QueueNode Pop();
5     }
```

Листинг 10. Интерфејс `IQueueRepository`

Репозиторијум реда је место где се врше позиви функција конкурентног реда. Као што је АФЦ инстанциран унутар репозиторијума посећених ентитета, на исти начин се класа `ConcurrentQueue` инстанцира унутар конструктора `QueueRepository` класе.

Асоцијације које су сачуване у реду морају да буду уписане у АФЦ како би постале видљиве приликом читања најпосећенијих асоцираних кључева. Упис асоцијација у АФЦ је реализован помоћу позадинског процеса који се активира након једнаких временских интервала. Конфигурација позадинског процеса која се дефинише у фајлу `Program.cs` је приказана у листингу **Листинг 11**.

```
 1    public static IHostBuilder CreateHostBuilder(string[] args) =>
 2            Host.CreateDefaultBuilder(args)
 3                …
 4                .ConfigureServices((hostContext, services) =>
 5                {
 6                    services.AddHostedService<InsertingJob<Guid, Guid>>();
 7                    services.AddHostedService<PersistingJob>();
 8                })
 9                …
10    }
```

Листинг 11. Регистрација позадинских процеса

Класа `InsertingJob` имплементира интерфејс `IHostedService`, који овој класи омогућава да буде регистрована као позадински процес. Када се позадински процес активира, позива се метода `StartAsync` класе `InsertingJob`. Ова метода пролази кроз све елементе конкурентног реда, ком приступа преко репозиторијума реда. Референца на репозиторијум реда се класи `InsertingJob` прослеђује кроз конструктор, коришћењем инјекције зависности. Приликом проласка кроз елементе конкурентног реда, сваки елемент бива избачен из реда и додат у АФЦ. Код класе `InsertingJob` је дат у листингу **Листинг 12**.

```
 1 public class InsertingJob<TKey, TAssociatedKey> : IHostedService,
 2 IDisposable
 3     {
 4         private readonly IQueueRepository<TKey, TAssociatedKey>
 5 _queueRepository;
 6         private readonly IAssociationRepository<TKey, TAssociatedKey>
 7 _associationRepository;
 8         private Timer _timer;
 9         private bool _loaded = false;
10
11         public InsertingJob(IQueueRepository<TKey, TAssociatedKey>
12 queueRepository, IAssociationRepository<TKey, TAssociatedKey>
13 associationRepository)
14         {
15             _queueRepository = queueRepository;
16             _associationRepository = associationRepository;
17         }
18
19         public Task StartAsync(CancellationToken cancellationToken)
20         {
21             _timer = new Timer(t =>
22             {
23                 if (!_loaded) _loaded = true;
24                 else
25                 {
26                     var placeHolder = _queueRepository.Pop();
27                     while (placeHolder != null)
28                     {
29                         ((AssociationRepository<TKey,
30 TAssociatedKey>)_associationRepository).SaveAssociationInBackground(pla
31 ceHolder.Key, placeHolder.AssociatedKey);
32                         placeHolder = _queueRepository.Pop();
33                     }
34                 }
35             }, null, TimeSpan.Zero, TimeSpan.FromMinutes(2));
36             return Task.CompletedTask;
37         }
38
39         public Task StopAsync(CancellationToken cancellationToken)
40         {
41             return Task.CompletedTask;
```

```
42          }
43
44          public void Dispose()
45          {
46              _timer?.Dispose();
47          }
48      }
```

Листинг 12. Имплементација позадинског процеса за упис асоцијација из конкурентног реда у АФЦ

5.6. Перзистенција података

Зарад постизања високих перформанси у погледу времена извршавања, АФЦ је дизајниран као структура података која се налази у оперативној меморији. Како важи да је оперативна меморија привремена меморија (Silberschatz, 2012:454), подаци смештени у АФЦ-у морају да буду сачувани трајно како би се спречило њихово губљење услед губитка напајања или престанка рада програма. Такође, конкурентни ред, у који се иницијално уписују нове асоцијације, може да садржи асоцијације у тренутку престанка рада програма, стога је неопходно да се и ова структура података трајно сачува.

За трајно чување ове две структуре података коришћен је принцип бинарне серијализације објеката. „Бинарна серијализација објеката представља серијализацију у којој су бајтови објеката записани на диск у онаквом стању у каквом се налазе“ (Soukup et Macháček, 2014:67). Перзистенција података се дешава периодично тако што се позива позадински процес који позива методе репозиторијума за перзистенцију података. Регистрација периодичног процеса који се активира на сваких 5 минута је приказана на листингу **Листинг 11**. Метода која се користи за перзистенцију АФЦ-а је приказана на листингу **Листинг 13**, док се аналогно имплементира и метода за серијализацију конкурентног реда.

```
1
2  public void PersistAFC()
3          {
4              StringBuilder builder = new StringBuilder();
5              builder.Append(DateTime.Now.ToString("yyyyMMddHHmmssffff"));
6              builder.Append("-");
7              builder.Append(_fileName);
8              var path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
9  builder.ToString());
10             try
11             {
12                 Stream saveFileStream = File.Create(path);
13                 BinaryFormatter serializer = new BinaryFormatter();
```

```
14                 serializer.Serialize(saveFileStream, _afc.ToDictionary(kvp =>
15 kvp.Key, kvp => kvp.Value));
16                 saveFileStream.Close();
17                 _logger.LogInformation($"AFC successfully saved at
18 {DateTime.Now.ToString(@"MM\/dd\/yyyy h\:mm tt")}");
19             }
20             catch (Exception e)
21             {
22                 _logger.LogError(e, $"AFC not saved at
23 {DateTime.Now.ToString(@"MM\/dd\/yyyy h\:mm tt")}");
24             }
         }
```

Листинг 13. Имплементација перзистенције АФЦ-а

Насупрот бинарној серијализацији, принцип бинарне десеријализације је коришћен приликом инстанцирања репозиторијума асоцијација у оквиру листинга **Листинг 9**. Приликом инстанцирања репозиторијума се врши покушај учитавања АФЦ -а и конкурентног речника са диска, и уколико је учитавање успешно, врши се десеријализација учитаних структура. На овај начин је омогућен наставак коришћења структура података које садрже податке које су садржале и пре него што је програм заустављен.

5.7. Анализа перформанси

Након приказа детаља имплементације свих функција које микросервис за високоперформантно бројање учесталости придруживања кључева мора да испуни, потребно је да се аналитичким путем израчуна временска сложеност функција. На овај начин треба да се докаже да ли су испуњени захтеви решења приказани у трећем поглављу.

Функција *упишиНовуАсоцијацију* се користи за упис нове асоцијације у конкурентни ред. Будући да је упис новог елемента у ред описан у другом поглављу и да је временска сложеност уписа нове вредности у ред $O(1)$, важи да функција *упишиНовуАсоцијацију* има константну временску сложеност.

Следећа функција, која уједно представља и најважнију функцију овог микросервиса, јесте функција *прочитајКНајфреквентнијихАсоциранихКључева*. У претходном тексту је објашњено да се операција читања најпосећенијих асоцираних кључева потпуно ослања на истоимену операцију АФЦ-а, стога је у наставку фокус пребачен на анализу извршавања функције у оквиру АФЦ-а. Наиме, први корак приликом позива функције *прочитајКНајфреквентнијихАсоциранихКључева* АФЦ-а

јесте приступ конкурентном речнику помоћу асоцијатора. У другом поглављу је дефинисано да операција приступа вредности речника помоћу кључа има временску комплексност $O(1)$. Након што је пронађен асоцијатор у речнику, врши се приступ показивачу „главе“ повезане листе АФЦ-а. Како су сви елементи у листи сортирани опадајуће према фреквенцији придруживања, да би се прочитало K најучесталијих асоцираних кључева из листе, потребно је само да се изврши пролазак кроз K елемената повезане листе. У одељку 5.5 је објашњено да се број K учитава из конфигурационих фајлова, тако да се он сматра константном вредношћу. Узимајући у обзир и време потребно да се приступи кључу речника, укупна временска комплексност износи:

$$= O(1) + O(K),\ K = const.$$

$$= O(1 + K), K = const.$$

$$= O(1)\ .$$

На овај начин је доказано да функција *прочитајКНајфреквентнијихАсоцираниxКључева* има константну временску сложеност.

Функција *обришиСвеАсоциранеКључевеСаАсоцијатором* се потпуно ослања на извршавање истоимене функције у оквиру АФЦ-а. Све што ова функција треба да уради јесте брисање кључа из речника. У одељку 2.3.3 је дефинисана операција брисања кључа из речника и показано је да је њена временска комплексност $O(1)$. На овај начин је доказано да функција *обришиСвеАсоциранеКључевеСаАсоцијатором* има константну временску сложеност.

6. Ограничења решења

Појам капацитета је карактеристичан за бројне хардверске компоненте, међу којима су оперативна и секундарна меморија (Tanenbaum, 2013:67). Капацитет представља меру ограничености одређеног ресурса и у контексту софтверских производа је тежња усмерена на то да капацитет буде што је могуће рационалније искоришћен. У поглављу које следи, објашњења су два најзначајнија ограничења микросервиса за високоперформантно бројање учесталости придруживања кључева са становишта капацитета. Након тога су представљени проблеми који су изазвани конкурентним режимом коришћења микросервиса.

6.1. Ограничење бројача фреквенција АФЦ-а

Како је објашњено у одељку 5.3, бројање учесталости асоцирања кључева се реализује кроз инкрементирање обележја „фреквенција“ чвора листе АФЦ-а. У листингу **Листинг 2** је приказана имплементација класе која представља чвор листе АФЦ-а и приказано је обележје `Frequency`, које чува учесталост асоцирања. Тип података којим је декларисано ово обележје је `uint`, који представља целобројну 32-битну вредност из опсега (0, 4.294.967.295) (Troelsen et Japikse, 2017:68). Ограничење да обележје `Frequency` може да има само вредности из представљеног опсега уједно представља ограничење бројача фреквенција АФЦ-а.

У случају када бројач достигне своју максималну вредност, имплеметирани механизам провере вредности бројача спречава да се изврши додатно инкрементирање вредности бројача. У том случају би чвор, који је достигао максималну вредност бројача,

остао на почетку листе АФЦ-а, али му се вредност бројача више не би инкрементирала. Уколико се деси да још чворова достигну максималне вредности бројача, у том случају ће инкрементирани чвор да буде постављен на почетак повезане листе како би се задржала хронологија уписа асоцијација. Појава више чворова са максималном вредношћу бројача представља јасан знак да треба да се размотри увођење 64-битног типа података за декларисање обележје које чува учесталост придруживања.

6.2. Ограничење оперативне меморије

Како је раније дискутовано, асоцијације се у оквиру микросервиса за високоперформантно бројање учесталости придруживања кључева складиште унутар АФЦ-а. АФЦ представља транзијентну структуру података која се налази у оперативној меморији. Услед постојања великог броја различитих асоцијатора и различитих асоцираних кључева по асоцијатору, величина АФЦ-а може да превазиђе доступну количину оперативне меморије.

Проблем да је процесима на рачунару потребно више радне меморије него што је доступно на рачунару је познати проблем и решава се коришћењем виртуелне меморије. „Виртуелна меморија је техника која омогућава да се процеси изврше иако се не налазе потпуно у меморији“. Употреба технике страничења на нивоу оперативног система подразумева да је процес подељен на станице, које се из секундарне меморије учитавају у оперативну меморију само када су потребне. (Silberschatz, 2012:389-390)

Како је за превазилажење проблема ограничења оперативне меморије неопходно да се користи секундарна меморија, извршавање операција микросервиса би било доста спорије. Будући да приступ секундарној меморији представља временски захтевну операцију, микросервис не би могао да постигне перформансе које су приказане у оквиру одељка 3.2.

6.3. Ограничење евентуалне конзистентности

Употреба конкурентног реда за оптимизацију времена извршавања функције уписа нове асоцијације уводи могућност да позадински процес може да закасни са уписом и не упише асоцијацију у АФЦ пре него што се од микорсервиса затраже најучесталији асоцирани кључеви. С обзиром на то да је пројектовани микросервис

првенствено намењен за коришћење у циљу подршке другим информационим системима, потенцијална погрешна позиција најучесталијих асоцираних кључева у једном тренутку представља занемарљиву грешку коју један овакав систем може да допусти, зарад обезбеђења високих перформанси. У случају коришћења микросервиса у ситуацијама где је неопходна потпуна конзистентност података, морало би да се изврши додатно прилагођавање пројектованог микросервиса.

6.4. Ограничење екстремног конкурентног приступа АФЦ-у

Услед коришћења микросервиса за високоперформантно бројање учесталости придруживања кључева од стране апликација које одликује екстремно велика количина саобраћаја, приступ АФЦ-у може да постане озбиљно ограничење. Наиме, уколико у истом тренутку стигне огромна количина ХТТП захтева за читање из АФЦ-а, а у оквиру позадинског процеса постоји велика количина асоцијација која треба да буде уписана, конкурентни речник АФЦ-а неће да дозволи да му више нити приступи у истом тренутку. Оваква ситуација би могла да проузрокује дуго време чекања на приступ АФЦ-у. Такође, док чека на приступ АФЦ-у, конкурентни ред може да буде закључан од стране једне нити и ово закључавање може знатно да успори упис нових асоцијација на нивоу микросервиса.

Како би наведене ситуације довеле до дугог времена чекања, операција читања најучесталијих придружених кључева би потенцијално могла да доведе до успоравања клијентских апликација. Уместо употребе конкурентног речника, који сам управља вишенитним приступом, употреба приоритетног реда (Cormen et al., 2009:162), који би вршио синхронизацију свих операција које приступају АФЦ-у, могла би да омогући превазилажење ограничења у одређеној мери, али проблем не би био у потпуности решен.

7. Евалуација

У поглављу које следи су дискутовани резултати добијени током тестирања имплементираног решења у локалном окружењу. Потом је представљено упоредно решење, које је имплементирано коришћењем базе података. На крају поглавља је приказано поређење резултата добијених тестирањем упоредног решења у локалном окружењу са резултатима решења које користи АФЦ.

7.1. Резултати тестирања микросервиса за високоперформантно бројање учесталости придруживања кључева у локалном окружењу

Након имплементације и аналитичког израчунавања перформанси пројектованог решења, следећи корак представља емпиријска валидација перформанси. Имплементирани микросервис је покренут на рачунару Леново Тхинкпад Т470с (енг. *Lenovo ThinkPad T470s*). За позив ендпоинта који служи за упис нових асоцијација у микросервис, коришћена је конзолна апликација написана у ДОТНЕТ-у, чији код је приказан у листингу **Листинг 17**.

Циљ тестирања је било утврђивање како се решење понаша када обрађује различите количине података и у којој мери је зависно од њих. Број података који се обрађује диктирају три улазна параметра (у даљем тексту параметри од значаја за тестирање):

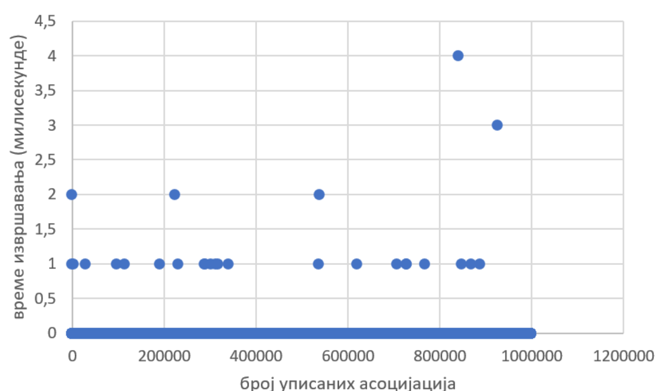
- број различитих асоцираних кључева по асоцијатору,
- највећи број понављања асоцијација по асоцијатору и

- број различитих асоцијатора.

Промена броја различитих асоцираних кључева по асоцијатору омогућава задавање већег броја *К* приликом читања *К* најпосећенијих асоцираних кључева, што омогућава тестирање времена читања при великим вредностима параметра *К*. Уједно, број различитих асоцираних кључева по асоцијатору утиче на повећање заузећа оперативне меморије, стога је важно да се тестира како он утиче на заузеће меморије. Највећи број понављања асоцијација по асоцијатору изазива бројна преповезивања унутар повезане листе АФЦ-а, па је корисно да се направи увид у то да ли овај број има утицај на време извршавања. Промена укупног броја асоцијатора је значајна са становишта процена перформанси речника АФЦ-а. Наиме, како је дискутовано у одељку 2.3.3, у одређеном тренутку долази до промене величине речника и до његовог реструктурирања. Ова операција би могла да резултује дужим временом извршавања приликом операције уписа нове асоцијације у микросервис. Иако се операција уписа нове асоцијације извршава у оквиру позадинског процеса, у одељку 7.1.2 су дискутовани резултати времена извршавања функције *упишиНовуАсоцијацију* у контексту АФЦ-а.

7.1.1. Резултати тестирања времена извршавања функција микросервиса

Током извршавања функције *упишиНовуАсоцијацију*, микросервис је показао изузетне перформансе, без обзира колике вредности су изабране за параметре од значаја. Резултати времена извршавања првих милион операција уписа нових асоцијација за 100 различитих асоцијатора и 100 различитих асоцираних кључева, при чему је број понављања асоцијација по асоциратору максимално 10, приказани су на графикону **Графикон 1**. Времена извршавања су изражена у милисекундама.



Графикон 1. Времена извршавања функције *упишиНовуАсоцијацију*

Резултати приказани на графикону **Графикон 1** нису изненађујући, будући да операција уписа нове асоцијације подразумева упис новог елемента у конкурентни ред. Како је само неколико операција уписа извршено за више од једне милисекунде, доказано је да се операција уписа нове асоцијације извршава са високом перформантношћу.

Извршавање функције *прочитајКНајфреквентнијихАсоциранихКључева* је показало јако слична времена извршавања за различите вредности параметара од значаја. Резултати тестирања времена извршавања приказани су у табели **Табела 4**.

Табела 4. Времена извршавања функције *прочитајКНајфреквентнијихАсоциранихКључева*

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	***К***	**време извршавања (милисекунде)**
100.000	100	5	20	**0,2**
100.000	100	5	50	**0,2**
100.000	100	5	100	**0,3**
100	100.000	5	20	**0,2**
100	100.000	5	200	**0,2**
100	100.000	5	2000	**0,2**

Изразито кратка времена извршавања приказана у табели **Табела 4** доказују изузетну перформантност функције *прочитајКНајфреквентнијихАсоциранихКључева*. Уједно, добијена времена извршавања доказују да су приступ речнику путем асоцијатора и пролазак кроз неколико хиљада чворова операције које се извршавају веома брзо. Осврћући се на дискусију у одељку 2.3.1, где је дискутовано да операција претраге повезане листе има линеарну временску комплексност, важно је да се напомене да би време проласка кроз повезану листу АФЦ-а линеарно расло са повећањем броја *К*. Ипак, због изузетно дугог времена које би било потребно да се упише велики број различитих асоцираних кључева по асоцијатору, највећа вредност броја *К* која је тестирана је 2.000.

Операција *обришиСвеАсоциранеКључевеСаАсоцијатором* је при сваком избору вредности параметара од значаја извршена за мање од милисекунде. Ова функција се извршава изузетно перформантно, што потврђује аналитичку процену перформанси из поглавља 5.

7.1.2. Резултати тестирања функције *упишиНовуАсоцијацију* АФЦ-а

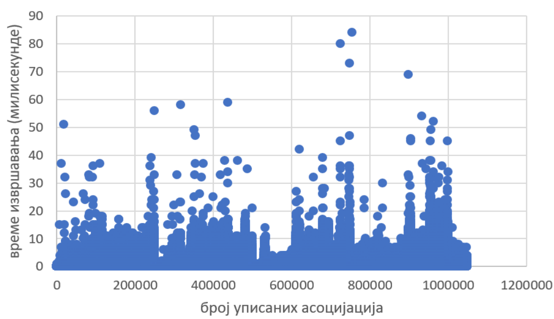
Иако се функција АФЦ-а *упишиНовуАсоцијацију* извршава у оквиру позадинског процеса и не утиче директно на перформансе функције *упишиНовуАсоцијацију* на нивоу микросервиса, важно је да се емпиријски утврди колико је времена потребно за извршавање ове функције. Тестирањем је покушано да се утврди како ће времена извршавања да се мењају када се величина речника значајно повећа, а како када се величине повезаних листа значајно повећају.

На графикону **Графикон 2** су приказани резултати уписа асоцијација у АФЦ, при чему је број различитих асоцираних кључева по асоцијатору изразито велики. Овај сценарио представља оптерећење величине повезаних листа АФЦ-а. Пажљивим посматрањем, на овом графикону могу да се примете региони које одликује раст вредности и који се смењују једни за другим. Услед велике количине података, није могуће да се експлицитно одреди облик функције којом би могли да се моделују ови региони. Ипак, водећи се аналитичким доказом времена извршавања из одељка 5.7, може да се претпостави да је функција која моделује ове регионе линеарна функција.

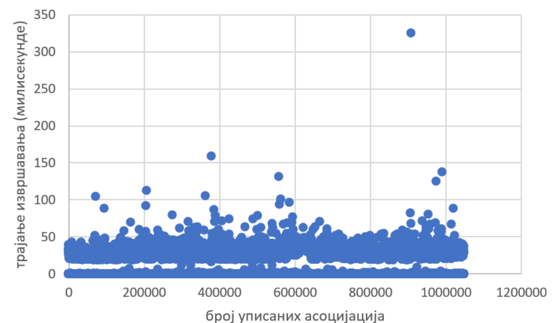
Сценарио у ком се врши упис асоцијација изразито великог броја различитих асоцијатора је дат на графикону **Графикон 3**. Овај сценарио представља оптерећење величине повезаних листа АФЦ-а. Важна особина графикона која може да се уочи јесте празан простор између горњег и доњег скупа тачака. Наиме, узрок постојања горњег скупа тачака јесте тренутак у ком се уписује асоцијација која садржи асоцијатор који претходно није био уписан у АФЦ-у. Приликом уписа таквих асоцијација потребно је да се инстанцира повезана листа АФЦ-а и да се креира први чвор те листе. Доњи скуп тачака представљају уписи свих асоцијација, осим претходно описаних. Будући да је број различитих асоцираних кључева по асоцијатору мали, а да је укупан број асоцијација јако велики, није могуће да се уоче региони линеарног раста који су били уочљиви на графикону **Графикон 2**.

На графиконима **Графикон 4** и **Графикон 5** су приказани сценарији у којима је највећи број понављања асоцијација по асоцијатору повећан, док су друга два параметра од значаја смањена. У оба сценарија је овај број постављен на 10, будући да би већи број проузроковао неприхватљиво време потребно да се упишу све асоцијације у условима тестирања. Посматрањем графикона **Графикон 4**, могуће је да се уочи веома слична

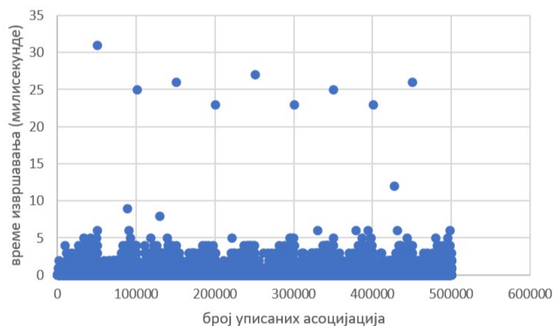
ситуација као на графикону **Графикон 2** – присуство региона са сличним обрасцем раста који се смењују дуж целе X осе . Будући да је број различитих асоцираних кључева по асоцијатору мањи него у случају приказаном на графикону **Графикон 2**, пропорционално томе је и раст региона мањи на графикону **Графикон 4**. Такође, на овом графикону је јасније уочљив раст времена извршавања узрокован истанцирањем повезаних листа АФЦ-а. Слична је ситуација и у случају резултата на графикону **Графикон 5**, где нема пуно одступања од резултата на графикону **Графикон 3**. Због мањег укупног броја асоцијација, празан простор између горњег и доњег скупа тачака је уочљивији.



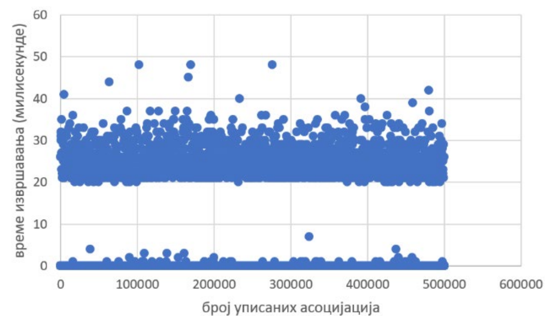
Графикон 2. Упис нових асоцијација: 100 различитих асоцијатора, 100.000 различитих асоцираних кључева по асоцијатору са највише 5 понављања асоцијација по асоцијатору



Графикон 3. Упис нових асоцијација: 100.000 различитих асоцијатора, 100 различитих асоцираних кључева по асоцијатору са највише 5 понављања асоцијација по асоцијатору



Графикон 4. Упис нових асоцијација: 10 различитих асоцијатора, 10.000 различитих асоцираних кључева по асоцијатору са највише 10 понављања асоцијација по асоцијатору



Графикон 5. Упис нових асоцијација: 10.000 различитих асоцијатора, 10 различитих асоцираних кључева по асоцијатору са највише 10 понављања асоцијација по асоцијатору

У складу са добијеним резултатима, утврђено је да преповезивање услед велике количине асоцираних кључева по асоцијатору и реструктурирање листе АФЦ-а, које се том приликом догађа, нема значајан утицај на време извршавања. Такође, повећање броја

различитих асоцијатора није резултовало у повећању времена извршавања. Са друге стране, на повећање времена извршавања операције *упишиНовуАсоцијацију* утиче упис асоцијатора који није претходно уписан у АФЦ. Овај утицај није превише велики, већ важи да се ова операција извршава дуже него операција уписа асоцијације кад асоцијатор већ постоји у АФЦ-у. Такође, на повећање времена извршавања утиче повећање броја различитих асоцираних кључева по асоцијатору, тако што доводи до линеарног раста времена извршавања.

7.1.3. Резултати тестирања заузећа меморијског простора микросервиса за високоперформантно бројање учесталости придруживања кључева

Како је дискутовано у поглављу 6, меморија представља ограничени ресурс и веома је важно да се њом управља на адекватан начин. Будући да је АФЦ структура података која се налази у оперативној меморији и треба да сачува велику количину података, веома је важно да се тестирањем утврди колико како се раст заузећа меморијског простора повећава са повећањем количине података која је сачувана у АФЦ -у. Поред АФЦ-а, конкурентни ред такође заузима меморијски простор, стога је важно да се анализира колико меморијског простора он заузима. Поред заузећа оперативне меморије, неопходно је да се испита и заузеће меморије на диску након што се АФЦ и конкурентни ред перзистирају.

Као и приликом тестирања времена извршавања, и у случају тестирања заузећа меморије је извршена провера у којој мери промена параметара од значаја утиче на заузеће меморијског простора. У табели **Табела 5** су приказане вредности параметара од значаја и заузеће оперативне меморије од стране речника и листа АФЦ-а.

Табела 5. Заузеће оперативне меморије од стране АФЦ-а

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	**заузеће меморије**
100	100	10	**0,46 MB**
10.000	100	5	**46,85 MB**
100.000	100	5	**244,2 MB**
100	10.000	5	**45,79 MB**
100	100.000	5	**233,6 MB**

Посматрајући резултате из табеле **Табела 5**, може да се примети да је заузеће оперативне меморије од стране АФЦ-а прихватљиво са становишта меморијских капацитета савремених рачунара, код којих се често среће доста више радне него што је било потребно АФЦ-у у условима тестирања. Повећање броја различитих асоцијатора, као и пораст броја различитих асоцираних кључева по асоцијатору, доводи до повећања заузећа оперативне меморије. Повећање највећег броја понављања асоцијација по асоцијатору због предугог времена извршавања није тестирано коришћењем великих вредности, али је јасно да овај параметар не доводи до креирања нових објеката у меморији и не утиче на заузимање простора.

Када је реч о конкурентном реду, због природе тока извршавања програма, није могуће да се прикажу експлицитне вредности заузећа оперативне меморије. Са друге стране, знајући да чворови конкурентног реда чувају оба кључа, услед великог броја уписа нових асоцијација, конкуренти ред би могао да постане далеко већа структура АФЦ-а. Да се овакав случај не би десио, позадински процес периодично празни ред, па се у реду не задржава превише елемената.

Са становишта меморије на диску коју заузимају АФЦ и конкурентни ред, и овом случају су добијени резултати задовољавајући. Конкурентни ред ни у овом контексту није погодан за анализу, будући да се након сваког позива позадинског процеса уклањају сви његови елементи. Из овог разлога није могуће да се прикаже конкретна бројна вредност заузећа меморије на диску за перзистенцију конкурентног реда. Са друге стране, простор на диску који АФЦ заузима након перзистирања је тестиран тако што су коришћене различите вредности параметара од значаја. Добијени резултати су приказани у табели **Табела 6**.

Табела 6. Резултати тестирања заузећа простора АФЦ-а на диску

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	**заузеће меморије**
100	100	10	**0,417 MB**
10.000	100	5	**42,47 MB**
100.000	100	5	**414,75 MB**
100	10.000	5	**42 MB**
100	100.000	5	**403,05 MB**

На основу приказаних резултата може да се закључи да промена параметара од значаја на исти начин утиче и на количину заузетог простора на диску као и на заузеће оперативне меморије.

7.2. Упоредно решење – микросервис за бројање учесталости придруживања кључева имплементиран коришћењем базе података

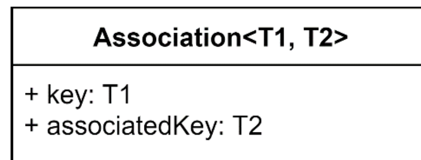
Како током израде овог рада није пронађено решење које би могло да послужи као упоредно решење микросервису за високоперформантно бројање учесталости придруживања кључева, имплементирано је решење у виду микросервиса са истим функционалностима, али које је засновано на употреби базе података уместо АФЦ-а. „База података представља скуп међусобно повезаних података којом се управља коришћењем система за управљање базама података“ (Teorey et al., 2009:2). Базе података су намењене за складиштење великих количина података, стога се подаци смештају на уређаје секундарне меморије како би се обезбедила перзистенција података (Silberschatz, 2020:18). Подацима у базама података се приступа коришћењем упита (Garcia-Molina et al., 2008:2). Упити омогућавају да се резултати групишу по одређеном критеријуму и сортирају у одређеном поретку (Churcher, 2016:135,171). Помоћу претходно наведених функционалности упита базе података је могуће да се имплементира претраживање најучесталијих придружених кључева.

Да би се приступило фајловима базе података, потребно је да се приступи секундарној меморији. Приступ фајловима на диску подразумева пребацивање блокова са диска у оперативну меморију (Silberschatz, 2020: 567). Такође, базе података могу да буду смештене на физички удаљеним ресурсима којима мора да се приступа преко мреже (Silberschatz, 2020:23). Приступ преко мреже уводи додатно време чекања, узроковано природом рачунарских мрежа. На основу ове две чињенице, приступ бази података се сматра „скупом“ операцијом са становишта времена извршавања. У складу са тим, постављена хипотеза да је микросервис имплементиран коришћењем АФЦ-а перформантније решење од микросервиса имплементираног коришћењем базе података са становишта времена извршавања.

7.2.1. Имплементација упоредног решења

Микросервис за бројање учесталости придруживања кључева који користи базу података је имплементиран као веб АПИ пројекат. Дијаграм класа упоредног решења је приказан на слици **Слика 20**. За разлику од решења које користи АФЦ, упоредно решење

поседује подешавања за повезивање са базом података, као и класе које омогућавају употребу објектно-релационог мапирања за приступ бази података.



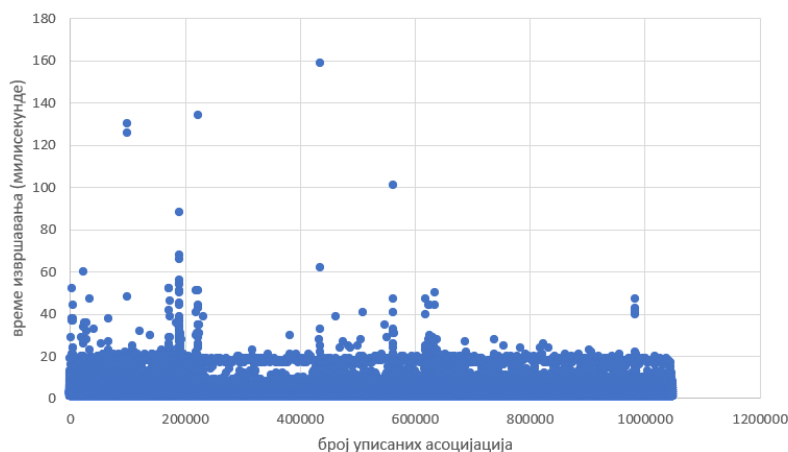
Слика 20. Дијаграм класа упоредног решења

На листингу **Листинг 16**, који је дат у прилогу, приказан је код за имплементацију репозиторијума из ког се врше позиви упита над базом података. Функција `GetMostFrequentAssociatedKeys` позива упит над базом података којим се преузима првих *K* торки, при чему су торке филтриране тако да садрже одређени асоцијатор и груписане према асоцираном кључу. Функције `SaveNewAssociation` и `DeleteAllAssociatedKeys` се извршавају употребом основних упита за упис и брисање података из базе података.

7.2.2. Резултати тестирања времена извршавања операција упоредног решења

База података коришћена за тестирање у локалном окружењу је `Microsoft SQL Server`. Пројектована је тако да садржи само једну релацију, чија су обележја аутоматски генерисани јединствени идентификатор типа `Guid`, асоцијатор и асоцирани кључ, који су такође типа `Guid`.

Резултати извршавања функције *упишиНовуАсоцијацију* су показали прилично уједначена времена без обзира на варијације вредности параметара од значаја. На графикону **Графикон 6** су приказани резултати уписа првих милион асоцијација, при чему је број различитих асоцијатора постављен на 100, број различитих асоцираних кључева по асоцијатору на 100.000, а највећи број понављања асоцијација по асоцијатору на 5. Како је наведено у раду „Системи складишта података – дизајн и имплементација (енг. *Data Warehouse Systems Design and Implementation*) (Vaisman et Zimányi, 2014:45), базе података су специјализоване за упис нових слогова и постижу високе перформансе у погледу времена извршавања. Резултати са графикона **Графикон 6** потврђују ову особину.



Графикон 6. Резултати времена извршавања функције *упишиНовуАсоцијацију* упоредног решења

Функција *прочитајКНајфреквентнијихАсоциранихКључева* је показала различите резултате зависно од постављених параметара. У табели **Табела 7** су приказани добијени резултати током тестирања.

Табела 7. Времена извршавања функције *прочитајКНајфреквентнијихАсоциранихКључева* упоредног решења

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	***K***	**време извршавања (милисекунде)**
1000	1000	10	20	**474**
1000	1000	10	1000	**435**
100.000	100	5	20	**16.764**
100.000	100	5	200	**14.349**
100	100.000	5	20	**17.274**
100	100.000	5	1000	**16.029**

На основу резултата приказаних у претходној табели, може да се закључи да са порастом укупног броја асоцијација расте и време претраге. Сва три параметра од значаја једнако утичу на укупан број асоцијација који ће да буде уписан, односно повећање њихових вредности је директно сразмерно повећању времена извршавања функције *прочитајКНајфреквентнијихАсоциранихКључева*. Промена броја најпосећенијих асоцираних кључева *K* није показала значајан утицај на време извршавања. Добијено време извршавања када је у бази података сачувано више од 10.000.000 асоцијација је више од 14 секунди, што је неприхватљиво време са становишта корисничког задовољства.

Функција *обришиСвеАсоциранеКључевеСаАсоцијатором* је током тестирања показала да се са порастом броја асоцијација у бази података повећава и време потребно да се изврши операција уписа. Добијени резултати су приказани у табели **Табела 8**.

Табела 8. Времена извршавања функције *обришиСвеАсоциранеКључевеСаАсоцијатором* упоредног решења

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	**време извршавања (милисекунде)**
100	100	5	**63,8**
10.000	100	5	**297,5**
100.000	100	5	**14.340,8**
100	10.000	5	**331,9**
100	100.000	5	**13.452,3**

У случају када је у бази података број сачуваних асоцијација мали, функција *обришиСвеАсоциранеКључевеСаАсоцијатором* је показала задовољавајуће резултате. Са друге стране, при великој количини сачуваних асоцијација, ова операција се извршава у неприхватљиво дугом времену.

7.2.3. Резултати тестирања заузећа меморијског простора упоредног решења

Упоредно решење које користи базу података не користи структуре података које се налазе у оперативној меморији, стога тестирање заузећа оперативне меморије није релевантно у овом случају. Тестирање заузећа меморије на диску је, као и у случају претходних тестова, извршено тако што су коришћене различите вредности параметара од значаја. Будући да детаљна анализа добијених резултата излази ван оквира овог рада, добијени резултати ће да буду искоришћени за поређење са резултатима микросервиса који користи АФЦ. Резултати тестирања заузећа простора на диску од стране базе података су приказани у табели **Табела 9**.

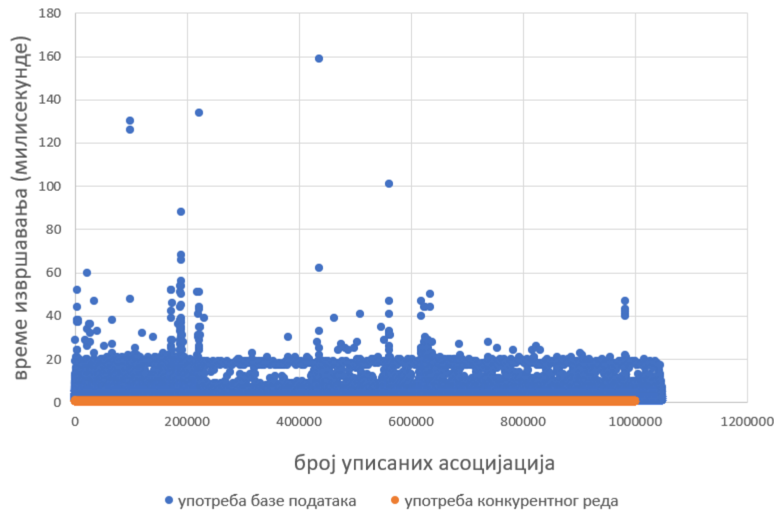
Табела 9. Заузеће простора на диску упоредног решења

број асоцијатора	**број различитих асоцираних кључева по асоцијатору**	**највећи број понављања асоцијација по асоцијатору**	**заузеће меморије**
100	100	5	**80 MB**
10.000	100	5	**464 MB**
100.000	100	5	**2.512 MB**
100	10.000	5	**336 MB**
100	100.000	5	**2.576 MB**

7.3. Поређење добијених резултата

Имплементација микросервиса за високоперформантно бројање учесталости придруживања кључева коришћењем базе података представља веома интуитивно решење. Коришћењем шаблона за веб АПИ и једноставног повезивања апликације са базом података које обезбеђује ДОТНЕТ, потребно је веома мало времена да се имплементира једно овакво решење. За разлику од имплементације коришћењем базе података, имплементацији микросервиса коришћењем АФЦ-а је претходио задатак пројектовања структуре података какав је АФЦ, а потом и оптимизације операције уписа нове асоцијације коришћењем реда. Узимајући у обзир да је микросервис који користи АФЦ специјализован да обезбеди високоперформантно бројање учесталости придруживања кључева, очекивани исход поређења је био да је микросервис који користи АФЦ перформантнији од упоредног решења.

Тестирање функције *упишиНовуАсоцијацију* је код оба микросервиса показало да време извршавања у оба случаја одликује константност. Ипак, иако је база података оптимизована за операцију уписа нових торки, она ипак захтева приступ диску, што се извршава пуно спорије него упис у конкурентни ред који се налази у оперативној меморији. Времена извршавања која су приказана на графикону **Графикон 7** сведоче да се функција *упишиНовуАсоцијацију* извршава перформантније од стране микросервиса који користи АФЦ.



Графикон 7. Приказ поређења времена извршавања функције *упишиНовуАсоцијацију*

Време извршавања функције *прочитајКНајфреквентнијихАсоциранихКључева* је приликом тестирања микросервиса који користи АФЦ у просеку било реда величине 0,2 милисекунде. Измерено време показује да је постигнуто изузетно ефикасно читање најфреквентнијих асоцираних кључева коришћењем АФЦ-а. Са друге стране, упоредно решење које користи базу података је показало прихватљиве перформансе када је у бази података сачуван релативно мали број асоцијација (435 милисекунди за 5 милиона асоцијација) . У случају када је у бази података број сачуваних асоцијација реда величине неколико десетина милиона, коришћење овог решења није прихватљиво са становишта корисничког искуства.

Извршавање функције *обришиСвеАсоциранеКључевеСаАсоцијатором* је током тестирања у оквиру микросервиса који користи АФЦ показало изузетно добре перформансе и независност од параметара од значаја. Са друге стране, упоредно решење је показало супротне карактеристике – зависност од параметара од значаја и лоше перформансе при постојању велике количине података унутар базе података.

Последњи аспект поређења решења јесте перзистенција податка. У случају базе податка, при извршавању теста у којем је обрађен највећи број асоцијација, за смештање базе податка на диск је било потребно 5 пута више меморијског простора. Иако је АФЦ показао далеко боље перформансе у погледу заузећа меморије, бинарана десеријализација која се извршава приликом учитавања АФЦ-а са диска захтева превише оперативне меморије. Наиме, пошто класа `ConcurrentDictionary` није серијабилна у оквиру ДОТНЕТ кора, потребно је да се изврши конверзија инстанце ове класе у инстанцу класе `Dictionary`. При учитавању АФЦ-а великог капацитета, конверзија која се дешава у захтева огромну количину оперативне меморије и доводи до значајног

успоравања покретања микросервиса. Због овог недостатка, база података се сматра перформантнијом структуром у контексту перзистенције података.

На основу поређења добијених резултата, микросервис који користи АФЦ је показао боље перформансе од упоредног решења које користи базу података. Иако је упоредно решење доста интуитивније и лакше за имплементацију, оно је показало озбиљна ограничења када је реч о коришћењу микросервиса у условима у којима је потребна обрада велике количине података.

8. Закључак

У овом раду је представљен један начин за имплементацију микросервиса који омогућава високоперформантно бројање учесталости придруживања кључева. Језгро микросервиса представља структура података која се назива бројач учесталости асоцијација (АФЦ). Ова структура података омогућава претрагу најучесталијих придружених кључева у константном – $O(1)$ времену. Поред операције претраге, АФЦ омогућава брисање кључева асоцираних са одређеним кључем, такође у константном – $O(1)$ времену. Последња операција, операција уписа нове асоцијације се извршава са линеарном временском комплексношћу – $O(n)$.

У локалном окружењу је извршено тестирање претходно описаних операција. Читање најучесталијих придружених кључева се, у условима где је у АФЦ-у сачувано више од 25 милиона асоцијација кључева, просечно извршавало за 0,2 милисекунде. Операција брисања асоцираних кључева са одређеним кључем је такође извршена у просеку за мање од милисекунде. Добијени резултати потврђују аналитички процењене временске комплексности, односно потврђују изузетну ефикасност ове две операције. Са друге стране, операцију додавања нове асоцијације у АФЦ одликује линеарни раст времена извршавања са порастом броја различитих асоцираних кључева. Током тестирања, операција додавања нове асоцијације је достизала време извршавања од 100 милисекунди.

Како је за потребе микросервиса за високоперформантно бројање учесталости придруживања кључева непоходно да се операција уписа нових асоцијација извршава у $O(1)$ времену, асоцијације су приликом стизања ХТТП захтева од клијентских апликација привремено складиштене у ред, из ког су у оквиру позадинског процеса

уписане у АФЦ. Након тестирања у локалном окружењу, операција уписа нове асоцијације се извршавала у просеку за мање од милисекунде. На овај начин је постигнуто да се све операције микросервиса извршавају изузетно перформантно.

Када је реч о меморијском простору који АФЦ заузима, након складиштења више од 25 милиона асоцијација, простор у оперативној меморији који АФЦ заузима је мањи од 250 мегабајта. Након перзистенције, која се реализује коришћењем бинарне серијализације, на диску је заузето мање од 415 мегабајта.

Додатна валидација имплементираног решења је извршена поређењем микросервиса који користи АФЦ са микросервисом који бројање учесталости кључева извршава уз помоћ базе података. Иако упоредно решење које користи базу података представља једноставније решење са становишта комплексности имплементације, перформансе које је постигао микросервис који користи АФЦ су далеко боље од перформанси упоредног решења. Наиме, операција читања најучесталијих придружених кључева се, при 25 милиона асоцијација које су сачуване у бази података, извршавала у трајању од око 16 секунди. Такође, извршавање операције брисања свих асоцираних кључева са одређеним кључем је трајало дуже од 10 секунди. Са друге стране, операција уписа нове асоцијације је трајала у просеку 20 милисекунди. Добијени резултати су показали да је микросервис који користи АФЦ далеко ефикаснији у контексту времена извршавања од микросервиса који користи базу података.

Када је реч о заузећу меморијског простора на диску, за складиштење 25 милиона асоцијација је при коришћену базе података потребно око 2,5 гигабајта меморијског простора. Упркос заузећу мање количине меморијског простора у случају микросервиса који користи АФЦ, техника бинарне десеријализације је показала веома лоше перформансе, стога је тешко да се утврди које решење је перформантније.

Иако је циљ овог рада постигнут, имплементирани систем оставља могућности за додатна унапређења. Наиме, поменута техника бинарне серијализације представља прилично неефикасно решење, стога би вредело да се у наредним истраживањима посвети пажња овом унапређењу. Са друге стране, потенцијална комерцијализација решења захтева да се одговори на питање на који начин би се вршило разликовање клијентских апликација од стране микросервиса за високоперформантно бројање учесталости придруживања кључева и смештање њихових података тако да је могуће да се разликује којој апликацији припадају. Поред овог питања, начин наплате коришћења микросервиса представља још једно питање вредно разматрања када је реч о могућности комерцијализације решења.

Референце

Nah, F. F. H. (2004). A study on tolerable waiting time: how long are Web users willing to wait?. *Behaviour & Information Technology*, *23*(3), 153–163.

Galletta, D. F., Henry, R., McCoy, S., & Polak, P. (2004). Web site delays: How tolerant are users?. *Journal of the Association for Information Systems*, *5*(1).

Hoxmeier, J. A., & DiCesare, C. (2000). System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, 140-145.

Neal, D. T., Wood, W., Labrecque, J. S., & Lally, P. (2012). How do habits guide behavior? Perceived and actual triggers of habits in daily life. *Journal of Experimental Social Psychology*, *48*(2), 492–498.

Felfernig, A., Jeran, M., Ninaus, G., Reinfrank, F., Reiterer, S., & Stettinger, M. (2014). Basic approaches in recommendation systems. *Recommendation Systems in Software Engineering*, 15-37.

Sullivan, D. (2018, April 20). *How Google autocomplete works in Search*. Google. Preuzeto 18. avgusta 2021, sa https://blog.google/products/search/how-google-autocomplete-works-search/

Falk, K. (2019). *Practical Recommender Systems*. Manning Publications.

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O’Reilly Media.

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116.

Larrucea, X., Santamaria, I., Colomo-Palacios, R., & Ebert, C. (2018). Microservices. *IEEE Software*, *35*(3), 96–100.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. McGraw-Hill Education.

Mathai, A. M., & Haubold, H. J. (2017b). *Probability and Statistics*. De Gruyter.

Triplett, J., McKenney, P. E., & Walpole, J. (2011). Resizable, scalable, concurrent hash tables. *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'11)*, 145-158.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2012). *Operating System Concepts*. Wiley.

Jones, R., Hosking, A., & Moss, E. (2011). *The Garbage Collection Handbook*. Taylor & Francis.

Soukup, J., & Macháček, P. (2014). *Serialization and Persistent Objects: Turning Data Structures into Efficient Databases*. Springer.

Tanenbaum, A. S., & Austin, T. (2013). *Structured Computer Organization*. Pearson.

Teorey, T. J., Buxton, S., Fryman, L., Güting, R. H., Halpin, T., Inmon, W. H., Harrington, J. L., Melton, J., Lightstone, S. S., Morgan, T., Nadeau, T. P., O'Neil, B., O'Neil, E., O'Neil, P., Schneider, M., Simsion, G., & Witt, G. (2009). *Database Design: Know It All*. Elsevier Gezondheidszorg.

Garcia-Molina, H., Ullman, J., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Pearson.

Churcher, C. (2016). *Beginning SQL Queries: From Novice to Professional* (2nd ed.). Apress.

Vaisman, A., & Zimányi, E. (2016). *Data Warehouse Systems: Design and Implementation (Data-Centric Systems and Applications)*. Springer.

Dale, N., Walker, H. M., & Nell Dale. (1996). *Abstract Data Types*. D.C. Heath.

Troelsen, A., & Japikse, P. (2017). *Pro C# 7: With .NET and .NET Core* (8th ed.). Apress.

Microsoft. (n.d.). *Reference Source*. Preuzeto 23. avgusta 2021, sa https://referencesource.microsoft.com/#mscorlib/system/Collections/Concurrent/ConcurrentDictionary.cs

Прилози

ПРИЛОГ А: ОПЕРАЦИЈА УПИСА АСОЦИЈАЦИЈЕ У АФЦ

```
 1     public void SaveNewAssociatedKey(T associatedKey)
 2         {
 3             if (firstElement == null)
 4             {
 5                 firstElement = new Node<T>(associatedKey);
 6             }
 7             else
 8             {
 9                 Node<T> previousElement = null;
10                 Node<T> lastElementInLessFrequentRegion = null;
11                 Node<T> lastElementInMoreFrequentRegion = null;
12                 var placeHolder = firstElement;
13                 while (placeHolder != null)
14                 {
15                     if
16 (placeHolder.AssociatedKey.Equals(associatedKey))
17                     {
18                         if (placeHolder.Frequency < uint.MaxValue)
19 placeHolder.Frequency++;
20                         if (previousElement == null) break;
21                         previousElement.NextNode =
22 placeHolder.NextNode;
23                         Node<T> nextNodeForPlaceholder;
24                         if (lastElementInLessFrequentRegion == null)
25                         {
26                             nextNodeForPlaceholder = firstElement;
27                             firstElement = placeHolder;
28                         }
29                         else
30                         {
31                             if (placeHolder.Frequency <
32 lastElementInLessFrequentRegion.Frequency)
33                             {
34                                 nextNodeForPlaceholder =
35 lastElementInLessFrequentRegion.NextNode;
36
37 lastElementInLessFrequentRegion.NextNode = placeHolder;
38                             } else
39                             {
40                                 if (lastElementInMoreFrequentRegion ==
41 null)
42                                 {
43                                     nextNodeForPlaceholder =
44 firstElement;
45                                     firstElement = placeHolder;
46                                 } else
47                                 {
48                                     nextNodeForPlaceholder =
49 lastElementInMoreFrequentRegion.NextNode;
```

```
50
51 lastElementInMoreFrequentRegion.NextNode = placeHolder;
52                                  }
53                              }
54                          }
55                          placeHolder.NextNode = nextNodeForPlaceholder;
56                          break;
57                      }
58                      if (placeHolder.HasNext())
59                      {
60                          var nextNode = placeHolder.NextNode;
61                          if (placeHolder.Frequency > nextNode.Frequency)
62                          {
63                              lastElementInMoreFrequentRegion =
64 lastElementInLessFrequentRegion;
65                              lastElementInLessFrequentRegion =
66 placeHolder;
67                          }
68                          previousElement = placeHolder;
69                          placeHolder = nextNode;
70                      }
71                      else
72                      {
73                          placeHolder.NextNode = new
74 Node<T>(associatedKey);
75                          break;
76                      }
77                  }
78              }
        }
```

Листинг 14. Операција уписа асоцијације у АФЦ

ПРИЛОГ Б: КОНТРОЛЕР ВЕБ АПИ-ЈА

```
 1     [ApiController]
 2     [Route("/associations")]
 3     public class AssociationController : ControllerBase
 4     {
 5         private readonly IAssociationRepository<Guid, Guid>
 6 _associationRepository;
 7
 8         public AssociationController(IAssociationRepository<Guid, Guid>
 9 associationRepository)
10         {
11             _associationRepository = associationRepository;
12         }
13
14         [HttpGet("{key:Guid}")]
15         public IActionResult
16 GetMostFrequentAssociatedKeys([FromRoute]Guid key)
17         {
18             var mostFrequentAssociatedKeys =
19 _associationRepository.GetMostFrequentAssociatedKeys(key);
20             return Ok(mostFrequentAssociatedKeys);
21         }
22
23         [HttpPost]
24         public IActionResult SaveNewAssociation([FromBody]
25 AssociationDTO associationDTO)
26         {
27
28 _associationRepository.SaveNewAssociation(associationDTO.Key,
29 associationDTO.AssociatedKey);
30             return NoContent();
31         }
32
33         [HttpDelete("{key:Guid}")]
34         public IActionResult DeleteAllAssociatedKeys([FromRoute] Guid
35 key)
36         {
37             _associationRepository.DeleteAllAssociatedKeys(key);
38             return Ok();
39         }
40     }
```

Листинг 15. Контролер веб АПИ-ја

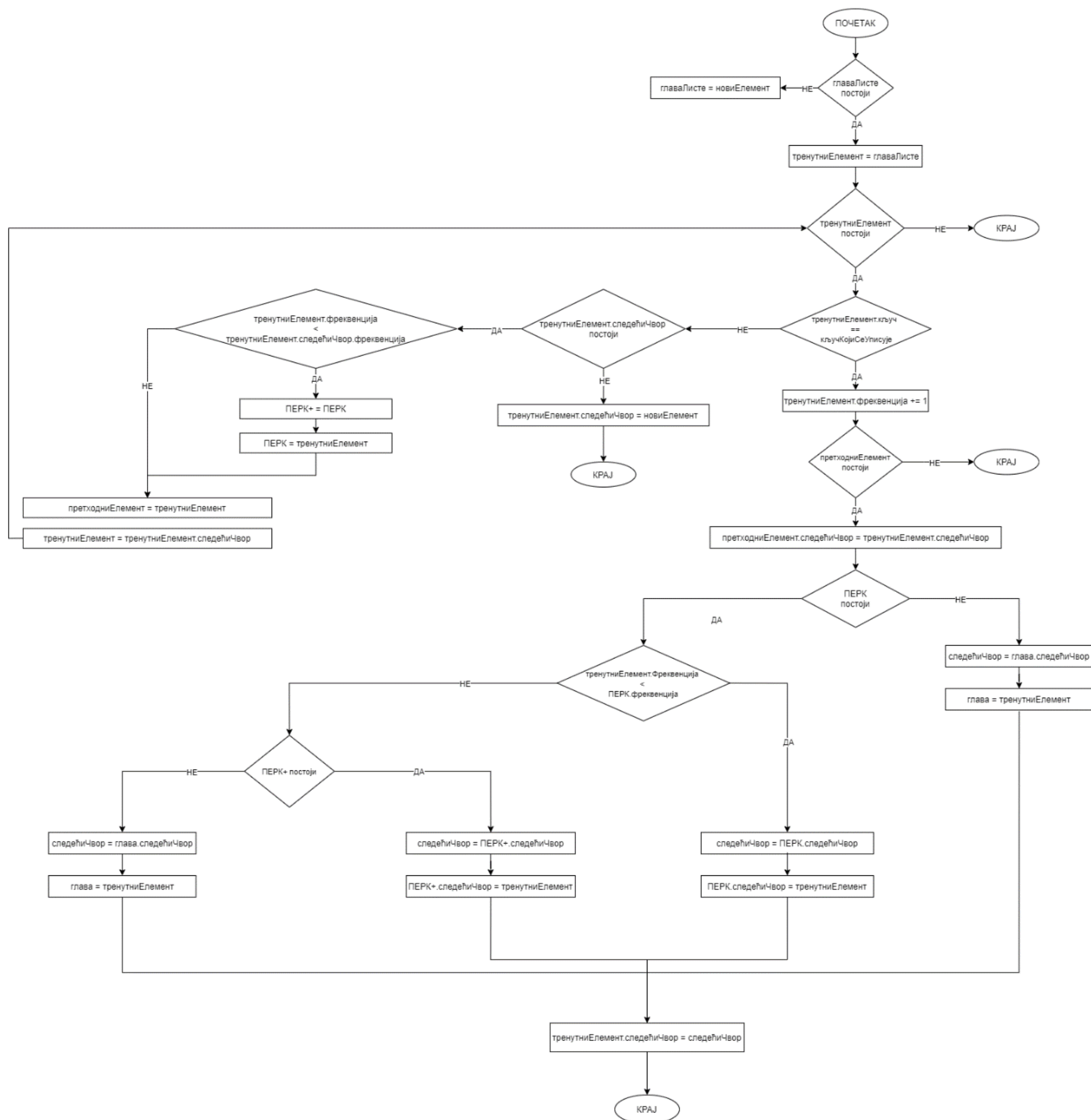
ПРИЛОГ В: ИМПЛЕМЕНТАЦИЈА РЕПОЗИТОРИЈУМА УПОРЕДНОГ РЕШЕЊА

```
 1     public class AssociationRepository : IAssociationRepository
 2     {
 3         private readonly AppDbContext _dbContext;
 4         private readonly ILogger _logger;
 5         private readonly IConfiguration _configuration;
 6
 7         public AssociationRepository(AppDbContext dbContext,
 8 ILogger<AssociationRepository> logger, IConfiguration configuration)
 9         {
10             _dbContext = dbContext;
11             _logger = logger;
12             _configuration = configuration;
13         }
14
15         public async Task<IEnumerable<ResponseNode<Guid>>>
16 GetMostFrequentAssociatedKeys(Guid key)
17         {
18             return await _dbContext.Association
19                 .Where(u => u.Key == key)
20                 .GroupBy(visitedItem => visitedItem.AssociatedKey)
21                 .Select(g => new
22                 {
23                     Count = g.Count(),
24                     AssociatedKey = g.Key
25                 })
26                 .OrderByDescending(u => u.Count)
27
28 .Take(_configuration.GetValue<int>("NumberOfElementsToRetrieve"))
29                 .Select(g => new ResponseNode<Guid>
30                 {
31                     Frequency = (uint)g.Count,
32                     AssociatedKey = g.AssociatedKey
33                 })
34                 .ToListAsync();
35         }
36         public async Task SaveNewAssociation(Guid key, Guid
37 associatedKey)
38         {
39             var newAssociation = new Association()
40             {
41                 Key = key,
42                 AssociatedKey = associatedKey
43             };
44             await _dbContext.Association.AddAsync(newAssociation);
45             try
46             {
47                 await _dbContext.SaveChangesAsync();
48             }
49             catch (Exception e)
50             {
```

```
51                  _logger.LogError(e, "Insert in DB not successful");
52              }
53          }
54          public async Task DeleteAllAssociatedKeys(Guid key)
55          {
56              using (var bench = new Benchmark($"Deleting records:"))
57              {
58                  string sqlDeleteStatement = "DELETE FROM Association
59 WHERE key = ";
60                  try
61                  {
62                      await
63 _dbContext.Database.ExecuteSqlInterpolatedAsync($"DELETE FROM
64 Association WHERE [key] = {key}");
65                      //await _dbContext.SaveChangesAsync();
66                  }
67                  catch (Exception e)
68                  {
69                      _logger.LogError(e, "Deletion in DB not
70 successful");
71                  }
72              }
73          }
74      }
```

Листинг 16 - Имплементација репозиторијума упоредног решења

ПРИЛОГ Г: БЛОК-ДИЈАГРАМ АЛГОРИТМА УПИСА НОВЕ АСОЦИЈАЦИЈЕ У АФЦ



Слика 21. Блок-дијаграм функције *упишиНовуАсоцијацију* АФЦ-а

ПРИЛОГ Д: КОНЗОЛНА АПЛИКАЦИЈА ЗА УПИС НОВИХ АСОЦИЈАЦИЈА

```
 1 class Program
 2     {
 3         private const int numberOfKeys = 100;
 4         private const int numberOfAssociatedKeys = 100;
 5         private const int numberOfAssociatedKeyRepetition = 10;
 6         static void Main(string[] args)
 7         {
 8             Guid[] keys = new Guid[numberOfKeys];
 9             for (int i = 0; i < numberOfAssociatedKeys; i++)
10             {
11                 keys[i] = Guid.NewGuid();
12             }
13             Guid[] associatedKeys = new Guid[numberOfAssociatedKeys];
14             for (int j = 0; j < numberOfKeys; j++)
15             {
16                 associatedKeys[j] = Guid.NewGuid();
17             }
18             using (var client = new HttpClient())
19             {
20                 client.BaseAddress = new Uri("http://localhost:59454/");
21                 var dto = new AssociationDTO();
22                 Random random = new Random();
23                 foreach (var key in keys)
24                 {
25                     foreach (var associatedKey in associatedKeys)
26                     {
27                         dto.Key = key;
28                         dto.AssociatedKey = associatedKey;
29                         var dtoAsJson =
30 JsonConvert.SerializeObject(dto);
31                         var requestBody = new StringContent(dtoAsJson,
32 Encoding.UTF8, "application/json");
33                         var numberOfRepetitions = random.Next(1,
34 numberOfAssociatedKeyRepetition);
35                         for (var i = 0; i < numberOfRepetitions; i++)
36                         {
37                             var responseTask =
38 client.PostAsync("associations", requestBody);
39                             responseTask.Wait();
40                             var result = responseTask.Result;
41                             if (!result.IsSuccessStatusCode)
42                             {
43                                 Console.WriteLine("ERROR: Key {0},
44 associatedKey {1} ", dto.Key, dto.AssociatedKey);
45                             }
46                         }
47                     }
48                 }
49             }
50         }
51     }
```

Листинг 17. Конзолна апликација за симулирање слања ХТТП захтева за упис нових асоцијација

Биографија

Лука Радујевић је рођен 23. септембра 1998. године у Краљеву. У родном граду уписује основну школу „Браћа Вилотијевић", коју завршава као носилац дипломе „Вук Караџић". Школовање наставља похађајући природно-математички смер гимназије у Краљеву.

Након завршене средње школе, високо образовање започиње 2017. године на Факултету техничких наука у Новом Саду. Наредне године постаје носилац „Изузетне награде за успех" Универзитета у Новом Саду. Током последње године основних академских студија постаје добитник стипендије „Доситеја". Основне академске студије завршава у септембру 2021. године са општим успехом 9,87.

Од 2020. године своју професионалну каријеру започиње као стипендиста компаније „ТИАК" (енг. *TIAC*) у Новом Саду, где се специјализује у области развоја веб апликација.