Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra Informačních Technologií

Diplomová práce

Porovnání NDB a ORM

Bc. Filip Boye-Kofi

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Filip Boye-Kofi

Systémové inženýrství a informatika
Informatika

Název práce

Porovnání NDB a ORM

Název anglicky

Comparison NDB a ORM

Cíle práce

Hlavním cílem této práce je porovnat systémy pro přístup k databázi podle několika systémových a funkčních vlastností.
Dílčí cíle práce jsou:
- vytvoření experimentální aplikace pro porovnání;
- vytvoření vhodné databázové struktury;
- zjištění výkonnosti dotazů těchto přístupů.

Metodika

Teoretická část práce bude založena na analýze odborných informačních zdrojů.

V rámci praktické části budou porovnány NDB (MySQL) a ORM (Doctrine) na základě výpočetního výkonu a implementaci těchto přístupů k databázi. V rámci experimentu bude testováno velké množství dat a budou prováděny složité dotazy, které umožní změřit jednotlivé aspekty. Bude vytvořen návrh databázové struktury a následně pomocí jazyka PHP k ní bude přistupováno ve třídách modelu. Práce s databází bude na úrovni dotazu nebo objektů. Pomocí vhodných nástrojů budou porovnány vybrané metriky.

Na základě teoretické a praktické části budou formulovány závěry.

Doporučený rozsah práce

60 – 70 stran

Klíčová slova

ORM, NETTE, PHP, DB, NDB, SQL, DI

Doporučené zdroje informací

Carr D, Gray M. Beginning PHP : Master the Latest Features of PHP 7 and Fully Embrace Modern PHP Development. Birmingham: Packt Publishing, Limited; ISBN: 9781789535907;2018.

Halpin, Terry. Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM – Technics Publications – 2015. ISBN-13: 978-1634620741

WELLING, Luke a Laura THOMSON. Mistrovství PHP a MySQL. Přeložil Ondřej BAŠE. Brno: Computer Press, 2017. ISBN 978-80-251-4892-1.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 26. 8. 2019

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 14. 10. 2019

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 31. 03. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Porovnání NDB a ORM" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2021 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Janu Masnerovi Ph.D. za veškeré rady, připomínky a za motivaci napsání této diplomové práce.

Porovnání NDB a ORM

Abstrakt

Hlavním přínosem této práce je velice podrobné testování ORM knihovny Doctrine 2 a Nette Database s přesnými postupy. Testování těchto jazyků bylo provedeno za pomocí různých query dotazů. Rovněž jsou detailně popsány části PHP jazyku pro provoz těchto knihoven a zároveň popis databází, s kterými je poté manipulováno v praktické části.

Pro získání přesných výsledků byl měřen čas exekučního zápisu a čas čtení z databázového systému za pomoci knihovny „tracy bar“. Výsledky změřené pomocí této knihovny byly poté analyzovány a na základě výstupů bylo určeno, která technika je výkonnostně rychlejší či pomalejší. Bylo tedy jednoznačně určeno, která z těchto knihoven byla výkonnější v komunikaci s databází.

Klíčová slova: ORM, NETTE, DOCTRINE, PHP, DB, SQL, DI

Comparison NDB and ORM

Abstract

The main benefit of this work is a very detailed testing of the ORM library Doctrine 2 and Nette Database with precise procedures. Testing of these languages has been demonstrated using various query terms. Parts of the PHP language for the operation of these libraries are also described in detail, as well as a description of the databases, which are then manipulated in the practical part.

To obtain accurate results, the time of execution write and time of reading from the database system was measured using the "tracy bar" library. The results measured using this library were then analyzed and the outputs were used to determine which library's performance is faster or slower. Thus, it was clearly determined which of these libraries was more efficient in communicating with databases.

Keywords: ORM, NETTE, DOCTRINE, PHP, DB, SQL, DI

Obsah

Seznam obrázků

Seznam tabulek

Seznam grafů

Seznam použitých zkratek

PHP - Hypertext Preprocessor

SŘBD - Systém řízení báze dat

Go - programovací jazyk Golang

JS - programovací jazyk JavaScript

HTML - HyperText Markup Language

ORM - Object-relational Mapping

JSON – Formát pro výměnu dat

XML - Extensible Markup Language

CLI - Command-line Interface – Příkazový řádek

CRON – démon ve formě procesu či skriptu

IDE - Vývojové prostředí

OOP – Objektově orientované programování

SQL - Structured Query Language

DBMS - Database management system

DQL - Doctrine Query Language

API - Application Programming Interface

IOC - Inversion of control

URL - Jjednotný lokátor zdroje

HTTP - Hypertext Transfer Protocol – Internetový protokol

1 Úvod

Vývoj webových aplikacích je v dnešním světě velmi populární odvětví. Pomocí jazyků jako je například PHP či Java lze během několika hodin vytvořit jednoduchou webovou aplikaci, která může ukládat data o uživateli do databáze a následně po vyžádání zobrazit na stránce. Dnešní svět je směřován na vývoj desktopových a webových aplikací. Jedna z nedílných součástí aplikace, ať už webové či desktopové, je možnost ukládat důležité hodnoty či informace do databáze.

Je obecně známo, že jazyky jako je PHP, JS či Go nejsou persistentní, a proto je potřeba přístup k databázi na lokálním počítači, na serveru nebo na *cloudu.* Původní aplikace byly zpracovávány s přímou vazbou na databází pomocí syntaxe, které rozuměl SŘBD. Tento způsob je i dnes hojně využíván. Díky příchodu objektového programování bylo možné problematiku reálného světa aplikovat do objektů, které PHP začalo v roce 2004 s příchodem verze 5 podporovat.

Díky objektovému přístupu bylo možné i tuto techniku a myšlenku převést na vrstvu mezi kódem a databází pojmenovanou Objektově Relační Mapování. Je nutné tedy brát v potaz náročnost na serveru pokud se využívá ORM technika pro zpracování několik záznamů. Po důkladném zkoumání veškeré literatury týkající se výkonnostních diferencí mezi Doctrine 2 a raw SQL jsem nenašel studii, kde by bylo poukazováno na výkonnostní rozdíly těchto technik, s přesně definovaným postupem. Tato práce se zabývá měřením těchto dvou technologií v kontextu výkonu. K dosažení a ověření je používán MVC patern a je sledován vliv na funkčnost během zpracování dat z databáze. Následně jsou vyhodnoceny výsledky a grafy tohoto experimentu. Nakonec jsou ze získaných poznatků formulovány závěry.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této práce je porovnat systémy pro přístup k databázi podle několika systémových a funkčních vlastností.

Dílčí cíle práce jsou:

- vytvoření experimentální aplikace pro porovnání
- vytvoření vhodné databázové struktury
- zjištění výkonnosti dotazů těchto přístupů

2.2 Metodika

Teoretická část práce bude založena na analýze odborných informačních zdrojů. V rámci praktické části budou porovnány NDB (MySQL) a ORM (Doctrine) na základě výpočetního výkonu a implementaci těchto přístupů k databázi. V rámci experimentu bude testováno velké množství dat a budou prováděny složité dotazy, které umožní změřit jednotlivé aspekty.

Bude vytvořen návrh databázové struktury a následně pomocí jazyka PHP k ní bude přistupováno ve třídách modelu. Práce s databází bude na úrovni dotazů nebo objektů. Pomocí vhodných nástrojů budou porovnány vybrané metriky. Na základě teoretické a praktické části budou formulovány závěry.

3 Teoretická východiska

3.1 PHP

PHP je jednoduchý a zároveň výkonný jazyk s mnoha funkcemi. Je tedy určený hlavně pro tvorbu HTML komponent. PHP umožňuje *server-side* architekturu, původ pro jeho existenci byla tato architektura kdy umožňuje tvorbu dynamické webové aplikace. Pokud uživatel chce získat HTML stránku, je nutné mít *Webový server* a na tomto serveru také PHP. Přes Webový server prochází složená HTML stránka zpět k uživateli. Ovšem díky popularitě PHP také umožňuje další zdroje informací, jako je například JSON, XML. Dále podporuje také CLI neboli *Command Line Interface*, to umožňuje uživateli spustit PHP skripta přímo z příkazového řádku. *Command Line* je využit, pokud je potřeba práce na pozadí serveru a není nutně určená koncovému uživateli. Jsou to tedy situace, kdy je potřeba logovat či zálohovat nějaká data či informace, nebo kdy je za potřebí spustit *CRON* tasky (Tatroe & MacIntyre, 2020).

PHP může běžet na mnoha operačních systémech, jako jsou například systémy postavené na Unix, což jsou distribuce jako je Ubuntu, Arch, Debian. Dále také lze spustit na Windows nebo na Mac OS. Díky flexibilitě lze souběžně fungovat s *Web serverama,* jako je například Nginx či Apache. Jazyk jako takový je velmi flexiblní a užitečný, není tedy nutné pouze vypisovat HTML stránky, ale například může pomocí *bulit-in* funkcí generovat PDF či XML. PHP dále podporuje širokou škálu databází, jako je například PostgreSQL, MySQL či MS-SQL. Také lze propojit NoSQL database, například MongoDB či Redis (Tatroe & MacIntyre, 2020) (Welling & Thomson, 2017).

3.1.1 Syntaxe jazyka a sémantika

Tyto pojmy v kontextu jazyka PHP znamenají formát kódu (syntaxe) a tvoření určitého vztahu pomocí textů a symbolů v kódu (sémantika). Soubory PHP poznáme, pokud název souboru má koncovku „.php“. Aby PHP mohl spustit daný soubor, musí na začátku daného *skriptu* „<?php“, tím říkáme PHP, že cokoliv bude v tomto tagu, bude muset být interpretováno v podobě výstupu. Veškeré příkazy, které musí PHP vykonat končí středníkem „ ; “. Proměnné můžeme volat s dolarem na začátku, ale

musí být v určitém kontextu, jinak se dolar explicitně před proměnou nepíše (Welling & Thomson, 2017).

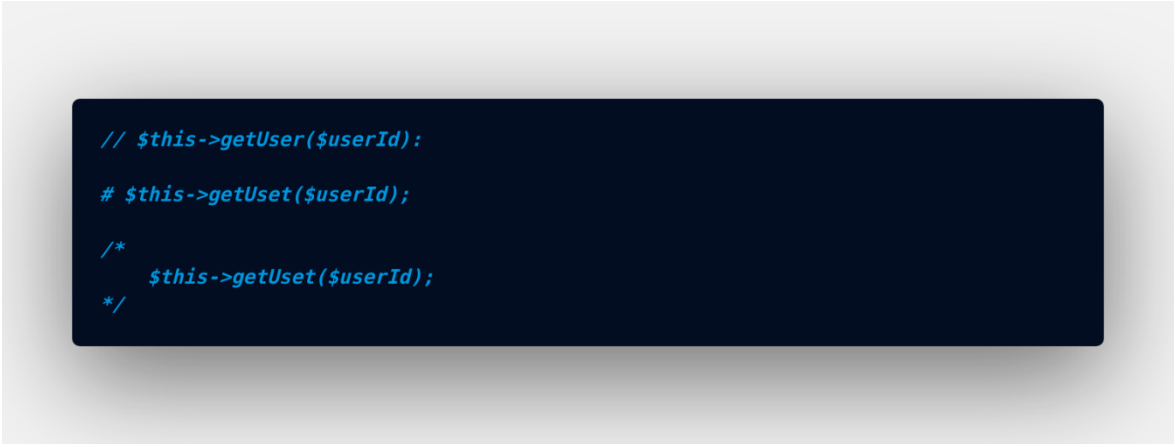
Pokud máme funkci mimo třídu, tak se napíše pouze jméno funkce se závorkami a zakončí se středníkem. Pokud funkce je součástí instance třídy (začínající "class InstanceTřidy") pak se zapisuje funkce *"$this->funkce()"* (zde je nutné zmínit že v situaci kdy funkce je definovaná uvnitř třídy se říká metoda), je to z toho důvodu, že říkáme třídě, ať použije svojí vlastní funkci (proto "this") sama v sobě (Welling & Thomson, 2017).

3.1.2 Komentáře

Komentáře jsou důležitým prvkem v programování, protože kdokoliv se správnou dokumentací může zjistit, co daná funkce dělá, aniž by prozkoumával programátor každý řádek po řádku. Tato praktika je důležitá, pokud chceme, aby náš kód byl používán v jiných projektech. Komentáře mají určitý standard. Snažíme se tedy psát výstižně a pokud možno co nejstručněji (Nixon, 2014).

Máme dva typy komentářů:

- *Single-line* (komentář na jeden řádek)
- *Multiple-lines* (komentář na více řádků)



Obrázek 1- Typy komentářů v PHP

Na obrázku 1 vidíme základní použití komentářů, PHP totiž ignoruje tyto řádky v *execution* a tudíž na výstupu nebude žádná chybová hláška. Je dobré se komentářům vyhýbat (zejména komentování funkcí a proměnných). Proto se používají pouze jako

DocBlock nebo ke komentování určitých proměnných, které na pozadí může knihovna, IDE či framework užívat.

Multiple-lines je tedy dobré používat jako *DocBlock*, to lze vidět na obrázku 2, kdy můžeme popsat třídu, jeho typy argumentů i return type deklarace (Nixon, 2014).

```php
/**
 * function validates apikey of application
 * @param array $config Config of application.
 * @return bool returns false if Apikey doesnt exist
 */
public function isValid(array $config) : bool {

    if (! $config['api_key']) {
        return FALSE;
    }

    return TRUE;
}
```

Obrázek 2- Docblock v PHP

3.1.3 Přiřazovací operátor

Základním operátorem pro přiřazování je "=". Pravá hodnota operandu je předávána levému operandu to lze vidět v tabulce 1. Je k dispozici i takzvaně kombinovaný operátor, kdy používáme hodnotu levého operandu a přiřadíme výslednou hodnotu pravého operandu zpět do levého operandu (PHP, 2021). Zde je seznam dostupných operátorů.

Přiřazení	*a = b*
Přidání	*a += b*
Odejmutí	*a -= b*
Pronásobení	*a *= b*
Dělení	*a /= b*
Modulus	*a %= b*

Tabulka 1- Popis operátorů a jejich značení

```php
function example1() {
    $array = [5,4,3];

    for ($i=0; $i < count($array) ; $i++) {
        $array[$i] += $i;
    }

    return $array;
}

function example2() {
    $array = [5,4,3];

    for ($i=0; $i < count($array) ; $i++) {
        $array[$i] .= 'is number';
    }

    return $array;
}
```

Obrázek 3- Implementace operátorů v PHP

Na obrázku 3 vidíme tři přiřazovací operátory.

- Základní, kdy pole *[5, 4, 3]* přiřazujeme do proměnné *array*.
- Kombinované kdy:
    - .=
        - přidává danou hodnotu v proměnné i do hodnoty pole s indexem *i*
    - +=
        - přičítá danou hodnotu v proměnné i do hodnoty pole s indexem *i*

Logické operátory

and / &&	*pokud $a a $b jsou pravdivý*
or / ||	*pokud $a nebo $b je pravdivé*
xor	*vrací true, právě jen když je pouze jedno z hodnot true*
!	*vrací true pokud proměnná není true*

Tabulka 2- Popis logických operátorů

Array operátory

Sjednocení	*$a + $b*
Rovnost	*$a == $b, vrací true/false*
Nerovnost	*$a != $b (<>), vrací true/false, pokud nemají stejné klíče a hodnoty*
Identický	*$a === $b, vrací true/false, pokud mají stejnou posloupnost a stejný datový typ*
Neidentické	*$a !== $b, vrací true/false, pokud pole nejsou identické*

Tabulka 3- Popis operátoru pro práci s polem

Porovnávací operátor

V této sekci jsou operátory, které se používají k porovnávání hodnot, kdy PHP kompiluje výsledkem *TRUE / FALSE* (Nixon, 2014).

Rovnost	==
Identitní	===
Neidentické	!==
Nerovné	!= / <>
Spaceship	<=>
Menší nebo rovno	<=
Větší nebo rovno	>=
Menší	<
Větší	>

Tabulka 4- Popis porovnávacích operátorů

3.1.4 Primitivní datové typy

PHP nabízí širokou škálu datových typů, přesněji deset datových typů, které může proměnná nabývat (Sklar & Trachtenberg, 2006).

Integer

Je interpretován v PHP jako celé číslo bez desetinné čárky.

Pravidla, jak rozpoznat *Integer* typ jsou:

- Musí mít alespoň jedno číslo
- Nesmí mít desetinnou čárku
- Může nabývat kladných čísel, ale i záporných čísel

Rozmezí hodnot, které lze nabývat je od -2147483648 až do 2147483647.

Boolean

Tento typ je skalární. Bool hodnota může nabývat pouze hodnotou *TRUE* nebo *FALSE*. Zároveň jsou tyto stavy předefinované konstanty.

Float

Float se vyznačuje tím, že obsahuje v číselné řadě desetinnou čárku. Několika způsoby lze vyjádřit *floating-point number* a to například 8.92 nebo 35E+20.

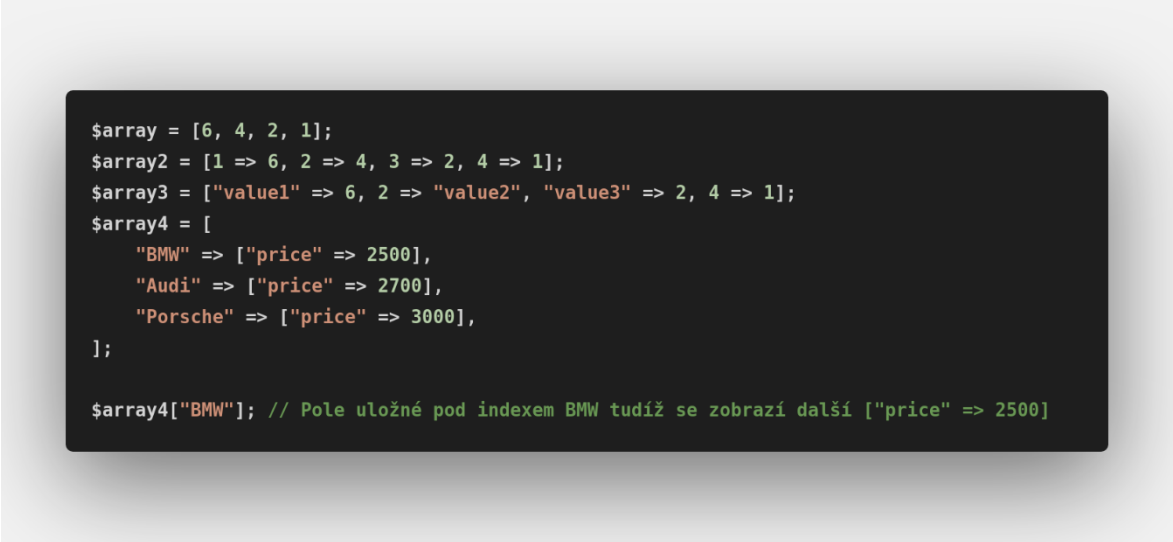
String

Je série po sobě jdoucích znaků, kde jeden znak je zastoupen jedním *byte*.

Array

Další datový typ je pole. Jedná se o mapu, který lze vidět na obrázku 4, kdy jedna entita v poli je zastoupena *key "=>" value*, díky tomu můžeme přiřazovat klíče k hodnotám a následně danou hodnoty dohledat pomocí klíče neboli *indexu*. Pole dokáže nahradit frontu nebo zásobník. Pole se rozlišuje na dva typy, a to asociativní a vícerozměrná.

Asociativní pole znamená, že *index* neboli *key* odkazuje na hodnotu. Vícerozměrné pole je pole, které obsahuje další pole. Tímto způsobem můžeme ukládat data a poté vykonat iteraci pro získání dat (Sklar & Trachtenberg, 2006). Pole se vytváří pomocí hranatých závorek *[]*, pole tedy přijímá n hodnoty oddělené čárkou, každý pár má mezi sebou symbol „=>".



```php
$array = [6, 4, 2, 1];
$array2 = [1 => 6, 2 => 4, 3 => 2, 4 => 1];
$array3 = ["value1" => 6, 2 => "value2", "value3" => 2, 4 => 1];
$array4 = [
    "BMW" => ["price" => 2500],
    "Audi" => ["price" => 2700],
    "Porsche" => ["price" => 3000],
];

$array4["BMW"]; // Pole uložné pod indexem BMW tudíž se zobrazí další ["price" => 2500]
```

Obrázek 4- Pole a jejich definování v PHP

3.1.5 Funkce

Funkce je znovupoužitelná instrukce kódu, která má přesně definované rozhraní, vstupy a výstupy. Tyto vstupy může transformovat do jiných hodnot a pak tyto hodnoty použít jako své vlastní výstupy (Thomas & Hunt, 2020). PHP rozlišuje, zda se jedná o funkci pomocí *keyword „function“*, kdy vždy dáváme po funkci ihned název funkce se závorkami na konci. Je dobré vhodně a výstižně pojmenovávat funkce.

Díky funkcím můžeme na serveru či počítači vykonávat strojově pokyny. Instrukce jsou zadávány ve složených závorkách. Funkce mohou, ale nemusí vracet hodnoty, pokud dojde k tomu, že funkce nevrací žádnou hodnotu můžeme této funkci přidat *return type hint void*. Hodnoty parametrů při tvorbě funkce se dají předefinovat, pokud se tak stane, volání funkce nemusí nutně být volána s argumentem vyhovující parametrům funkce, neboť má určenou výchozí hodnotu (Holzner, 2008).

3.1.6 Třídy a Instance objektů

PHP disponuje objektovým orientovaným programováním, to umožňuje abstrakci reálného světa. Třída se vyznačuje speciálním slovem class, poté následuje jméno třídy a složené závorky, kde je logika dané třídy. Pro přístup k metodám či proměnným se používá pseudoproměnná “$this”. Třída má několik magických metod.

Jedna z nich je konstruktor neboli __construct, která umožňuje vytvořit „blueprint“ třídy.

Třídy jako takové, mohou mít předka nebo mohou být samy rodiči. Pokud je třída rodič, pak jeho potomci přebírají schopnosti rodiče a zároveň mají svoje schopnosti. Schopnostmi myslíme metody či atributy.

V kontextu tříd se označuje funkce metodou, tato problematika je rozebrána a popsána v knize *PHP Objects, Patterns, and Practice* od Zandstra. Třída může být finální, to znamená, že nemůže mít potomka, tato třída má keyword final. Opak final je abstraktní třída (abstract), ta nemůže být instancovaná, tím pádem se musí z této třídy dědit, v této třídě lze nadefinovat abstraktní metodu, ale nelze napsat tělo této metody.

Poslední, nejkomplexnější a zároveň nejabstraktnější jsou rozhraní neboli interface, kdy určujeme, jaké metody musí třída, která implementuje toto rozhraní mít (Zandstra, 2016).

3.1.7 Scope

Scope neboli viditelnost, je důležitá pro zabezpečení chodu programu proti samotným programátorům. Jde o situace, kdy máme důležitou proměnou či metodu, která je vysloveně užívána pro danou třídu, ovšem i tato třída může být rodič, a tudíž z ní mohou dědit veškeré proměnné a metody, což může vést k nekonzistenci výstupu. Jde o situace, kdy máme bankovní účet typu Rodič, který dokáže provádět internetové platby (Nixon, 2014). Díky dědičnosti můžeme redukovat duplicitní kód tím, že podědíme klíčovým slovem *extends* a tím můžeme užívat veškeré základní funkce, jako je například výpis z účtu. Když ale dědit bude účet typu Junior, tak nežádoucí efekt bude schopnost užívat internetové platby, díky tomuto přichází na řadu *scope* kdy můžeme tzv. “zakázat” tuto funkci. Tato logika je aplikována i na proměnné tříd, kde se zmiňuje (Böhmer, 2015).

Máme tři druhy viditelnosti:

- *public*
- *protected*
- *private*

Public

Metoda či proměnná, která je zpřístupněná všem třídám, které mají přístup k dané třídě. Tato metoda začíná *keyword public,* následuje *function* a hned poté název metody.

Protected

Metoda či proměnná, která je zpřístupněná jen pro ty třídy, které dědí danou třídu obsahující protected metodu nebo proměnnou.

Private

Metoda či proměnná, která není zpřístupněná nikde a pouze jen v samotné třídě.

3.2 Objektově orientované programování

Je paradigma v programování, široce užíváno pod názvem OOP neboli objektově orientované programování. OOP umožňuje programátorům řídit a vytvářet přirozeně softwarové systémy pomocí abstrakcí, díky objektům. Tento způsob je flexibilní a lze jednoduše pochopit daný problém. Umožňuje opětovné užívání komponent (Böhmer, 2015). Zlepšuje také udržovatelnost kódu pomocí zjednodušení procesů úprav či rozšiřování komponent. Objektově orientovaný přístup se běžně používá během fáze vývoje softwaru. OOP snižuje celkový náklad na vývoj a údržbu softwaru. Objektově orientovaný model zjednodušuje tvorbu programů o této problematice se zmiňuje (Harle, 2010).

Důležité je mít na paměti, že objektově orientovaný přístup je užíván během každé fáze vývoje. OOP již prokázalo využití ve vývojovém světě, kdy lze zredukovat náklady na vývoj, co se týče samotného vývoje tak i testování a vše potřebné k vývoji produktu. Cituji zakladatele OOP (Booch, 2007): “Object-oriented design is a method of design encompassing the process of object- oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.”.

Třídy a objekty

Třídy a objekty jsou páteří objektového orientovaného programování, rozdíl mezi těmito pojmy je obecné pravidlo. Třída je rozhraní/blueprint, který definuje metody a vlastnosti specifického objektu. Objekt je instance z třídy (Lee & Tepfenhart, 2002). Třída je abstrakce, zatímco objekt je opravdová entita. Každý objekt je nadefinován pomocí identifikace, která zajišťuje unikátní jméno pro třídu. Stavy, které jsou reprezentovány jako *properties* objektu. Chování objektu je nadefinováno pomocí metod.

Základním konceptem jsou tedy dva objekty z třídy Students: John Doe, Adam Novák. Jejích vlastností/stavem může být věk nebo barva vlasů. Tyto objekty mohou mít poté schopnost chodit do školy, pokud je to definované v blueprint ve Students (Pecinovský, 2010).

Metody a vlastnosti

Metoda je operace, která může být vykonávána objektem. Vlastnosti neboli *property*, umožňují objektu (cizímu) z venčí zasahovat do vlastností objektu. V PHP jsou definovány jako *"$this->(vlastnost)"*. Pomocí *getters* a *setters*, lze kontrolovat přístupnost vlastnostem objektů (Zandstra, 2016).

Abstrakce a zapouzdření

Abstrakce je proces extrahování důležitých informací o objektu, kdy specifické detaily jsou ignorovány. Díky tomu lze užít objekt bez znalostí vnitřních detailů. Zapouzdření je proces pro seskupení vlastností do *„kontejneru"* (Böhmer, 2015). Vytváří tedy bariéru objektu proti vnějšímu světu. Účelem není schovávat informace. Je to tedy mechanismus kdy se minimalizuje interakce mezi objekty pomocí *keyword public a private.* Ke komunikaci s okolím se užívá *public keyword* a pro uzavření před jinými objekty se užívá *private*. Zde lze použít opět *getters a* setters pro přístup k informacím objektu (Lethbridge & Laganiere, 2005).

Dědičnost

Dědičnost je založena na stromovém způsobu seřazení všech objektů, kdy objekty mohou dědit z jiných objektů. Tímto způsobem získají schopnost jiných objektů (Thomas & Hunt, 2020).

Je to tedy mechanismus k definování třídy v kontextu předchozích definovaných tříd. To umožňuje vývojářům, definovat několik tříd, které si mohou sdílet některé vlastnosti či metody. Stručně řečeno je to způsob, jak předchozí definování metody přepsat a rozšiřovat s novými metodami a vlastnostmi (Pecinovský, 2010).

Polymorfismus

Polymorfismus nastává, pokud více objektů přepisuje tvz. *Overriding* metodu zděděnou od rodiče, tak, aby dělala vykonávala určitou operaci (Zandstra, 2016).

Delegace

Objekt, který je schopen používat služby jiných objektů. Tento objekt požádá určitý objekt o provedení dané metody. Tento daný objekt „vystavuje“ cíleně svoji metodu k obsloužení (Lethbridge & Laganiere, 2005).

Kompozice

Způsob, jak v objektovém programování dosáhnout kombinaci základních objektů vložené do jednoho objektu. Je to tedy způsob či nástroj pro tvorbu pokročilých datových struktur, jako je například *Linked List* (Harle, 2010).

3.3 Databáze

Databází je myšleno určité uložiště dat. Na počátcích vývoje byly tyto databáze děleny do dvou hlavních typů, hierarchické a síťové. Je to tedy místo pro perzistentní data, která jsou poté využívána systémy. Aktuálně existují další databáze, jako je relační databáze, objektově-relační nebo objektově-orientovaná databáze, které na sebe nabalují prvky z objektového programování (Stephens, 2008).

V této souvislosti rozlišujeme několik pojmů, a to báze dat a DBMS (v anglickém jazyce DataBase Managment Systém, v českém jazyce SŘBD – Systém Řízení Báze Dat), pomocí tohoto systému lze manipulovat s fyzickým uložištěm a operacemi nad nimi, a to na abstraktní úrovní. Databáze má několik vlastností, jako je například tvorba omezujících práv pro přístup či manipulaci s daty. Dále umožňuje přístup více uživatelům s omezujícími právy. Aktuálně existuje řada komerčních

SŘBD. Mezi nejznámější patří MySQL a PostgreSQL. Tyto systémy pracují za pomocí SQL (Simple Query Language) jazyka (Coronel & Morris, 2018).

3.4 Relační databáze

Relační databáze jejímž stavebním kamenem je relační model. Tímto pojmem lze označovat kromě softwarového řešení i *database*. Je tedy založena na tabulkách, které mají nadefinované vztahy. Tyto tabulky obsahují řádky, které lze chápat jako záznamy. Sloupce v relační databází lze chápat jako omezující podmínku, jejíž hodnota musí splnit, k uložení do tohoto sloupce. Sloupec dále slouží i jako konkrétní hodnota daného záznamu. Sloupce, ve kterých je uložen cizí klíč lze chápat jako uložení informace o vztahu mezi záznamy (Harrington, 2016). Relace jsou jedním z hlavních pilířů relační databáze. Relace jsou dvourozměrné struktury. Množina přípustných hodnot, které jsou dovoleny pro daný sloupec se nazývá doména, například doména můžou být telefonní čísla, bankovní účty. Doména je spjatá s datovým typem, ale každý má jiný účel (Stephens, 2008).

Relační schéma

Toto schéma definuje složení daného záznamu. Je to tedy kombinace dvojic domény a atributu.

Atribut

Atribut je sloupec v tabulce, který nám umožňuje určit vlastnost či datový typ entity v tabulce. Dále lze určit tzv. doménu, což je rozsah hodnot, které může sloupec nabývat (Ramakrishnan & Gehrke, 2003). Při tvorbě databáze se na základě této domény určí datový typ. SŘBD má poté za úkol zajistit, že do sloupce bude možné zapisovat pouze hodnoty, které vyhovují datovému typu, např: datový typ integer (INT) nebude do takového sloupce možné zapisovat čísla s desetinnou čárkou (Coronel & Morris, 2018).

Atomicita

Atribut sloupce u záznamu je vždy atomický, to znamená, že hodnotu nelze rozložit na další atributy, např: bankovní účet 235860686/0800.

Klíče

V relační databázi existuje řada klíčů, které mají své opodstatnění. Lze je chápat jako kombinace jednoho či více sloupců, které lze použít pro nalezení daného záznamu v tabulce. Například v tabulce studenti lze použít studentID pro nalezení daného studenta (Hernandez, 2013).

Composite key

Composite key je klíč, který obsahuje více jak jeden sloupec, například lze použít kombinaci jména a příjmení během hledání v tabulce studenti (Stephens, 2008).

Super klíč

Super klíč je kombinace sloupců v tabulce, pro které nelze mít dva záznamy se stejnými hodnotami. Tento klíč je také někdy nazýván jako unikátní klíč (Stephens, 2008).

Kandidátní klíč

Kandidátní klíč je sloupec, který identifikuje řádek v tabulce. Tento klíč může být použit jako primární klíč. Na druhou stranu klíče, které se nestaly primárním klíčem, se nazývají alternativní klíče (Coronel & Morris, 2018).

Primární klíč

Primární klíč umožnuje označit záznam unikátní hodnotou v celé tabulce. Sloupec, kde je primární klíč musí obsahovat nenulovou hodnotu. To znamená, že se ve sloupci nemůže záznam nabývat hodnotou *NULL*. Každá tabulka musí mít vlastní primární klíč a záznam musí obsahovat tuto hodnotu a musí být v celé relaci jedinečný (např. nelze mít stejnou imatrikulaci v letectví) (Ramakrishnan & Gehrke, 2003).

Cizí klíč

Tento klíč má jediný účel, a to pro vyjádření vztahů mezi tabulkami. Jedná se tedy o sloupec, který spojuje záznamy z různých tabulek navzájem (Ramakrishnan & Gehrke, 2003).

Propojení tabulek

Při návrhu databáze lze určit, které relace budou mezi sebou propojeny. Například databáze banky bude obsahovat tabulku pro klienty a tabulku pro účty. V tomto případě musíme popsat nejdříve vztah, jaký mezi těmito tabulkami probíhá, tj. klient John Doe si může pořídit několik účtů. Pro tento vztah existuje pojem kardinalita vztahů (Skřivan, 2008).

Vztahy mezi tabulkami

Určuje vztah záznamu například z tabulky č.2 se odkazuje na záznam v tabulce č.1. V příkladu uvedeném v sekci propojení tabulek je potřeba zjistit vztah, tedy zda jeden účet může patřit více uživatelům a zda jeden uživatel může mít více účtu. Tento vztah je potřeba určit a poté v SŘBD zapsat. Vztah klienta k bankovním účtům bude: 1:n, tj. jeden klient může mít 0 až n účtů, ale účet nemůže patřit více klientům (patří pouze jednomu). Pokud nastane situace m:n, je potřeba tento vztah rozdělit pomocí

pivota, kdy jedna tabulka propojí primární klíče z tabulky č.1 a primární klíče z tabulky č.2 (Tok Wang Ling, 2013).

3.5 DBMS

Je softwarové řešení, které nám umožňuje manipulovat s databází. Je to tedy abstraktní a aplikační vrstva oddělená od uložených dat. Je nutné, aby program byl schopen manipulovat s určitým množstvím dat a zároveň mít schopnost řídit a definovat strukturu dat (Chopra, 2016).

Každý software by měl umožňovat řadu operací, jako je spravování klíčů, kdy může obsahovat specifickou logiku. Spouštěče jako jsou *triggers* nebo i autentizace nad uživateli pro provedení operací nad daty. Dále by měl umožňovat integritu dat, kdy neumožní uživateli uložit nulovou hodnotu do nenuloveho sloupce a atomicitu (Coronel & Morris, 2018). Též by měl umožnit užívání jazyka pro práci s daty, jako je například SQL. Známe softwarové řešení je MySQL, PostgreSQL.

3.6 SQL

Je standardizovaný jazyk pro správu. SQL umožňuje manipulaci s daty obsažené v databázi. Je využíván programátory pro tvorbu aplikací. Díky SQL lze definovat také data. To znamená definování struktury tabulek, vkládání data (záznamů) do tabulek a určování vztahů a organizování vztahů mezi entitami tabulky (Beaulieu, 2005). Dále lze kontrolovat přístup k datům, tedy odebírání a přiřazování přístupových oprávnění jakýkoliv úrovních. Díky tomu je zajištěna ochrana dat proti úmyslným či náhodným zničením. SQL disponuje sdílením užívaných dat v databázi a umožňuje bezproblémový průběh operací, pokud k datům přistupuje více uživatelů současně (Sadowski, 2020).

SQL nelze řadit mezi plnohodnotné samostatné programovací jazyky. Lze to pocítit v situacích, kdy je absence řídicích programových konstrukcích a dalších požadovaných prvků v implementaci, které obsahuje každý programovací jazyk. SQL je tedy nástroj pro správu relačních databází. Nezastupuje databázový systém jako takový, ale je spíše integrovanou součástí systému řízení bází dat neboli SŘBD (Sadowski, 2020).

SQL je velmi srozumitelný jazyk, protože data interpretuje v podobě tabulek. Komunikuje s relačními databázemi, ve kterých jsou data interpretovány v podobě množin tabulek, které jsou navzájem propojeny. Tabulka obsahuje množinu dat. Pokud uživatel chce získat hodnotu uloženou v databázi musí se provést selektový výběr (Winand, 2012).

Výsledkem popsaným v SQL je určitá množina dat z jedné či více tabulek. Finální výstup z operace může sloužit jako množina vstupních údajů pro další operace. SQL může sloužit jako univerzální jazyk, především v provozu na serverech, na kterých se používají různé SŘBD, také tento jazyk umožňuje tvořit náhledy, čímž lze pro různé uživatele vytvořit množinu dat z databáze. Konečný výsledek pohledu je tedy tabulka. SQL, jakožto jazyk je rozdělen do čtyř částí:

- DDL
  - *Data Definition Language* – příkazy pro definici dat, jako je například CREATE TABLE či ALTER TABLE
- DML
  - *Data Manipulation Language* – příkaz pro manipulaci s daty, zde jsou nejdůležitější operace jako je SELECT N_1, N_2, N_3 ... FROM x WEHRE y.
- DCL
  - *Data Control Language* – příkaz pro řízení přístupových práv
- TCC
  - *Transaction Control Commands* – příkaz pro řízení transakcí

3.7 Objektová databáze

Databáze jejímž základním prvkem je prace s objekty. Umožňuje určitou syntaxi dotazů pro získání objektu z databáze. Tento způsob také umožňuje ukládat změny objektů zpět do databáze. Objektová databáze je označován někdy též jako objektově-orientovaná databáze či *object database managment systems (ODBMS).* Tento způsob dává smysl tam, kde vývojové prostředí a architektura je založena na objektech. Není potřeba pracovat s komplexní dotazem nad daty (Stephens, 2008).

Oproti relačnímu paradigma, je zde finální výstup zabalen do objektů, které úzce souvisí s objektovými jazyky jako je například Java, csharp či PHP. Existují

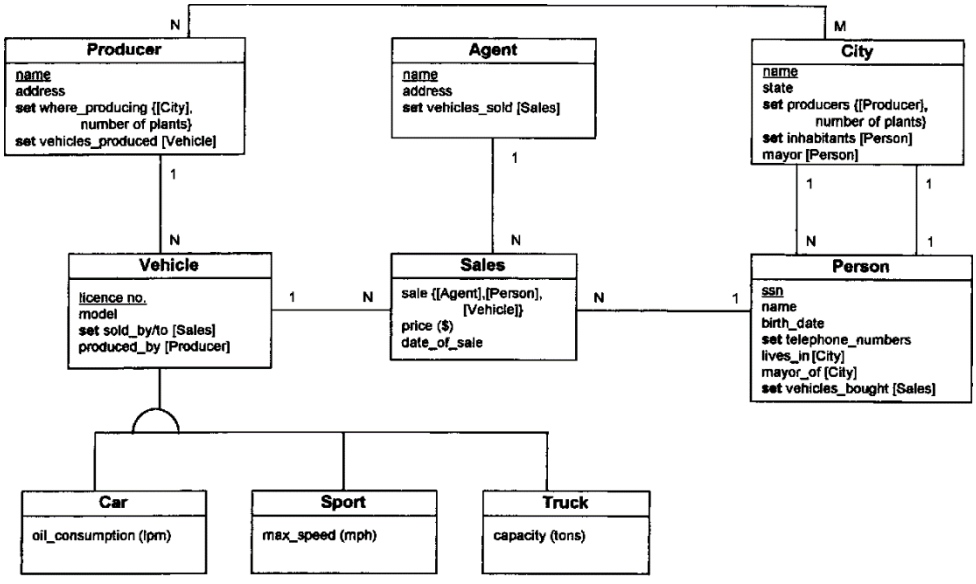
několik kritérií pro ODBMS, jako je například perzistence dat, která umožňuje ukládat data na dlouhou dobu. *Concurrency* je schopnost, která umožnuje přístup více uživatelům. Dále musejí nabízet obnovitelnost dat, která by se mohla při kolizi systému ztratit (Holub, 2007). Musí mít tedy schopnost je obnovovat do určité podoby.

Dále jsou zde požadavky z pohledu OO jako je například.

Kompozice, kdy komplexní objekt obsahuje a je složen z menších objektů či primitivních datových typů, jako je například *integer* či *string*. Dále by měl umožňovat zapouzdření, kdy data (stavy) a metody (operace) objektů jsou před klientem skryty. Také by měl umožňovat tvorbu tříd, díky kterým lze objekty vytvářet a sjednocovat do kolekcí. Dále by měla být podpora dědičnosti pomocí hierarchie, kdy dítě obdrží chování a strukturu rodiče. Poslední důležitou věcí je přetížení metod, kdy lze měnit chování zděděné metody dle požadavků (Holub, 2007).

Datový model

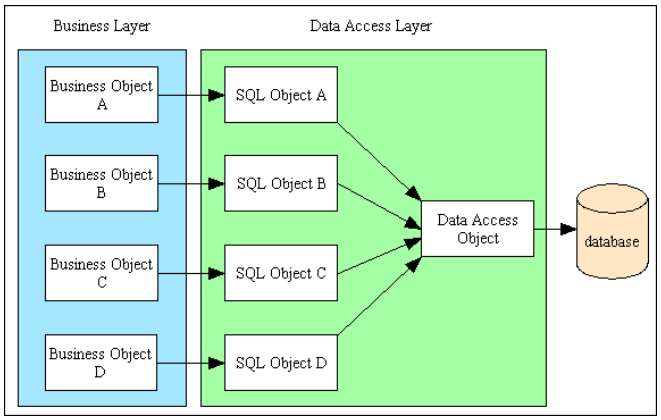
Tento model obsahuje několik prvků. Jedním z nich je objekt. Objekt je tedy instance určité třídy. Objekt obsahuje vlastní atribut a typické chování dle třídy. Dále lze vidět na obrázku 5 dědičnost, která znázorňuje vztah a chování objektu s rodiči. Poslední prvek je třída, která obsahuje strukturu složenou z atribut a také je složena z metod (Holub, 2007).



Obrázek 5- OO diagram (zdroj: https://www.semanticscholar.org/paper/Entity-Relationship-and-Object-Oriented-Data-of-Shoval-Shiran/add1b051fcf6094a2e2e7d586fc7495c802a399f)

3.8 Datová vrstva

Třívrstvá architektura obsahuje nejen aplikační a prezenční vrstvu, ale také datovou vrstvu, která je reflektována pomocí modelu (Marston, 2003). Tento model tedy odděluje byznys logiku od přímé komunikace s databází, který lze vidět na obrázku 6.



Obrázek 6- Popis datové vrstvy s využitím DAO (zdroj: (Marston, 2003))

Samotné PHP, tedy nabízí jednotné API rozhraní pomocí ucelené knihovny PDO, které lze užít pro jakýkoliv databázový systém (Moravec, 2009). PHP nelze zkompilovat, tudíž lze měnit DSN neboli *data source name* pro připojení k databází. Důvod tvoření abstrakce před SQL, je robustnost dotazů v SQL jazyce.

Tato abstrakce jako nabízí PDO, řeší problematiku jako je *SQL injection*, tvorba a složení komplexních dotazů. PHP a objektové programování nabízí také abstrakci v podobě návrhových vzorů, které popisují přístup a manipulaci s daty z databáze (Moravec, 2009). Neboť knihovna PDO získá data, ale v základním stavu. Tyto data získávají význam až na aplikační úrovni. Pro tento mezičlánek existuje tedy ORM, který dokáže díky *mapping* spojit relační databází s objekty v daném objektově orientovaném jazyce.

3.9 ORM

Objektově relační mapování v informatice je technika, kdy lze převést data různými systémy. Je to tedy proces mapování objektů do relační databáze. Data, které jsou uloženy v databázi lze získat pomocí objektového jazyka. Výsledky nejsou uloženy v objektech, ale jako surová data získány do pole, které pak musí být zpracovány do objektů. Získaná data z databáze jsou vloženy do pole či kolekce objektů, které jsou instancovány podle náležité třídy.

Při tvorbě aplikace je důležité zachytit veškeré prvky, které reflektují realitu. Objekt je poté v aplikaci označen jako *Entita*. Řádek z relační databáze je poté zpracován jako entita a tato entita je z pravidla reprezentována jako instance třídy (Dunglas, 2013).

Díky rozdílným entitám vznikla technika pro konverzi mezi relační databází a objekty, které jsou nadále zpracované OO jazykem. ORM poté umožňuje programátorovi přístup k danému objektu, se kterým může dále pracovat. Díky tomu je programátor při manipulaci s daty z databáze odkazován na objekty, které byly v aplikace vytvořené či zpracované do Entit. Do jisté míry je programátor distancován od užívání SQL dotazů. Pomocí ORM lze jednoduše provádět operace nad databází, což je čtení, mazání či modifikování dat. Dále je nutné umožnit persistenci dat (Mokruša, 2015).

Dále obsluhuje konverzi různých datových typů mezi OO jazykem a SŘBD. Tato technika dále umožňuje synchronizaci mezi vytvořenými objekty ve vývojovém prostředí a jejich reprezentací v SŘBD tak, aby nedocházelo k redundanci dat a umožnovala perzistenci dat. Mnoho ORM knihoven umožňuje odstínit programátora od vytváření SQL query dotazů do databáze. Některé implementace ORM obsahují vlastní dotazovací jazyk pro získaní objektů z databáze, například v Doctrine je to DQL (Mokruša, 2015).

Jedna z výhod ORM je nezávislost webové aplikace na určitém databázovém systému, což umožňuje zvolit jiný SŘBD, aniž by došlo k rozbití či poškození dané aplikace.

Konverze mezi objektem a relační databází se pojí s řadou problémů, které se označují za *object-relational impedance mismatch.* Dále existují několik návrhových vzorů pro práci s objekty pomocí ORM techniky.

- Table Data Gateway
- Row Data Gateway
- Active Record
- Data Mapper

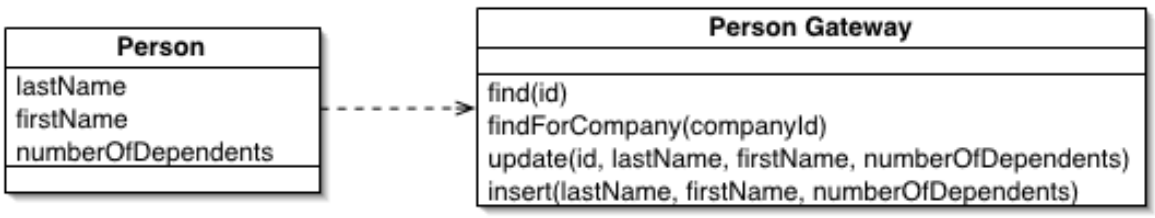
```
SELECT `name`, `school_email`, `age`
FROM `students`
WHERE `age` > 21
ORDER BY `name` DESC

SELECT s
FROM Namespace\Student s
WHERE s.id > 10
ORDER BY s.name DESC
```

Obrázek 7- ORM dotaz a SQL dotaz

3.9.1 Table Data Gateway

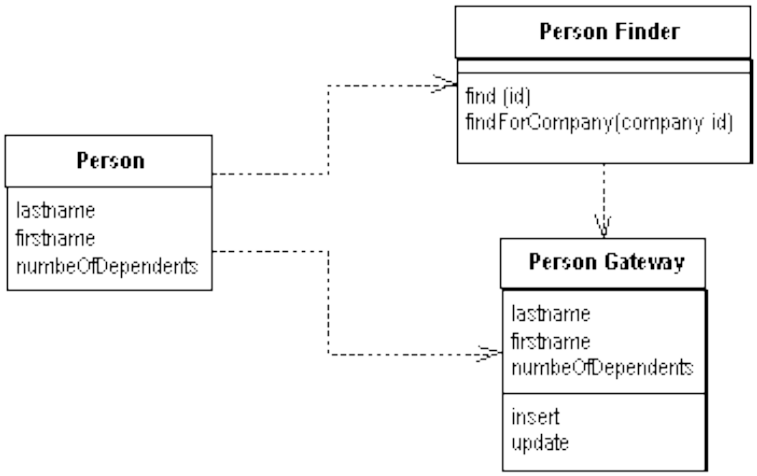
V tomto paternu se jedná o prostředníka či instance třídy (objekt), který využívá jeden z principů OOP, a to zapouzdřený přístup k určité tabulce v relační databázi. Tato instance musí znát atributy, datové typy určité tabulky, díky tomu může provádět operace nad nimi. Daný objekt může poté zpracovat základní operace jako je insert, select, find (s argumentem id integer), update nebo delete, tyto metody lze vidět na obrázku 8. Tyto metody jsou poté reflektovány v SQL jako insert, select, update či delete (Moravec, 2009).



Obrázek 8- Popis Table Data Gateway paternu a jeho interakce v kódu (zdroj: (Fowler, 2019))

3.9.2 Row Data Gateway

Je to návrhový vzor, který zapouzdřuje operace objektu, jako je find, update či insert nad entitou či řádkem z relační tabulky. Objekt Gateway je tedy reflektován jako specifický záznam v tabulce. Pomocí objektu *Finder* na obrázku 9 lze získat příslušný záznam a poté je „předán“ objekt Gateway, s kterým lze dále manipulovat (Mokruša, 2015).



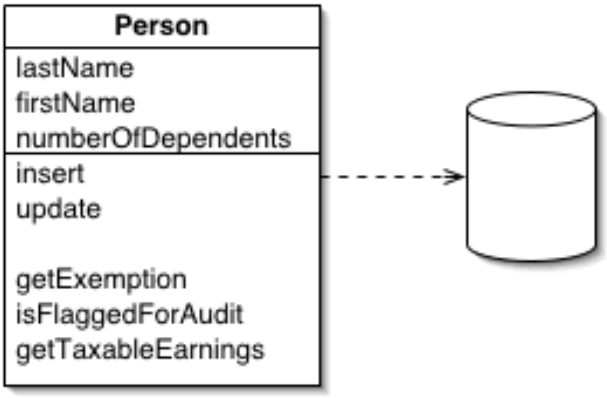
Obrázek 9- Patern Row Data Gateway a jeho popis na objektech (zdroj: (Fowler, 2019))

3.9.3 Active Record

Je velmi populární návrhový vzor a hlavně nejvyužívanější v ORM technice. Je to tedy obdoba Row Data Gateway, ale přidává navíc aplikační logiku (Moravec, 2009). Fowler zde popisuje tento patern, kde objekt obsahuje data a chování byznys

logiky, které je zapouzdřeno a je poté potřeba tento objekt uložit do databáze, kterou lze vidět na obrázku 10.

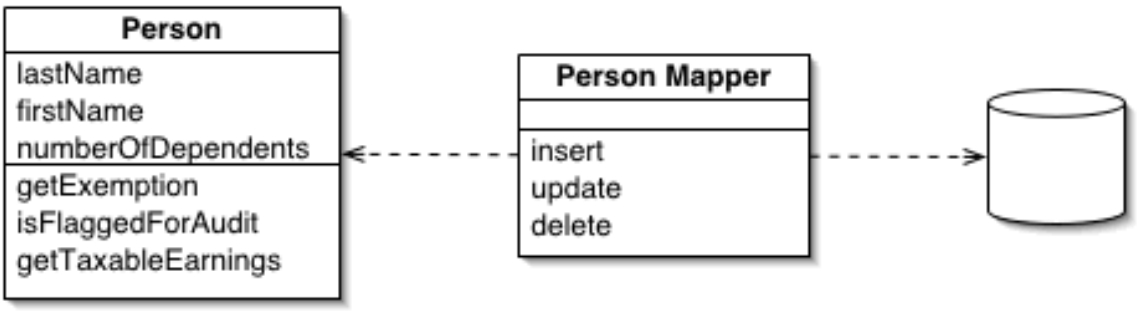
Objekt implementující tento patern by měl umožnit tvorbu instance ze záznamu, dále tvorbu a úpravu záznamu či jejich hledání. Z pohledu objektového programování by měl dále nabízet Get a Set metody (Fowler, 2019).



Obrázek 10- Patern Active Record a zobrazení jeho metod a atriutů (zdroj: (Fowler, 2019))

3.9.4 Data mapper

Je patern, kdy objekt nemá stejnou struktura jako tabulka v relační databázi nelze takto uložit. Vzor Data Mapper je software vrstva, která odděluje objekty od databáze (Moravec, 2009). Jeho účel je tedy přenést data mezi databází a objektem a izolovat jejich komunikaci mezi nimi, to lze vidět na obrázku 11. Data Mapper umožňuje objektům odstínit SQL rozhraní a strukturu databázového schématu. Mapper vyžaduje k práci veřejné metody dané entity (Fowler, 2019).

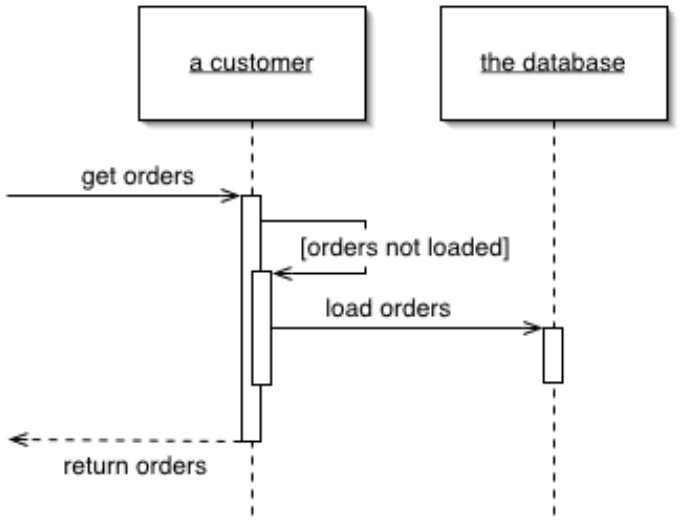


Obrázek 11- Data Mapper a jeho popis fungování (zdroj: (Fowler, 2019))

3.9.5 Lazy loading

Tato metoda umožňuje načítat data z databáze přímo do paměti. Projevuje se v případě, kdy načteme objekt, který nás opravdu zajímá. Díky této metodice se také načtou objekty, které jsou spjaté s daným objektem (Mokruša, 2015). Tento postup hraje důležitou roli ve výkonu a optimalizaci aplikace, která pracuje s relační databází, neboť v situacích, kdy je načítáno velké množství dat s relacemi vnořených objektů, výkon naší aplikace rapidně klesá a ve většině případů nejsou ani tyto atributy dále využity (Mokruša, 2015).

Tento postup provádí operace ve chvíli, kdy jsou data entity opravdu zapotřebí naší aplikací. Je tedy vhodný v situacích, kdy objekt načítá své relace, které jsou dost velkým výkonnostním faktorem na aplikaci (Fowler, 2019). Celý proces lze vidět na obrázku 12.

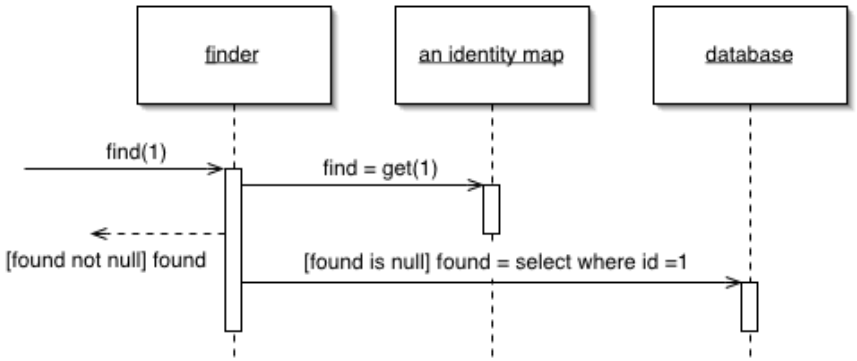


Obrázek 12- Lazy Load a jeho interpretace (zdroj: (Fowler, 2019))

3.9.6 Identity map

Předchází výkonnostnímu problému, kdy jsou načteny data více než jednou. V aplikaci lze vytvářet stejnou entity a pracovat s ní na různých místech. Její změna na jednom místě se nemusí projevit i na dalších místech. Identity Map si udržuje záznam o všech objektech, které byly získány z databáze. Pokaždé, kdy aplikace bude chtít daný objekt, vždy projde tímto procesem a zjistí se, zda už neexistuje daný objekt. Entity jsou tedy rozlišeny klíčem. Zde lze vidět na obrázku 13, kdy objekt *finder*

pomocí metody find s parametrem 1 hledá objekt. Nejdříve zkontroluje Identity Map, pokud zde nic nenajde, získá záznam z databáze, který se uloží do této mapy a poté se vrátí k objektu *finder*. Pokud v mapě již existuje, vrátí se tento objekt k finderu (Fowler, 2019).



Obrázek 13- Identity map a popis jeho fungování (zdroj: (Fowler, 2019))

3.9.7 Unity of Work

Je to design pattern, který je využíván k udržování stavů objektů, které jsou zpracovávány Data Mapperem. Databázová synchronizace se děje v situacích, kdy je volána z entityManager metoda, jako je například flush a je poté zpracován v transakci. To znamená, že pokud dojde k chybě, zatímco se synchronizuje entita do databáze, veškeré úkony se vrátí do původního stavu před touto synchronizací (Fowler, 2019). Díky implementaci tohoto paternu lze například v Doctrine minimalizovat dotaz do jednoho SQL dotazu zobrazeného na obrázku 14 a 15.

```
$ourEntity->name = 'Filip';
$ourEntity->name = 'Adam';
$em->flush($ourEntity);
```

Obrázek 14- Příklad popisující Unit of Work patern

```
UPDATE OurEntity SET name="Adam" WHERE id=1;
```

Obrázek 15- Optimalizovaný dotaz pomocí Unit of Work

3.10 Doctrine 2

Doctrine 2 je *ORM* knihovna pro PHP. Je to knihovna, která umožňuje využít abstrakci objektového programování od psaní čistých SQL dotazů. Slouží k propojování relací databáze s objekty. Pomocí *EntityManager*, lze získávat hodnoty či entity bez pomoci dotazovacích jazyků, díky základním metodám jako je find či findAll. (Doctrine, 2021).

V situacích, kdy je potřeba provádět složité operace nad daty, nám Doctrine 2 nabízí vlastní řešení v podobě dotazovacího jazyka DQL. Doctrine sice nepřichází v balíčku společně s Nette, ale lze ji nainstalovat přímo do existujícího projektu pomocí *composeru* (Tichý, 2010).

Doctrine 2 dokáže komunikovat s databází pomocí *Data mapper*. Tento patern nám pomáhá pracovat s daty z databáze. Doctrine 2 umožňuje přístup k relační databázi prostřednictvím objektového programování. Každý model vytvořený v aplikace lze nastavit vazbu na tabulku v databázi a instance modelu poté prezentuje jeden řádek v tabulce, atributy modelu jsou reflektovány jako sloupce tabulky. Jakmile instanci vytvoříme a uložíme, přibude jeden nový řádek do databáze. Pokud budeme chtít číst určitý řádek, lze si načíst "instanci" třídy, která odpovídá tabulce v naší databázi (Dunglas, 2013). ORM musí vyřešit několik problémů, jedním z nich je *Impedance mismatch* mezi objektem modelu a databází. *Mismatch* problém vzniká v důsledku způsobu, jakým jsou data uložena a reprezentována v relační databáze a OOP. Celý balíček je inspirován Hibernate v Java. Definování vlastních entit se využívají komentářové anotace (Dunglas, 2013).

Díky Doctrine 2 není potřeba řešit databázovou strukturu, neboť si sama generuje na základě entit strukturu. Doctrine 2 místo SQL používá DQL, kdy se v dotazech zapisují místo názvů relace/tabulky a atributu spíše názvy entit a atributy této

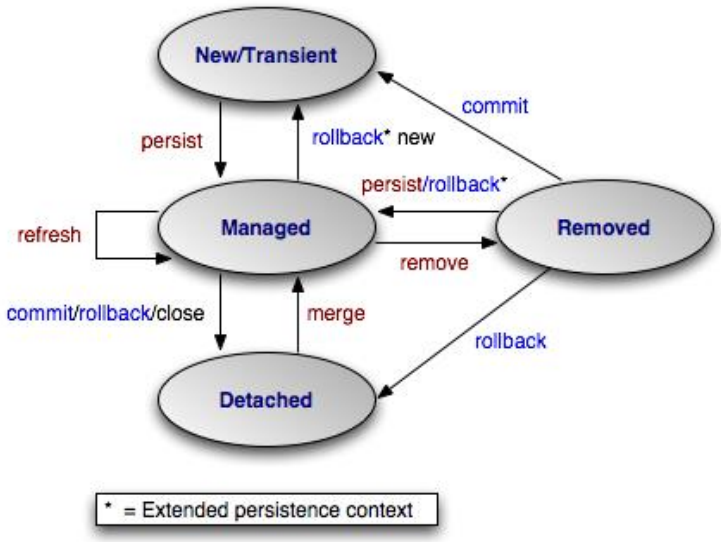
entity. Snaží se být tedy rychlým řešením, a tudíž se tato knihovna snaží cachovat vše potřebné k optimalizaci výkonu.

Doctrine 2 lze rozdělit do 3 vrstev:

- DBAL neboli DataBase Abstraction Layer
  - Abstrahuje aplikaci od konkrétního SŘBD. Je to rozšíření PDO, ale tato vrstva podporuje i jiné SŘBD řešení. Tato vrstva zavádí též DQL.
- Common
  - Základní rozhraní pro třídy a knihovny. Nástroje pro práci s collections, anotacemi či cachovaním.
- ORM (Object-Relation Mapping)
  - Nejabstraktnější vrstva zajišťující mapování objektu v aplikaci na naší relační databázi. Zajišťuje persistenci dat. Tato vrstva je závislá, jak na DBAL tak i na Common (Tichý, 2010).

3.11 EntityManager

Po celé tvorbě entity, je *entityManager* fasáda, která lze použít jako spoj mezi aplikací a databází. Tato fasáda umožňuje pomocí stavů objektů ovlivňovat data v databázi. Díky této *service* lze tedy vkládat či mazat entity v databázi.



Obrázek 16- Stavy entity (zdroj: https://i.stack.imgur.com/XLzpf.png)

Na obrázku 16 lze vidět stavy konkrétní entity a její cyklus během manipulování s *entityManager*. Pokud budeme naší novou entitu ukládat, použije se metoda *persist*. Pro opačný příklad, kdy budeme chtít určitou entitu odebrat z databáze stačí užít metodu *remove*, kdy argument bude entita a přejde tedy do stavu *Removed*.

3.12 Perzistence dat

Bohužel objekty jsou dočasně uloženy v RAM. V Objektovém programování se počítá s dočasnou manipulací objektů, které se ztratí po vypnutí počítače. Bohužel OOP nenabízí žádný mechanismus pro tvorbu persistenci dat. Kvůli tomu je potřeba například v PHP připojit k databázi například přes PDO, pomocí kterého lze uskutečnit persistenci (Tatroe & MacIntyre, 2020). Relační databáze je v dnešní době nejvíce užívaná databázová technologie. Existuje mnoho produktů či softwarového řešení založené na relační databázi například SŘBD. Mezi takováto řešení lze označit PostgreSQL či MySQL.

Každý takovýto produkt je založen na SQL. Toto paradigma aplikuje teorii množin. Díky tomu lze objekty aplikace prezentovat jako tabulky, kde sloupce jsou odpovídající atributy objektů a řádky jsou již samotné objekty. Vztahy objektů v aplikaci jsou reprezentovány jako klíče, a jejich reference jsou reprezentovány jako cizí klíč. Normalizace je souhrn konceptů pro chránění či zachování dat a zároveň eliminovat redundanci dat a eliminovat také nesrovnalosti v datech (Pravda, 2007).

3.13 Framework

Je platforma pro tvorbu softwarových a webových aplikací. Framework nabízí základy, díky kterým může developer tvořit aplikaci rychleji a zaměřovat se na specifické úkony. Například může mít předdefinované třídy a funkce, které komunikují se soubory.

Díky tomuto nemusí programátor vytvářet danou věc opakovaně a může se zaměřovat na určitou problematiku (Tatroe & MacIntyre, 2020). Framework lze porovnat k *Application Programming Interface* tzv. *API*. Nette, Laravel či Symfony obsahují API. Tudíž Framework poskytuje základy pro programování a *API* poskytuje přístup k určitým elementům. Framework může též obsahovat knihovny a další jiné programy pro tvorbu aplikace.

Framework jako takový je postaven na knihovnách, není to tedy samotná knihovna. Rozdíl mezi knihovnou a frameworkem je ve volání. Curl je knihovnička v PHP. Když zavoláme jednu z jeho funkcí, tak náš kód volá kód knihovny, ale pokud toto provedeme ve frameworku tak vztah je v tu chvíli obrácený, neboť framework volá veškeré dependency/kód ve frameworku, tomuto cyklu se říká *IoC*. Programovací jazyk říká počítači, co musí vykonat za práci. Framework je nástavba dnešních jazyků pro zkrácení duplicitních kódu (Lethbridge & Laganiere, 2005). Existuje několik druhu frameworků pro různé účely v programování.

- Web Application
- Data Science
- Mobile

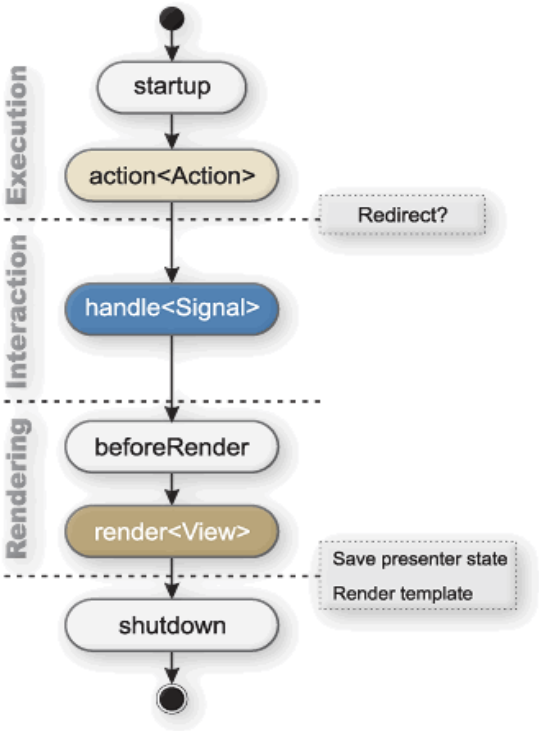
3.14 MVP Nette

Webová aplikace v dnešní době stojí na třech hlavních komponentech, které zastupují určitou logiku. První z těchto komponent je presenter, která komunikuje s uživateli. Uživatel přepošle parametry této komponentě, ta následně vrátí určitou HTML šablonu. Presenter ve většině případů předá parametry s*ervice* či *modelu*, od kterých získá response v podobě dat. Tyto data jsou následně vloženy do šablon. Tato šablona není nic jiného než HTML s vloženými PHP tagy. Presenter poté pošle tuto HTML stránku zpět uživateli. Další komponenta je model, který popisuje logiku aplikace. To zahrnuje práci s databází či soubory. Každá entita má svůj datový model, například UserModel či StudentModel. Poslední komponenta je Pohled, v tomto případě *latte* šablona. Je to tedy šablonovací jazyk, pomocí kterého lze vkládat logiku do HTML kódu.

3.15 Koloběh requestu v Nette

Klient či browser zadá URL, ten se vytvoří do HTTP requestu. HTTP request se pošle na server, kdy máme několik metod *GET, PUT, PATCH, DELETE, POST*. Poté request „míří“ do souboru index.php v *www/* adresáři. Inicializuje se prostředí, získá továrnu. Spustí se poté aplikace, kde router zpracuje request. Ten podle adresy zjistí, jaký presenter a metoda má být volána na základě této URL adresy a předá jí již zbytek adresy. Presenter může přečíst jaké parametry request obsahuje a jakou logiku má vykonat. Pomocí modelu, ke které má přístup zjistí potřebné údaje z databáze. Poté presenter pošle tyto údaje do *latte* značek, které díky *Latte engine* obstará jejích funkčnost. Server následně pošle response v podobě HTML stránky (Čápka, 2015). Zde lze vidět cyklus presenteru, který je zobrazen na obrázku 17.

- Execution (startup a action)
- Interaction (handle)
- Rendering (beforeRender a render)
- Shutdown



Obrázek 17- Cyklus presenteru (zdroj: (Nette, 2021))

Startup

Po obdržení požadavku se zavolá tato metoda, kterou můžeme využít k ověření uživatelských oprávnění.

Action

Metoda zpracovává požadavek bez vykreslování template. Tato metoda se spouští dříve než Render, takže v této metodě lze nastavit i jinou šablonu pomocí setView. Zde v této metodě budou zpracovávány testovací query dotazy do databáze v praktické části.

Handle

Metoda, která zpracovává signály, je zaměřena na práci s komponenty a zpracování AJAX požadavků.

BeforeRender

Tato metoda se volá před každou render metodou. Používá se pro předání proměnných pro layout.

Render

Je metoda, kde připravujeme šablonu pro vykreslení a předáváme jí data.

AfterRender

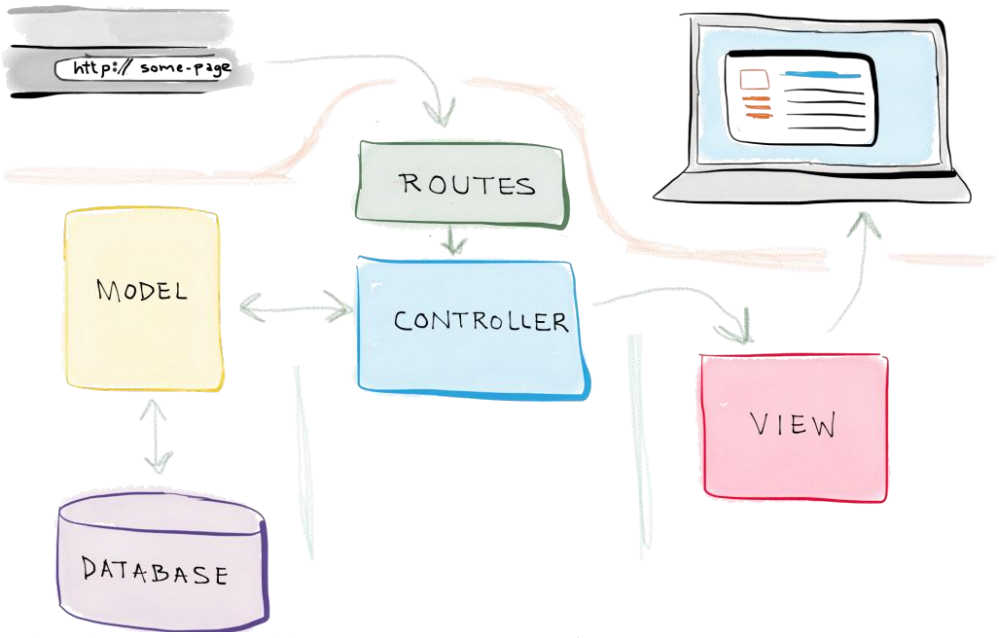
Metoda, která se volá po každé render metodě.

Shutdown

Metoda, která se volá na konci cyklu presenteru.

3.16 Koloběh MVC architektury

Požadavek je odeslán pomocí *request* a přijímán na web serveru (Nette). Požadavek je odeslán přes *router* pomocí *HTTP* s určitou metodou. *Router* obdrží požadavek a přepošle příslušnému *presenteru* a specifické metodě. *Presenter* komunikuje s Modelem a Model komunikuje s databází. Databáze přijímá data z Modelu a vrací odpověď. Model poté předá *response Controlleru*. *Controller* přepošle pohledu (*View*) výsledky podle byznys logiky (Gourley & Totty 2002). Pohled je vygenerován v prohlížeči uživatele. Návrh a celý koloběh lze vidět zde na obrázku 178.



Obrázek 18- Koloběh Model-View-Controlleru (zdroj: selftaughtcoders.com/web-application-pages-mvc-bootstrap/)

3.17 Nette Database

Nette umožňuje rozhraní pro práci s tabulkami, kde lze lehce řetězovat query dotazy a díky tomu se vyhnout čistým SQL dotazu. Tato vrstva umožňuje získat data z databáze jednoduše pomocí abstrakce. Myšlenka je získávat data z jedné tabulky pouze jednou do *collection* s názvem ActiveRow (Nette, 2021). Konfigurace tohoto *extension* je snadná, např. takto na obrázku 19.

```
$database = new Nette\Database\Connection($dsn, $user, $password);
```

Obrázek 19- Konfigurace Nette Database

Na tomto obrázku 20 vidíme objekt database, která je instance obdobná PDO. Jádro funkcionality je zpřístupněno pomocí třídy *Nette\Database\Context*. Poté pomocí *Nette\Database\Table* vrstvě, která vylepšuje dotazování na tabulku. Veškeré připojení je vytvořeno tedy jako „lazy“. To znamená že připojení k databázi se provede jen tehdy kdy je opravdu potřeba, toto chování lze vypnout pomocí *„lazy=> FALSE”* v konfiguraci. Díky třídě context lze používat metodu query s protekcí proti *SQL injection*.

```
$database->query('SELECT * FROM users W
HERE email = ?', $email);

$database->query('SELECT * FROM users W
HERE email = ? AND active = ?', $name,
 $active);
```

Obrázek 20- Použití Nette Database

V prvním dotazu se vyberou veškerá data z tabulky *users*, kde email použije *wildcard* “?” a dosadí se pomocí proměnné *$email*. Druhý dotaz má další omezující podmínku, a to zda je uživatel aktivní či nikoliv. *Wildcards* se používá jako prevence před SQL injection, kdy je vsunut kód do neochráněného dotazu (Nette, 2021).

Zde stojí také za zmínku třída *Nette\Database\Table*. Je to vrstva umožňující komunikovat s databází jednodušeji. Hlavní důvod této třídy je získat data pouze z jedné tabulky a pouze jednou. Data jsou poté vložena do *ActiveRow* instance. Data z jiných tabulek, které jsou propojeny mezi sebou jsou volány jiným dotazem, který obstarává *Database\Table* (Nette, 2021).

3.18 Modularita

Schopnost spravovat a udržovat webovou aplikaci lze umožnit pomocí modularizace projektu. Podle Roba Pike je klíčem modularity jednoduchost k úpravě jakékoliv části software k zvýšení výkonu či k lehké opravě chyb. Byznys logika a její objekty musí být navržené s určitou pečlivostí tak, aby tuto schopnost umožnovali. Jelikož modularita hraje důležitou roli v udržitelnosti webové architektury. Musí být dosaženo nízké závislosti mezi objekty (Harle, 2010).

4 Vlastní práce

Projekt byl založen na výkonnostním ohodnocením NDB a ORM, dle jejich výkonů. Cíl byl tedy tento postup implementovat několikrát nad ORM a nad Nette DataBase technologiemi. Pro každou implementaci byly zpřístupněny tři objektové stromy získané z relační databáze, které reflektovaly složitosti dotazů. Na základě výsledků se porovnaly technologie a určovalo se, která technologie je ideální pro dané typy projektů.

4.1 Diagram tříd

Na diagramu 1 je prezentován diagram tříd bankovní aplikace. Byl nakreslen pomocí aplikace pro vizualizaci daného problému. Důležité v tomto obrázku jsou vztahy, kdy jeden klient má několik bankovních účtů. Každý bankovní účet má několik prémiových výhod. Prémiových výhod je několik druhů.

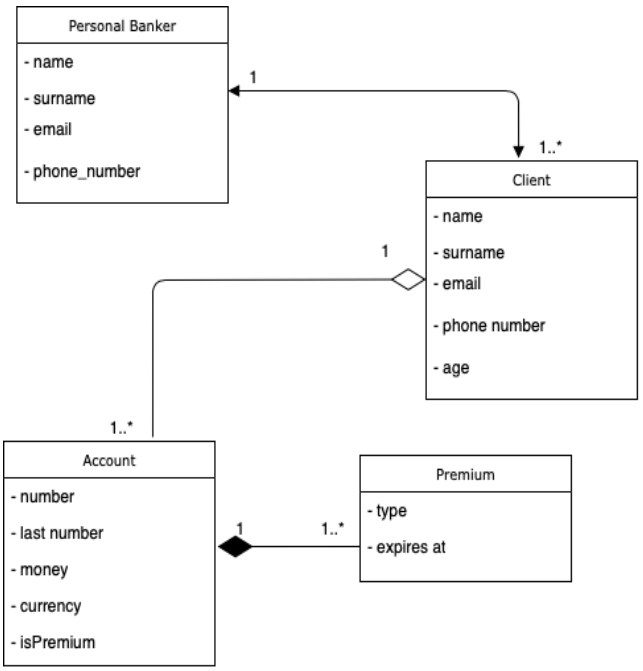


Diagram 1- popis diagram tříd

4.2 Databázová struktura

Podle předchozího diagramu má databáze čtyři tabulky, každá pro jednu třídu. V tomto projektu se použil vztah *many-to-one*. Díky této kardinalitě, *many-to-one* je tento vztah aplikován na *account-client*, kdy jeden klient má více řádku s různými účty. Na diagramu 2 vidíme databázový diagram.

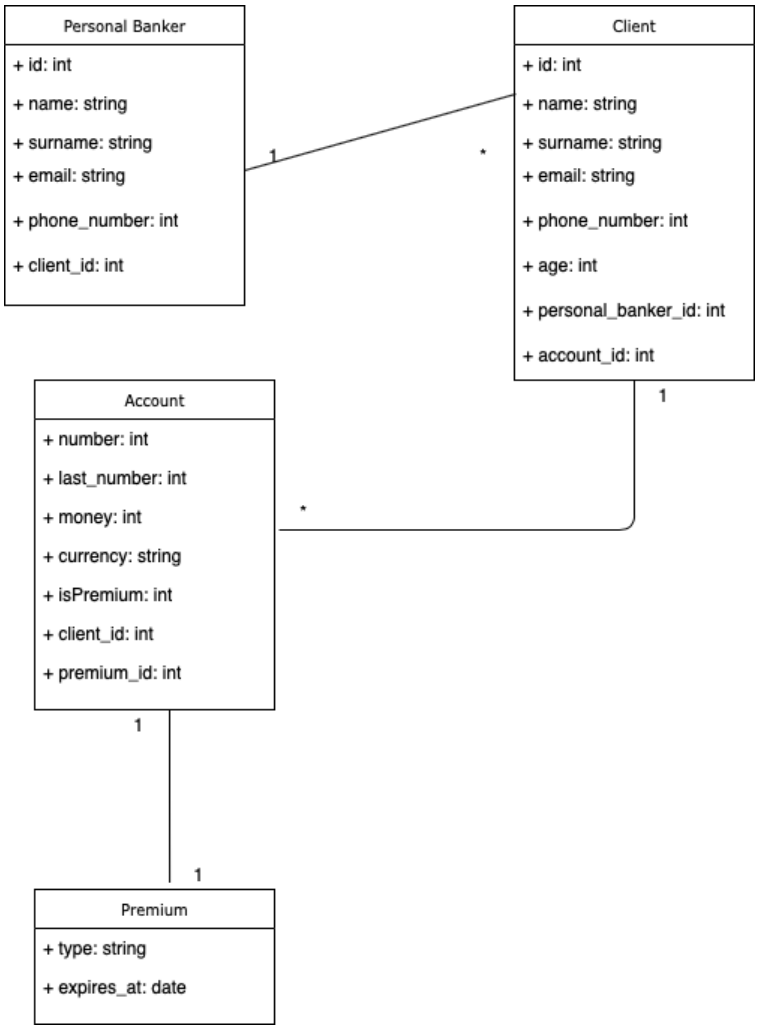


Diagram 2- popis databázového diagramu

4.3 Komplexita dotazů

Výsledek získáme pomocí čtení z databáze a následnou proměnou získaných dat do sady objektů. Na základě tohoto procesu se nadefinovaly tři komplexní dotazy. Každý komplexní dotaz reflektuje jeden diagram, které jsou zobrazeny na diagramu 3. Každý diagram reprezentuje hierarchickou strukturu objektu, které jsou získané pomocí komplexního dotazů.

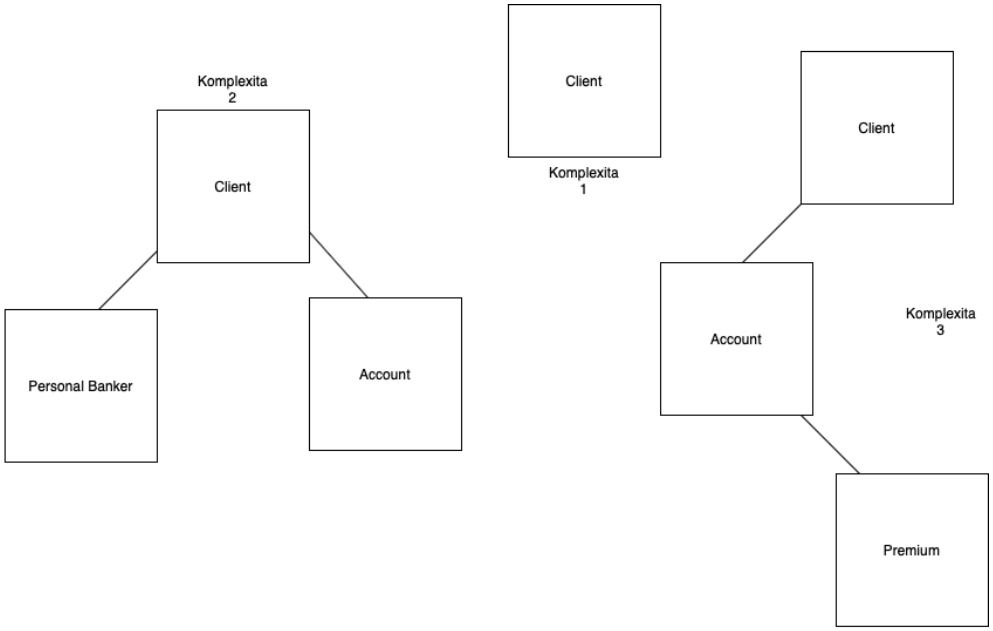


Diagram 3- úrovně složitosti

4.3.1 První složitost – Klient

Nejnižší a zároveň nejjednodušší dotaz. Na této úrovni lze získat list klientů banky, v podobě objektů či řádků. V tomto projektu je tedy tento dotaz označován, co se týče složitosti, jako nejjednodušší, neboť čte z databáze pouze z jedné tabulky a vytvoří také pouze jednu kolekci objektů nebo pole záznamů.

4.3.2 Druhá složitost – Účet a Osobní bankéř

Tato úroveň je mírně komplexnější, neboť lze získat kolekci klientů, osobních bankéřů a účtů v podobě objektů nebo jako pole řádků. Tento dotaz bude čerpat ze tří různých tabulek, které jsou navzájem propojeny.

4.3.3 Třetí složitost – Účet a Premium

Tato složitost je ta nejkomplexnější. V tomto dotazu lze získat opět kolekci klientů, osobních bankéřů, účtů a zároveň prémiových typů těchto účtů, jak lze vidět na diagramu 3. Tato úroveň je komplexní z několika důvodů. Musíme nejdříve vyselektovat účty, které mají další referenci a to na premium tabulku. Tento proces může mít delší odezvu, pokud databáze obsahuje veliký počet záznamů nebo dotazů, které nejsou zoptimalizovány.

4.4 Postup a testování

V souladu s cílem projektu bylo nutné dodržet určitá opatření, aby velikost databáze a složitost dotazů byly jediné proměnné v měření výkonnosti při zpracování dotazu. Jeden z hlavních kroků bylo dodržování postupu dle oficiální dokumentace a nastavení konfigurace všech technologií použitých pro bankovní aplikaci, zároveň konfigurace pro náš SŘBD. Dle doporučení oficiální dokumentace Doctrine2 s verzí 2.8.0 bylo implementováno vše v anotacích. Dále verze pro PHP byla vybrána 7.4, která podporuje jak objektové přístupy, tak i připojení k databázi. Při konfiguraci bylo zjištěno, že SŘBD zlepšuje rychlost provedení query dotazu, kdy stejný výsledek je získán za kratší čas. Kvůli tomu bylo třeba brát při měření výkonnosti vše v potaz, neboť doba exekuce se měnila při opakovaném spuštění. Také je výkon ovlivněn cache systémem a to u všech technologií, které byly využity. Tudíž bylo potřeba otestovat několikrát po sobě veškeré dotazy do databáze a poté zprůměrovat výstupy tyto dotazy.

Abychom se vyvarovali těmto proměnným, bylo zapotřebí do dotazů, které se několikrát spouštěly, vždy pozměnit dotaz. Ale zároveň bylo potřeba zachovat stejnou množinu výstupů. Tím jsme omezily dotazy z cache databáze. Každá kombinace proběhla dvacetkrát, jak pro ORM, tak i pro NDB. Každý testovací dotaz byl spouštěn na třech velikostí záznamů či objektů:

- 1000 jednotek
- 2000 jednotek
- 3000 jednotek

Průměrná doba odezvy je užitečná v odhadu doby, za kterou bude aplikace schopna zpracovávat vstup (dotaz) v reálné produkci. Myšlenka této analýzy bylo napodobit chování webové aplikace, kterou použije současně několik klientů naráz.

Každý test proběhl následovně:

- Každé spuštění bylo prováděno na jednom serveru
- Každý test použil stejnou databázovou strukturu
- Každý test obsahoval stejná data, během složitějších testů se zvyšovala pouze kvantita dat a složitost
- Operace byla spuštěna dvacetkrát a následně byl výpočet kvantifikován na základě průměru *response time*
- Po insert testu se musela smazat celá databáze a veškeré cache či dočasné soubory, které by mohly ovlivňovat další testování.

Také bylo nutné v PHP nastavit několik přídavných modulů (*extensions),* které podporují komunikaci s databází. Následně v projektu musel být použit parametr *memory_limit*, který nastavuje skriptu velikost paměti, kterou může PHP alokovat. Je důležité zmínit, že při implementaci testovacích use casů se drželo dle definovaných podmínek a metodik. Tato možnost byla použita při skládání složitých ORM objektů, které byly navrženy v sekci 4.4. MySQL ve verzi 8.0.19 byla vybrána jako DBMS (DataBase Managment Systém), kde bylo nutné předejít cachování, a proto bylo nutné veškeré dotazy pozměnit, ale zachovat jejich výstup stejný. Každá operace byla s danou složitostí opakována dvacetkrát kvůli budoucímu výpočtu průměru a následné analýze.

4.5 Technologické podmínky

Tato studie bylo opřena od reálného projektu pro klienta, kde bylo použito ORM a obdobné SQL dotazy. Nicméně většina web serverů, database serverů běží na Linux prostředí. Aby tedy testování bylo validní a mohlo být užito pro reálné situace, musel tento projekt fungovat na Linux prostředí, přesněji na Ubuntu 20.04. Projekt byl vypracován na Intel Core I5-33 CPU@3.7Ghz x4. Projekt funguje na nově instalovaném operačním systému.

Ke zvýšení kvality tohoto porovnávání bylo vynaložené veškeré úsilí na snížení vedlejších vlivů během získávání výsledků z databáze. Dodržovaly se určité strategie. Všechny strategie byly použity na stejném počítači, ve stejném dni.

4.5.1 Testovací nástroje

Tato sekce popisuje nástroje, které byly implementovány či použity k dosažení cílů práce. Programátor či tester může ke své práci použít celou škálu nástrojů. Tyto nástroje poté dokáží práci zjednodušit, zefektivnit či zrychlit.

Faker

Nedílnou součástí tohoto testování byla generace dat pro databázi různých entit. Díky této knihovně lze efektivně naplnit databázi testovacími daty a manipulovat s vygenerovanými daty dle požadavku. Pomocí této knihovny lze nasimulovat reálna jména, emailové adresy či telefonní čísla. Tyto data byla generována před provedením všech testů.

Tracy bar

Tento nástroj slouží k měření výkonu jak ORM techniky, tak i NDB techniky. Nette nástroj *Tracy*, byl zapnutý a vložený do všech testovacích metod. Tento nástroj je zobrazen v pravém dolním rohu. Pokud test byl proveden korektně, zobrazí se poté v pravém dolním rohu, spuštěný SQL dotaz a čas, který byl potřeba k dokončení exekuce. V tomto projektu bylo tedy cíleno na každou operaci v databází.

4.6 Bankovní aplikace

Jak bylo již zmíněno, bylo potřeba navrhnout a uskutečnit relační databázi, pomocí SŘBD a zároveň implementovat za pomocí objektového programování, kde díky tomu bylo možné uskutečnit *object relational mapping*. Jak lze vidět, datový model ukazuje či definuje formát tabulek, sloupců, cizí klíčů a vztahů mezi tabulkami.

Samotný SQL může poté přímo přistupovat k testovací databázi pomocí nastavení databázového připojení za použití knihovny od Nette, které umožní tvořit query/dotazy ke komunikaci s databází. Pro Doctrine 2 je potřeba nastavit také připojení k databázi. Doctrine 2 poté umožní propojit modely pro každou tabulku.

Každá tabulka v Databázi má svojí vlastní určenou třídu, která je nutná pro interakcí s tabulkou.

4.7 Databáze

Databázová struktura tohoto projektu byla vytvořena ze čtyř entit, to znamená ze čtyř relačních tabulek. Typicky základní datový model pro část klienta, účtů, prémiových typů či osobních bankéřů bude složen a uspořádán z entity *Client*, entity *Account*, entity *Premium* a entity *PersonalBanker*. Tabulky budou stejně pojmenované s jediným rozdílem, a to že počáteční písmeno bude malým písmenem a v množném čísle. *Account* bude mít vztah s entitou *Premium Many-To-One*, díky které lze definovat slevy či výhody k danému účtu. Entita *premium* bude mít vztah *One-To-Many* s *Account* entitou.

4.7.1 Atributy

Atributy entit neboli sloupce relací budou obsahovat jak primární, tak i cizí klíče. *Client* entita bude obsahovat sloupec *name* s datovým typem *string*, *surname* s datovým typem *string*, email s datovým typem *string*, *phoneNumber* s datovým typem *integer*, *age* s datovým typem *integer* a samozřejmě primární a cizí klíč, jako je *personalBanker_id* bude obsahovat *integer* typ.

Z pohledu MySQL či Postgresu budou obdobné názvy sloupců, pouze se zvolí datový typ *VARCHAR(255)* pro *name, email* a *surname*. *Number* bude mít datový typ *INT(11)* spolu s *age*, reference bude ovšem na *PersonalBanker* pomocí sloupce *personalBanker_id* s datovým typem *INT(11)*, které bude odkazovat na tabulku personalBanker.

Account entita obsahuje atributy *number integer*, *lastNumber integer*, *money integer*, *currency string*, *client_id* jako cizí klíč *a premium_id*, také jako cizí klíč.

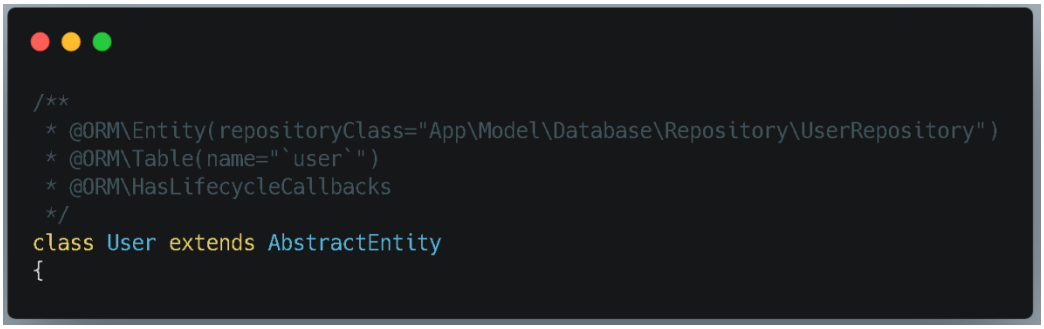
Z pohledu MySQL bude reference na tabulku Premium pomocí *premium_id* s datovým typem *INT(11)*. Sloupec *number* bude datového typu *INT(11)*, která bude uchovávat informace o bankovním účtu v číselném formátu. Sloupec *lastNumber* bude sloužit pro budoucí snazší zobrazování informací v podobě *INT(11)*, kdy se bude jednat o bankovní institut. Sloupec *money* uchovává pouze v číselném formátu

INT(11) informaci o počtu peněz klienta. *Currency* sloupec bude datového typu *VARCHAR(255)*.

Poslední entita personalBanker obsahuje atributy *name s datovým typem* string a *surname* s datovým typem *string*. Z pohledu MySQL bude sloupec *name* a *surname* obsahovat datový typ *VARCHAR(255)*.

4.7.2 Mapping

Nedílnou součástí pro implementaci ORM je *mapping* neboli mapování mezi objektovým modelem naší aplikace a relační tabulky v databázi. *Mapping* musí umožňovat veškeré operace nad tabulkou jako SŘBD. Pro takovýto zápis lze užít anotace. U každé entity se začíná anotací ihned nad třídou, kdy Doctrine si přečte, zda jde o entitu. Zjistí také z anotace, jakou *repository class* má spárovat a zjistí i na jakou relační tabulku se má daná entita spárovat či namapovat. V příkladu *Client* je počáteční mapování zobrazeno na obrázku 21.



Obrázek 21- Hlavní mapování entity s tabulkou

Poté následuje samotná definice sloupců či atributů, kdy se zapouzdří celý objekt a tudíž má veškeré stavy v *private*. Definování atributů v Doctrine 2 lze vidět na obrázku 22.

```
/**
 * @var string
 * @ORM\Column(type="string", length=255, nullable=FALSE, unique=false)
 */
private $name;

/**
 * @var string
 * @ORM\Column(type="string", length=255, nullable=FALSE, unique=false)
 */
private $surname;

/**
 * @var string
 * @ORM\Column(type="string", length=255, nullable=FALSE, unique=TRUE)
 */
private $email;

/**
 * @var integer
 * @ORM\Column(type="integer", length=9, nullable=FALSE, unique=TRUE)
 */
private $phoneNumber;
```

Obrázek 22- Anotace všech atributů

Definování atributů se provádí pomocí komentářů popsaných v sekci PHP. Díky tomu umí Doctrine 2 přečíst jakého datového typu má být sloupec v databázi a zároveň ví, jak velkou informaci lze do ní ukládat.

V anotacích se také definuje, zda sloupec může obsahovat nulovou hodnotu (pomocí *FALSE/TRUE)* a taktéž se definuje, zda se může v relační databázi opakovat. Po veškerém mapování je potřeba předat těmto atributům hodnoty, to jsou záznamy či entity, které budou obsaženy v databázi.

4.7.3 Getters a Setters

Jak bylo již zmíněno ORM neboli *Object Relational Mapping* vychází a je založen na objektovém programování, proto k přístupu k objektu musíme přistupovat zvenčí, aby se dodržel jeden z pilířů objektového programování – tzv. “zapouzdření“. Tahle metoda vrací hodnotu, která musí vyplývat z názvu. Příklad lze vidět na obrázku 23.

```php
public function getEmail(): string
{
    return $this->email;
}

public function setEmail($email): void
{
    $this->email = $email;
}

public function getPhoneNumber(): integer
{
  retun $this->phoneNumber;
}

public function setPhoneNumber($number):void
{
  $this->phoneNumber = $number;
}

public function getName(): string
{
    return $this->name;
}
```

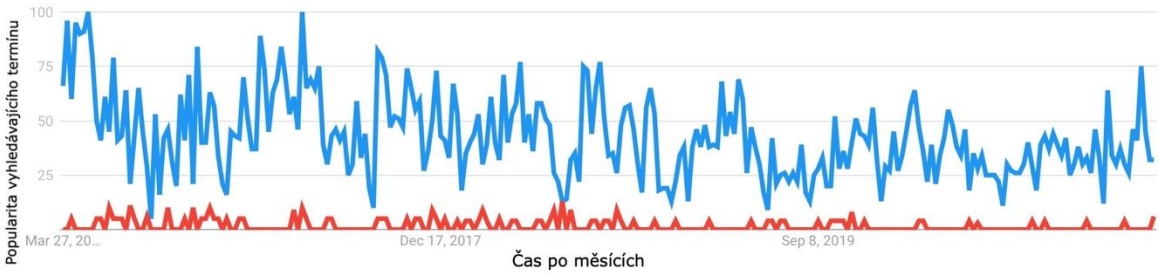
Obrázek 23- Gettery a Settery entity

V těchto *setterech* či *getterech* lze přidávat potřebnou logiku dle požadavků, jako je například telefonní předčíslí pro českou republiku (+420). Hlavní myšlenka ORM je proto dlouhodobá udržitelnost a škálovatelnost.

5 Výsledky a diskuse

Výběr jazyka bylo postaveno na základě podpory OOP a také možností připojení k databázi. Poté byly vybrány dva hlavní přístupy k datům a samotné databázi. Tento výběr či kombinace technologií umožňovala posoudit, zda se výkonnostně liší užití přístupu k databázi v čase.

Na grafu 1 lze vidět modrou křivku, která zastupuje Doctrine 2 a červenou křivku, která zastupuje NDB. Dá se tedy říci, že NDB je česká záležitost a demografie také ukazuje vyhledávání převážně z České republiky.



Graf 1- Statistika počtu vyhledávání pomocí google.com

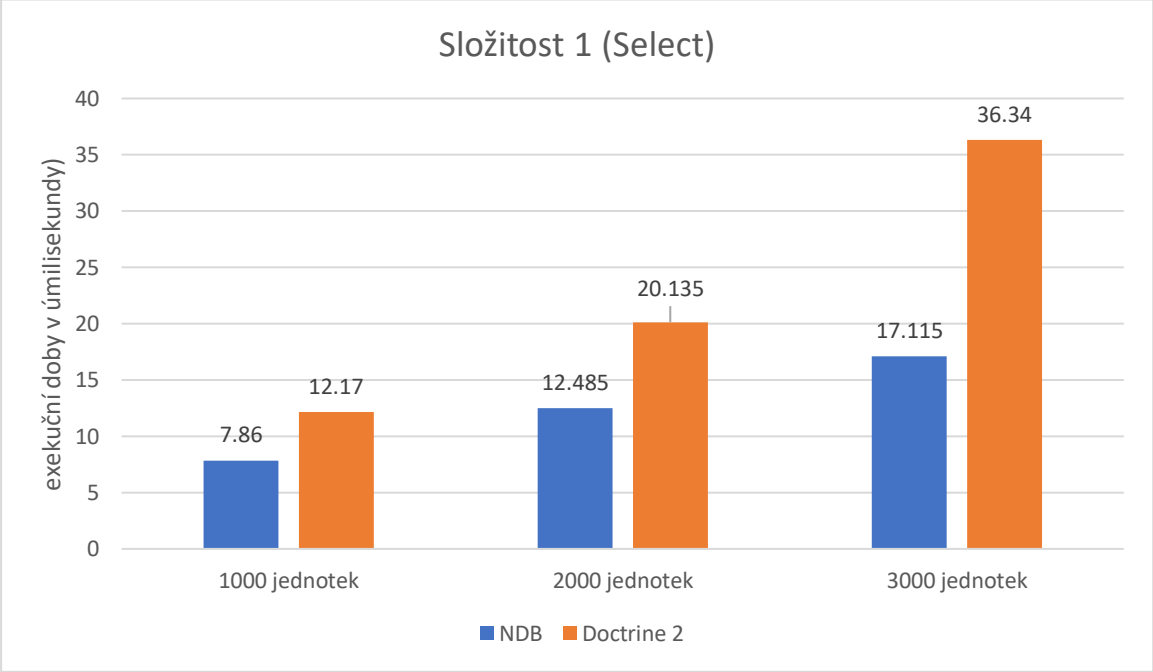
5.1 Datová analýza

Tato kapitola je věnována výsledkům získaných z praktické část a je potřeba tedy analyzovat veškeré zápisy do databáze, nejvyšší výkyvy hodnot, první a poslední testy, tendence grafů či průměrné doby odezvy. Práce byla vytvořena pro webovou aplikaci za pomocí dvou přístupů k databázi.

Výstupy těchto technik byly objekty či pole. Díky diagramu tříd bylo možné navrhnout vztahy objektů či tabulek, které v tomto projektu znázornily vztahy mezi *client, account, premium, personalBanker,* kde byly zmíněny kardinality jako je *OneToMany, ManyToOne.*

5.1.1 Analýza pro první složitost

V testu SELECT se složitostí 1 pro 1000 jednotek byl výsledek v prvním testu pro Nette 7,5 milisekund. Výsledek pro ORM bylo 11,8 milisekund. Poslední test pro Doctrine 2 vyšel 11,6 milisekund a pro Nette byl výsledek z posledního testu 7,6 milisekund. Během těchto testů bylo kontrolováno vždy cache systém, zda nebylo využito. Průměrný *response time* ($\bar{x}$) výsledku pro NDB byl 7,86 milisekund. V případě Doctrine 2 byl průměr ($\bar{x}$) výsledku 12,17 milisekund, což lze vidět vidět na grafu 2



Graf 2- Zobrazení průměrů pro operaci Select se složitost 1

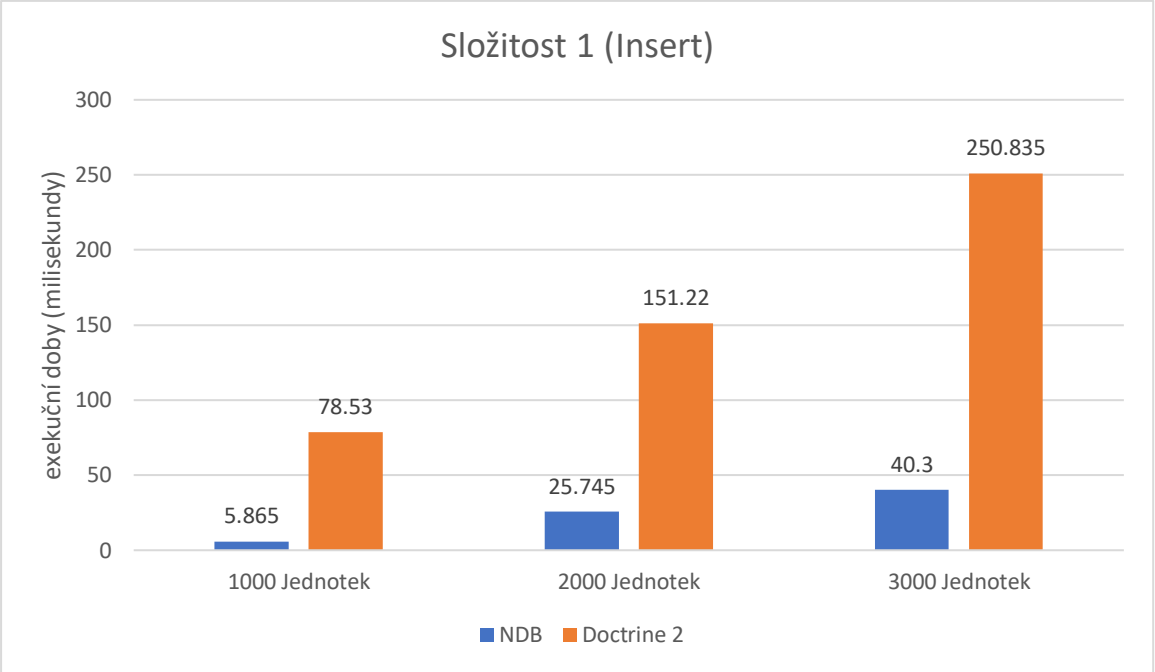
Na křivce v grafu 13, lze vidět mírný výkyv ve výsledku Doctrine 2 a to 15,5 milisekund, poté se výsledky ustálily a křivka měla konstantní tendenci. V případě Nette byla křivka po celou dobu testování konstantní. Nejvyšší hodnota v testu byla pro Doctrine 2 15,5 milisekund a pro Nette 8,6 milisekund.

Dalším use case byl *SELECT* záznamů pro 2000 jednotek. První test měl rychlou odezvu pro Doctrine 2, kde zpracoval celých 2000 záznamů do objektů v čase 19,7 milisekund. Nette získalo a zpracovalo stejnou množinu dat v čase 13,9 milisekund. Hodnoty posledního opakování byly velmi podobné prvním, a to pro NDB 11,2 milisekund a pro Doctrine 20,9 milisekund.

Průměrná doba ($\bar{x}$) výsledků měřené pro ORM bylo 20,135 milisekund a průměrná doba pro Nette byla 12,485 (viz. graf 2). Nejvyšší hodnoty v tomto test case byla pro Nette 13,9 milisekund a pro ORM 22,5 milisekund. Rozdíl byl tedy 8,6 milisekund. Křivka pro Nette vykazovala mírné odchylky mezi pátou a šestou iterací, a to 1,6 milisekund, dále byla odchylka mezi devatenáctou a dvacátou iterací, lze vidět na grafu 8.

Poslední use case byl pro 3000 jednotek. V prvních výsledcích byl opět nejslabší ve výkonu ORM s odezvou 35,2 milisekund. Nette dokázalo výsledky z databáze získat za polovinu času a to během 16 milisekund. Výsledek posledního opakování pro NDB bylo 16,9 milisekund a pro ORM byl výsledek 44 milisekund. Průměrná hodnota ($\bar{x}$) byla pro Doctrine 2 36,34 milisekund, zatímco Nette zvládlo veškeré operace v průměru 17,115 milisekund. Nejvyšší možná hodnota pro ORM byla 44 milisekund a pro NDB pouze 23,9. Na křivce v grafu 9 můžeme vidět, že se v druhé iteraci zvýšila hodnota pro Nette na 23,9 milisekund. Zde je dobré si uvědomit, že Doctrine skládá hodnoty, a i datové typy datetime uvedené v databázi, které musí následně zkonvertovat do třídy DateTime.

Při prvním spuštění test case INSERT s 1000 jednotek byl výsledek pro Doctrine 88,6 milisekund a pro Nette Database 15 milisekund. V tomto případě rozdíl výsledku těchto technik byl 73,6 milisekund. Je důležité zdůraznit, že oba testy zapsaly úspěšně do databázového systému své výsledky. Poslední test proběhl pro Doctrine 2 v čase 77,7 milisekund a pro Nette Database 17,9 milisekund. Průměrná hodnota ($\bar{x}$) pro Nette byla 15,865 milisekund pro Doctrine 2 78,53 milisekund, výsledky průměrných hodnot lze vidět na grafu 3.



Graf 3- Zobrazení průměrů pro operaci Insert se složitost 1

Nejvyšší možná hodnota pro Nette Database byla 17,9 milisekund a pro ORM 96,9 milisekund. Můžeme vidět, že křivka pro Nette kolísala v třetí a v devatenácté iteraci, lze vidět na grafu 10.

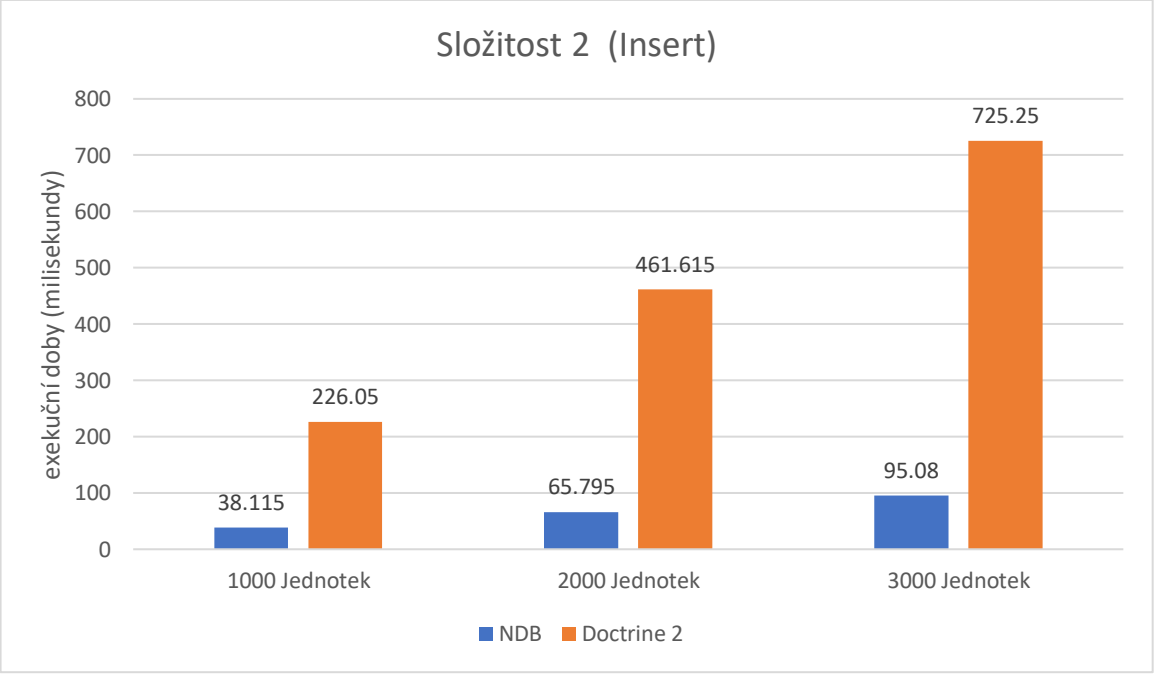
Dalším test case bylo testování s nárůstem o 1000 jednotek, tedy množina výstupu pro Doctrine a NDB byla 2000 záznamů či objektů z databáze. Zde chování PHP serveru nevykazovalo žádnou nestabilitu, prvotní test vyšel pro ORM s odezvou 175,8 milisekund. Naproti tomu výsledek pro Nette vyšel na 23,9 milisekund, zde byl masivní rozdíl mezi výsledky. Důvod rozdílu těchto výsledků je tvorba a skládání 2000 záznamů do 2000 objektů, které na sebe dále implementují interní třídy Doctrine. Poslední hodnoty v tomto testu bylo pro Doctrine 2 147,9 milisekund a pro NDB 24,7 milisekund. Průměrné doby ($\bar{x}$) vyšly v sumarizaci u *insert* s 2000 jednotek pro Nette 25,745 milisekund. Doctrine 2 měla průměrnou dobu ($\bar{x}$) odezvy 151,22 milisekund. V grafu 11, lze vidět nejvyšší hodnoty, které patří ORM z prvního testu a to 175,8 milisekund. Oproti tomu NDB měla nejvyšší možnou hodnotu 20 milisekund. Křivky pro Nette a Doctrine 2 byly převážně stálé a stabilní.

Poslední test byl opět se stejnou složitostí ovšem s opětovným nárůstem o 1000 jednotek. Výsledky se zvýšily v případě Nette o 10,3%, na 39,2 milisekund. V případě ORM výsledek prvního pokusu byl 246,6 milisekund, což je znatelný rozdíl v *response time* z uživatelského hlediska. Výsledek posledního testu pro objektový přístup byl

252,9 milisekund, kde oproti tomu NetteDatabase zapsalo ty stejné hodnoty za 40,2 milisekund. Průměrná doba ($\bar{x}$) pro Nette 40,3 milisekund a pro ORM 250,835 milisekund. Nejvyšší hodnota pro Doctrine 2 byla 269,7 milisekund a pro Nette database 48 milisekund.

5.1.2 Analýza pro druhou složitost

V další sekci se prováděly testy se složitostí 2, kdy se vytvářely objekty s relací, pomocí *insert* operace. Zde byl nárůst času značně vysoký. Testování bylo prováděno způsobem, kdy objekty byly vloženy do zásobníku a poté se zavolala funkce, která všechny tyto dočasně uložené objekty uložila do databáze. První test proběhl pro ORM v čase 233,3 milisekund, oproti tomu Nette zvládlo vkládání záznamů do databáze za 34,3 milisekund. Test posledního opakování dopadl pro Doctrine 2 207,7 milisekund a pro NDB 39,7 milisekund. Průměrná hodnota doby ($\bar{x}$) pro ORM byla 226,05 milisekund. Nette měla průměrnou dobu ($\bar{x}$) odezvy 38,115 milisekund, výsledky lze vidět v grafu 4.



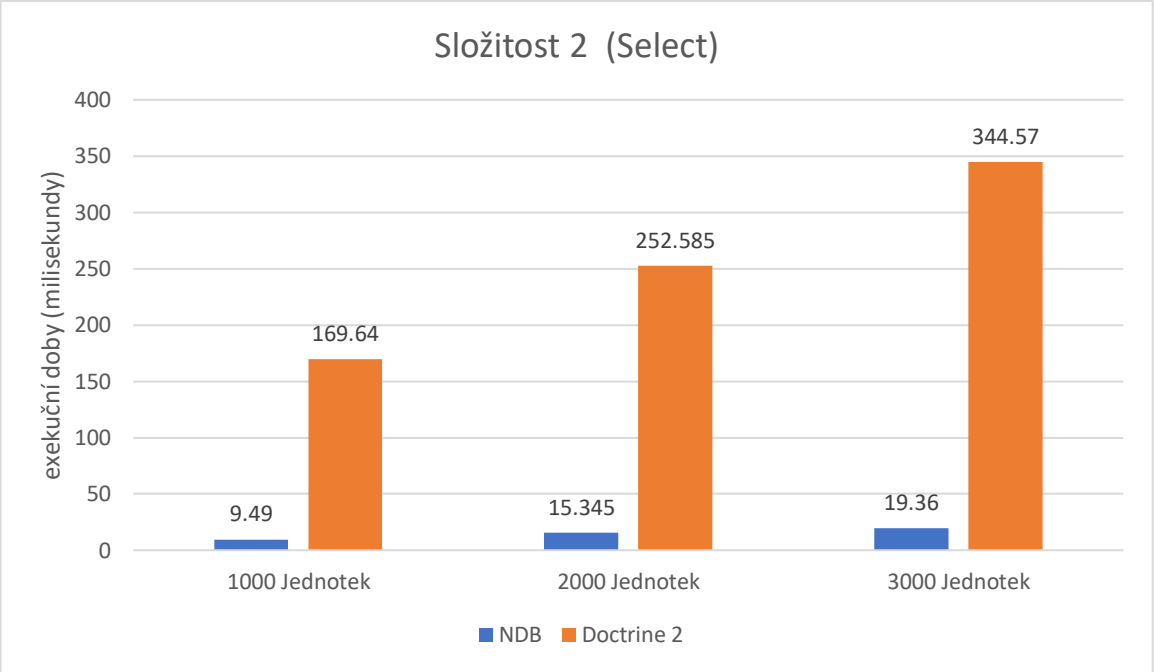
Graf 4- Zobrazení průměrů pro opearci Insert se složitostí 2

Nejvyšší hodnota pro Nette byla 45,2 milisekund a pro Doctrine 2 nejvyšší hodnota dotazu byla zpracována v čase 243,3 milisekund. Křivka pro ORM měla z prvních testů skákavou tendenci, oproti tomu křivka pro Nette byla opět konzistentní.

Další test case byl proveden na 2000 jednotek. V tomto případě Doctrine 2 provedla operace v čase 505,8 milisekund. Pro Nette se dvojnásobně zvýšil čas pro zpracování na 88,8 milisekund, lze vidět na grafu 17. V posledním opakováním NDB zpracovalo operace v čase 59,7 milisekund. Pro ORM byl výsledek 451,2 milisekund. Průměr ($\bar{x}$) výsledných testů pro Nette byl 65,795 milisekund, zatímco Doctrine 2 měla průměr 461,615 milisekund. Nejvyšší možná hodnota tohoto test case byla pro Doctrine 2 hodnota z prvního testu 505,8 milisekund, totéž platí i pro Nette, které má hodnotu z prvního testu 88,8 milisekund. Křivka pro NDB byla opět konstantní, zatímco mírné výkyvy byly zaznamenány mezi první a druhou iterací a také mezi jedenáctou a dvanáctou iterací v křivce pro Doctrine, lze vidět na grafu 7.

Poslední test case byl pro 3000 jednotek. První zkouška pro Nette k zapsání 3000 jednotek trvala 119,7 milisekund a pro Doctrine 2 804,5 milisekund. Poslední iterace proběhla pro Doctrine 2 nejpříznivěji ze všech testů, a to během 675,6 milisekund a pro NDB 93,9 milisekund. Průměrná hodnota ($\bar{x}$) vyšla pro Doctrine 2 725,25 milisekund. Oproti tomu Nette mělo průměr 95,025 milisekund. Nejvyšší hodnota pro NDB byla z prvního testu a to 119,7 milisekund, pro ORM byla hodnota také z první iterace a to 804,5 milisekund. Křivka pro Doctrine 2 měla mírné výkyvy v šesté, sedmé, deváté a v třinácté iteraci, což lze vidět na grafu 12.

Pro poslední testovací scénář se složitostí 2 byla vybrána operace *select*. V prvním testu pro 1000 jednotek vyšlo pro NDB přesně 8 milisekund ke zpracování a ORM získalo stejná data za 160 milisekund. Výsledek poslední iterace test case byla pro ORM 159,6 milisekund a pro Nette 8,5 milisekund. Průměrný čas ($\bar{x}$) získaný během databázových operací byl pro Nette 9,49 milisekund. Oproti tomu ORM získalo data v průměru ($\bar{x}$) za 169,64 milisekund, což lze vidět na grafu 5.



Graf 5- Zobrazení průměrů pro opearci Select se složitostí 2

Nejvyšší hodnota pro Doctrine 2 byla 198,1 milisekund v páté iteraci, NDB měla nejvyšší hodnotu 13,9 milisekund. Křivka pro Doctrine 2 měla kolísavé hodnoty v páté a v desáté iteraci a poté mezi osmnáctou a devatenáctou iterací, což lze vidět na grafu 13.

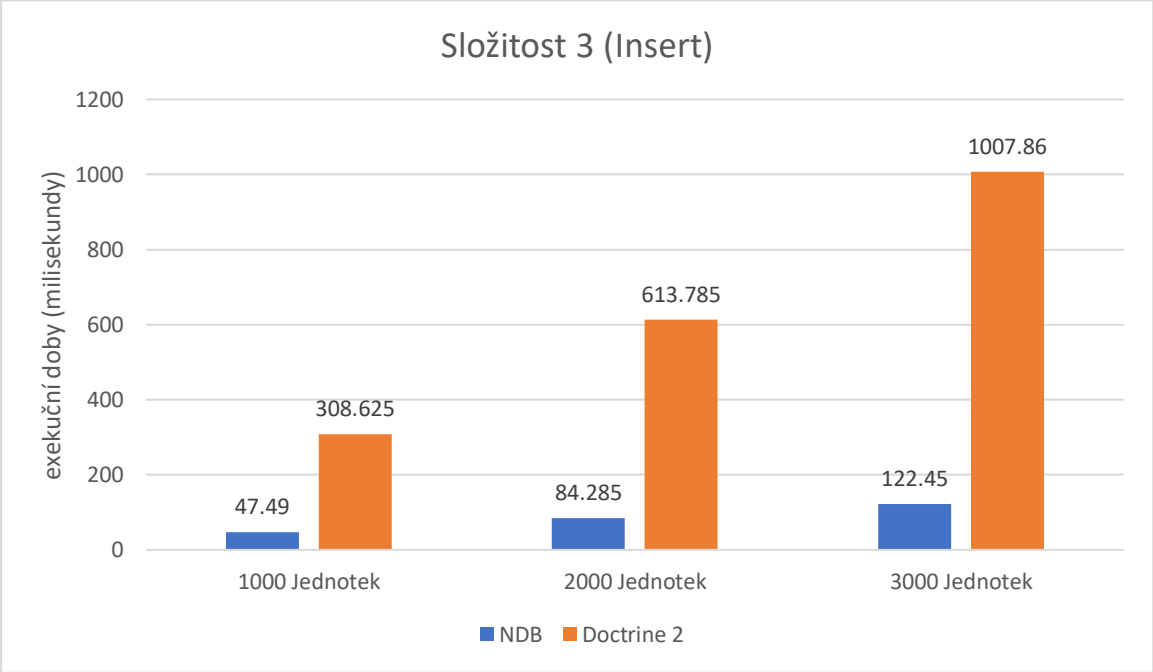
Další *select* byl navýšen o 1000 jednotek, z databáze bylo tedy získáváno 2000 jednotek, kdy každá jednotka obsahovalo relaci na jiné tabulky či objekty. První test byl pro Doctrine 2 263,3 milisekund. Hodnota pro Nette byla 12,7 milisekund. Průměrné hodnota ($\bar{x}$) byla pro Nette 15,345 milisekund a pro ORM byl průměr vypočítán na 252,585 milisekund. Nejvyšší hodnota naměřena pro Nette byla 20,9 milisekund, pro ORM byla nejvyšší naměřená hodnota 281,1 milisekund pomocí „*tracy bar*“. Křivka pro ORM měla výkyvy v desáté, patnácté a sedmnácté iteraci, což lze vidět na grafu 14.

Poslední testovací scénář vyšel v první iteraci pro Nette 18,9 milisekund, Doctrine 2 získalo stejná data za 457,2 milisekund. Poslední iterace tohoto testu byla pro NDB přesně 20 milisekund, pro ORM byla 375,8 milisekund. Průměrná hodnota ($\bar{x}$) časů pro NDB byla 19,36 milisekund, zatímco ORM technika získávala data v průměru 344,57 milisekund. Nejvyšší hodnota byla naměřena pro ORM a to 457,2 milisekund a pro Nette 20 milisekund. Křivka pro Nette měla sice mírné výkyvy, ale

poté se ustálila. Doctrine 2 měla výkyvy v první, páté a třinácté iteraci, lze vidět na grafu 15.

5.1.3 Analýza pro třetí složitost

První test byl pro 1000 jednotek s operací *insert*, která se vyznačuje tím, že jedna operace vyžaduje zápis do tří různých vnořených tabulek v databázi. Výsledek zde vyšel pro Nette 42,7 milisekund a pro Doctrine 375,8 milisekund. Poslední iteraci zpracovalo Nette za 43,1 milisekund, pro ORM byla hodnota z poslední iterace 272 milisekund. Nejvyšší hodnota pro Nette v tomto testu byla 57 milisekund, pro Doctrine 2 byla nejvyšší hodnota 381,6 milisekund. Průměrná doba ($\bar{x}$) exekučního času byla pro Doctrine 2 308,625 milisekund, zatímco pro Nette byla hodnota 47,49 milisekund, výsledky lze vidět na grafu 6.



Graf 6- Zobrazení průměrů pro operaci Insert se složitostí 3

Z grafu 16 lze vypozorovat, že křivka Doctrine 2 měla výkyv ve čtvrté, páte,třínácte a v čtrnácté iteraci.

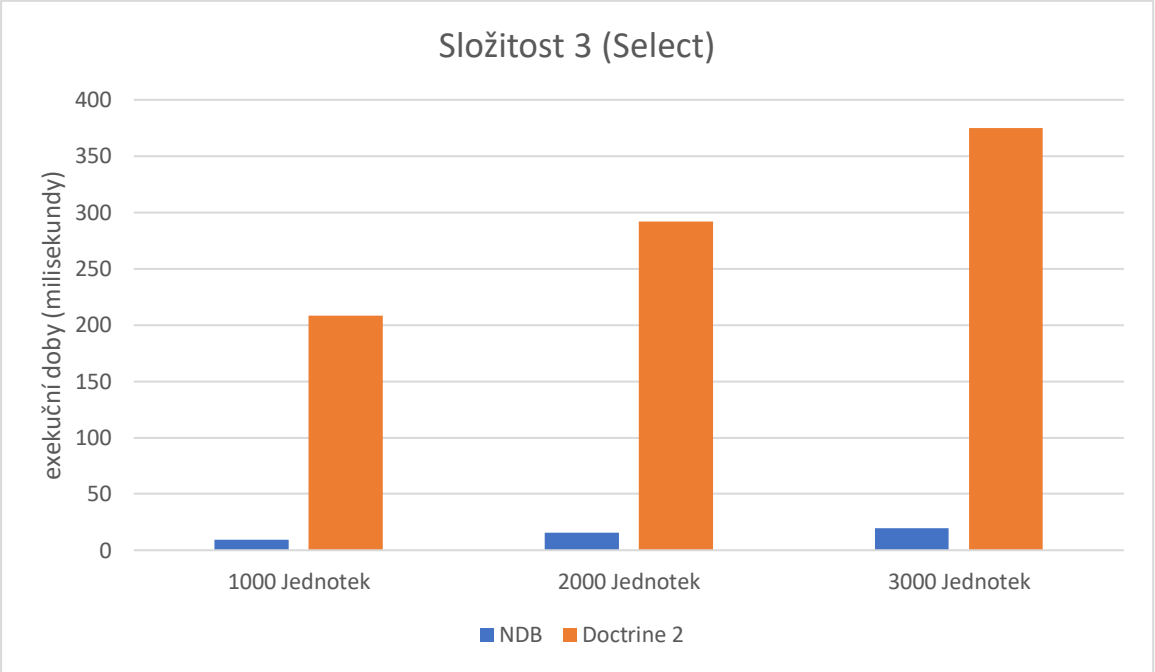
Další test case byl opět proveden na 2000 jednotek. Zde NDB provedla operace v čase 84,5 milisekund. Pro ORM se dvojnásobně zvýšil čas pro zpracování operace na 631 milisekund. V posledním opakováním Doctrine 2 zpracovalo operaci v čase 571,1 milisekund. Pro Nette byl výsledek za 97,8 milisekund. Nejvyšší hodnota tohoto

test case byla pro Doctrine 2 hodnota z předposlední iterace, 640,2 milisekund. Nette zpracovalo 2000 jednotek za 97,8 milisekund.

Průměrná ($\bar{x}$) hodnota pro Doctrine 2 byla 613,785 milisekund a pro Nette byla 84,285 milisekund. Křivky pro NDB a ORM byly konstantní až na mírné výkyvy.

V test case pro 3000 jednotek s operací insert, vyšel jako první výsledek pro Nette 105,9 milisekund, pro Doctrine 2 989,9 milisekund. Poslední iterace proběhla pro NDB příznivě a to během 108,8 milisekund, hodnota pro Doctrine byla 2 977,9 milisekund. Průměrná hodnota ($\bar{x}$) vyšla pro Doctrine 2 1007,86 milisekund. Naproti tomu NDB mělo průměr ($\bar{x}$) 122,45 milisekund. Nejvyšší hodnota pro ORM byla z osmnácté iterace a to 1153,4 milisekund, zatímco pro Nette byla hodnota z třetí iterace a to 198,8 milisekund. Křivka pro Nette měla klidný průběh až na třetí iteraci, kdy vyskočila o 76 milisekund oproti průměrné hodnotě. Dále měla tato křivka výkyvy v deváté a čtrnácté iteraci, dle grafu 18.

Poslední test toho projektu byla operace *select* se složitosti 3. V první iteraci byla výsledná hodnota pro Nette 9,7 milisekund, pro Doctrine 2 byla hodnota 204,7 milisekund. Poslední test vyšel pro ORM s hodnotou 209,9 milisekund, pro Nette vyšel poslední test o 0,1 méně než v prvním testu, tedy 9,6 milisekund. Nejvyšší naměřená hodnota v testu pro Nette byla 13,7 milisekund. Doctrine 2 naproti tomu potřebovala 219,3 milisekund ke zpracování. Průměrná hodnota ($\bar{x}$) pro ORM byla 208,13 milisekund a pro Nette vyšel velmi překvapivý výsledek 9,6 milisekund, lze vidět na grafu 7.



Graf 7- Zobrazení průměrů pro operaci Select se složitostí 3

Křivka v případě ORM měla malé odchylky v řádu 10 milisekund. Oproti tomu křivka pro Nette měla odchylky v sedmé a v jedenácté iteraci, což lze vidět na grafu 19.

Druhý test case se prováděl na 2000 jednotkách. Zde v iteraci č.1 vyšla hodnota pro Doctrine 2 283,4 milisekund. Pro Nette byla hodnota z této iterace 14,2 milisekund. Oproti tomu vyšla v poslední iteraci pro Doctrine 2 jedna z nejnižších hodnot, 275,8 milisekund, pro Nette vyšlo 15,3 milisekund. Průměrná hodnota ($\bar{x}$) byla pro Nette 15,345 milisekund a pro Doctrine 2 291,925 milisekund. Nejvyšší hodnota vyšla pro Doctrine 2 334,7 milisekund a pro Nette byla hodnota 22 milisekund.

Poslední test case byl pro 3000 jednotek se složitostí 3. Hodnota z první iterace vyšla pro Nette 21,1 milisekund a pro Doctrine 2 374,7 milisekund. Ovšem hodnota z poslední iterace pro Nette byla jedna z nejnižších hodnot a to 18,9 milisekund a pro Doctrine 2 byla tato hodnota 383,8 milisekund. Průměrná hodnota ($\bar{x}$) byla pro Doctrine 2 374,705 milisekund a pro Nette 19,62 milisekund. Nejvyšší hodnota byla pro Nette 21,9 milisekund a pro Doctrine 2 414,6 milisekund. Křivka zde byla pro Doctrine 2 a Nette spíše v rovině.

6 Závěr

V první části diplomové práci byla rozebrána problematika databáze a samotný přístup k databázi. Bylo také nutné uvést obecnou charakteristiku těchto termínu, spolu s jejich výhodami či nevýhodami. Také bylo potřeba rozebrat objektový přístup v programování, na kterém objektově relační mapování stojí a navazuje na něj. V této první teoretické kapitole se tedy rozebírá do podrobností relační databáze a její model. Dále bylo popsáno objektově relační mapování a způsoby návrhu této techniky, pomocí objektů. Poté se rozebírá Nette Database a její knihovny pro prací s daty.

Během shromžďování informací jsem nenalezl studii, která by srovnávala Doctrine 2 s Nette Database. Byly však zpracované studie zaobírající se Doctrine 2 z jiného úhlu pohledu, například od Dominika Firla s názvem *"Srovnání řešení správy dat v MySQL a MongoDB při použití Doctrine 2 ve frameworku Symfony 2",* kde autor srovnává MySQL a MongoDB. Dále byla od Michaela Voříška zpracována studie o výkonnosti ORM s názvem *"Akcelerace ORM prostřednictvím transparentního dávkového zpracování"*. Shledal jsem tedy nedostatek v hledané literatuře ohledně výkonnu Doctrine 2 oproti knihovnám, které obstarávají abstrakci a SQL, jako je PDO či Nette Database. Tato mezera ve studiích byla má motivace systematicky testovat dotazy a poté výstupy těchto dotazů podrobně rozebrat.

Na základě teoretické částí a poznatků byl navržen diagram tříd pro znázornění vztahů. Tyto vztahy jsme mohli poté převést do aplikační logiky. Poté bylo (pomocí konfigurace) Doctrine 2 a Nette úspěšně připojeno k prázdné databázi. Díky mapování, které se rozebíralo na teoretické úrovni se vytvořila databáze pomocí přístupu *code first*. Na základě tvorby databáze a mapování je možné pracovat s daty jako s entitami a následně tyto objekty ukládat jako řádky do databáze.

Testovaly se stejné dotazy ve stejných podmínkách, a to pro oba přístupy. Veškeré testy proběhly v jednom dni, na stejném serveru, se základním nastavení v Linuxu a zároveň se používala stejná databáze. Tím jsme omezili výkonnostní proměnné pro toto porovnání.

V datové analýze lze vidět, z pohledu výkonnosti, jednoznačné výsledky pro Nette. Ovšem přehlednost v kódu je podstatně horší, neboť syntaxe SQL dotazů jsou vesměs robustní. Oproti tomu Doctrine 2 nebyla ve výkonu rychlejší v žádném testu, v porovnání s Nette. Doctrine 2 měla exekuční čas vždy delší, jednak kvůli méně

optimalizovaným dotazům a také díky tvorbě objektů. Na druhou stranu je výhoda této techniky určité rozhraní pro práci entit. Například na grafu 12 lze vidět nárůst exekučního času během *Insert* dotazů a to hlavně kvůli složitějším dotazům, které způsobují více vnořených objektů do sebe.

Je tedy nutné si stanovit, pro koho je jaká technika určená. Jedna z nevýhod Doctrine 2 jsou situace, kdy si vývojář nemusí plně uvědomit náročnost dotazů pro získání určité množiny objektů. Je tudíž nutné mít na paměti tyto výkonnostní problémy, neboť takovéto operace mohou narušit běh webového serveru. To může vyřešit speciální jazyk od Doctrine 2, který se nazývá Doctrine Query Language. Díky němu lze předejít N+1 problému. Na druhou stranu snadná, a vcelku intuitivní API od Nette přináší řadu výhod, v případě využití této knihovny. Například řešení N+1 problému nebo jednoduchého vkládání více záznamů pomocí pole. Nette Database je tedy dobré využít na komplexní data, kdy je opravdu získáno takové množství, které by PHP nedokázalo vkládat do objektů. Dále se nabízí jako řešení v situacích, kdy není potřeba manipulovat s daty jako s objekty, například výpis kategorií e-shopu.

Pomocí přesného testování bylo potvrzeno, že Nette Database má znatelně vyšší výkon než Doctrine 2.

7 Seznam použitých zdrojů

[1] TATROE, Kevin, MACINTYRE, Peter, 2020. *Programming PHP : creating dynamic web pages*. Cambridge: O'Reilly. ISBN 9781492054139.

[2] NIXON, Robin, 2014. *Learning PHP, MySQL, JavaScript, CSS & HTML5*. Beijing Sebastopol, CA: O'Reilly Media. Print. ISBN 978-1491949467.

[3] SKLAR, David, TRACHTENBERG, Adam, 2006. *PHP cookbook*. Sebastopol, CA: O'Reilly. ISBN 978-0596101015.

[4] WELLING, Luke, THOMSON, Laura, 2017. *PHP and MySQL Web development, fourth edition*. Upper Saddle River: Addison-Wesley. ISBN 978-0321833891.

[5] ZANDSTRA, Matt, 2016. *PHP Objects, Patterns, and Practice*. Berkeley, CA: Matt Zandstra. ISBN 9781484219959.

[6] LEE, Richard, TEPFENHART, William, 2002. *Practical object-oriented development with UML and Java*. Upper Saddle River, New Jersey: Prentice Hall. ISBN 0130672386.

[7] BÖHMER, Marian, 2015. *Návrhové vzory v PHP*. Brno: Computer Press. ISBN 9788025133385.

[8] HOLZNER, Steven, 2008. *PHP: the complete reference*. New York: McGraw-Hill. ISBN 978-0071508544.

[9] THOMAS, Dave, HUNT, Andy, 2020. *The pragmatic programmer*. Boston: Addison Wesley. ISBN 9780135957059.

[10] HARLE, Robert,. *Object Oriented Programming* [online]. 2010. [cit. 2021-01-12]. Dostupné z: https://www.cl.cam.ac.uk/teaching/0910/OOProg/OOP.pdf

[11] LETHBRIDGE, Timothy, LAGANIERE, Robert, 2005. *Object-oriented software engineering*. Berkshire: McGraw-Hill Education. ISBN 978-0077109080.

[12] BOOCH, Grady, 2007. *Object-oriented analysis and design with applications*. Upper Saddle River, NJ: Addison-Wesley. ISBN 978-0201895513.

[13] HERNANDEZ, Michael J, 2013. *Database design for mere mortals: a hands-on guide to relational database design*. Boston: Addison-Wesley. ISBN 9780321884497.

[14] STEPHENS, Rod, 2008. *Beginning database design solutions*. Indianapolis, IN: Wiley Pub. ISBN 978-0470385494.

[15] CORONEL, Carlos, MORRIS, Steven, 2018. *Database systems: design, implementation, and management*. Boston, MA, USA: Cengage Learning. ISBN 978-1337627900.

[16] RAMAKRISHNAN, Raghu, GEHRKE, Johannes, 2003. *Database management systems*. Boston: McGraw-Hill. ISBN 978-0072465631

[17] HARRINGTON, Jan L, 2016. *Relational database design and implementation*. Amsterdam Boston: Morgan Kaufmann/Elsevier. ISBN 978-0128043998.

[18] SKŘIVAN, Jaromír,. *Datové modely a návrhy relačních schémat* [online]. 2008. [cit. 2021-01-18]. Ústav informačních studií a knihovnictví FF UK v Praze Dostupné z:
https://sites.ff.cuni.cz/uisk/wp-content/uploads/sites/62/2016/01/Datov%C3%A9-modely-a-n%C3%A1vrhy-rela%C4%8Dn%C3%ADch-sch%C3%A9mat_Sk%C5%99ivan.pdf

[19] LING, Tok Wang, 2013. *Database Schema Using Entity-Relationship Approach [online]*. [cit. 2020-12-15]. National University of Singapore. Dostupné z: https://www.comp.nus.edu.sg/~lingtw/cs4221/er.pdf

[20] PRAVDA, Michal, 2007. *Výhody a nevýhody použití perzistence objektů v jazyce Java*. Praha, Diplomová práce. Univerzita Karlova, Matematicko-fyzikální fakulta. RNDr. Michal Kopecký, Ph.D.

[21] MARSTON, Tony. *Radicore Development Infrastructure* [online]. 2020-03-06 [cit. 2020-12-13]. Dostupné z: https://www.tonymarston.net/php-mysql/infrastructure-faq.html

[22] MORAVEC, Petr, 2009. *Přístup k databázím v PHP frameworcích Zend a Nette*. Praha, Bakalářská práce. Vysoká škola ekonomická v Praze: Ing. Luboš Pavlíček.

[23] HOLUB, Vít, 2007. *Metody transformace relačních databázových systémů do objektových*. Praha, Doktorská disertační práce. Česká zemědělská univerzita. Prof. RNDr. Jiří Vaníček, CSc.

[24] MOKRUŠA, Petr. *Objektově-Relační Mapování na Platformě PHP*. Brno, 2015. Brno, Bakalářská práce. Vysoké Učení Technické v Brně. Ing. Radek Burget, Ph.D.

[25] DUNGLAS, Kévin, 2013. *Persistence in PHP with Doctrine ORM: build a model layer of your PHP applications successfully, using Doctrine ORM*. Birmingham, UK: Packt Pub. ISBN 978-1782164104.

[26] CHOPRA, Rajiv, 2016. *Database Management Systems (DBMS)*: S CHAND & CO LTD,. ISBN 978-9385676345.

[27] BEAULIEU, Alan, 2005. *Learning SQL*. Sebastopol, CA: O'Reilly Media. ISBN 978-0596520830.

[28] WINAND, Markus, 2012. *SQL Performance Explained everything developers need to know about SQL performance*. Wien: M. Winand. ISBN 978-3950307825.

[29] GOURLEY, David, TOTTY, Brian, SAYER, Marjorie, AGGARWAL, Anshu, REDDY, Sailu, 2002. *HTTP: The Definitive Guide*. O'Reilly Media. ISBN 978-1565925090.

[30] PHP Documentation Group, *Documentation* [Online]. 2021-03-04. [cit. 2021-03-10]. Dostupné z: https://www.php.net/.

[31] Doctrine 2. *Doctrine 2 ORM's documentation* [online]. 2021-01-01. [cit. 2021-02-28]. Dostupné z: https://www.doctrine-project.org/

[32] TICHÝ, Jan. *Seriál: Doctrine 2* [online]. 2010-07-21 [cit. 2020-12-25]. Dostupné z: http://www.zdrojak.cz/clanky/doctrine-2-uvod-do-systemu/

[33] FOWLER, Martin. *Catalog of Patterns of Enterprise Application Architecture. In: martinfowler.com* [online]. 2019-08-01. [cit. 2020-12-12]. Dostupné z: http://martinfowler.com/eaaCatalog/

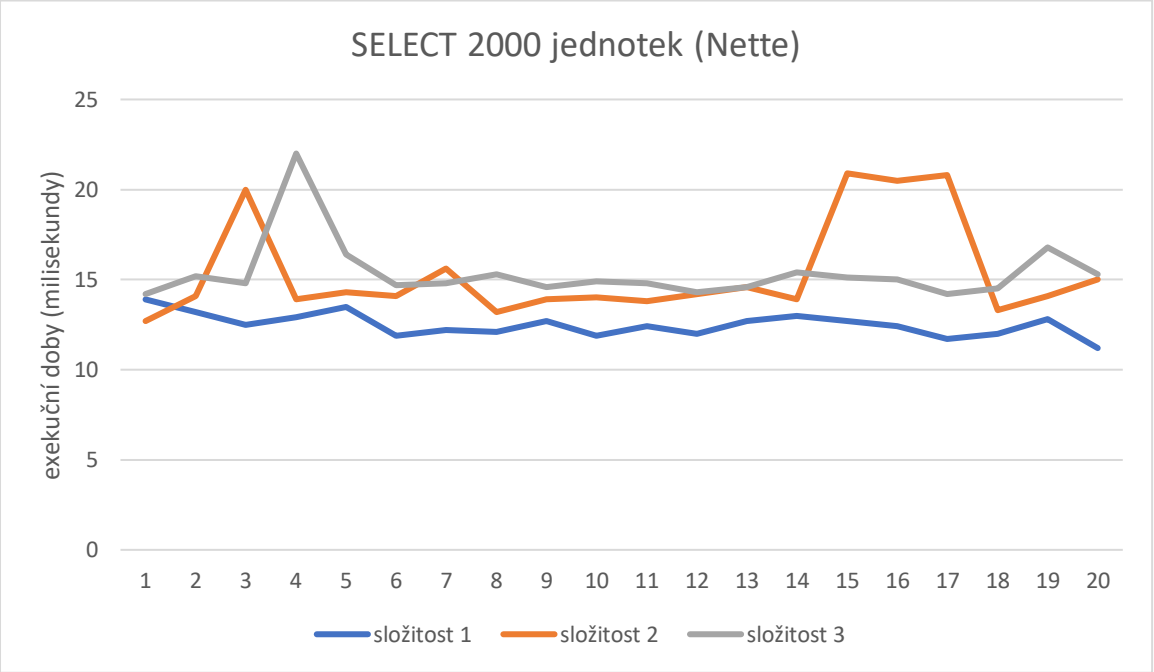
[34] Framework Nette. *Dokumentace*. [online]. 2021-01-01. [cit. 2021-03-10]. Dostupné z: http://doc.nette.org/cs/

[35] Sadowski, 2020.*SQL-PRVN* [Online] [cit. 2020-11-25]. Dostupné z: http://books.fs.vsb.cz/SQLReference/Sadovski/SQL-PRVN.HTM

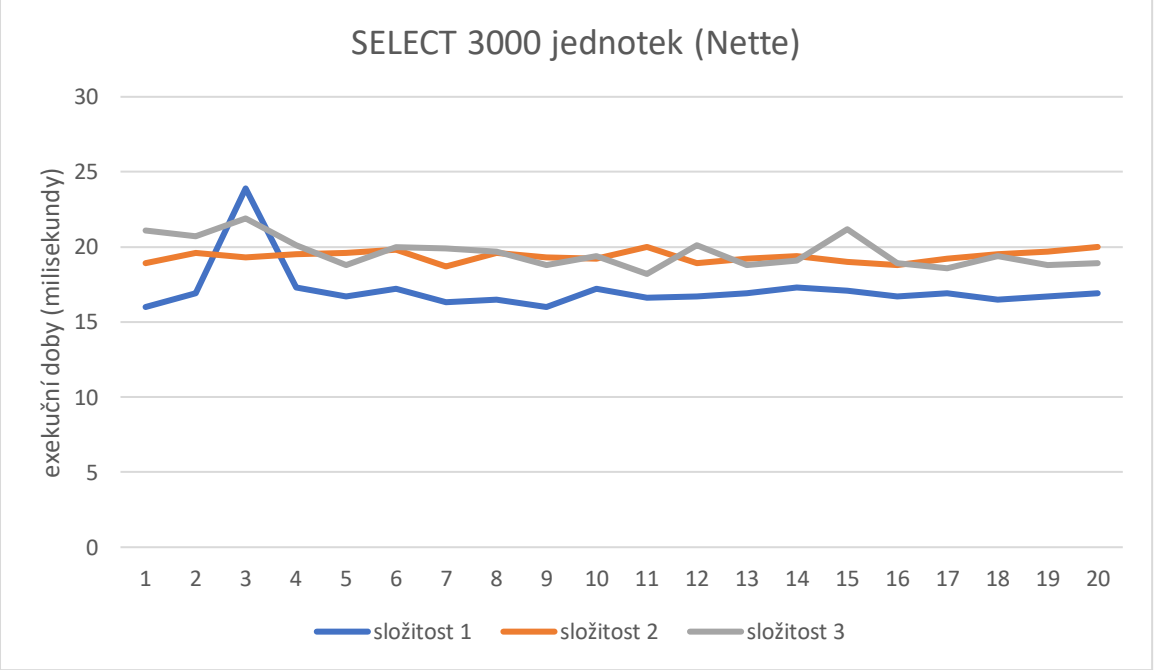
[36] PECINOVSKÝ, Rudolf, 2010. *OOP: naučte se myslet a programovat objektově*. Brno: Computer Press. ISBN 978-80-251-2126-9.

[37] ČÁPKA, David. *Úvod do Nette frameworku pro PHP*. ITnetwork.cz [online]. 2015. [cit. 2020-11-21]. Dostupné z: http://www.itnetwork.cz/php/nette/doctrine

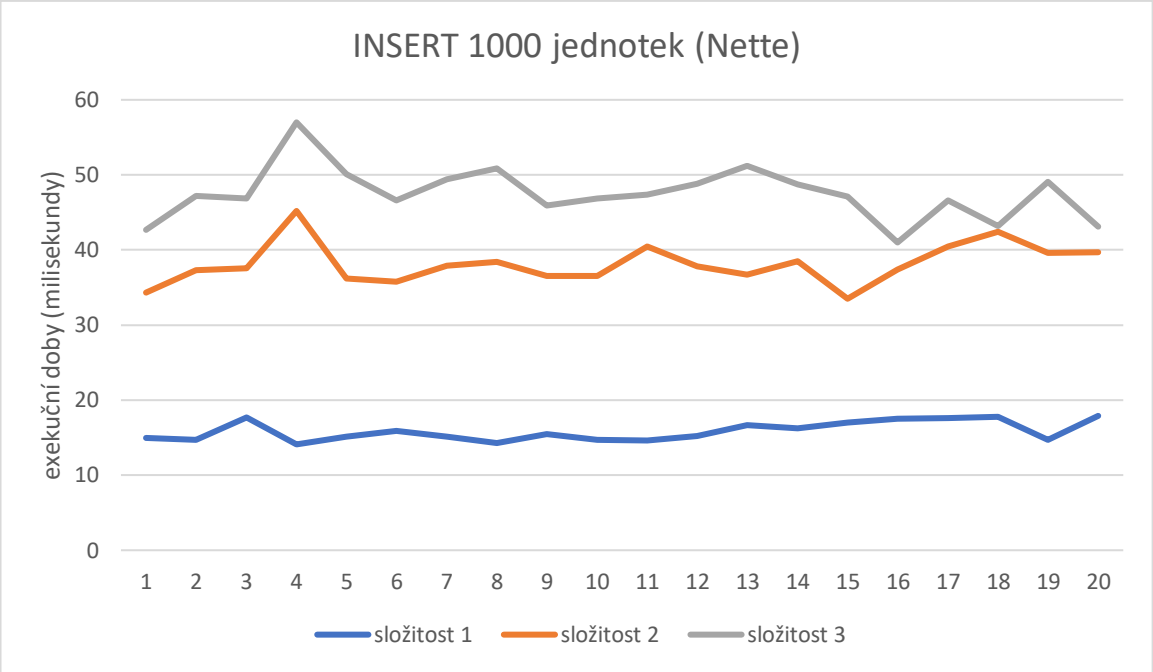
8 Přílohy



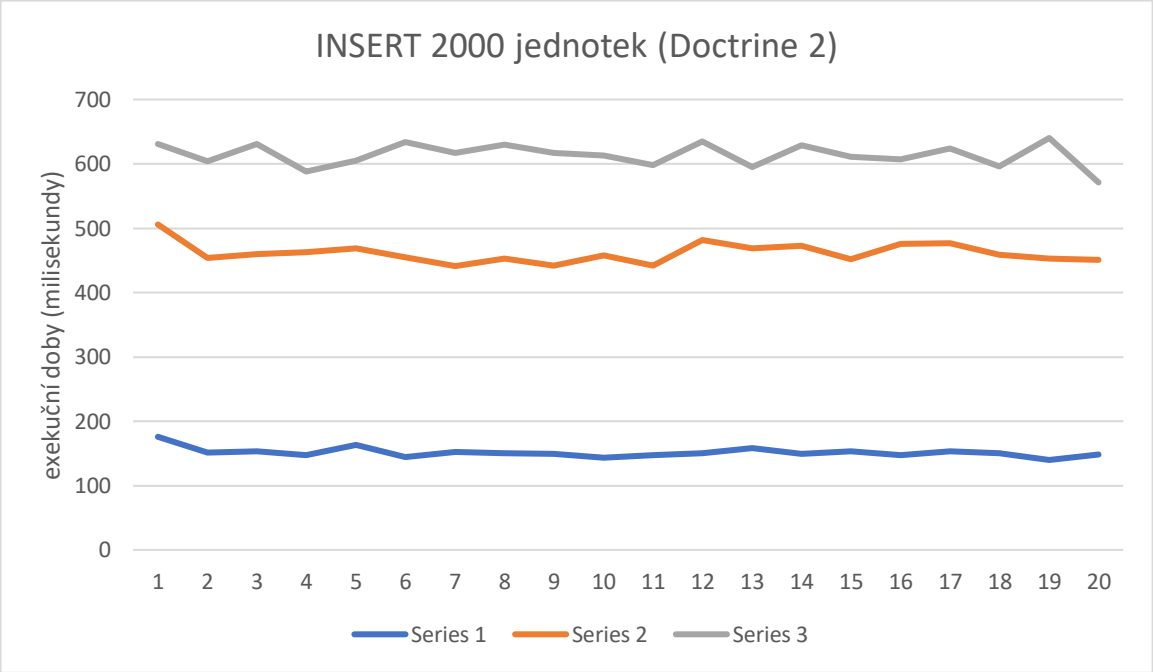
Graf 8- srovnání všech třech složitostí pro Select s 2000 jednotek v Nette



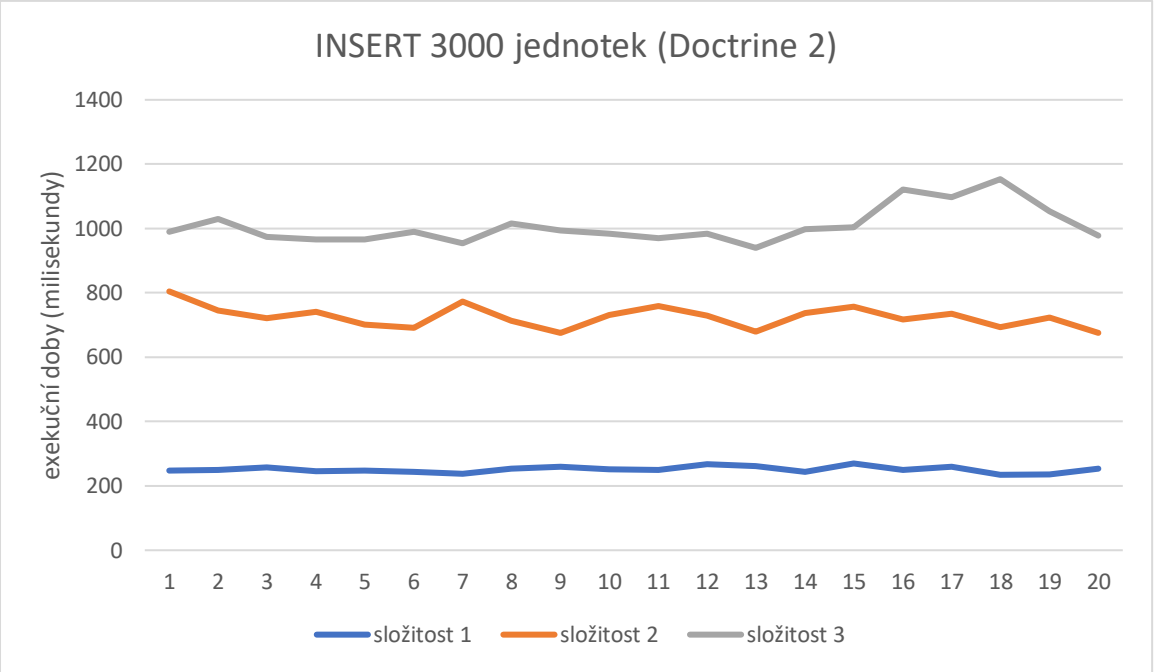
Graf 9- srovnání všech třech složitostí pro select s 3000 jednotek v Nette



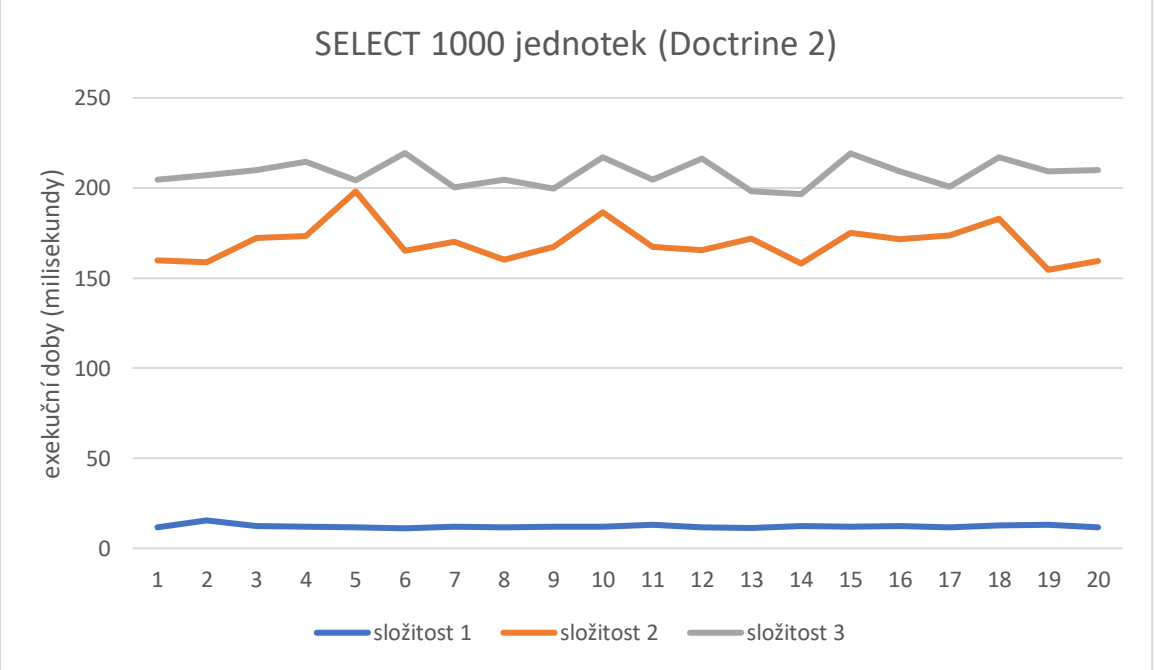
Graf 10- srovnání všech třech složitostí pro Insert s 1000 jednotek v Nette



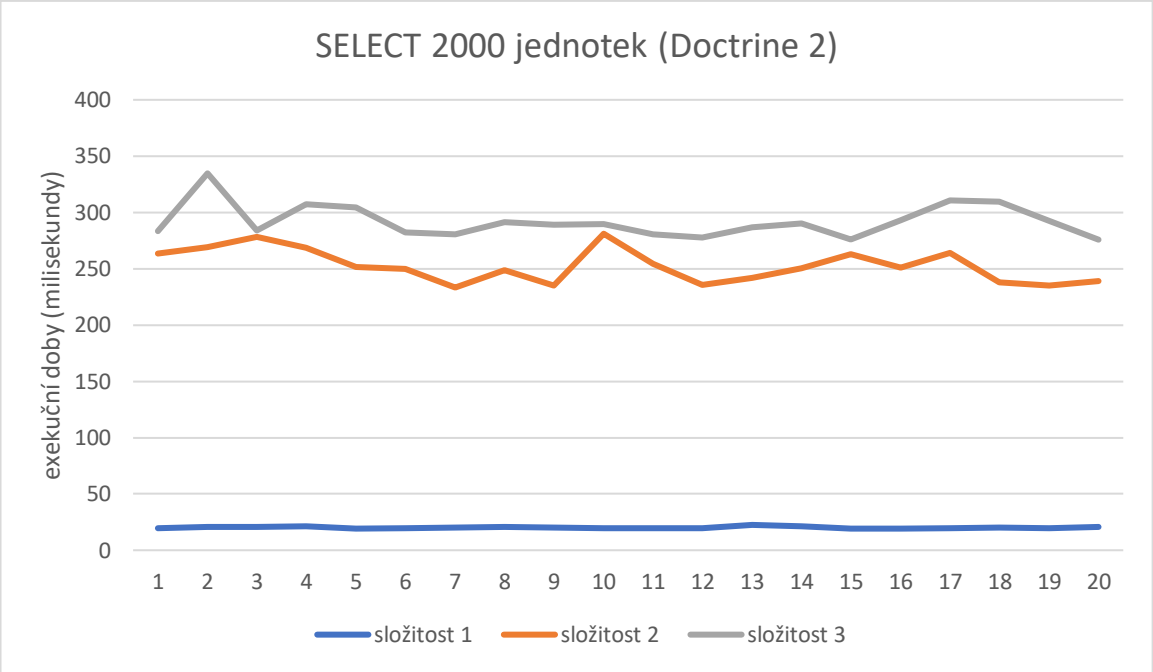
Graf 11- srovnání všech třech složitostí pro Insert s 2000 jednotek v Doctrine 2



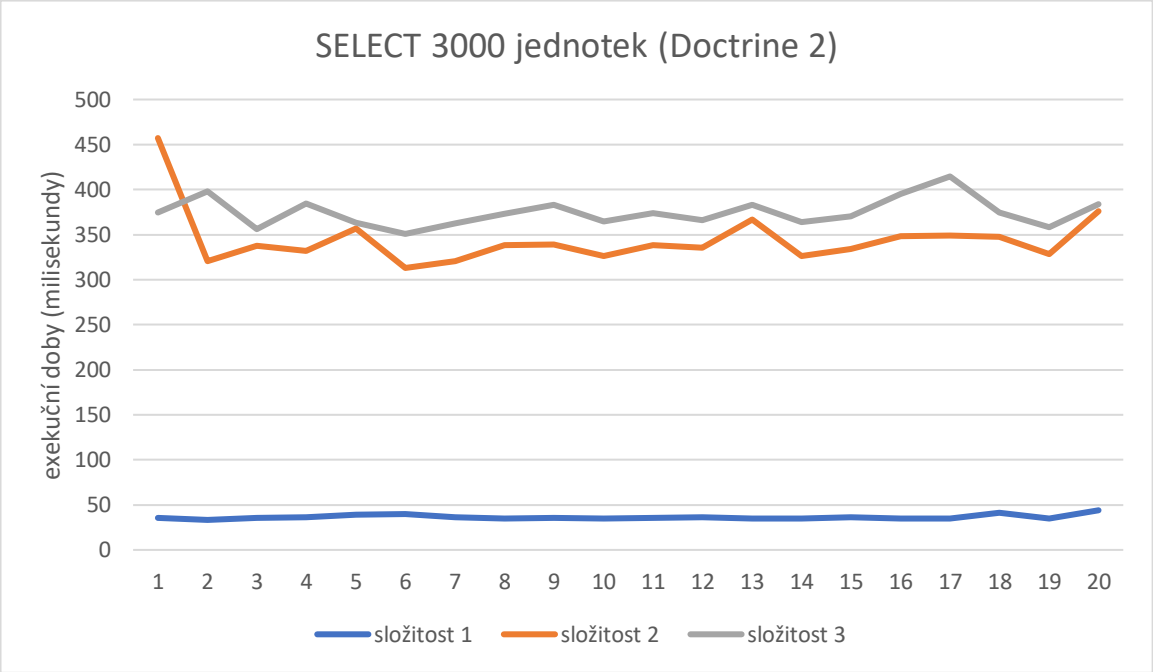
Graf 12- srovnání všech třech složitostí pro Insert s 3000 jednotek v Doctrine 2



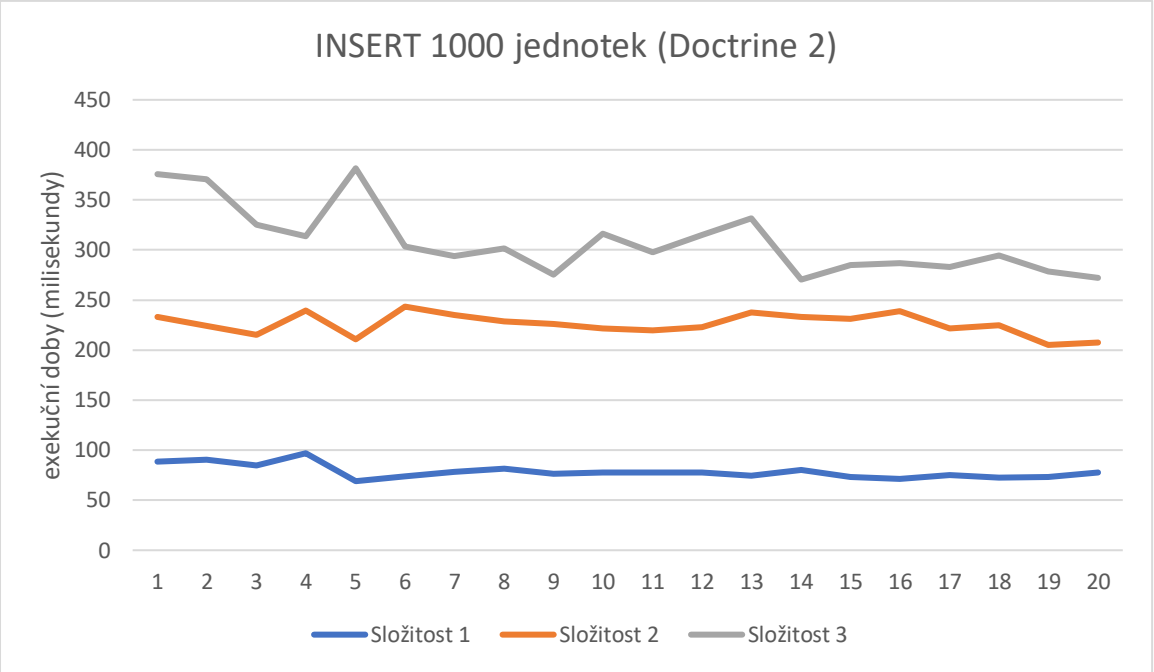
Graf 13- srovnání všech třech složitostí pro Select s 1000 jednotek v Doctrine 2



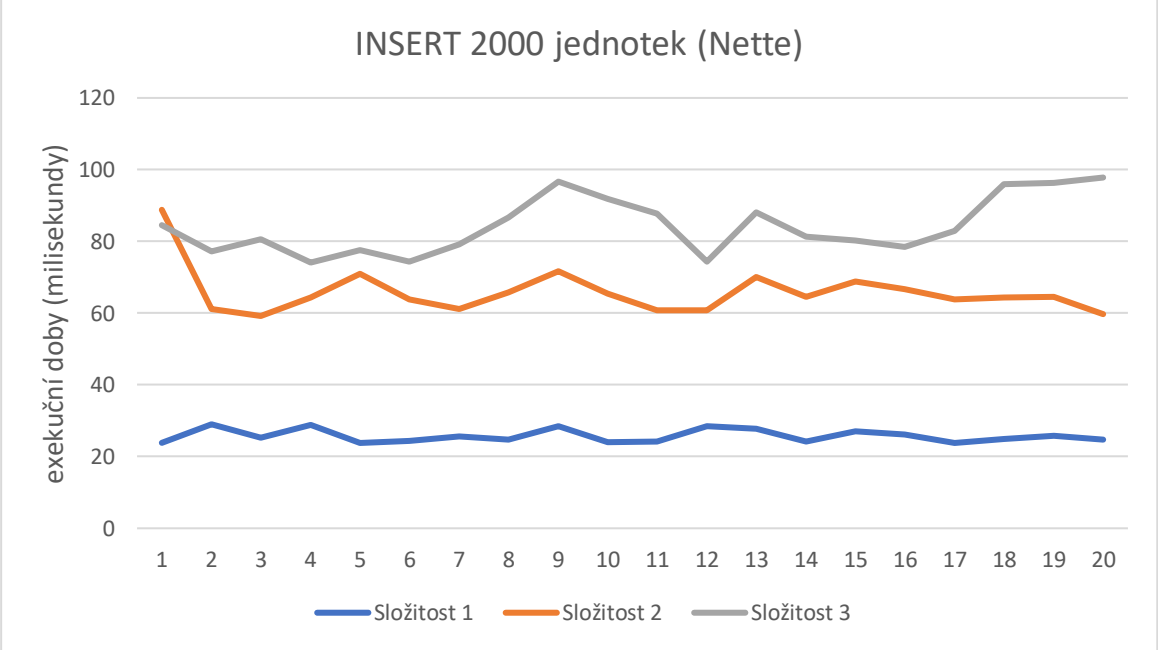
Graf 14- srovnání všech třech složitostí pro Select s 2000 jednotek v Doctrine 2



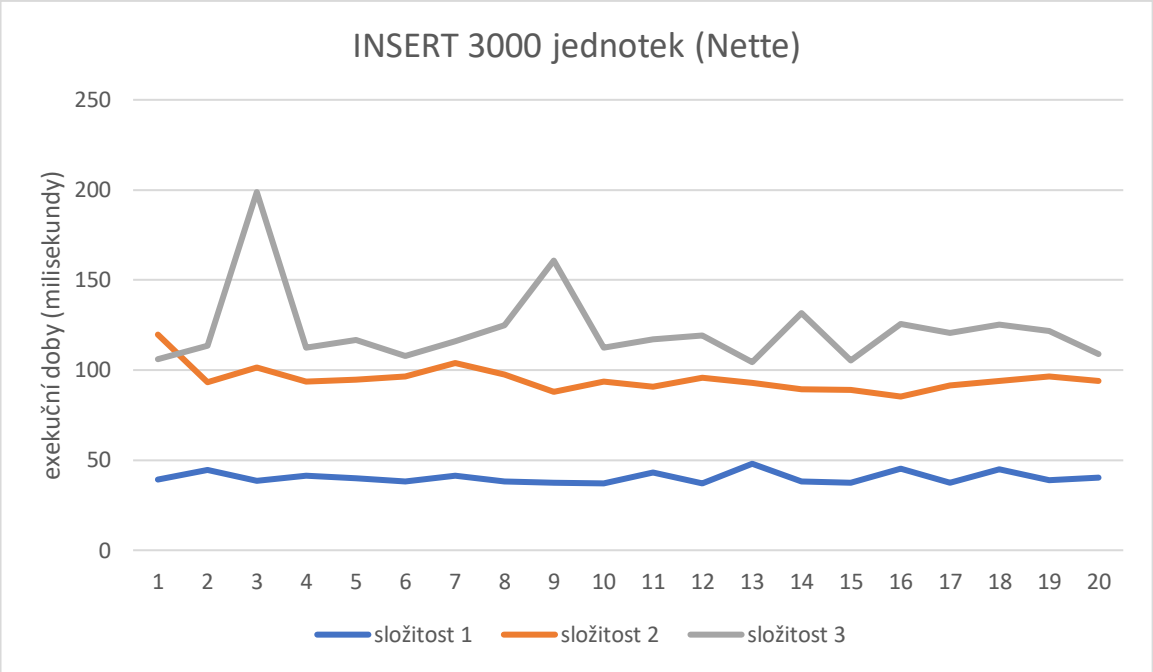
Graf 15- srovnání všech třech složitostí pro Select s 3000 jednotek v Doctrine 2



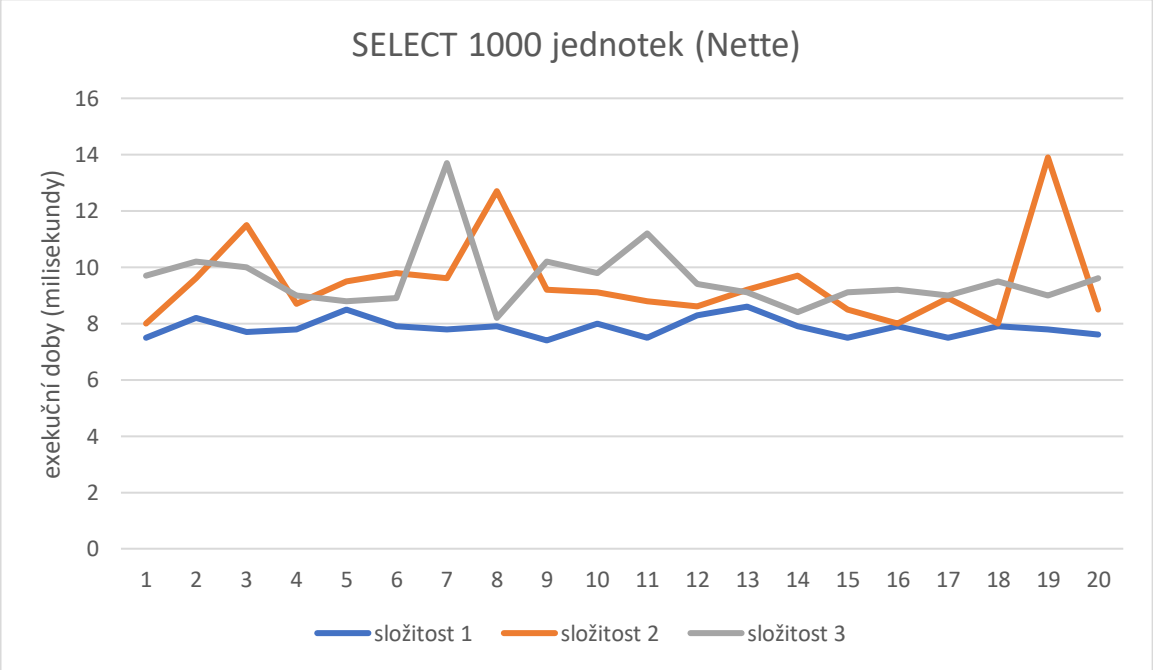
Graf 16- srovnání všech třech složitostí pro Insert s 1000 jednotek v Doctrine 2



Graf 17- srovnání všech třech složitostí pro Insert s 2000 jednotek v Nette



Graf 18- srovnání všech třech složitostí pro Insert s 3000 jednotek v Nette



Graf 19- srovnání všech třech složitostí pro Select s 1000 jednotek v Nette