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

Милош Гравара

**Анализа имплементација и перформанси алата за објектно-релационо мапирање**

Дипломски рад

- Основне академске студије -

Нови Сад, 2023.

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Датум: |
|  |
| **ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА** | Лист: |
| 1/1 |

*(Податке уноси предметни наставник - ментор)*

| Врста студија: | **Основне академске студије** |
| --- | --- |
| Студијски програм: | **Рачунарство и аутоматика** |
| Руководилац студијског програма: | **проф. др Милан Рапаић** |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент: | **Милош Гравара** | Број индекса: | **RA 60/2019** |
| Област: | **Електротехничко и рачунарско инжењерство** | | |
| Ментор: | **проф др. Горан Сладић** | | |
| НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:   * проблем – тема рада; * начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; * литература | | | |

**НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:**

|  |
| --- |
| **Анализа имплементације и перформанси изабраних алата за објектно-релационо мапирање** |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
| Анализирати принципе рада објектно-релационих мапера. Упознати се са концептима и архитектуром најчешће коришћених имплементација. Специфицирати тестове за анализу перформанси алата. Документовати решења и запажања. |

|  |  |
| --- | --- |
| Руководилац студијског програма: | Ментор рада: |
|  |  |

|  |
| --- |
| Примерак за:  - Студента;  - Ментора |

**САДРЖАЈ**

[1. Увод 9](#_Toc146036668)

[2. Објектно-релационо мапирање 11](#_Toc146036669)

[2.1 Наслеђивање 12](#_Toc146036670)

[2.1.1 Једна табела по ланцу наслеђивања 12](#_Toc146036671)

[2.1.2 Једна табела по класи 13](#_Toc146036672)

[2.1.3 Конкретна табела по ланцу наслеђивања 14](#_Toc146036673)

[2.2 Асоцијације 15](#_Toc146036674)

[2.3 Добављање података 16](#_Toc146036675)

[*2.3.1* *Lazy Loading* 16](#_Toc146036676)

[2.3.2 Проблем N+1 упита 16](#_Toc146036677)

[2.3.3 *Eager Loading* 17](#_Toc146036678)

[2.4 Кеширање 18](#_Toc146036679)

[2.5 Обрасци понашања ОРМ алата 18](#_Toc146036680)

[*2.5.1* *Identity Map* 18](#_Toc146036681)

[*2.5.2* *Unit of Work* 19](#_Toc146036682)

[2.6 Употреба ОРМ алата у развоју апликација 19](#_Toc146036683)

[3. Преглед релевантне литературе 21](#_Toc146036684)

[3.1 Методологија претраге литературе 21](#_Toc146036685)

[3.2 Опис претходних истраживања 21](#_Toc146036686)

[3.3 Уочени недостаци у литератури 23](#_Toc146036687)

[3.3.1 Непотпуност 23](#_Toc146036688)

[3.3.2 Одсуство анализе Пајтон ОРМ алата 23](#_Toc146036689)

[4. Преглед изабраних ОРМ алата 25](#_Toc146036690)

[*4.1* *Еntity Framework* 26](#_Toc146036691)

[4.1.1 Aрхитектура алата 26](#_Toc146036692)

[4.1.1.1 DbContext и DbSet 26](#_Toc146036693)

[4.1.1.2 Entity Data Model 28](#_Toc146036694)

[4.1.1.3 Руковање конекцијама 29](#_Toc146036695)

[4.1.1.4 Праћење промена ентитета 30](#_Toc146036696)

[4.1.2 Извршавање упита 31](#_Toc146036697)

[4.1.2.1 Language Integrated Query (LINQ) 31](#_Toc146036698)

[4.1.2.2 Превођење LINQ упита 31](#_Toc146036699)

[4.1.2.3 Maтеријализација 32](#_Toc146036700)

[4.1.3 Aспекти побољшања перформанси 33](#_Toc146036701)

[4.1.3.1 Lazy Loading и Eager Loading 33](#_Toc146036702)

[4.1.3.2 Индексирање 33](#_Toc146036703)

[4.1.3.3 Профилисање 34](#_Toc146036704)

[4.1.3.4 Кеширање 35](#_Toc146036705)

[4.1.4 Oстале кључне особине 36](#_Toc146036706)

[4.1.4.1 Миграције 36](#_Toc146036707)

[4.1.4.2 Контрола конкурентног приступа 37](#_Toc146036708)

[4.2 Hibernate 38](#_Toc146036709)

[4.2.1 Архитектура алата 38](#_Toc146036710)

[4.2.1.1 ЈDBC 38](#_Toc146036711)

[4.2.1.2 SessionFactory 39](#_Toc146036712)

[4.2.1.3 Session 40](#_Toc146036713)

[*4.2.1.4* *EntityManager* у *SpringJPA* 40](#_Toc146036714)

[4.2.1.5 Животни циклус перзистенције ентитета 41](#_Toc146036715)

[4.2.2 Мапирање ентитета 42](#_Toc146036716)

[4.2.2.1 Основе 43](#_Toc146036717)

[4.2.2.2 Композитни кључ 43](#_Toc146036718)

[4.2.2.3 Мапирање асоцијација 44](#_Toc146036719)

[4.2.3 Извршавање упита 46](#_Toc146036720)

[4.2.3.1 Hibernate Query Language (HQL) 46](#_Toc146036721)

[*4.2.3.2* *Criteria Queries* 46](#_Toc146036722)

[4.2.3.3 Генерисање SQL наредбе 47](#_Toc146036723)

[4.2.3.4 Хидрација објекта 47](#_Toc146036724)

[4.2.4 Keширање 48](#_Toc146036725)

[4.2.4.1 Кеш првог нивоа 48](#_Toc146036726)

[4.2.4.2 Кеш другог нивоа 48](#_Toc146036727)

[4.3 SQLAlchemy 50](#_Toc146036728)

[4.3.1 Архитектура алата 50](#_Toc146036729)

[4.3.1.1 DBAPI 50](#_Toc146036730)

[*4.3.1.2* *Engine* 52](#_Toc146036731)

[4.3.1.3 Session 53](#_Toc146036732)

[4.3.1.4 Животни циклус ентитета 55](#_Toc146036733)

[4.3.2 Мапирање ентитета 55](#_Toc146036734)

[4.3.2.1 Класично мапирање 56](#_Toc146036735)

[4.3.2.2 Декларативно мапирање 56](#_Toc146036736)

[4.3.2.3 Aнатомија мапирања 57](#_Toc146036737)

[4.3.3 Извршавање упита 60](#_Toc146036738)

[4.3.3.1 Query API 60](#_Toc146036739)

[4.3.3.2 Изразно стабло 61](#_Toc146036740)

[4.3.4 Остале кључне особине 63](#_Toc146036741)

[4.3.4.1 Јединица рада и тополошко сортирање 63](#_Toc146036742)

[4.3.4.2 Систем догађаја 64](#_Toc146036743)

[5. Aнализа перформанси ОРМ алата 65](#_Toc146036744)

[5.1 Тестно окружење 65](#_Toc146036745)

[5.1.1 Хардверске и софтверске спецификације 65](#_Toc146036746)

[5.1.2 Шема базе података 66](#_Toc146036747)

[5.2 Тестирање 68](#_Toc146036748)

[5.2.1 Поставка 68](#_Toc146036749)

[5.2.2 Упити 68](#_Toc146036750)

[5.2.3 Анализа добијених резултата 70](#_Toc146036751)

[5.2.4 Упоређивање објектно-релационих мапера 76](#_Toc146036752)

[6. Закључак 77](#_Toc146036753)

[7. Литература 79](#_Toc146036754)

[БИОГРАФИЈА 83](#_Toc146036755)

[КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА 85](#_Toc146036756)

[KEY WORDS DOCUMENTATION 87](#_Toc146036757)

# Увод

Модерне објектно-оријентисане апликације, поготово веб апликације, користе релационе базе података за перзистенцију података. Због разлика објектне и релационе парадигме, повезивање апликације са релационом базом података и писање упита је често захтеван посао за програмере који може да одузме доста времена у процесу развоја софтвера. Како би се олакшао и убрзао овај процес, заживела је идеја објектно-релационих мапера (ОРМ) који аутоматски премошћавају разлике у наведеним парадигмама, формално познатије као *Object-relational impedance mismatch*. Са друге стране, заступљено је мишљење како коришћење ових механизама може знатно да наруши перформансе апликација приликом комуникације са базом података, због чега се у многим артиклима на интернету наводи како су ОРМ алати анти-шаблон у развоју [13, 19, 20].

Циљ овог рада је, да се за најпопуларније програмске језике у развоју апликација, истраже начини имплементације одговарајућих алата за објектно-релационо мапирање, као и да се одреди њихов утицај на деградацију перформанси приликом употребе. За потребе истраживања, одабрани алати су: *Entity Framework* [9]*, Hibernate* [9]и *SQLAlchemy* [10]због велике заступљености језика С#, Јава и Пајтон у развоју модерних апликација.

Први део рада, поглавље 2, односи се на генералан поглед на ОРМ алате и основне концепте који су заступљени у њиховој имплементацији. У наредном поглављу, пружен је кратак увид у релевантну литературу која се бавила овом тематиком и идентификоване су празнине које овај рад тежи да попуни. Четврто поглавље, представља детаљан опис сваког од одабраних ОРМ алата и објашњена је имплементација најбитнијих механизама којима се користе наведени алати. У петом поглављу, представљена је мeтодологија којом се анализирао утицај изабраних ОРМ алата на перформансе апликације, као и резултати спроведеног истраживања. Последње поглавље представља свеукупан поглед на вршено истраживање и закључке изведене из добијених резултата. Додатно, наведене су могућности за проширивање рада у будућим истраживањима.

# Објектно-релационо мапирање

Као што је већ наведено у претходном поглављу, сврха ОРМ алата је да премосте разлику између објектне и релационе парадигме, познатију као oбјектно-релациона неусклађеност импеданси (енгл. *Object-relational impedance mismatch*) [5]. Овај термин је преузет из електроинжењерства и представља проблем који се јавља приликом повезивања два струјна кола са различитим импедансама, услед којег долази до слабљења преноса снаге [7]. Генерално, термин неусклађеност импеданси, користи се у инжењерству и рачунарским наукама да се укаже на немогућност ефикасне интеракције између два система.

С тим на уму, може се повући паралела како би се описале некомпатибилности између објектно-оријентисане и релационе парадигме које изазивају проблеме у дизајну и имплементацији апликација заснованих на њиховој комбинацији.

Неусклађеност две парадигме настаје због тога што оне, у основи, имају различит поглед на концепт података. Програми написани у објектно-оријентисаним језицима се фокусирају на објекте, инстанце класе која садржи атрибуте, који представљају структуру, али и методе којима се дефинише понашање инстанце. Објекти се не гледају само кроз призму података које у себи енкапсулирају, већ и кроз способност трансформације тих података и интеракције са другим објектима са којима могу податке да размењују. Са друге стране, релациона парадигма заснована је на математичком појму релације и основни концепт инстанце представљен је редом у табели. Овим се акценат ставља на структуру података, док је могућност трансформације података одвојена од структуре.

Разлике објектно оријентисане и релационе парадигме које изазивају проблеме у развоју могу да се манифестују кроз неколико различитих концепата који ће бити наведени у наредним секцијама.

## Наслеђивање

У објектно-оријентисаном програмирању, наслеђивање омогућава класама да усвајају атрибуте и методе својих наткласа, изграђујући тако хијерархију објеката. Meђутим, наилази се на изазове када се покуша применити ова хијерархија на релациону парадигму. Релациони модели су фокусирани на структуралне односе између података, без обраћања посебне пажње на хијерархију између табела.

Објектно-релациони мапери пружају неколико начина за решавање овог проблема :

* Једна табела по ланцу наслеђивања
* Једна табела по класи
* Конкретна табела по ланцу наслеђивања

### Једна табела по ланцу наслеђивања

A diagram of a customer

Description automatically generatedСве класе из хијерархије наслеђивања се представљају једном табелом [11]. Колоне табеле представљају својства свих класа. Увођењем посебног дискриминатор обележја, за сваки ред у табели се означава која је његова одговарајућа класа у дефинисаној хијерархији, чиме се омогућава мапирање. Пример ове стратегије дат је на слици 2.1.

Слика 2.1 Једна табела по ланцу наслеђивања

Предност овог приступа је што добро прати еволуцију објектног модела, јер су модификације у шеми знатно олакшане, као и што нема потребе за операцијама спајања (енгл. *join*) које су скупе из аспекта извршавања упита.

Са друге стране, јавља се проблем неискоришћеног простора. Пошто сваки ред у табели има колоне сваког подтипа, овакво мапирање ће резултовати у мноштву колона са непостојећим (енгл. *null*) вредностима. Такође, величина овакве табеле може да представља уско грло система приликом операција читања и писања.

### Једна табела по класи

A diagram of a customer

Description automatically generatedОсновна (родитељска) класа има своју табелу која садржи ссвојства која су заједничка за све изведене класе [11]. Свака изведена (дечја) класа такође има своју табелу која садржи само она својства која су јој специфична. Табеле изведених класа обично имају страни кључ који се повезује са примарним кључем родитељске табеле. Пример стратегије приказан је на слици 2.2.

Слика 2.2 Једна табела по класи

Предност овог приступа je што је база података нормализована, дакле неће постојати додатне колоне које носе непостојеће вредности као што је то случај са претходним начином.

Ипак, проблем код овог приступа је што читање целокупног објекта често захтева операције спајања између основне и изведене табеле, што може да деградира перформансе.

### Конкретна табела по ланцу наслеђивања

У овој стратегији, свака конкретна класа, односно, класа која може бити инстанцирана у објектно-оријентисаном моделу, има своју табелу у релационој бази података [11]. Ова табела садржи колоне за сва својства класе, укључујући својства која наслеђује из својих родитељских класа. Пример стратегије приказан је на слици 2.3.

Предност овог приступа је што су упити знатно поједностављени у односу на претходну стратегију пошто не захтевају операције спајања. Такође, уколико се промени неко својство изведене класе, промене ће бити независне од других табела.

A diagram of a customer

Description automatically generatedСа друге стране, јавља се проблем редундансе, с обзиром на чињеницу да свака табела садржи сва својства једне конкретне класе. Осим што се, очигледно, повећава величина простора у бази података, овакав приступ резултује проблемима са ажурирањем основне класе, јер свака измена у основној класи мора да се пропагира на све изведене табеле.

Слика 2.3 Конкретна табела по ланцу наслеђивања

## Асоцијације

Услед разлика у начинима представљања односа између ентитета две парадигме, може настати неколико проблема. Да би се разумели проблеми који настају, неопходно је разумети кључне разлике. Релационе базе података успостављају односе између табела помоћу страних кључева ка другим табелама, док објекти енкапсулирају референце ка другим објектима које се чувају у меморији током извршавања програма [7]. Поред тога, објекти, кроз употребу колекција, могу да чувају референце ка мноштву других објеката помоћу само једног поља. Са друге стране, релациона база података, због нормализације, захтева да све везе ка другим табелама буду изражене кроз атомичне вредности. Ово доводи до инверзије у структурама података између објеката и табела. Ова појава се лако може објаснити кроз пример:

Посматра се апликација која опслужује веб продавницу и нуди могућност да се различити производи из понуде сместе у корпу за куповину. Посматрано из апсекта објектне парадигме, објекат „корпа“ ће садржати у себи листу ставки које представљају производе који су тренутно смештени у њој. Ако би се ово представило у релационој бази података, табела ставка би, кроз колону која би представљала идентификациону ознаку корпе, требало да има ограничење страног кључа ка табели корпа, јер табела корпа, због прве нормалне форме, не сме да има колоне које садрже неатомичне вредности.

Решење које је могуће у овом случају је, да се за сваки објекат чува поље које репрезентује примарни кључ у одговарајућој табели, како би се могло извршити мапирање између референци објеката и страних кључева. Другим речима, реализује се кроз ограничење страног кључа.

## Добављање података

При раду са релационим базама података, поготово у модерним апликацијама, начин на који систем добавља податке игра кључну улогу у његовој перформантности. Две основне стратегије за добављање података у објектно-релационим маперима су *Lazy Loading* и *Eager Loading*. Иако обе стратегије нуде различите приступе у решавању истог проблема, свака има своје специфичне предности и изазове у коришћењу. Неадекватно коришћење датих механизама може непосредно довести до деградације у перформансама система, које могу резултовати непотребним упитима ка бази података, изазивајући додатно опетерећење у систему и продужити време одзива апликације. Стога је од велике важности разумети како ове стратегије функционишу, како би се, правилном применом, постигле жељене перформансе система.

### *Lazy Loading*

*Lazy Loading* је механизам који омогућава учитавање повезаних података само када су они заиста потребни [11]. Другим речима, подаци се не довлаче из базе података све док апликација експлицитно не захтева приступ тим подацима, што може побољшати ефикасност система. Међутим, непромишљено коришћење *Lazy Loading-a* може довести до такозваног ***N+1 query*** проблема, који ствара значајно оптерећење базе података.

### Проблем N+1 упита

Већина модерних ОРМ алата користе *Lazy Loading* као основно понашање за добављање података. С обзиром на чињеницу да се не добављају подаци које апликација експлицитно не затражи, у тренутку када они постану неопходни за обраду, систем ће слати упите ка бази за сваки неопходан податак. Овакво понашање неретко резултује проблемом N+1 упита ка бази [7]. Ова појава се лако може објаснити на примеру:

Претпоставка је да постоји база података која опслужује друштвену мрежу у којој су моделоване табеле Објава (Post) и Ознака (Tag). Однос између ових табела је један према више, односно једна објава може да садржи више ознака. Уколико би се у систему користио *Lazy Loading*, добављање података би се свело на:

* Основни упит којим би се добавиле све објаве.
* За сваку објаву која се добави, прави се додатни упит како би се добавиле све ознаке везане за конкретну објаву.

За N објава у бази, уместо да се шаље само један упит, ово би резултовало са N+1 упитом ка бази података, одакле и потиче назив проблема. N+1 проблем може да изазове спор одзив система, као и да лимитира скалабилност система, стога је битно узети у обзир неколико могућих солуција. Једна од њих је *Eager Loading*.

using (var context = new BlogContext())

{

var posts = context.Posts.ToList();

foreach (var post in posts)

{

var tags = context

.Tags

.Where(t=>t.post\_id == post.post\_id)

.ToList();

}

}

Листинг 2.1 N+1 проблем

### *Eager Loading*

*Eager Loading* подразумева учитавање свих повезаних података у исто време када и основни објекат [11]. Ово значи да апликација одмах добавља све неопходне релације и податке, минимизујући број упита ка бази података. Иако овај приступ решава проблем N+1 упита и користан је када су сви подаци одмах потребни, може представљати оптерећење у случајевима када нису неопходни сви подаци.

Из претходно наведеног може се закључити да избор између две наведене стратегије треба да буде базиран на конкретним потребама апликације и перформансама система.

## Кеширање

Кеширање у системима објектно-релационог мапирања је једна од основних техника за повећање ефикасности апликација, која редукује потребу за учесталим упитима ка бази података. Чувањем често коришћених података у меморији за брже преузимање, ова функционалност се позиционира између апликативног слоја и слоја базе података [7]. Међутим, како база података може бити измењена од стране других корисничких процеса или апликација, кеширани подаци могу постати застарели и неосвежени у кешу, што може довести до проблема са актуелношћу података. Због тога је инвалидација кеша од суштинског значаја, јер осигурава да кеширани подаци остану релевантни и синхронизовани са актуелним стањем у бази. ORM алати обично нуде опције за конфигурисање политика кеширања на нивоу појединачних класа, омогућавајући различите стратегије за различите типове података.

## Обрасци понашања ОРМ алата

У контексту потребе за перзистенцијом података у апликацијама, често се примењују специфични дизајн обрасци како би се обезбедила ефикасност, конзистентност и одрживост кода. Ови обрасци пружају систематизоване начине за решавање уобичајених проблема који настају при интеракцији између апликације и базе података. Концепти као што су *Identity Map* и *Unit of Work* [11] представљају дубоко укорењене обрасце који играју кључну улогу у ефективном раду ОРМ алата. У наредним секцијама пружен је дубљи увид у ова два обрасца.

### *Identity Map*

*Identity Map* је дизајн образац који има циљ да одржава конзистентност учитаних објеката у рам меморији одређене сесије или трансакције [11]. Главна предност овог образца је у спречавању многобројних учитавања истог објекта из базе података током једне сесије чиме може доћи до аномалија модификације. Основна идеја овог обрасца је да систем, у специјализованим мапама, чува објекте који су претходно добављени из базе података. За избор кључа мапе, најчешће се узима кључ табеле у бази података. Када систем треба да учита објекат, прво се врши провера да ли се тражени објекат налази у мапи, и уколико се налази у меморији, систем не мора да шаље упит ка бази. Са друге стране, уколико се тражени објекат не налази у мапи, систем шаље упит ка бази, добавља га, и смешта у мапу за будуће коришћење.

### *Unit of Work*

*Unit of Work* [11] обезбеђује механизам за праћење промена над објектима. Његова сврха је да се обезбеди атомичност у примени промена: или се све промене успешно примене, или ништа од њих. Oвим се осигурава конзистентност података и ефикасна администрација ресурса. Синхронизација са базом података се може вршити са сваком променом објекта, али то може да доведе до великог броја позива ка бази података, што потенцијално доводи до деградације перформанси. Идеја *Unit of Work* обрасца је да постоји централна компонента која води рачуна о свим променама над објектима модела и управља трансакцијама и синхронизацијом са базом података.

## Употреба ОРМ алата у развоју апликација

Објектно-релациони мапери представљају моћан инструмент у развоју савремених софтверских апликација и омогућавају рад са базама података на вишем нивоу апстракције. Стварна моћ OРM-ова лежи у њиховој способности да преведу објектно-оријентисане моделе у релационе упите и обрнуто, без потребе за ручним писањемем *Structured Query Language* (*SQL)* упита. Ово не само што значајно поједностављује развој и одржавање кода, већ, такође, смањује могућност грешака у комуникацији са базом података. Међутим, употреба OРM-а такође захтева дубоко разумевање њихових механизама и образаца, како би се избегли потенцијални проблеми у перформансама или неслагања у репрезентацији података. Управо из тог разлога, неопходно је ставити акценат на комбиновању предности OРM алата са добрим познавањем основа релационих база података.

У овом поглављу разматрани су теоријски аспекти и изазови који се јављају при мапирању између објектно-оријентисане и релационе парадигме. Приказани концепти и изазови служе као основа за разумевање значаја и сложености која се крије иза употребе објектно релационих мапера.

У наредним поглављима, ставиће се акценат на конкретне ОРМ алате, где ће бити пружен увид у то како су различити алати приступили решавању наведених изазова.

# Преглед релевантне литературе

Проучавање релевантне литературе је кључни корак у истраживачком процесу који помаже у разумевању постојећих методологија. У контексту тематике објектно-релационих мапера, критички осврт на литературу обезбеђује основу за сагледавање предности и недостатака различитих решења, као и мотивацију за писање овог рада, како би се попуниле одређене празнине у литератури.

## Методологија претраге литературе

Претрага је вршена коришћењем различитих научних база података, укључујући *Scopus*, *IEEE Xplore*, *Research Gate*. Неке од кључних речи које су коришћене за претрагу су: "објектно-релационо мапирање", "анализа перформанси ОРМ алата" и "Object-relational impendace mismatch". Такође, с обзиром на чињеницу да је подељењо мишљење о употреби ОРМ алата у развоју софтвера, један део пажње је усмерен на артикле написане од стране признатих научника и програмера у области објектно-релационих мапера и софтвер архитектуре. Поред научних чланака и радова, у обзир су узете и књиге посвећене овој тематици, где се посебно истичу дела Мартина Фаулера [11], Мајкла Бајера [14] и Влада Михалцеа [12], који се сматрају за референтне личности у области објектно-релационог мапирања.

## Опис претходних истраживања

У овој подсекцији представљена је синтеза најзначајнијих закључака из претходних истраживања о објектно-релационим маперима. Истраживања су се фокусирала на различите аспекте: од теоријске основе OРM-а, преко различитих техничких имплементација, до перформанси и реалних примена у различитим сценаријима.

У свом познатом чланку, *The Vietnam of Computer Science* [13], Тед Њуард прави аналогију објектно-релационог мапирања са америчком војном кампањом у Вијетнамском рату, при чему наводи да употреба алата за објектно-релационо мапирање у раним фазама развоја софтвера пружа ефикасност, док у каснијим фазама, са све већом употребом и улагањем времена, деградације перформанси нису занемарљиве због превелике разлике између две парадигме. Његова главна теза је да не постоји добро решење за премошћавање наведених разлика и да сва постојећа решења укључују болне компромисе.

У раду Цветковића и Јанковића [6] излагана је детаљна анализа и поређење два популарна OРM алата за .NET: *Entity Framework* (EF) и *NHibernate*. Противећи се распрострањеном мишљењу да OРM алати уводе значајан додатни трошак (енгл. *overhead*) у односу на традиционалне приступе приступу подацима, аутори показују да је, за већину уобичајених упита, перформанса EF и *NHibernate-а* слична директним SQL упитима. Иако су примећени изузеци у неким случајевима, као што су масовна ажурирања или груписање у EF, основна порука рада је да се рупа у перформансама значајно сужава код савремених OРM оквира као што су EF и *NHibernate*.

Koли, Стајнер и Асадузаман у свом раду [4] разматрају изазове који произлазе из коришћења OРM алата при превазилажењу проблема неусаглашености између објекатно-оријентисаних програмских језика и релационих база података. Аутори детаљно приказују анти-шаблоне узроковане коришћењем OРM-a који утичу на перформансе упита, укључујући проблеме као што су превише жељни захтеви, непотребно угњежђивање подупита и лоша читљивост упита. Иако представљају стратегије за оптимизацију, аутори истичу да је приступ решавању проблема са OРM-ом комплексан и често захтева разматрање на нивоу целокупне шеме базе података. Ово истраживање поставља значајне основе за разумевање и оптимизацију учинка OРM алата на перформансе релационих упита и подстиче даља истраживања у правцу интеграције објекатно-оријентисаних и релационих приступа.

Освртањем на неколико радова може се закључити да међу научним колективом влада подељено мишљење о употреби алата за објектно-релационо мапирање. У наредном потпоглављу, биће пружен осврт на неке од недостатака у литератури које ће овај рад покушати да попуни.

## Уочени недостаци у литератури

На основу анализе релевантних радова, у наставку су описани недостаци и ограничења у постојећој литератури о објектно-релационим маперима које треба узети у обзир:

### Непотпуност

Многи извори пропуштају дискусију о релевантним аспектима мапирања и њиховом утицају на карактеристике квалитета система, услед чега је отежано разумевање предности и мана разуличих стратегија мапирања [4,6,21]. Уколико би се читаоци фокусирали само на неколико радова, не би се могла створити шира слика о свим потенцијалним компромисима који се морају направити приликом коришћења алата.

### Одсуство анализе Пајтон ОРМ алата

Иако је Пајтон (енгл. Python) један од најпопуларнијих програмских језика данашњице, чини се да литература занемарује OРM алате специфичне за овај језик. С обзиром на растућу употребу Пајтон програмског језика у развоју веб апликација, потребна је дубља анализа и рецензија ових алата у контексту мапирања.

Иако постојећа литература обухвата велики део аспеката објектно-релационог мапирања, ипак постоје значајни недостаци. У наредним поглављима биће пружена посебна на пажња на изабране алате за објектно-релационо мапирање, са циљем да се попуне неке од уочених недостатака у литератури

# Преглед изабраних ОРМ алата

У претходним поглављима, представљене су основе објектно-релационог мапирања и проблеми који ови алати решавају. Постоји много различитих алата за објектно-релационо мапирање, али, због своје велике заступљености у развоју модерних апликација, изабрани су *Entity Framework*, *Hibernate* и *SQLAlchemy* за детаљну анализу начина на који су они имплементирани.

Сваки од ових алата има своју јединствену архитектуру, начине извршавања упита и механизме оптимизације. Циљ овог поглавља је да пружи техничке детаље, предности и могуће изазове који произилазе из коришћења наведених алата.

Разумевање техничких аспеката ових алата, попут начина на који обрађују упите, управљају кеширањем и врше оптимизације, може значајно помоћи програмерима да максимално искористе њихове потенцијале и избегну типичне замке.

Анализа ће започети детаљима о имплементацији *Entity Framework-a*, затим ће се прећи на *Hibernate* и завршити детаљним прегледом *SQLAlchemy* радног оквира.

## *Еntity Framework*

Entity Framework (ЕF) представља интегрални део .NET платформе и представља водећи алат за објектно-релационо мапирање унутар *Microsoft* екосистема [18]. Од своје појаве, EF је доживео бројне ревизије и промене, при чему је свака верзија доносила побољшања у функционалностима и флексибилности.

У овом поглављу ће бити представљене архитектурне основе *Entity Framework-a*, како он трансформише *Language Integrated Query* (LINQ) упите у SQL, и који механизми стоје иза управљања подацима и оптимизације. Такође, биће представљени различити аспекти који утичу на перформансе и добре праксе за достизање оптимизованих резултата при раду са EF.

### Aрхитектура алата

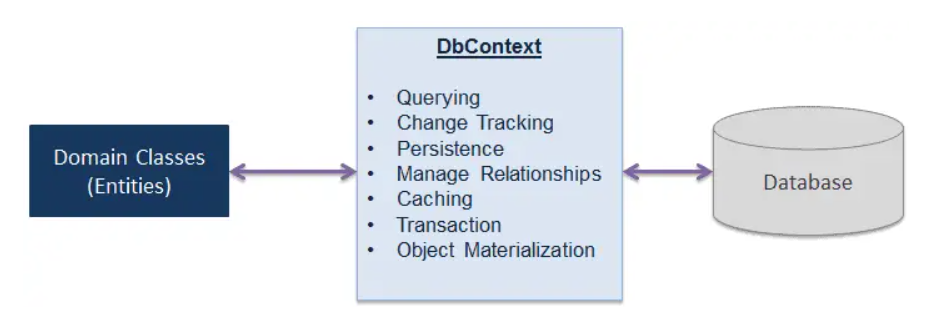
#### DbContext и DbSet

DbContext представља основу класу одговорну за интеракцију са базом података. Ова класа служи као интерфејс између доменских класа и табела у бази података [9].

Одговорности DbContexta у раду са базом података су:

* **Управљање упитима**: Превођење *LINQ-to-Entities* упита у SQL упит који се шаље ка бази података
* **Праћење промена ентиета**: DbContext представља имплементацију *Unit of Work* шаблона, стога ова класа прати све промене које се дешавају над ентитетима и осигурава да база података буде у конзистентном стању
* **Перзистенција података**: На основу испраћеног стања ентитета, DbContext извршава одговарајуће *Data Manipulation Lagunage* (DML) наредбе над базом података
* **Кеширање**: Пружа могућност кеширањa на првом нивоу и, за трајање свог животног циклуса, чува ентитете који су били добављени неким од претходних упита
* **Материјализација**: Процес током којег се подаци добављени из базе конвертују у објекте ентитета

На слици 4.1 приказана је позиција DbContext класе у ЕF алату.

[[1]](#footnote-1)Слика 4.1 Позиција *DbContext* класе у *Entity Framework-y*

DbSet је класа која представља интерфејс помоћу којег се врши комуникација ка једној табели у бази података. Пружа могућност вршења Add, Remove, Find, Attach операција помоћу којих се генеришу неопходне DML операције над базом података. Такође, помоћу метода Local, SqlQuery и DbQuery пружа могућност, приступу локалним подацима у кеш меморији DbContexta, директно писање SQL упита, као и писање LINQ упита над ентитетима. DbSet се најчешће декларише као поље класе која наслеђује DbContext, чиме се остварује мапирање ентитета класа на записе у табелама релационе базе података и координише свим инстанцама.

На листингу 4.1 приказана је типична употреба DbContext и DbSet класа приликом конфигурације апликација заснованих на EF алату.

public class BenchmarkDbContext : DbContext

{

public BenchmarkDbContext() {}

public DbSet<User> Users { get; set; }

public DbSet<Profile> Profiles { get; set; }

}

Листинг 4.1 Препоручен начин коришћења DbContext и DbSet класа

#### Entity Data Model

*Entity Data Model* (EDM) је скуп концепата који описују структуру података независно од начина на који се они чувају. ЕДМ је, заправо, репрезентација у меморији (енгл. *in-memory*) свих мета података неопходних за објектно-релационо мапирање и састоји се од: концептуалног модела, модела складишта (eнгл. *storage model*) и мапирањима између њих. Овакав концепт омогућава логици апликације да се фокусира на домен, изолојући је од детаља приступа бази [9].

Концептуални модел чине доменске класе апликације, *DbContext* класа и *DbSet* колекције. Складишни модел заснован је на шеми базе података са којом је апликација повезана. Информације о мапирањима су неопходне како би се могло ефикасно врши пребацивање из концептуалног модела у складишни и обрнуто.

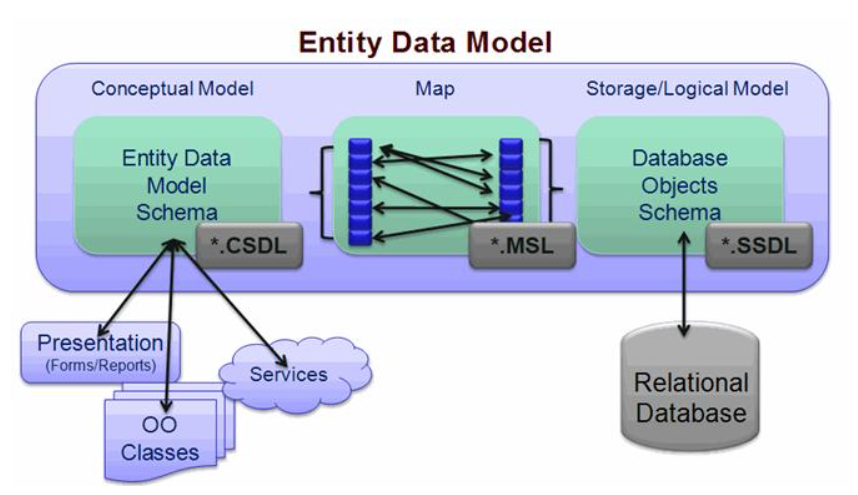
Како би ово мапирање било ефикасно, ЕF користи три језика заснована на *еXtensible Markup Language* (XML) језику: *Conceptual Schema Definition Language* (CSDL), *Store Schema Definition Language* (SSDL) и *Mapping Specification Language* (MSL).

**CSDL** се фокусира на концептуални део апликације. Језик описује ентитете и њихове односе, без узимања у обзир специфичних детаља или структуре конкретне базе података.

**SSDL** описује логичку структуру података базе података повезане са апликацијом, укључујући табеле, колоне, везе и остале релационе атрибуте.

Да би се осигурало да оба света могу да комуницирају без ометања, **MSL** постоји као мост између ова два језика. Овај језик дефинише правила и начине на које се концептуални модели из CSDL-a мапирају на енитете у бази података описаних путем SSDL-a.

На слици 4.2 приказана је архитектура *Entity Data Model-a.*

[[2]](#footnote-2)Слика 4.2 Приказ архитектуре ЕДМ-а

#### Руковање конекцијама

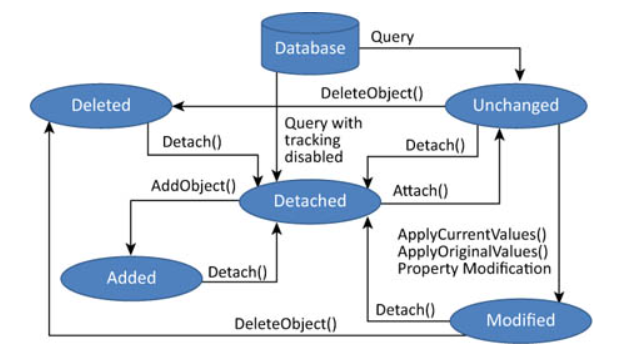
У раду са базом података, битно je разумети како се управља конекцијама, почевши од њиховог успостављања до затварања, како не би дошло до ситуација у којима су угрожене перформансе или безбедност система. ЕF пружа ефикасан и оптимизован систем за руковање конекцијама и он се састоји од неколико кључних сегмената:

* **Успостављање конекције:** Када се креира DbContext инстанца, EF не отвара одмах конекцију са базом података. Уместо тога, конекција ће бити отворена само када је то потребно, приликом извршења упита или када је потребно сачувати податке.
* **Животни циклус конекције:** У својој основној поставци, ЕF користи краткотрајне конекције, што значи да се конекција отвара само када је потребно и брзо се затвара после коришћења чиме се минимизују ресурси које конекција заузима. Међутим, у неким случајевима, као што су трансакције, конекција може да буде отворена дужи временски период.
* **Затварање и ослобађање конекције:** Када је рад са базом података завршен, EF аутоматски затвара конекцију како би осигура да не дође до непотребног трошења ресурса. Такође, препоручено је да се користи using исказ приликом рада са инстанцом DbContext класе како би се осигурало правилно ослобађање ресурса.

#### Праћење промена ентитета

*Entity Framework* користи механизам праћења промена како би идентификовао и записао све измене над ентитетима од последњег учитавања из базе података. Ова способност је кључна за рад са ОРМ системима и осигурава ефикасност и усаглашеност при ажурирању базе података.

Ентитет може имати једно од пет стања (слика 4.3) која одређују како ће се на њега гледати при следећем позиву методе SaveChanges инстанце класе DbContext:

* **Unchanged (Непромењен)**: Ентитет је учитан из базе података и није било измена на њему од тада.
* **Modified (Измењен)**: Након учитавања из базе података, вршене су промене на ентитету. Ова променаће бити сачувана у бази података при следећем позиву методе *SaveChanges*.
* **Deleted (Обрисан)**: Ентитет је обележен за брисање и биће уклоњен из базе података при следећем позиву методе *SaveChanges*.
* **Added (Додат)**: Ентитет је додат у контекст али још увек не постоји у бази података. Биће креиран нови запис у бази података при следећем позиву *SaveChanges* методе.
* **Detached (Одвојен)**: Ентитет је у контексту али не прати промене на њему. Ниједна промена овог ентитета неће бити рефлектована у бази података.

[[3]](#footnote-3)Слика 4.3 Животни циклус ентитета

### Извршавање упита

#### Language Integrated Query (LINQ)

LINQ представља интегрисан систем за упите који је основни део .NET радног оквира. Омогућава директно писање упита у C# програмском језику, користећи синтаксу која је читљива и интуитивна. Применом LINQ језика могу се лако изводити различите операције као што су селекција, филтрирање, сортирање, груписање и спајање података из различитих извора. *LINQ to Entities* је врста LINQ jeзика намењена за слање упита ка бази података користећи концептуалне моделе и саставни је део *Еntity Framework-a*.

#### Превођење LINQ упита

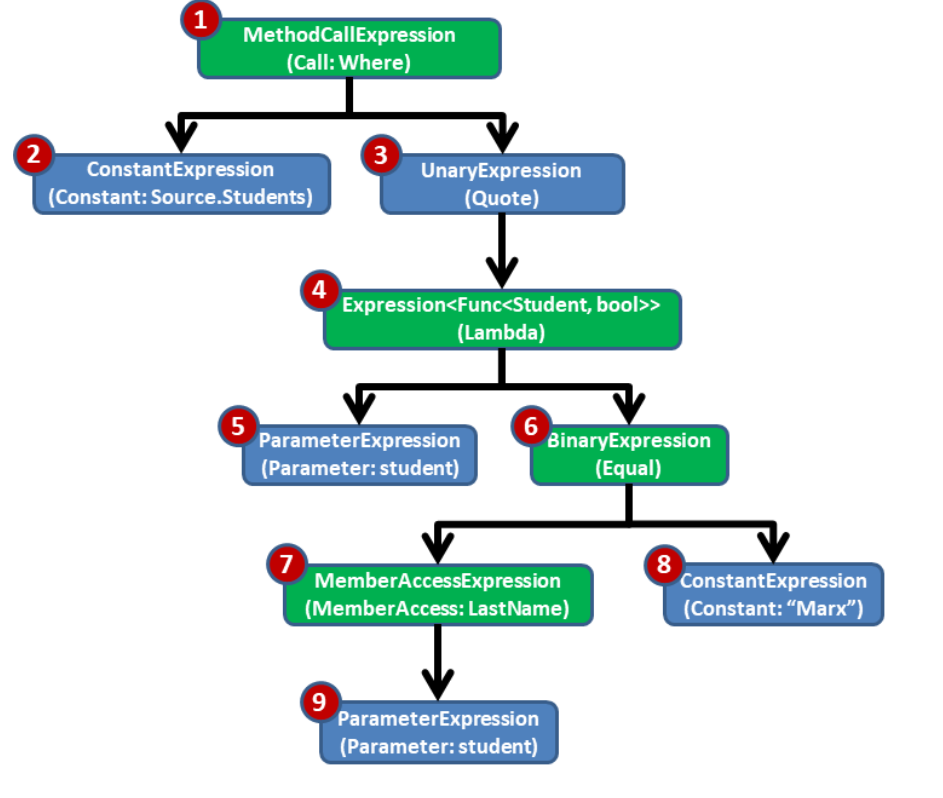
Интерно, LINQ упити се представљају као изразна стабла (енгл. *expression trees*), структуре података које представљју упите као скуп објеката [21]. На овај начин се омогућава алату да "разуме" упит на вишем нивоу апстракције, чиме се омогућава оптимизација упита пре превођења у SQL. Када LINQ упит треба да се изврши, изразно стабло се прослеђује упитном каналу за обраду (енгл. *pipeline*) *Entity Framework-a*. Oвај канал за обраду садржи компоненте које анализирају изразно стабло са циљем генерисања одговарајућег SQL упита. Генерисани SQL упит се затим прослеђује провајдеру базе података (енгл. *database provider*), библиотеци намењеној за комуникацију са одговарајућом базом података, који се повезује са базом и извршава упит.

За упит наведен у листингу 4.2:

Source.Students

.Where(student=>student.LastName == “Marx);

Листинг 4.2 Пример LINQ to Entities упита

[[4]](#footnote-4)Изгенерисаће се следеће изразно стабло на слици 4.3:

Слика 4.4 Изразно стабло LINQ упита

#### Maтеријализација

Након што се SQL упит изврши и врати резултате, Entity Framework мора да преведе те резултате у инстанце објеката који су дефинисани у апликацији. Овај процес је познат као материјализација. Он укључује креирање објеката, попуњавање њихових својстава са вредностима из резултата упита и, у неким случајевима, повезивање објеката у сложене структуре као што су листе или графови.

Након разматрања процеса превођења LINQ упита у SQL у одељку 4.1.2, у следећем одељку 4.1.3 детаљно ће се размотрити стратегије и методе за оптимизацију и подешавање перформанси упита.

### Aспекти побољшања перформанси

#### Lazy Loading и Eager Loading

EF подразумевано користи *Lazy Loading* стратегију за добављање података, што значи да се повезани подаци са циљаним ентитетом не преузимају, одмах, приликом добављања главног ентитета, већ по експлицитном захтеву [9]. То се постиже тако што ЕF креира посредничке (енгл. *proxy*) објекте, који, на захтев, комуницирају са базом како би добавили повезане податке, који још нису преузети. Као што је већ напоменуто, ова стратегија учитавања може да изазове N+1 проблем, стога ЕF нуди могућност избора *Eager Loading* стратегије. Oво се постиже коришћењем Include методе у LINQ-y изразy, што се може видети у наредном примеру на листингу 4.3:

return context.Users.Include(x => Posts).ToList();

Листинг 4.3 Избор *Eager Loading* стратегије

Oвим се у једном упиту учитавају и подаци о главном ентитету (*User*), као и о повезаном ентитету (*Post*). Meђутим, треба још једном напоменути да прекомерно коришћење ове методе може негативно утицати на перформансе.

#### Индексирање

Ефикасно индексирање може драстично побољшати перформансе упита. Стварање индекса на колонама које се често користе у упитима или приликом операција спајања, може помоћи у бржем добављању података. EF нуди могућност креирања основних и композитних индекса над атрибутима ентитета и то се може извршити на два начина: уз помоћ анотација (листинг 4.4), или коришћењем *FluentAPI-a* (листинг 4.5), посебног начина за конфигурацију модела.

public class Tag

{

public int Id {get; set;}

[Index]

Public string Name {get; set;}

}

Листинг 4.4 Креирање индекса уз помоћ анотација

protected override void

OnModelCreating (DbModelBuilder builder)

{

builder.Entity<Tag>()

.Property(t => t.Name)

.HasColumnAnnotation(

“Index”,

new IndexAnnotation(

new IndexAttribute(“Name\_Index”)

));

}

Листинг 4.5 Креирање индекса уз помоћ *FluentAPI-ja*

#### Профилисање

Профилисање у EF алату представља битан инструмент за оптимизацију и надзор комуникације апликације са базом података. Омогућава програмерима да идентификују неефикасне упите, пружа увид у преведене SQL упите и помаже у дијагностиковању потенцијалних проблема у перформансама. У својој основи, *Entity Framework* омогућава логовање путем којег се могу приказати информације о извршеном упиту, као што су време извршавања, добављена количина података, као и преведена LINQ наредба у нативни SQL.

За постављен LINQ упит приказан у листингу 4.6:

return context.Users.Where(x => x.DoB >= year).ToList();

Листинг 4.6 Пример упита за који се врши профилисање

Профилисањем се утврђује одговарајући SQL упит који је *Entity Framework* генерисао (листинг 4.7):

SELECT u.user\_id, u.date\_of\_birth

FROM Users AS u

WHERE u.date\_of\_birth >= @\_\_year\_0

Листинг 4.7 Генерисан SQL упит од стране EF-a

#### Кеширање

*Entity Framework* омогућава неколико облика кеширања података [9]:

* **Кеширање објеката** (eнгл. *Object caching*): Посебан ObjectStateManager објекат, који је уграђен у контекст класу, води рачуна о објектима који су претходно добављени и налазе се у меморији. Ово је познатије и као кеширање првог нивоа (енгл. *First level cache*).
* **Keширање плана извршавања упита**: Чување изгенерисане SQL наредбе за упите који се понављају неколико пута.
* **Кеширање метаподатака** - Служи за дељење метаподатака о моделу између неколико различитих конекција.

Приликом кеширања првог нивоа, ObjectStateManager објекат води рачуна да ли се ентитет, који се тражи упитом, већ налази у меморији. Ако се у меморији налази ентитет са идентичним кључем који је требао да се добави, Entity Framework ће га укључити одмах у резултат упита. Треба напоменути да ће Entity Framework свакако извршити упит ка бази података, али ће се заобићити фаза материјализације објекта чиме се штеди на ресурсима. Са друге стране, треба узети у обзир да уколико се превише ентитета налази у меморији, то може негативно да утиче на перформансе упита.

Кеширање метаподатака је битно за ефикасну интеракцију са моделом података у EF. Метаподаци представљају структурне и конфигурационе информације о моделу, као што су дефиниције ентитета и везе између ентитета, као и подаци неопходни за успостављање конекције са базом података. Приликом креирања инстанце DbContext класе, EF интерпретира ове метаподатке и додељује одговарајући објекат конекције контексту. С обзиром на чињеницу да је животни век контекст класе везан за трајање трансакције, EF кешира метаподатке како би брзо приступио конфигурационим информацијама приликом креирања нових инстанци контекста. Такође, уколико се креира нови објекат конекције који има исти конекциони стринг, као и неки од постојећих објеката конекције, EF ће, ради ефикасности, искористити кеширане метаподатке.

### Oстале кључне особине

#### Миграције

Кључна способност коју нуди *Entity Framework* је инкрементална миграција података за ажурирање шеме базе података, уз промене у доменском моделу апликације [9]. EF пружа функцију миграција која омогућава систематско развијање шеме базе података у складу с променама направљеним у ентитетским класама у коду.

Миграције су обично *code-first*, што значи да је ентитетски модел у коду сматран извором истине и база података треба да буде ажурирана да би се успоставила сагласност с њим. Када се направе промене у ентитетским класама, као што су додавање или уклањање атрибута, EF може генерисати миграциону скрипту која садржи кораке потребне за ажурирање базе у складу с тим. Свака миграција има Up() методу која примењује промене шеме и Down() методу која их враћа, што омогућава да се, ефикасно, измењена шема базе података, врати на претходно стање.

Миграције садрже само инкременталне промене, а не целокупну шему. EF прати до сада примењене миграције преко историјске табеле. Пре него што примени даље миграције, упоређује снимак (енгл. *snapshot*) у историји са тренутним моделом да би применио само потребне промене. За ове потребе, ЕF садржи посебан фајл под називом *Snapshot* у којем се налази тренутно стање шеме базе података. Миграције могу креирати, брисати или модификовати структуралне компоненте базе података попут табела, колона, индекса и ограничења да би се шема усагласила.

Командна линија EF алата пружа команде за генерисање, примену и враћање миграција. За постојеће базе података, EF Core такође пружа функционалност за инверзно инжењерство модела и миграција из шеме базе података. У глобалу, миграције омогућавају управљање променама шеме на систематизован и аутоматизован начин у различитим окружењима за релационе базе података као што су SQL Server, PostgreSQL и MySQL.

#### Контрола конкурентног приступа

С обзиром на чињеницу да је база података дељени ресурс у времену, што значи да више корисничких процеса могу да јој приступају истовремено, *Entity Framework* пружа механизме којима се може контролисати вишекориснички режим рада [9]. Најчешћи метод је оптимистичка контрола конкуренције путем верзионисања реда у табели, што се постиже увођењем посебне колоне која представља верзију ентитета. Пример оптимистичке контроле конкуренције приказан је у листингу 4.8. Ово повезује временску ознаку или инкрементирајући број верзије реда са сваким записом, који се ажурира при свакој измени. При чувању, EF проверава да ли је верзија промењена од добављања, и уколико је то био случај, изазове се *DbUpdateConcurrencyException* изузетак указујући на застарелу верзију података. Са друге стране, песимистичка контрола конкуренције користи закључавање базе података да би спречила конфликте ажурирања. Закључавање може бити имплицитно спроведено од стране *Entity Framework-a* или експлицитно наведено за одређене ентитете или упите. Ипак, треба узети у обзир да, иако песимистичко закључавање спречава настајање конфликата ажурирања, оно умањује могући ниво конкурентности при комуникацији са базом података.

public class Person

{

public int Id {get; set;}

[Timestamp]

public byte[] Version {get; set;}

}

Листинг 4.8 Оптимистичко закључавање

*Entity Framework* представља снажан алат за управљање релационим базама података у .NET апликацијама. Кроз различите могућности као што су миграције података, контрола конкуренције и интегрисане оптимизације упита, EF омогућава програмерима да ефикасно мењају шему базе података и управљају подацима у својим апликацијама. Са правилном имплементацијом и коришћењем најбољих пракси, максимално се могу искористити предности које овај OРM радни оквир нуди.

## Hibernate

*Hibernate* представља алат за објектно-релационо мапирање, развијен за Јава програмски језик [8]. Захваљујући својој стабилности и поузданости, као и квалитетној документацији и активној заједници програмера који раде на његовом константном унапређењу, истакао се као један од најпопуларнијих објектно-релационих мапера за Јава екосистем.

С друге стране, JPA (*Java Persistence API*) је Јава спецификација за управљање перзистенцијом података [15]. *Hibernate* је, у ствари, имплементација JPA спецификације, чиме је омогућено да развојни тимови користе JPA стандарде и анотације у *Hibernate* апликацијама.

У оквиру овог поглавља, детаљно ће се размотрити архитектура *Hibernate* алата, његове основне концепте и функционалности, као и начине којима се овај алат интегрише са SpringJPA, делом *Spring* радног оквира који олакшава комуникацију са базама података у Јава апликацијама [16].

### Архитектура алата

#### ЈDBC

*Java Database Connectivity* (ЈDBC) представља програмски интерфејс развијен у Јава програмском језику за комуникацију са релационим базама података [22]. Он омогућава Јава апликацијама да упућују упите према бази података користећи стандардизоване SQL наредбе, без обзира на конкретну базу података која се користи. Такође, обезбеђује драјвере који служе као мост између апликације и базе података, што доприноси преносивости кода. Омогућава извршавање CRUD операцијa (Create, Read, Update, Delete), управљање трансакцијама и многе друге операције које се односе на базе података. При томе, JDBC пружа могућност за ефикасно и сигурно управљање ресурсима, конекцијама са базом и грешкама које могу настати при комуникацији.

#### SessionFactory

*Hibernate* SessionFactory објекат је одговоран за конфигурисање интеракције са базом података. Основни задатак овог објекта је да управља конекцијама са базом, као и метаподацима неопходним за мапирање ентитета релационе базе података на доменске објекте апликације [8]. Такође, игра кључну улогу у креирању Session објеката којима се врши интеракција ка бази. Конструисање SessionFactory објекта је скупа операција, стога се, приликом покретања апликације, креира једна инстанца која се користи током њеног рада.

Традиционални приступ конфигурацији SessionFactory објекта представља декларативну конфигурацију, уз помоћ XML датотека. Ове датотеке садрже конекциони *Uniform Resource Locator* (URL) ка бази података, креденцијале приступа, информације о драјверу и дијалекту SQL језика базе података, метаподатке мапирања и подешавања механизма кеширања. *Hibernate*, при покретању, учитава конфигурационе датотеке како би конструисао SessionFactory објекат. Пример XML конфигурације приказан је на слици 4.5. Са друге стране, могуће је, уз помоћ *FluentAPI-ја*, у програму конструисати овај објекат употребом SessionFactoryBuilder објекта и уланчавањем позива метода којима се постављају претходно поменути параметри конфигурације. Пример овакве конфигурације дат је у листингу 4.9.

A screen shot of a computer code

Description automatically generatedСлика 4.5 XML конфигурација *Hibernate-a*

SessionFactory factory = new SessionFactoryBuilder()

.setProperty(„hibernate.dialect“,

„org.hibernate.dialect.MySQLDialect“)

.setProperty(„hibernate.connection.ur“,

„jdbc:mysql://localhost:3306/mydb“)

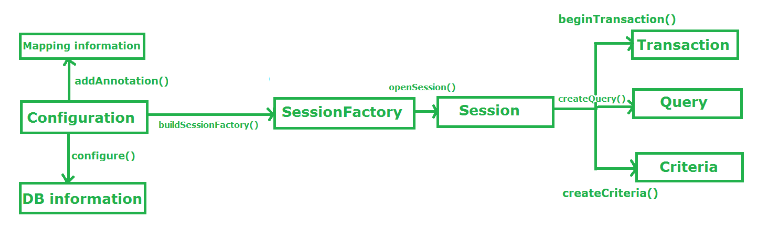
.configure()

.buildSessionFactory();

Листинг 4.9 Конфигурација *Hibernate-a* коришћењем *FluentAPI-ja*

#### Session

Сесија (енгл. Session) представља имплементацију Unit of Work обрасца за рад са базом података и обавија JDBC Connection интерфејс. Пружа интерфејс за извршавање CRUD операција над ентитетима, као и за припрему и извршавање упита [8]. Такође, управља механизмом кеширања ентитета који су учитани у току једне сесије. Поред тога, има кључну улогу у креирању трансакционих објеката и управља променама стања у животном циклусу ентитета у оквиру јединице рада.

[[5]](#footnote-5)Слика 4.6 Архитектура *Hibernate* алата

#### *EntityManager* у *SpringJPA*

EntityManager у Спринговој JPA имплементацији пружа интерфејс за управљање ентитетима који прати стандарде JPA спецификације [15, 16]. Представља омотач (eнгл. wrapper) oко *Hibernate* Session објекта, пружајући сличне могућности перзистенције и креирања упита. Такође, као што постоји SessionFactory објекат уз помоћ којег се управља конфигурацијом комуникације са базом података и креираjy објекти сесије, EntityManagerFactory је одговоран за креирање EntityManager oбјеката и управљање конфигурацијом. Такође, као и SessionFactory објекат, креира се једном, приликом покретања апликације, путем XML или програмске конфигурације.

Када је потребно, помоћу инстанце EntityManagerFactory класе, креира се EntityManager објекат, помоћу којег се врше операције приступа подацима у оквиру једне трансакције. Попут објекта сесије, представља контекст у оквиру којег се врши конверзија ентитета, задужен је за кеширање и управљање променом стања у оквиру животног циклуса ентитета у апликацији.

#### Животни циклус перзистенције ентитета

Апликација која има могућност трајног чувања података мора да комуницира са механизмом за перзистенцију, сваки пут када је неопходно да се тренутно стање пренесе из меморије у базу података, или обрнуто. Ово значи да је потребно позвати *Hibernate* интерфејсе како би се сачували или учитали подаци.

Приликом интеракције са објектно-релационим мапером, неопходно је да се води рачуна о стањима инстанце ентитета у односу на његову перзистенцију. Ово се назива **животни циклус перзистенције** и представља скуп стања кроз који прође инстанца ентитета током свог живота у апликацији [12]. Како апликација не би морала да се брине о праћењу промена стања, механизам перзистенције нуди услугу којом се памте све промене стања ентитета. Ово се назива **контекст перзистенције** [12]. Ентитети у *Hibernate* aлату пролазе кроз неколико различитих стања. Иницијално, након креирања, a пре повезивања са сесијом и базом, ентитет се налази у **транзијентном** стању (енгл. *Transient*). У овом стању, ентитет још увек није уписан у базу и не постоји његова идентификациона ознака. Када се ентитет сачува први пут, помоћу сесије, он прелази у **перзистентно** стање (енгл. *Persistent*) где постаје управљан од стране *Hibernate* сесије. Док је у перзистентном стању, све измене ентитета се прате и синхронизују са базом података. Taкође, ентитет може бити обрисан, што га ставља у **уклоњено** стање (енгл. *Removed*) при чему он више не постоји у бази. Након затварања сесије, ентитет постаје **одвојен** (енгл. Detached) од сесије али и даље садржи податке из базе, уколико није уништен из меморије. Објекат у одвојеном стању може поново да се припоји новој сесији у оквиру које може бити ажуриран или обрисан. Треба напоменути да, уколико дође до поништавања трансакције, ефекти синхронизације са базом података не морају бити трајни. Животни циклус перзистенције приказан је на слици 4.6.

A diagram of a computer

Description automatically generated[[6]](#footnote-6)Слика 4.7 Животни циклус перзистенције

### Мапирање ентитета

Како би апликација могла да користи перзистиране податке неопходно је, најпре, извршити мапирање класа ентитета на табеле у релационој бази података. У овом поглављу, детаљно ће бити дискутовано o начинима на који *Hibernate* учитава и чува инстанце класа доменског модела. Иако постоји неколико начина путем којих је могуће извршити ово мапирање, посебна пажња ће бити пружена на мапирање помоћу анотација. Анотације представљају специјална обележја у Јава коду које указују на то како треба интерпретирати одређене делове кода приликом приступа бази података.

#### Основе

Како би било могуће мапирати Јава класу на табелу у релационој бази података, постоји неколико анотација које треба узети у обзир [16]:

* **@Entity**: Ова анотација се користи како би се Јава класа обележила као ентитет, што указује да ће њене инстанце бити мапиране на табелу у бази података
* **@Table**: Користи се за дефинисање имена табеле на коју ће се ентитет мапирати. Уколико се ова анотација не користи, име Јава класе ће бити коришћено као име табеле
* **@Id**: Означава примарни кључ ентитета, при чему сваки ентитет мора имати једно поље обележено овом анотацијом
* **@Column**: Користи се за мапирање конкретног поља на колону у бази података. Такође, може се користити за дефинисање својстава колоне, као што су дужина, јединственост итд.

За успешно мапирање ентитета на табелу, неопходно је навести барем **@Entity** и **@Id** анотације. Пример мапирања приказан је у листингу 4.10

@Entity

@Table(name = “MyEntity”)

public class MyEntity {

@Id

@GeneratedValue

Private Long id;

@Column(name = “first\_name”)

private String firstName;

}

Листинг 4.10 Мапирање уз помоћ анотација

#### Композитни кључ

Композитни кључ је примарни кључ који се састоји из две или више колона у табели базе података. У Hibernate, композитни кључеви се могу креирати коришћењем посебним анотацијама [16]:

* **@Embeddable**: Ова анотација се користи да обележи Java класу која представља композитни кључ. Класа која је обележена овом анотацијом не сме имати своју табелу; уместо тога, њена поља представљају колоне у табели ентитета који је користи.
* **@EmbeddedId**: Ова анотација се користи у ентитетској класи да обележи атрибут који представља композитни кључ. Атрибут мора бити типа класе која је обележена @Embeddable анотацијом.

Пример ентитета са композитним кључем приказан је у листингу 4.11:

@Embeddable

public Class CompositeKey implements Serializable {

@Column(name = “first”)

private Long first;

@Column(name = “second”)

private Long second;

}

@Entity

@Table(name = “MyEntity”)

public class MyEntity {

@EmbeddedId

private CompositeKey id;

}

Листинг 4.11 Пример дефинисања композитног кључа

#### Мапирање асоцијација

Мапирање асоцијација *у Hibernate-у* означава процес повезивања двa или више ентитета. Ове везе могу бити једносмерне или двосмерне и обично одговарају релационим везама у бази података, као што су „један-на-један“, „један-на-више“ и „више-на-више“ [8, 15, 16].

Основне анотације које се користе за мапирање асоцијација укључују @OneToOne, @OneToMany, @ManyToOne и @ManyToMany. Ове анотације нуде различите конфигурације, укључујући и одређивање смера везе (једносмерна или двосмерна) и каскадне операције.

Уколико се два ентитета налазе у односу "један-на-један" или "један-на-више" конфигурација мапирања се врши на начин описан у листингу 4.12:

@Entity

public class Person {

@Id

private Long id;

@OneToOne

@JoinColumn(name = “document\_id”)

private Document document;

}

Листинг 4.12 Веза један-на-један између два ентитета

Са друге стране, уколико је однос између два ентитета "више-на-више", приступа се стандардном решењу у релационим базама података, креирању посебне табеле везе. Помоћу *Hibernate-a*, то се постиже на начин описан у листингу 4.13:

@Entity

public class Student {

@Id

private Long id;

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name =

"course\_id")

)

private Set<Course> courses = new HashSet<>();

}

Листинг 4.13 Мапирање односа више-на-више у *Hibernate-y*

Приликом мапирања асоцијација, често је потребно одредити начин на који се повезани ентитети учитавају. Мапирање може бити конфигурисано тако да користи *Lazy Loading* или *Еаger Loading* стратегијy добављања повезаних ентиета. То се постиже на следећи начин описан у листингу 4.14:

@OneToMany(fetch=FetchType.EAGER)

public Set<Order> getOrders() {

return orders;

}

Листинг 4.14 Конфигурација *Eager Loading* стратегије

Такође, битно је напоменути да је *Lazy Loading* подразумевани начин добављања повезаних ентитета.

### Извршавање упита

#### Hibernate Query Language (HQL)

За разлику од стандардног SQL језика који је прилагођен за релационе базе података, HQL је објектно-оријентисани језик за писање упита, специфично развијен за Hibernate. HQL oмогућава писање упита над ентитетима користећи доменске моделе и њихове атрибуте, чиме се пружа могућност писања интуитивнијих упита над базом података. Пример HQL упита приказан је у листингу 4.15

query = session.createQuery(„from User where id = :id“);

query.setLong(„id“, 3);

User u = (User) query.uniqueResult();

Листинг 4.15 Пример HQL упита

#### *Criteria Queries*

*CriteriaAPI* нуди могућност за програмско конструисање упита коришћењем Јава објеката и метода [8]. За разлику од HQL упитa који се конструишу помоћу стрингова, *Criteria* упити се генеришу скупом метода које нуди програмски интерфејс *CriteriaAPI*. Главна предност овог приступа је што *CriteriaAPI* омогућава динамичко креирање упита, као и могућност ранијег откривања грешака, током компилације. Ипак, битно је да се напомене да упити засновани на HQL стринговима и *Criteria*API програмским интерфејсом имају исту ефикасност и перформансе. Пример CriteriaAPI упита дат је у листингу 4.16.

CriteriaBuilder cb = session.getCriteriaBuilder();

CriteriaQuery<User> cr = cb.createQuery(User.class);

Root<User> root = cr.from(User.class);

cr.select(root)

Query<User> query = session.createQuery(cr);

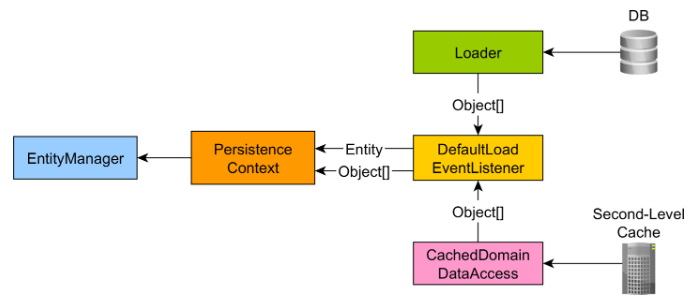
List<User> users = query.getResultList();

Листинг 4.16 Пример *CriteriaAPI* упита

#### Генерисање SQL наредбе

Без обзира на приступ упиту, *Hibernate* генерише SQL упит тако што парсира HQL или *Criteria*API упит и трансформише га у оптимизоване SQL исказе на основу метаподатака о мапирању ентитета и дијалекта базе података, који су енкапсулирани у SessionFactory објекту. Такође, на основу конфигурационих података, *Hibernate* може да оптимизује упит, узимајући у обзир информације о индексима, кеширању и другим факторима оптимизације. Резултујући SQL упит се прослеђује бази података, где се извршава.

#### Хидрација објекта

Хидрација објекта је процес у којем *Hibernate* преводи редове добијене из базе података у објекте Јава апликације (слика 4.8) [8]. Ово је виталан процес објектно-релационог мапирања и oтпочиње тако што се прикупе подаци генерисани упитом, који се најчешће налазе у табеларној форми, у оквиру JDBC скупа резултата (*ResultSet*). Затим, за сваки ред резултата, Hibernate интерпретира конфигурационе метаподатке како би утврдио који Јава објекат одговара датом реду и која поља тог објекта треба да се попуне вредностима реда. Уколико је објекат повезан са другим објектима, *Hibernate* ће индефиковати и иницијализовати ове асоцијације, уз могуће коришћење *Lazy Loading* стратегије. Једном када је објекат хидриран, он се ставља у контекст перзистенције у виду EntityEntry објекта којем је могуће пратити промене стања и синхронизовати га са базом података, по потреби.

[[7]](#footnote-7)Слика 4.8 Хидрација објекта

### Keширање

Кеширање је битан механизам за побољшање перформанси апликације засноване на перзистенцији података. У *Hibernate-у*, постоје два основна нивоа кеширања: кеш првог нивоа (*First Level Cache*) и кеш другог нивоа (*Second Level Cache*).

#### Кеш првог нивоа

Први ниво кеша је увек активан и не може се онемогућити. Он је асоциран са *Hibernate* објектом сесије и осигурава да се, у току комуникације, исти објекат не учитава више пута из базе података.

#### Кеш другог нивоа

Док је први ниво кеша везан за конкретну сесију, други ниво кеша је везан за SessionFactory и дељен је између свих сесија. Он може кеширати ентитете, колекције и упите.

За активирање кеша другог нивоа потребно је конфигурисати *Hibernate* да користи конкретан пружалац кеша, као што је *EhCache* или *Hazelcast*. Уколико се користи конфигурација помоћу XML датотеке, активирање кеша се може извршити на следећи начин (листинг 4.17):

<property name="hibernate.cache.use\_second\_level\_cache">

true

</property>

<property name="hibernate.cache.region.factory\_class">

org.hibernate.cache.ehcache.EhCacheRegionFactory

</property>

Листинг 4.17 Koнфигурација кеша другог нивоа у *Hibernate-y*

Када је конфигурисан, ентитети се могу кеширати користећи анотацију @Cache (листинг 4.18):

@Entity

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY)

public class Person

Листинг 4.18 Употреба кеша другог нивоа

Механизам кеширања у *Hibernate-у* може знатно побољшати перформансе апликације, али треба бити пажљив са његовом употребом. Неправилно конфигурисан кеш може довести до неконзистентних података или деградације перформанси.

## SQLAlchemy

SQLAlchemy је алат за објектно-релационо мапирање развијен за потребе апликација направљених у Пајтон програмском језику [3]. Oд свог настанка, 2005. године, тежи да пружи комплетан систем за рад са релационим базама података у Пајтону, ослањајући се на DB API програмски интерфејс за интеракцију са базама података. Због својих кључних карактеристика, попут способности за ефикасну обраду сложених SQL упита и имплементације обрасца јединице рада (*Unit of Work*), *SQLAlchemy* обезбеђује високо аутоматизован механизам за перзистенцију података и сматра се де факто стандардом за рад са релационим базама података у Пајтону.

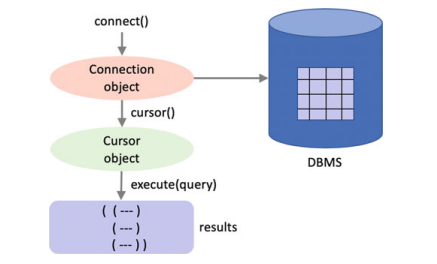
У наредном поглављу, детаљно ће се ући у имплементацију битних механизама за објектно-релационо мапирање у SQLAlchemy алату, почевши са прегледом његове архитектуре и механизмима за праћење стање ентитета. Након тога, биће пружен увид у начине путем који се врши мапирање класа ентитета на табеле релационе базе података. Такође, детаљно ће се разматрати процес извршавања упита над базом података, од декларисања до креирања објеката над добављеним подацима. Поглавље се завршава, кратким увидом у аспекте и механизме којима се долази до побољшања перформанси упита извршаваних над базом података.

### Архитектура алата

#### DBAPI

*Python Database API*, или DB API скраћено, представља стандардизовани интерфејс за начине којима Пајтон модули интерагују са базама података [23]. DB API спецификација пружа скуп метода и конвенција којих се драјвери база података морају придржавати како би били у складу са DB API стандардом. Својом апстракцијом приступа базама података, DB API има за циљ да обезбеди конзистентни интерфејс за остваривање конекције са базом, извршавање упита, као и руковање са добављеним подацима, чиме се омогућава aпликацијама развијеним у Пајтону да лако комуницирају са било којим системима за управљање базама података.

Постоји неколико кључних елемената у DB API-jу:

* **Конекциона функција** - Помоћу које се остварује иницијално повезивање са базом података. Као повратну вредност враћа објекат конекције (Connection).
* **Објекат конекције** - Омогућава приступ бази података. Основни задатак објекта конекције је да пружи могућност слања упита, што се остварује кроз креирање посебних курсорских објеката (Cursor) oбјеката. Такође, помоћу њега се управља трансакцијама, кроз commit() и rollback() методе, и затвара конекција са базом података.
* **[[8]](#footnote-8)Курсорски објекат** - Представља контекст у оквиру којег се извршавају упити ка бази података и добављају подаци који су резултат извршавања. Eнкапсулира интеракцију за базом података и апстракцију курсора базе података. Уз помоћ метода execute() и executemany() пружа могућност извршавања упита, као и управљање подацима добављеним подацима у скупу резултата кроз методе fetchone(), fetchmany() и fetchall() . Након што се заврши рад са базом података, неопходно је затворити курсоре како би се могли потврдити или понишитити ефекти трансакција.

Слика 4.9 Елементи DBAPI програмског интерфејса

#### *Engine*

Engine je oсновна компонента архитектуре *SQLAlchemy* aлата, која представља капију између *SQLAlchemy-ја* и саме базе података [10]. Задатак ове компоненте је да управља повезивањем са базом података, користећи одговарајући DBAPI драјвер и да одржава базен конекција (енгл. *connection pool*) са циљем остваривања ефикасне комуникације са базом података током животног века апликације. Поред тога, нуди интерфејс којим се енкапсулирају разлике специфичне за дијалекте база података, услед којих је могуће вршити комуникацију са различитим системима за управљање базама података на јединствен начин. Такође, чува метаподатке о шеми базе података и мапирањима између табела базе и класа апликације. Руковањем свим овим одговорностима, Engine ствара основу за ефикасно функционисање *SQLAlchemy-ja*.

Како би се креирао *Engine* објекат, неопходно је проследити конекциони стринг, који садржи све информације неопходне за повезивање са базом података. Битно је напоменути да се, приликом инстанцирања овог објекта, конекција са базом података не отвара одмах, већ по експлицитном позиву методе connect() прописаном путем DB API стандарда.

Tипичан облик конекционог стринга, прописан је RFC-1738 стандардом, и приказан је у листингу 4.19:

dialect+driver://username:password@host:port/database

Листинг 4.19 Пример конекционог стринга

A diagram of a pool and a dialogue

Description automatically generated[[9]](#footnote-9)Слика 4.10 Архитектура Engine објекта

Информације о дијалекту специфичном за повезану базу података, енкапсулиране су унутар Dialect објекта чија референцa се налази унутар Engine oбјекта, док се управљање над базеном конекција врши кроз посебан Pool објекат.

#### Session

За разлику од *Engine* објекта који упоставља комуникацију са базом података и омогућава мапирање између класа и табела, кроз *Session* објекат се реализује сва комуникација са базом података и он представља зону надзора (енгл. *holding zone*) за све учитане објекте, којима прати и управља променама у стањима животног циклуса [14]. Као и у одговарајућим примерима претходно наведених објектно-релационих мапера, *Session* објекат представља имплементацију обрасца јединице рада (*Unit of Work*). Такође, *Session* представља и имплементацију *Identity Map* шаблона, чиме се спречава учитавање редундантних објеката у меморију.

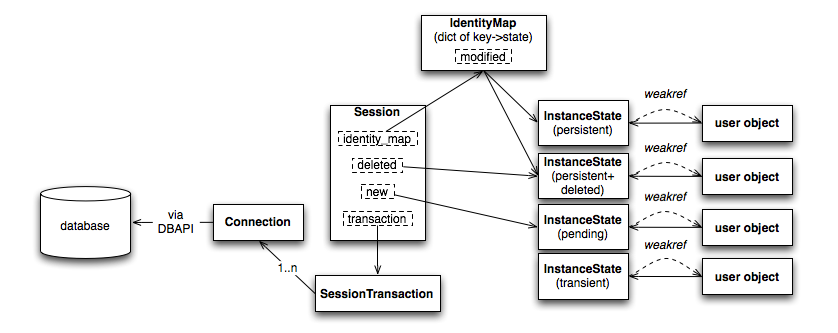
Приликом инстанцирања, објекат сесије је без стања (енгл. *stateless*), што значи да још није асоциран ни са једном конекцијом ка бази података. Приликом слања упита, сесија, од асоцираног Engine објекта, добија ресурс конекције уз помоћ ког се отпочиње трансакција ка бази података. Ефекти трансакције се затим, на експлицитан захтев, кроз посредство сесије, потврђују или поништавају.

Приликом било које измене над објектима који се налазе у сесији, генерише се догађај промене који се записује у сесији. Ове измене су енкапсулиране у оквиру InstanceState објекта, чија се референца налази у оквиру објекта сесије, са циљем заштите модификованих или обрисаних објеката од garbage collection механизма. Уколико се шаље упит ка бази података, или треба да се изврши потврда трансакције, објекат сесије, најпре, "почисти" (енгл. *flush*) све измене записане у меморији и прослеђује их ка бази података ради синхронизације. Овај процес синхронизације се врши кроз неколико корака:

* **Преглед промена**: Сесија прво прегледа све објекте које прати у зони надзора и идентификује све промене које треба да примени на базу података
* **Формирање SQL наредби**: За сваку детектовану промену, сесија формира одговарајућу SQL наредбу. На пример, уколико је нови објекат додат у сесију, генерисаће се INSERT наредба
* **Извршавање наредби**: Једном када су све SQL наредбе формиране, сеисја их извршава над базом података у одговарајућем редоследу

Важно је напоменути да, иако објекат сесије синхронизује промене у бази података, то не значи да су промене трајне, јер оне могу бити опозване уколико се поништи трансакција у оквиру које су ове измене начињене.

Објекте, које прати сесија, могуће је откачити (енгл. *detach*) од сесије, чиме се процес аутоматске синхронизације за њих обуставља. Након што се откаче, могуће их је и даље користити у оквиру апликације, с тим што се промене над њима неће рефлектовати у бази података. Уколико има потребе за поновном синхронизацијом са базом података, могуће их је поново повезати са новом или постојећом сесијом, чиме њихово стање поново бива одражено у бази кроз објекат сесије.

[[10]](#footnote-10)Слика 4.11 Илустрација објекта сесије

#### Животни циклус ентитета

Kao што је већ наведено, објекти ентитета могу проћи кроз неколико стања током свог животног циклуса у апликацији [14]. Основна стања у којима објекат може бити су:

* **Transient**: Представља инстанцу која се не налази у сесији и још није сачувана у бази података. Једина веза коју објекат у транзијентном стању има са ОРМ-ом је путем метаподатака који повезују његову класу са одговарајућом табелом у бази.
* **Pending**: Након што се инстанца у транзијентном стању дода у сесију, она прелази у стање чекања, чиме се означава да ће, приликом следећег позива *flush* механизма, она бити уписана у базу података.
* **Persistent**: Инстанца ентитета у перзистентном стању присутна је у сесији и за њу постоји одговарајући запис у бази података. Ентитет може прећи из стања чекања у перзистентно стање, уколико се његове измене синхронизују са базом података, позивом flush() методе. Такође, сви ентитети добављени упитима над базом података су иницијално у перзистентном стању.
* **Deleted**: Обрисано стање ентитета означава да се захтев за брисање инстанце у апликацији синхронизовао са базом података, али ефекти трансакције и даље нису потврђени. Уколико дође до потврде трансакције, објекат из обрисаног, прелази у одвојено (енгл. *detached*) стање, чиме се више не налази у сесији. Са друге стране, уколико дође поништавања ефеката трансакције, објекат се враћа из обрисаног у перзистентно стање.
* **Detached**: Представља инстанцу која се односи, или се односила, на запис у бази података, али тренутно није везана ни за једну сесију, што означава да се промене над њеним стањем неће рефлектовати у бази података, процесом синхронизације.

### Мапирање ентитета

Приликом отпочињања рада са *SQLAlchemy* објектно-релационим мапером, неопходно је, најпре, извршити мапирање класа, дефинисаних у апликацији, на табеле у релационој бази података [10]. Поред тога, потребно је дефинисати мапирање веза између апликативних класа на везе између табела. Овим се формирају метаподаци који се чувају у оквиру Engine објекта којима се омогућава ефикасно претварање из инстанци класа ентитета у записе у табелама базе података, и обрнуто.

#### Класично мапирање

Класично мапирање, познато и као императивно мапирање, у *SQLAlchemy* алату се односи на мапирање већ постојећих класа у апликацији на постојеће табеле у релационој бази података [10]. Ово значи да у *SQLAlchemy* сматра да постоје два одвојена објекта: Table објекат, који енкапсулира метаподатке о структури табеле у бази, и, у апликацији дефинисан, објекат ентитета класе, који се повезују помоћу посебне map\_imperatively методе mapper\_registry модула. Након овог повезивања, објекат ентитета класе добија нове атрибуте, кореспондентне колонама у табели (листинг 4.20):

user\_table = Table(

"user",

mapper\_registry.metadata,

Column("id", Integer, primary\_key=True),

Column("username", String(50)),

)

class User:

pass

mapper\_registry.map\_imperatively(User, user\_table)

Листинг 4.20 Класично мапирање у *SQLAlchemy-jy*

Такође, mapper функција има могућност и додавања посебних атрибута у кориснички дефинисаној класи, који су неопходни за мапирање веза између објеката. Овакав поступак се у Пајтон свету назива инструментација класа (енгл. *class instrumentation*).

Ипак, битно је напоменути да је класично мапирање застарео начин повезивања класа и табела и није више препоручен начин употребе *SQLAlchemy-ја*.

#### Декларативно мапирање

За разлику од класичног мапирања, модерна употреба *SQLAlchemy-ја* за мапирање ентитета заснива се на декларативном мапирању, налик оном у *Hibernate* алату [14]. У оваком начину мапирања, крајњи корисник експлицитно дефинише атрибуте класа који ће се мапирати на одговарајућу табелу. Типичан начин коришћења је да се најпре дефинише DeclarativeBase наткласа, која омогућава употребу декларативних процеса у свим доменским класама које је наследе. У листингу 4.21, приказан је типичан начин декларативног дефинисања мапирања.

class Base(DeclarativeBase):

pass

class User(Base):

\_\_tablename\_\_ = "user"

id: Mapped[int] = mapped\_column(primary\_key=True)

username: Mapped[str]

Листинг 4.21 Декларативно мапирање у *SQLAlchemy-jy*

DeclarativeBase класа садржи референцу на registry објекат који садржи метаподатке о табелама у релационој бази и њиховом мапирању на класе.

#### Aнатомија мапирања

На слици 4.11, приказан је механизам мапирања који је састављен од два слоја интеракције између кориснички дефинисаних класа и метаподатака табела који су повезани [14]. Леви слој представља претходно поменуту инструментацију класа, док су функционалности везане за базу података и мапирање представљене десним слојем

У склопу инструментације класа, компонента ClassManager повезана је са кориснички дефинисаном класом која се мапира и садржи колекцију InstrumentedAttribute oбјеката повезаних са мапираним атрибутима мапиране класе.

A diagram of a company

Description automatically generated.[[11]](#footnote-11)Слика 4.12 Анатомија мапирања у *SQLAlchemy-jy*

У десном слоју, слоју мапирања, компонента Mapper представља везу између кориснички деифнисане класе и претраживе јединице (енгл. *selectable unit*). У највећем броју случајева је то табела релационе базе, мада може бити извршено и мапирање кориснчки дефинисане класе на поглед базе података. Мapper у себи садржи колекцију MapperProperty објеката који се брину о SQL репрезентацији кореспондентних атрибута кориснички дефинисаних класа. ColumnProperty представља варијанту претходно наведеног MapperProperty објекта и представља мапирање извршено са атрибута мапиране класе на колону табеле базе података. Са друге стране, RelationshipProperty је такође варијанта MapperProperty објекта и представља везу ка другом маперу, што осликава везе између табела.

RelationshipРroperty сваком атрибуту делегира одговарајуће понашање учитавања (енгл. *loading behaviour*), чиме се назначава која се стратегија учитавања асоцираних објеката користи. Ова информација је енкапсулирана у склопу LoaderStrategу објекта и могуће стратегије су: *lazy* (одложено), *eager* (рано) или *immeadiate* (непосредно) учитавање. Подразумевана стратегија учитавања се одређује током конфигурације мапера, уз могућност коришћења алтернативних опција током извршавања упита. Takoђе, у склопу RelationshipProperty oбјекта постоји референца на DependencyProcessor компоненту која управља асоцијацијама током процеса синхронизације, односно позива flush() методе.

Приликом креирања Mapper објеката и одговарајућих RelationshipProperty објеката, формира се структура графа, у којој су Mapper објекти чворови, а ивице су RelationshipProperty објекти. Овакав граф је кључна структура за функционисање SQLAlchemy алата, зато што игра битну улогу у операцијама које захтевају каскадну пропагацију измена ка повезаним ентитетима, приликом раног учитавања (енгл. *Eager Loading*), као и током процеса синхронизације објеката који у себи садрже асоцијације.

### Извршавање упита

#### Query API

*Query API* у *SQLAlchemy* алату представља динамични програмски интерфејс, који омогућава програмерима да једноставно и ефективно креирају и извршавају упите над базом података [10]. Пружа мост између програмских конструкција високог нивоа и SQL наредби, чиме се ствара интуитиван интерфејс за комуникацију са базом података. У овом поглављу ће детаљно бити појашњен начин на који *Query API* функционише, почевши од декларисања упита до конструкције објеката у апликацији.

У основи *Query* програмског интерфејса је способност да се креирају упити. Ово се обично изводи кроз објекат сесије (Session). Kao што је већ наведено, сесија у *SQLAlchemy* је главна тачка интеракције апликације са базом података.

Једном када је основни упит креиран, *Query* програмски интерфејс нуди различите методе за специфицирање жељеног упита ка бази података, попут филтрирања, омогућено filter() методом, у којој се наводи услов по којем се врши филтрирање, затим сортирање, што се ради уз помоћ order\_by() методе, при чему је могуће навести по којем атрибуту се врши сортирање и у којем редоследу. Поред тога, могуће је вршити и груписање на основу специфицираних поља кроз методу group\_by(). Пример *QueryAPI* упита, приказан је у листингу 4.22

query = session

.query(orders.c.customer\_id)

.filter(orders.c.quantity > 10)

.order\_by(orders.c.date\_ordered)

Листинг 4.22 *Query* API упит

Овако изграђен упит се извршава позивом метода .all(), којом се враћа листа свих резултата, .first(), којом се враћа први елемент у листи резултата или .one(), чиме се враћа тачно један резултат или изазива изузетак.

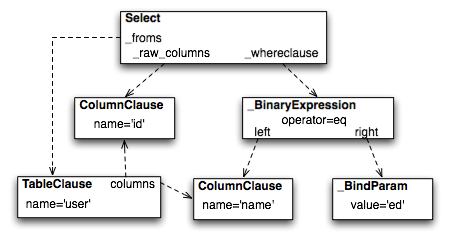
#### Изразно стабло

Као што је наведено у претходном поглављу, *Query* програмски интерфејс омогућава коришћење Пајтон објеката како би се представили делови SQL израза, који се, затим, комбинују како би се креирао еквивалентан *QueryАРI* израз. Oвако креирани изрази, могу бити представљени помоћу изразних стабала (енгл. expression trees).

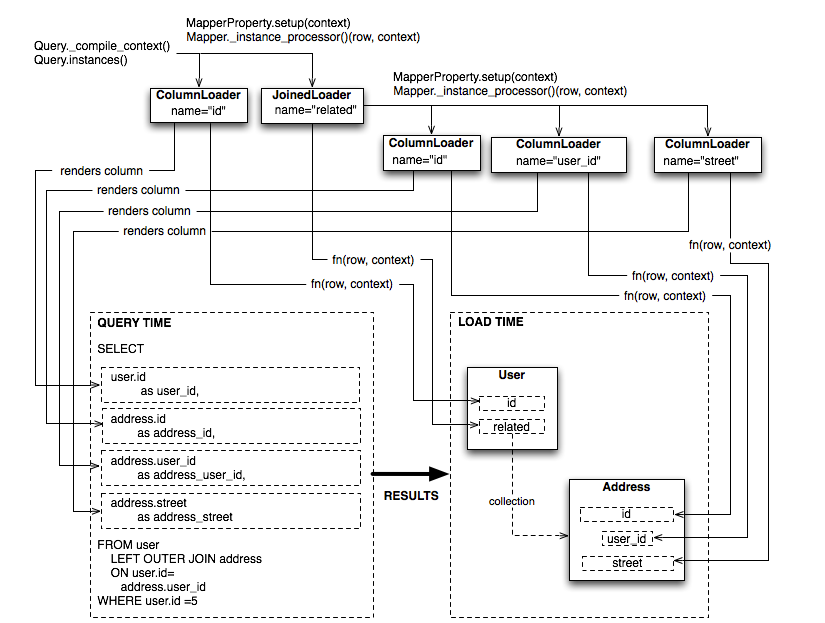
Сваки *Query* објекат, у корену изразног стабла које га представља, садржи SELECT чвор, који представља почетак одговарајуће SQL наредбе. Како се позивају друге *Query* методе, попут filter() или order\_by(), тако се изразно стабло попуњава новим чворовима са одговарајућим деловима SQL наредбе. Другим речима, позивањем метода над *Query* објектом, инкрементално се додају нови чворови у изразно стабло. Такође, уколико су у упиту специфициране операције спајања више ентитета (јоin) или је као подразумевана стратегија изабрано рано учитавање, генерисање изразног стабла се ослања и на граф Mapper објеката, поменутих у поглављу **4.3.2.3**. Приликом позива извршавања *Query* објекта, овако креирано изразно стабло се прослеђује *SQLAlchemy* компајлеру, који га преводи у одговарајући SQL израз, са постављеним параметрима упита, уз обраћање пажње на дијалект система за управљање базом података са којим се врши комуникација. Процес превођења се назива **фаза рендеровања** SQL израза (енгл. *SQL rendering*).

query = session.query(User.id).filter(User.name == 'ed')

Листинг 4.23 *SQLAlchemy* упит који се изражава путем изразног стабла

Израз задат у листингу 4.23 представљен је изразним стаблом на слици 4.12.

Слика 4.13 Пример изразног стабла *Query* API израза

[[12]](#footnote-12)Након што се изгенерише SQL упит и изврши над базом података, записи из базе података, захваћени упитом, се морају претворити у доменске објекте апликације. Овај процес се назива **фаза учитавања** података (енгл. *data loading*) и ослања се на рекурзивни обилазак структуре графа, формиране током фазе мапирања ентитета, при чему се узимају у обзир сви MapperProperty oбјекти и одлучује се да ли је атрибут, који је представљен овим објектом, неопходно иницијализовати подацима из базе. Ово се врши уз посредство LoaderStrategy објеката. Након што се фаза учитавања заврши, објекти су спремни за обраду у апликацији.

Слика 4.14 Фаза рендеровања и фаза учитавања

### Остале кључне особине

#### Јединица рада и тополошко сортирање

A diagram of a flowchart

Description automatically generated[[13]](#footnote-13)Као што је већ наведено у претходним поглављима, објекат сесије у *SQLAlchemy-jy* представља имплементацију обрасца јединице рада (*Unit of Work*), са циљем управљања променама над ентитетима. Како би се извршила синхронизација свих измена са базом података, неопходно је извршити наредбе перзистенције (DML наредбе) над базом. С обзиром на то да ентитети могу бити повезани, као и да промена над једним ентитетом мора да се пропагира на повезане ентитете, неопходно је наредбе перзистенције сортирати како не би дошло до нарушавања интегритета базе података [14]. Као пример, може се узети брисање ентитета који је референциран путем ограничења страног кључа. Неопходно је прво, или обрисати све повезане ентитете, или извршити модификацију над њима, како се не би нарушило ограничење страног кључа.

Слика 4.15 Тополошки сортиране DML наредбе

Из тог разлога, наредбе се смештају у граф зависности (слика 4.14) који се, затим, тополошки сортира како би се произвео правилан редослед извршавања. Сортирање се врши на основу претходно конструисаног графа зависности између Mapper објеката. Након што се све наредбе сортирају у правилном тополошком редоследу, оне бивају извршене над базом података како би се стања апликације и базе синхронизовали.

#### Систем догађаја

Систем догађаја (енгл. *Event System*) *SQLAlchemy* алата омогућава корисницима да креирају посебне Пајтон функције које су пријављене (енгл. *subscribe*) на oдређене догађаје у систему, чиме се омогућава да се одговарајући код изврашава током кључних фаза животног циклуса ентитета или операције [10]. Овакве функције се називају слушаоци (енгл. *Listeners*). Tипични примери догађаја су прелази ентитета из једног стања у друго, учитавање података или синхронизација. Уопштено, систем догађаја омогућава увид у интерне механизме *SQLAlchemy-ja* и подржава проширивост и подешавање алата потребама апликације.

У претходном поглављу изложени су *Entity Framework*, *Hibernate* и *SQLAlchemy* алати за објектно-релационо мапирање. Кроз детаљну анализу стечено је разумевање о основама ових алата, њиховим битним карактеристикама и начинима на које се могу интегрисати у различите системе. С обзиром на чињеницу да је ефикасност кључна карактеристика за сваки софтверски производ, неопходно је разумети како ови алати функционишу ради остварења оптималних перформанси апликација заснованих на објектно-релационим маперима као механизмима за перзистенцију података.

Из наведеног разлога, у наредном поглављу биће приказано темељно истраживање о утицају коришћења ових алата на перформансе апликација и пружен одговор на питање које се често појављује у различитим дискусијама међу програмерима: Да ли је коришћење објектно-релационих мапера анти-шаблон?

# 

# Aнализа перформанси ОРМ алата

У претходном поглављу, детаљно је описана имплементација и начин рада одабраних објектно-релационих мапера. У овом поглављу пажња ће бити усмерена на анализу њиховог утицаја на перформансе апликација.

Истраживање ће бити спроведено на унапред дефинисаном скупу SQL упита, при чему ће се мерити време извршавања истих упита коришћењем чистог SQL приступа, без посредовања OРМ алата и поредиће се са временом извршавања када се исти упити пошаљу кроз објектно-релациони мапер. Резултати ће показати да ли постоји значајна разлика у перформансама између ова два приступа.

Поред тога, биће дат критички осврт на предности и мане сваког појединачног алата. На крају ће бити изведен генералан закључак о утицају ОРМ алата на перформансе апликација.

## Тестно окружење

### Хардверске и софтверске спецификације

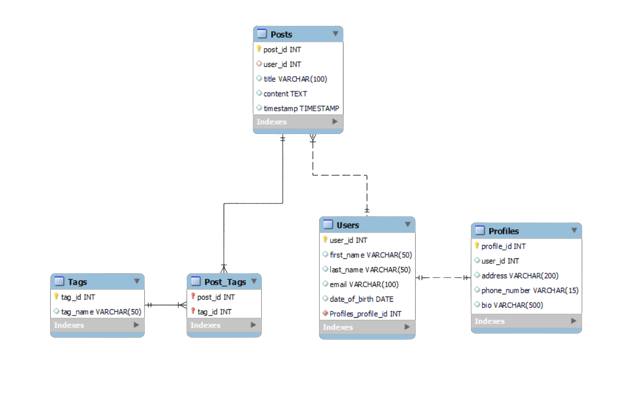
За потребе истраживања, коришћен је рачунар са 16GB RАМ меморије, AMD Ryzen 7 процесором од 16 језгара на 3.2GHz фреквенције и Samsung PM991A ССД диском од 512GB и оперативним системом Windows 10 Pro 64-bit. За систем за управљање базом података одабран је MySQL верзије 8.0.34 због своје доступности и једноставности за коришћење и креирање базе података, као и због своје широке примене у развоју модерних информационих система [17]. Тестиране су следеће верзије наведених објектно-релационих мапера: *Entity Framework Core* 7.0.9, *Hibernate* 6.2.6 и *SQLAlchemy* 1.4.49 [1,2,3]. Хардверска и софтверска конфигурација тестног окружења је пажљиво одабрана како би се обезбедили релевантни и упоредиви резултати перформанси изабраних ОРМ алата.

### Шема базе података

За потребе тестирања наведених алата, коришћена је шема базе података друштвене мреже, приказана на слици 5.1. Основни ентитети који чине шему су: Корисник, Профил, Објава и Ознака.

* **Корисник** (User) - представља особу која користи систем или платформу. У основи, то је налог преко кога особа приступа и користи различите услуге или функције платформе.
* **Профил** (Profile) - садржи личне и додатне податке о кориснику. То могу бити информације попут слике профила, контактних информација, биографије и других детаља који помажу у идентификацији или представљању корисника на платформи. У овој табели се налази страни кључ *user\_id* који директно повезује профил са конкретним корисником.
* Сваки Корисник има један и само један Профил који садржи додатне информације о њему. Профил је повезан са корисником преко страног кључа *user\_id*. Ова веза један према један осигурава да сваки корисник има јединствен профил на платформи.
* **Објава** (Post) - представља текстуални садржај који корисник дели на платформи. Свака објава има временску ознаку (енгл. *timestamp*) која указује на тачно време када је објава креирана или постављена. Ова временска ознака помаже у праћењу и сортирању објава на хронолошкој основи.
* Један корисник може да има више објава, док једна објава може бити везана за једног и само једног корисника.
* **Ознака** (Таg) - представља кључну реч или ознаку која помаже у категоризацији и претрази садржаја. Тагови омогућавају корисницима да брзо пронађу сродне објаве или чланке на основу интересовања или теме.
* **Објава\_Oзнака** (Post\_Tag) - представља везу између објава и одговарајућих тагова. Ова табела омогућава свакој објави да има више тагова и, истовремено, сваки таг може бити придружен више објава. Ова веза помаже у ефикасној организацији и претрази садржаја на платформи.

Разлог због ког је одабрана ова шема базе података је да се омогући увид у механизме објектно релационих мапера за мапирање веза између ентита шеме базе података и веза између објеката у одговарајућим програмским језицима. Поред тога, осликава шему базе

Такође, услед постојања веза између ентитета, ствара се могућност тестирања перформанси алата у реализацији комплексних упита.

Слика 5.1 Шема базе података коришћење у тестирању перформанси

Табеле Корисник и Профил, попуњене су са по 80 хиљада насумичних података, при чему је акценат био на поштовању задатих ограничења шеме базе података. Табела Објава садржи 200 хиљада насумичних података, док су табеле Ознака и Ознака\_Објава попуњене са по 100 хиљада насумичних података. Посебна пажња је обраћена да подаци буду разноврсно распоређени, како би се створили реални услови тестирања. Ово се односи на чињеницу да поједини корисници платформе могу имати више објава везаних за своје налоге, док неки корисници уопште немају објаве. Такође, поједине објаве имају неколико ознака, док неке немају ниједну.

Подаци неопходни за тестирање генерисани су помоћу сачуваних процедура (енгл. *stored procedures*) MySQL-овог *SQL/Persistent Stored Modules* jeзика, који представља екстензију SQL језика.

## Тестирање

### Поставка

У циљу анализе и поређења перформанси изабраних алата за објектно-релационо мапирање, извршено је тестирање брзине извршавања упита. Перформансе изабраних алата поређене су са перформансама извршавања чистих SQL упита задатих кроз апликацију путем одговарајућих програмских интерфејса за повезивање са базама података. За *Entity Framework* поређење се врши са извршавањем упита путем ADO.NET *SqlClient-a* у С# програмском језику. Времена добијена извршавањем *Hibernate* алата пореде се са онима добијеним коришћењем JDBC интерфејса у Јави, док се перформансе *SQLAlchemy-ja* упоређују са временима добијеним коришћењем MySQL Connector/Python интерфејса, базираног на DBAPI спецификацији [23].

Тестирање је извршено дефинисањем 10 различитих упита над базом података, при чему је сваки упит покретан 100 пута како би се ублажио утицај случајних фактора, попут тренутног оптерећења система на којем се врши тестирање и како би се добила прецизнија статистика о просечном времену извршавања за сваки упит. На овај начин омогућено је прецизно и реалистично поређење брзина извршавања између различитих алата за објектно-релационо мапирање.

### Упити

Како би се испитао утицај ОРМ алата на перформансе различитих операција над базом података, одабран је репрезентативан скуп SQL упита који покривају типичне сценарије приступа и манипулације над подацима. Скуп упита чине једноставни и сложени SELECT упити, упити са JOIN клаузулама, агрегација и сортирање резултата применом GROUP BY односно ОRDER BY клаузула, као и DML наредбе за манипулацију над подацима, које укључују INSERT, UPDATE и DELETE операције. У табели 5.1 приказан је скуп за скуп упита, као и њихове одговарајуће SQL наредбе.

|  |  |
| --- | --- |
| Упит | SQL наредба |
| Q1: Пронаћи кориснике рођене после задате године. | SELECT \*  FROM User  WHERE date\_of\_birth >= @param |
| Q2: Одредити укупан број објава за корисника са задатом имејл адресом | SELECT u.id, COUNT(p.id)  FROM User as u  JOIN Post as p  ON u.id = p.user\_id  WHERE u.email = @param  GROUP BY u.id |
| Q3: Пронаћи топ 5 корисника по броју објава. | SELECT u.name,  COUNT(p.id) as total\_posts  FROM User as u, Post as p  WHERE u.id = p.user\_id  GROUP BY u.id  ORDER BY total\_posts  LIMIT 5 |
| Q4: Добавити све кориснике који у биографији имају кључну реч | SELECT u.id, u.name  FROM User u, Profile p  WHERE u.id = p.user\_id  AND p.bio LIKE ’%keyword%’ |
| Q5: Пронаћи све различите ознаке за корисника са задатом имејл адресом. | SELECT DISTINCT t.name  FROM User as u, Post as p,  Tag as t, Post\_Tag as pt  WHERE u.id = p.user\_id  AND p.id = pt.post\_id  AND t.id = pt.tag\_id  AND u.email = @param |
| Q6: Пронаћи све ознаке које се налазе на 5 или више објава | SELECT t.name, COUNT(p.id) as total\_posts  FROM Post as p, Tag as t, Post\_Tag as pt  WHERE p.id = pt.post\_id AND t.id = pt.tag\_id  GROUP BY t.name  HAVING total\_posts >= 5 |
| Q7: Пронаћи укупан број ознака за сваког корисника | SELECT p.user\_id, COUNT(pt.tag\_id) as total\_tags  FROM Post as p,Post\_Tag as pt  WHERE p.id = pt.post\_id  GROUP BY p.user\_id |
| Q8: Додати новог корисника | INSERT INTO User  VALUES (@first\_name, @last\_name, @email, @date\_of\_birth) |
| Q9: Обрисати све профиле корисника чији број телефона почиње са задатим цифрама | DELETE Profile FROM Profile  WHERE Profile.phone\_number  LIKE ’prefix%’ |
| Q10: Изменити све објаве које потичу од задатог корисника | UPDATE Post  SET content = @param  WHERE user\_id = @param |

Табела 5.1 Упити коришћени у тестирању

Скуп упита, приказан у табели 5.1, је одабран из разлога што покрива све типичне случајеве коришћења приликом интеракције са базом података. Такође, састоји се из једноставних упита, који се односе на само једну табелу, али и из упита који захтевају спајање неколико табела и операције агрегације, чиме се постиже разноврсност и стварају добре основе за веродостојност тестова.

### Анализа добијених резултата

Упоредна евалуација перформанси *Entity Framework-a* и *SqlClient-a* приказана графички је приказана на дијаграму **5.1**.

Дијаграм 5.1 Упоређивање перформанси EF-a и SqlClient-a

Може се уочити да је *SqlClient* пружио боље перформансе у смислу брзине извршавања, иако су поједини упити имали приближно време извршавања. Примећује се да за упите који захтевају комбиновање операција агрегације и спајања више табела, конкретно упите **Q6** и **Q7**, EF пружа значајно лошије перформансе, што се дешава због начина на који је ЕF превео основни LINQ израз. У листингу **5.1** приказан је SQL израз који је EF генерисао приликом извршавања упита **Q7**.

SELECT `p`.`user\_id` AS `Id`, (

SELECT COUNT(\*)

FROM `Post\_Tags` AS `p0`

INNER JOIN `Tags` AS `t` ON `p0`.`tag\_id`=`t`.`tag\_id`

WHERE `p`.`post\_id` = `p0`.`post\_id`) AS `Count`

FROM `Posts` AS `p`

GROUP BY `p`.`user\_id`

Листинг 5.1 Комплексан SQL израз генерисан од стране *Entity Framework-a*

A screenshot of a diagram

Description automatically generatedПримећује се разлика између, иницијално постављеног, упита **Q7** у табели **5.1** и упита који је ЕF изгенерисао, с обзиром на то да садржи угњеждени упит који представља додатно оптерећење на перформансе апликације. Стога, како би се проверио утицај овакве конструкције упита на перформансе апликације, извршено је профилисање, иницијално постављеног упита **Q7**, и упита генерисаног од стране EF алата, приказаног у листингу 5.1. На слици 5.2 приказана је визуелизација оба упита применом наредбе EXPLAIN *MySQL Workbench* алата за рад са *MySQL-ом*.

Слика 5.2 Визуелизација извршавања упита Q7

Као главна метрика за ефикасност извршаваног упита над MySQL системом за управљање базом података узима се вредност параметра цене (енгл. *cost*), која назначава колико су наредбе упита скупе за сервер базе података. На слици 5.2 се види да је цена упита Q7 генерисаног помоћу EF алата скоро 2 пута већа него оригиналан упит.

Перформансе *Hibernate-a*, приказане на дијаграму 5.2приближније су онима добијеним извршавањем упита уз помоћ JDBC програмског интерфејса, уз неколико одступања.

Дијаграм 5.2 Упоређивање перформанси *Hibernate-a* и JDBC интерфејса

Ипак, уочава се да су времена извршавања упита **Q1** и **Q3** уз употребу *Hibernate-a*, скоро три пута дужа у односу на времена извршавања JDBC интерфејса. Профилисањем *Hibernate-a* приликом извршавања упита **Q1,** примећује се да је генерисан SQL израз идентичан као и, иницијално постављени упит **Q1.** Анализом статистике коју пружа *Hibernate*, увиђа се да време неопходно да се HQL упит припреми, односно преведе у одговарајући SQL израз, износи 0.18ms, време потребно да се изврши упит, односно да се пошаље ка бази података, изврши на серверу базе података, и резултати пошаљу назад, износи 41.8ms, док је преостало време утрошено на процес хидрације објеката, што говори да, уколико је упит адекватно преведен, инстанцирање објеката је најзахтевнија операција. Резултати профилисања приказани су у листингу 5.2

25300 nanoseconds spent acquiring 1 JDBC connections;

0 nanoseconds spent releasing 0 JDBC connections;

183800 nanoseconds spent preparing 1 JDBC statements;

41777100 nanoseconds spent executing 1 JDBC statements;

Листинг 5.2 Профилисање *Hibernate* алата

Са друге стране, профилисање упита **Q3** указује на чињеницу да упит који садржи операције агрегације и спајања, неће увек бити преведен у адекватан SQL израз, што, као и у случају *Entity Framework-a* изазива погоршање времена извршења. У листингу 5.3 приказан је изгенерисан SQL упит Q3 уз помоћ *Hibernate* алата.

select u1\_0.user\_id,u1\_0.email,

(select count(1) from Posts p2\_0

where u1\_0.user\_id=p2\_0.user\_id)

from Users u1\_0 join Posts p1\_0

on u1\_0.user\_id = p1\_0.user\_id

group by u1\_0.user\_id

order by count(p1\_0.post\_id) desc limit 5

Листинг 5.3 Упит **Q3** генерисан од стране *Hibernate-a*

Анализом резултата, приказаних у дијаграму 5.3, добијених извршавањем упита употребом *SQLAlchemy* алата, уочено је неколико ствари.

Дијаграм 5.3 Упоређивање *SQLAlchemy-a* и pymysql DBAPI интерфејса

Најпре, логовањем генерисаних SQL наредби примећује се да је свака од наредби адекватно преведена из израза написаног уз помоћ Query програмског интерфејса. Ово значи да, чак и код комплексних упита који садрже агрегацију и операције спајања, није генерисан ниједан угњежден упит који би успорио време извршавања.

Пример SQL наредбе упитa **Q7**, генерисане од стране *SQLAlchemy* алата, може се видети на листингу 5.4. Уочава се да, за разлику од *Entity Framework-a*, генерисан упит је идентичан, иницијално постављеном, упиту **Q7**.

SELECT `Posts`.user\_id AS `Posts\_user\_id`, count(`Post\_Tags`.tag\_id) AS total\_tags

FROM `Posts`

INNER JOIN `Post\_Tags`

ON `Posts`.post\_id = `Post\_Tags`.post\_id

GROUP BY `Posts`.user\_id

Листинг 5.4 Пример SQL наредбе генерисане од стране SQLAlchemy алата

Са друге стране, посматрањем дијаграма **5.3**, уочава се да поједини упити имају неколико пута веће време извршавања у односу на њихово извршавање употребом pymysql DBAPI интерфејса.

С обзиром на чињеницу да је за сваки упит генерисан адекватан SQL израз, ствара се претпоставка о нарушењу перформанси због поступка инстанцирања објеката након извршавања упита. Профилисањем извршавања проблематичних упита, уз помоћ уграђеног cProfile модула Пајтон програмског језика, примећује се да је апликација највећи део времена провела у оквиру \_instantiate методе *loading.py* модула изворног кода *SQLAlchemy* алата. Ова метода је задужена за поступак креирања објеката на основу података добијених извршавањем упита. Ово даје јасне назнаке да је инстанцирање објеката, добијених упитом, захтевна операција за *SQLAlchemy*, због комплексности имплементације објекта сесије који у себи садржи неколико различитих структура података за манипулацију стањима објеката. То потврђује и званична документација *SQLAlchemy* алата [10], у којој се наводи да се, уколико није неопходно, не добављају комплетни објекти, већ само неопходна поља. Такође, ова чињеница је потврђена у оквиру изворног кода *SQLAlchemy-ja* [3], где је наведено да је метода \_instantiate(), најкритичнија секција целокупног алата. У листингу 5.6, приказани су резултати профилисања извршавања упита ка бази.

ncalls tottime cumtime percall filename:ln(function)

463812 0.196 0.423 0.00 loading.py:883(\_instance)

60259 0.010 0.010 0.00 instrumentation.py:405(get\_impl)

2260235 0.118 0.118 0.000 protocol.py:62(read)

1 0.610 0.610 0.610 session.py:1563(execute)

Листинг 5.6 Профилисање извршавања упита Q2

Што се тиче извршавања DML наредби, показује се да операције уписивања и пакетне измене (енгл. *batch update*) имају приближна времена извршавања између ОРМ алата и чистог SQL кода, док je операција пакетног брисања (енгл. *batch delete*) у случају *Entity Framework-a* 10 пута спорија, а у случају *Hibernate-a* 3 пута спорија. Извршавање операције брисања кроз SQLAlchemy алат има приближно време као и у случају извршавања чистог SQL кода.

### Упоређивање објектно-релационих мапера

На дијаграму 5.4 приказана су времена извршавања изабраних алата за објектно-релационо мапирање.

Листинг 5.4 Упоређивање перформанси између изабраних ОРМ алата

Посматрањем дијаграма 5.1, 5.2, 5.3 и 5.4, може се закључити да, уз изузетак упита **Q3**, *Hibernate* алат за објектно-релационо мапирање има најстабилније перформансе, сличне онима добијеним извршавањем JDBC интерфејса.

Разлог због којег *Entity Framework* није ефикасан као *SqlClient* у упитима **Q6** и **Q7** је тај што наведени упити садрже операције груписања, које EF не преводи директно. Уместо коришћења изворнe SQL GROUP BY клаузуле, EF преузима све повезане записе и затим обрађује груписање у меморији апликације, што може довести до потенцијалних неефикасности.

Иако *SQLAlchemy* генерише адекватне упите, постоји велики проблем приликом инстанцирања објеката, што знатно нарушава његове перформансе.

# Закључак

Објектно-релациони мапери представљају корисно средство за развој модерних апликација јер омогућавају ефикасну интеграцију објектно оријентисаних језика и релационих база података. Међутим, њихова употреба захтева добро разумевање интерних механизама како би се избегли уобичајени недостаци који могу довести до лоших перформанси.

Спроведено истраживање је показало да су *Entity Framework*, *Hibernate* и *SQLAlchemy* релативно ефикасни за већину једноставних SQL операција. Ипак, код комплекснијих упита са спајањем, груписањем и агрегацијом података, примећене су значајније разлике у перформансама у односу на извршавање чистог SQL кода.

Главни узроци смањених перформанси су неоптимално генерисани SQL изрази, додатно оптерећење на апликацију због обраде резултата упита и скупи процеси инстанцирања објеката. Ово указује на значај правилног коришћења ОРМ алата и њихово комбиновање са чистим SQL-ом у одређеним ситуацијама.

И поред одређених недостатака у перформансама, предности које ОРМ алати доносе у продуктивности развоја, једноставнијем одржавању кода и смањењу грешака чине их корисним у већини случајева. Њихова широка заступљеност указује да многи развојни тимови сматрају да су компромиси у перформансама оправдани.

У будућим истраживањима би требало проширити обим тестирања на већи број различитих упита и операција. Такође, потребно је испитати утицај ОРМ алата на скалабилност система под оптерећењем. Ово би пружило још јаснију слику о њиховим предностима и манама.

# Литература

[1] EFCore. (n.d.). Retrieved from <https://github.com/dotnet/efcore> (Приступ 14.09.2023.)

[2] Hibernate. (n.d.). Retrieved from <https://github.com/hibernate/hibernate-orm> (Приступ 14.09.2023.)

[3] SQLAlchemy. (n.d.). Retrieved from <https://github.com/sqlalchemy/sqlalchemy> (Приступ 14.09.2023.)

[4] Colley, D., Stanier, C., & Asaduzzaman, M. (2018). The Impact of Object-Relational Mapping Frameworks on Relational Query Performance. In \*2018 International Conference on Computing, Electronics & Communications Engineering\* (iCCECE) (pp. 47-52). <https://doi.org/10.1109/iCCECOME.2018.8659222>

[5] Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009). A Classification of Object-Relational Impedance Mismatch. In \*2009 First International Conference on Advances in Databases, Knowledge, and Data Applications\* (pp. 36-43). <https://doi.org/10.1109/DBKDA.2009.11>

[6] Cvetkovic, S., & Janković, D. (1970). A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment. \*6348\*(pp. 147-158). <https://doi.org/10.1007/978-3-642-16092-9_14>

[7] Barnes, J. M. (2007). Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications. \*Mathematics, Statistics, and Computer Science Honors Projects, 6\*. <https://digitalcommons.macalester.edu/mathcs_honors/6> (Приступ 24.08.2023.)

[8] Hibernate. (n.d.). <https://hibernate.org/orm/documentation/6.2/> (Приступ 14.09.2023.)

[9] Entity Framework Core. (n.d.). <https://learn.microsoft.com/en-us/ef/core/> (Приступ 14.09.2023.)

[10] SQLAlchemy. (n.d.). <https://docs.sqlalchemy.org/en/20/> (Приступ 14.09.2023.)

[11] Fowler, M. (2002). \*Patterns of enterprise Application Architecture\*. Добављено са <https://www.marcosavard.com/PageProfessionnelle/books/PatternsOfEnterpriseApplicationArchitecture/PatternsOfEnterpriseApplicationArchitecture2006Nov13.pdf> (Приступ 23.08.2023.)

[12] Mihalcea, V. (2016). \*High-performance Java Persistence\*. Cluj-Naoca, Romania.

[13] Neward, T. (2006). The Vietnam of Computer Science.

[14] Bayer, M. (2012). SQLAlchemy. In A. Brown & G. Wilson (Eds.), \*The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks\*.

[15] Object Management Group (2006). Java Persistence API (JPA) Version 2.0. Добављено са [http://www.jcp.org/en/jsr/detail?id=338] (Приступ 08.09.2023.)

[16] Pivotal Software, Inc. (2018). Spring Data JPA reference documentation. Добављено са [https://docs.spring.io/spring-data/jpa/docs/current/reference/html/] (Приступ 06.09.2023.)

[17] Oracle. (2019). MySQL 8.0 reference manual. Добављено са [https://dev.mysql.com/doc/refman/8.0/en/] (Приступ 08.09.2023.)

[18] Microsoft. (2017). .NET framework documentation. [https://docs.microsoft.com/en-us/dotnet/framework] (Приступ 03.09.2023)

[19] Khorikov, V. (2020, August 5). Why you should avoid ORM frameworks. (<https://enterprisecraftsmanship.com/posts/avoid-orm-frameworks/>) (Приступ: 06.09.2023.)

[20] Larman, C. (n.d.). Object-relational mappers: Anti-patterns, pain without gain. [<https://craiglarman.com/wiki/index.php?title=Object-relational_mappers:_anti-patterns,_pain_without_gain>] (Приступ 06.09.2023).

[21] Мicrosoft. (2023). Expression Trees. Преузето са <https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/> (Приступ: 03.09.2023.)

[22] Oracle. JDBC overview. Преузето са <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

[23] Python Software Foundation DBAPI 2.0 спецификација. Преузето са <https://peps.python.org/pep-0249/>

[24] Hunt, J. (2019). Python DB-API. In: Advanced Guide to Python 3 Programming. Undergraduate Topics in Computer Science. Springer, Cham. <https://doi.org/10.1007/978-3-030-25943-3_24>

[25] Bauer, C., & King, G. (2015). Java persistence with Hibernate (2nd ed.). Manning Publications.

# БИОГРАФИЈА

Mилош Гравара рођен је 20. јануара 2000. године у Новом Саду. Основну школу „Јожеф Атила“ у Новом Саду завршио је 2015. године. Након тога уписује гимназију „Исидора Секулић“ у Новом Саду, коју завршава 2019. године. Исте године уписује се на Факултет техничких наука, смер Рачунарство и Аутоматика. Школске 2021/22. се опредељује за усмерење Примењене рачунарске науке и информатика, након чега се школске 2022/23. опредељује за модул Информациони системи. Положио је све испите предвиђене планом и програмом.

# КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

|  |  |
| --- | --- |
| Редни број, **РБР**: |  |
| Идентификациони број, **ИБР**: |  |
| Тип документације, **ТД**: | монографска публикација |
| Тип записа, **ТЗ**: | текстуални штампани документ |
| Врста рада, **ВР**: | дипломски рад |
| Аутор, **АУ**: | Милош Гравара |
| Ментор, **МН**: | проф. др Горан Сладић, ФТН Нови Сад |
| Наслов рада, **НР**: | Анализа имплементација и перформанси алата за објектно-релационо мапирање |
| Језик публикације,**ЈП**: | српски |
| Језик извода, **ЈИ**: | српски / енглески |
| Земља публиковања, **ЗП**: | Србија |
| Уже географско подручје, **УГП**: | Војводина |
| Година, **ГО**: | 2023 |
| Издавач, **ИЗ**: | ауторски репринт |
| Место и адреса, **МА**: | Нови Сад, Факултет техничких наука,  Трг Доситеја Обрадовића 6 |
| Физички опис рада, **ФО**: | бр. поглавља 7/ страница 70/ цитата 25/ табела 2/ слика 20/ графикона 4/ прилога / листинга 30 |
| Научна област, **НО**: | Електротехничко и рачунарско инжењерство |
| Научна дисциплина, **НД**: | Софтверско инжењерство |
| Предметна одредница/кључне речи, **ПО**: | Објектно-релационо мапирање, перформансе, SQL упити |
| **УДК** |  |
| Чува се у, **ЧУ**: | Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад |
| Важна напомена, **ВН**: |  |
| Извод, **ИЗ**: | Описане су имплементације три најзаступљенија алата за објектно-релационо мапирање - *Entity Framework*, *Hibernate* и *SQLAlchemy* Анализиран је њихов утицај на перформансе приликом употребе насупрот чистим SQL упитима. Резултати су показали деградацију у перформансама за скуп комплекснијих упита, док су за једноставне упите забележене занемарљиве разлике. |
| Датум прихватања теме, **ДП**: |  |
| Датум одбране, **ДО**: |  |
| Чланови комисије, **КО**: |  |
| председник |  |
| члан |  |
| ментор | проф. др Горан Сладић, ФТН Нови Сад |
| Потпис ментора | |

# KEY WORDS DOCUMENTATION

|  |  |
| --- | --- |
| Accession number, **ANO**: |  |
| Identification number, **INO**: |  |
| Document type, **DT**: | monographic publication |
| Type of record, **TR**: | textual material |
| Contents code, **CC**: | BSc thesis |
| Author, **AU**: | Miloš Gravara |
| Mentor, **MN**: | Goran Sladić, PhD, full prof., FTN Novi Sad |
| Title, **TI**: | Analysis of implementations and performance of object-relational mapping tools |
| Language of text, **LT**: | serbian |
| Language of abstract, **LA**: | serbian / english |
| Country of publication, **CP**: | Serbia |
| Locality of publication, **LP**: | Vojvodina |
| Publication year, **PY**: | 2023 |
| Publisher, **PB**: | author’s reprint |
| Publication place, **PP**: | Novi Sad, Faculty of Technical Sciences, Square Dositej Obradović 6 |
| Physical description, **PD**: | no. of chapters 8/ pages 70/ quotes 25/ tables 2/ pictures 20/ graphs 1/ appendix / listings 30 |
| Scientific field, **SF**: | Electrical and computer engineering |
| Scientific discipline, **ND**: | Software engineering |
| Subject / Keywords, **S/KW**: | object-relational mapping, performance, SQL queries |
| **UDC** |  |
| Holding data, **HD**: | Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad |
| Note, **N**: |  |
| Abstract, **AB**: | The implementations of the three most common tools for object-relational mapping - *Entity Framework*, *Hibernate* and *SQLAlchemy* - are described. Their impact on performance against native SQL queries is analyzed. The results showed a degradation in performance for a set of more complex queries, while for simple queries negligible differences were noted. |
| Accepted by sci. board on, **ASB**: |  |
| Defended on, **DE**: |  |
| Defense board, **DB**: |  |
| president |  |
| member |  |
| mentor | Goran Sladić, PhD, full prof., FTN Novi Sad |
| Mentor’s signature | |

1. Слика преузета са <https://www.entityframeworktutorial.net/entityframework6/> [↑](#footnote-ref-1)
2. Слика преузета са: <https://www.codemag.com/article/0711051/Introducing-ADO.NET-Entity-Framework> [↑](#footnote-ref-2)
3. Слика преузета из [9] [↑](#footnote-ref-3)
4. Слика преузета са: <https://www.codeproject.com/Articles/1240553/LINQ-Part-An-Introduction-to-IQueryable> [↑](#footnote-ref-4)
5. Слика преузета са <https://www.geeksforgeeks.org/hibernate-architecture/> [↑](#footnote-ref-5)
6. Слика преузета из [25] [↑](#footnote-ref-6)
7. Слика преузета са <https://vladmihalcea.com/> [↑](#footnote-ref-7)
8. Слика преузета из [24] [↑](#footnote-ref-8)
9. Слика преузета из [14] [↑](#footnote-ref-9)
10. Слика преузета из [14] [↑](#footnote-ref-10)
11. Слика преузета из [14] [↑](#footnote-ref-11)
12. Слика преузета из [14] [↑](#footnote-ref-12)
13. Слика преузета из [14] [↑](#footnote-ref-13)