České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra mikroelektroniky
Obor: Elektronika

Akcelerace ORM prostřednictvím transparentního dávkového zpracování

ORM acceleration based on transparent batched processing

DIPLOMOVÁ PRÁCE

Vypracoval: Michael Voříšek
Vedoucí práce: Ing. Petr Macejko
Datum odevzdání: 23. června 2020

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Voříšek** Jméno: **Michael** Osobní číslo: **434944**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Elektronika a komunikace**

Studijní obor: **Elektronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Akcelerace ORM v PHP prostřednictvím transparentního dávkového zpracování

Název diplomové práce anglicky:

ORM Acceleration in PHP Based on Transparent Batched Processing

Pokyny pro vypracování:

1) Prostudujte problematiku ORM (Object-Relational Mapping) v PHP a shrňte aktuální stav implementací.
2) Na základě získaných poznatků navrhněte koncept pro uložení požadavků a jejich dávkového odesílání do databáze. Návrh přizpůsobte tomu, aby z pohledu programátora bylo vše transparentní a systém se choval stejně jako by data byla odesílána sekvenčně včetně případných výjimek při ukládání.
3) Návrh realizujte formou znovupoužitelné knihovny v jazyce PHP.
4) Knihovnu použijte pro reálnou implementaci API pro zpracování dat z embedded HW / USB HID zařízení (např. klávesnice) a ověřte parametry.
5) Kriticky zhodnoťte Vámi zvolený postup a implementaci. Navrhněte doporučení pro případný další postup řešení problému.

Seznam doporučené literatury:

Persistence in PHP with the Doctrine ORM, Kévin Dunglas, ISBN: 1782164103
Hibernate Tips: More than 70 solutions to common Hibernate problems, Thorben Janssen, ISBN: 1544869177
Principles of Distributed Database Systems, M. Tamer Özsu, Patrick Valduriez, ISBN: 1441988335
Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert C. Martin, ISBN: 0134494164

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Peter Macejko, katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.10.2019** Termín odevzdání diplomové práce: **14.08.2020**

Platnost zadání diplomové práce: **30.09.2021**

Ing. Peter Macejko podpis vedoucí(ho) práce	prof. Ing. Pavel Hazdra, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Prohlášení

Čestně prohlašuji, že jsem práci vypracoval samostatně. Škole poskytuji plnou možnost práci posoudit v souladu se zákonem ČVUTu o závěrečných pracích, zejména zda splňuje požadavek na prokázání, zda student umí samostatně pracovat. Jakékoliv další práva se neposkytují a musejí být dopředu konzultovány.

V Praze dne 23. června 2020

....

Michael Voříšek

Poděkování

Na tomto důležitém místě děkuji vedoucímu práce, Ing. Petru Macelkovi, za jeho skvělou dostupnost a velmi individuální spolupráci v odborné i formální rovině.

Děkuji rodičům, kteří se o mě od -0. věku starali, dodali mi úžasné zázemí a přežili to se mnou. Mami, tati, díky!

Děkuji učitelům ze ZŠ, SŠ, VŠ, osobním mentorům, koučům, co mi denně ukazovali správnou cestu a otevřeně mi dávali ten nejupřímnější feedback.

Děkuji spoluzakladateli mé firmy, Mgr. Henrichu Paľovi, za uplynulých více než 10 let, kdy jsme se potkali u vaření čajů, jako absolutní ňoumové v podnikání pokusili vystřelit do světla přelomový produkt až po skvělé osobní zážitky a postupné odhalování pravidel hry. Velké poděkování patří také našim zaměstnancům, dodavatelům a v neposlední řadě také zákazníkům.

Děkuji také všem kamarádům a kamarádkám za všechny výhody, názory a nezapomenutelné chvíle.

Také děkuji všem, které jsem nezmínil, a v neposlední řadě také haterům za jejich názory a motivaci.

Přátelé, děkuji, jsem Vám zavázaný!

Michael Voříšek

Název práce:
Akcelerace ORM prostřednictvím transparentního dávkového zpracování

Autor: Michael Vořísek

Studijní program: Elektronika a komunikace
Obor: Elektronika
Druh práce: Diplomová práce

Vedoucí práce: Ing. Petr Macejko
Katedra mikroelektroniky,
Fakulta elektrotechnická,
České vysoké učení technické v Praze

Abstrakt: Práce ukazuje, že PHP je dostatečně rychlý jazyk i pro velké objemy dat, a to i při použití ORM a zachování výhod z toho plynoucích. Úzkým hrdlem současných systémů je ukládání záznamů do databáze po jednom, což je pomalé. Nejpopulárnější ORM knihovna současnosti, Doctrine ORM, podporu hromadného zpracování nenabízí, přitom právě využití ORM nabízí perfektní znalosti o datech, aby je bylo možno spravovat hromadně a zrychlit tím výsledný čas minimalizací celkového počtu dotazů. Jako ideální se ukázalo slučovat desítky až stovky řádků pro uložení současně a výsledně tím bylo dosaženo až 50x zrychlení oproti stávajícím řešením.

Klíčová slova: databáze, php, orm, doctrine, optimalizace

Title:
ORM acceleration based on transparent batched processing

Author: Michael Voříšek

Abstract: This thesis shows PHP is fast enough language also for huge data even if they are managed thru ORM while keeping all it's benefits. The bottleneck of current systems is the updates are done one-by-one which is very slow. The most popular ORM library currently, Doctrine ORM, does not offer bulk processing, but in fact the use of ORM offers a perfect knowledge about the data to be processed in bulk and decrease the total time by minimizing RTD/query count. It showed up it is ideal to group tens to hundreds of rows to update at once resulting in up to 50x faster processing vs. the current solutions.

Key words: database, php, orm, doctrine, optimalization

Obsah

Úvod **13**

1 Motivace a cíle **15**
1.1 Motivace . . . 15
1.2 Cíle . . . 16

2 Seznámení s jazykem PHP a jeho základními koncepty **17**
2.1 PHP . . . 17
2.2 Reference . . . 18
2.3 Třída . . . 18
2.4 Interface . . . 18
2.5 Instance / Objekt . . . 18
2.6 Call stack / debug trace . . . 19
2.7 GC . . . 19
2.8 Throw výjimky . . . 19
2.9 Kontext funkce . . . 19
2.10 Late static binding . . . 19
2.11 Trait . . . 20

3 Analýza problémů a použité technologie **21**
3.1 Slepá cesta - celá vlastní implementace . . . 21
3.2 Slepá cesta - mezivrstva mezi ORM a databází . . . 22
3.3 ORM . . . 22
3.4 Active pattern . . . 23
3.5 Data mapper . . . 24
3.6 Entity manager . . . 25
3.7 RDB . . . 26
3.8 Současná struktura a typické workflow Doctrine ORM . . . 26

4 Nutné úpravy a související problémy **29**
4.1 Přímé úpravy Doctrine ORM . . . 29
4.2 Co představuje údržba forku? . . . 30
4.3 Jak lokalizovat chybu v anonymní funkci? . . . 30
4.4 Omezení interface pro použití s traitem . . . 31
4.5 Použití profileru na nízkoúrovňové metody . . . 31
4.6 Xdebug a jeho vylepšení . . . 32

5 Použití v praxi **33**

5.1 Hromadný import/úprava dat 33
5.2 Zpracování dat z embedded hw 33

6 Zhodnocení výsledků a diskuse **37**
6.1 Porovnání stavu před a po 37
6.2 Kompatibilita 37
6.3 Benchmark - testovací scénář 38
6.4 Benchmark - ukázka generovaných SQL příkazů 39
6.5 Benchmark - vyhodnocení zrychlení 39
6.6 Jak moc to bylo náročné? 41
6.7 Výrazné přínosy pro celou PHP komunitu 41
6.8 Možné pokračování dalšího vývoje 41

Závěr **43**

Literatura **43**

Přílohy **47**

Úvod

Data jsou všude kolem nás. Mohou to být dokumenty, fotky, data z IoT, like na soc. síti, nebo třeba i jen nevinný záznam o pohybu myši.

Za poslední desítky let lze vypozorovat jasně rostoucí trend v objemu dat celkově, per zařízení a per proces/vlákno programu. Klientské zařízení typicky produkují data postupně a postupné zpracování není typicky zásadním limitem.

Centralizované servery obsahují typicky rádově větší objemy dat a rychlost práce s nimi je zásadní konkurenční výhodou. Technologie na této straně se nazývají backend.

Efektivnímu zpracování na straně backendu se věnuji již od střední školy a proto jsem se rozhodl na toto téma navázat a využít zkušeností, kdy jsem již několikrát začal úplně od nuly, protože původní cesta nebyla špatná, ale nebyla také ideální a spousta rozhodnutí byla potřeba od základu učinit jiná. Zároveň jsem za tuto dobu vyzkoušel snad všechny programovací jazyky - Javu, C Sharp, C, Python... ale pro kompromis mezi C syntaxí a flexibilitou zůstal u PHP. Samotné PHP nyní také pomáhám přímo rozvíjet, stejně jako některé open source knihovny v něm napsané.

V této práci se zabývám analýzou stávajících technologií, jejich omezením, výhodami, jejich vysvětlením. V implementační kapitole se potom zaměřuji na zaintegrování těchto postřehů a zkušeností do nejpopulárnější ORM knihovny pro PHP - knihovny Doctrine ORM.

ORM pro svoji důležitost vysvětlím stručně v úvodu: Object-relational mapping - Obousměrné mapování dat mezi databází a objekty v cílové aplikaci. Používá se pro trvalé uložení dat bez nutnosti psát individuální SQL dotazy. Pro uživatele/programátora je to přínosné proto, že nemusí řešit specifika databáze, návrh je výrazně rychlejší, nelze takto snadno vytvořit bezpeční trhlinu a takto napsanou aplikaci lze podstatně snáze rozvíjet/refaktorovat.

Závěrem práce je diskuse řešení, možné kroky do budoucna a samotný závěr.

Kapitola 1

Motivace a cíle

1.1 Motivace

Než se čtenář ponoří do samotného obsahu, je vhodné si připomenout typickou webovou aplikaci/službu.

Webserver přijme požadavek a zavolá jeden specifický vstupní skript, kterému předá parametry uživatelského požadavku.

Vstupní skript spustí inicializaci, tj. načte konfiguraci, další skripty, naváže spojení s databází, ...

Poté se typicky zavolá metoda, která má za úkol požadavek vyřídit a její práce s daty typicky vypadá následovně:

1. načti/vytvoř záznam
2. uprav záznam
3. ulož záznam

Pokud je požadavkem úprava jednoho nebo několika málo záznamů, tento postup je při dodržení standardních programovacích zásad rychlý.

Pokud je však vstupem např. seznam vyrobených dílů za celý den, který může mít i tisíce či milióny záznamů, tento způsob zpracování je sice formálně nadále správný, ale z pohledu uživatelské přívětivosti katastrofální. Výsledkem je obvykle pomalu přibývající progress bar a většinou nespokojený uživatel.

Problémem je sériové zpracování záznamů ve smyčce, která obvykle provede hned několik dotazů do databáze popř. na disk či jiné persistentní úložiště.

Uvědomělý čtenář zbystří, a správně se dotáže, proč záznamy do databáze neuložit

najednou. Ano, to lze, ale za předpokladu, že autor softwaru má tuto vrstvu v aplikaci pod kontrolou, typicky pokud záznamy ukládá přímo pomocí SQL nebo vrstvy, která nativní SQL abstrahuje na podobnou úroveň.

V momentě, kdy autor softwaru používá vyšší míru abstrakce a využívá např. ORM, je nutné, aby skupinové změny použitá knihovna podporovala.

Autor práce se historicky snažil kvůli těmto omezením knihovnám s vyšší abstrakcí vyhnout a dokonce stojí za několika zdárnými implementacemi vlastních řešení, ale vlastní řešení obsahují nespočet limitací a ne vždy je snadné je dále integrovat. Autor práce tedy postupně dospěl k názoru, že vlastní řešení postupně převede na industry standard technologie. Paradoxně, nejrozšířenější ORM knihovna pro PHP - Doctrine v poslední verzi 2.7 (v době psaní této práce) však data ukládá záznam po záznamu. Již několik let se vyvíjí nová verze 3.0, ale její autoři si zřejmě dali až moc vysoké požadavky a datum vydání je tak velmi nejistý, autor spíše očekává, že tento rok k vydání nedojde.

Proto se autor práce rozhodl pomalé, postupné zpracování adresovat a to přímou úpravou samotné Doctrine knihovny.

1.2 Cíle

Cílem je maximalizovat rychlost při zachování stávajícího rozhraní definovaného knihovnou Doctrine. Dávkové ukládání je z pohledu aplikace plně transparentní.

Operace, které nelze provést dávkově se nesmí takto ani zpracovávat.

Operace, které se standardně prováděly v určitém pořadí, které ale není přesně definováno, mohou mít pořadí jiné, ale musí splnit specifikaci, tedy nutné/explicitní požadavky na jejich pořadí.

Kapitola 2

Seznámení s jazykem PHP a jeho základními koncepty

V této sekci autor práce vysvětluje základní bloky programovacího jazyka PHP. Předpokládá se průměrná znalost programování.

Autor práce se rozhodl pro seznámení s těmito pojmy najednou v této úvodní kapitole pro pohodlí čtenáře. Doctrine knihovna je sice silně inspirovaná knihovnou Hibernate[7] pro Javu, ale implementace se liší pro efektivní využití jazyka jak z pohledu výkonu, tak z pohledu intuitivního použití uživatelem.

2.1 PHP

Programovací jazyk spojující výhody C syntaxe (složené závorky, ...) a výhody vyšších jazyků (automatická alokace paměti, není nutné typovat proměnné, ...).

Mezi nestandardní vlastnosti patří:
- `string` a `array` jsou datové typy předávané standardně jako hodnota, tedy změna hodnoty v novém kontextu nezpůsobuje změnu v kontextu/call stacku nadřazeném.
- jazyk je v základu single thread, což klade důraz na rychlý kód, příp. paralelizaci hned na vstupu - výsledkem jsou minimální nároky na zamykání/synchronizaci uvnitř aplikace
- proménné, metody, apod. lze odkazovat za běhu dynamicky - př. `$x->a` je ekvivalentní `$x->'a'`
- možnost rozšíření/úprava prakticky čehokoliv pomocí rozšíření bez nutnosti rekompilovat celý interpreter

Autor této práce nesouhlasí, že jde o jazyk specializovaný na webové stránky. Jazyk je dle autora velmi použitelný i na systémové skripty.

Mezi nevýhody patří nekonzistentní názvosloví/pořadí parametrů a nižší výkon oproti nativnímu C pro velmi nízkoúrovňové použití.

2.2 Reference

Ukazatel na proměnnou. V PHP se, na rozdíl od jiných jazyků, řetězce `string` a pole `array` předávají jako hodnoty a to stylem copy-on-modify, tj. předávání je rychlé, k fyzické kopii dojde až při změně dat, pokud je na ně v okamžiku změny více než jedna reference.

2.3 Třída

Elegantní zápis pro definici proměnných a metod. Definici třídy nelze po načtení dané třídy nikterak upravit (bez modifikace/vlastní kompilace, nebo rozšíření). Doctrine však potřebuje pro lazy loading třídy modifikovat. PHP sice umožňuje vytvořit anonymní třídu programově, ale předka nelze definovat proměnnou (omezení parseru). Řešením je negenerovat kód třídy a tento kód načíst.

Pro načtení se nabízí `eval` funkce, ta je však obecně nebezpečná, na některých systémech tedy i zakázaná, a zároveň obsahuje obrovskou nevýhodu - vstup nelze cachovat. Cachovat na úrovni parseru. Parsování kódu a jeho optimalizace před spuštěním je náročná operace. PHP již několik let nabízí cachování mezikódu rozšířením opcache. Pokud tedy nagenerované třídy uložíme na disk, mohou se tyto dříve načtené třídy, pokud nedošlo k jejich změně, při dalším požadavku načíst rovnou naparsované.

2.4 Interface

Definice prototypů metod, česky rozhraní. Třída jej může implementovat a potom se lze spolehnout, že třída tyto metody bude poskytovat. PHP v rozhraní, v době psaní této práce, nepodporuje jakoukoliv implementaci a dokonce ani jakoukoliv jinou viditelnost než public.

2.5 Instance / Objekt

Data definovaná třídou. V PHP je však možné vlastnosti instance úplně odebrat nebo dokonce přidat další dříve nedefinované.

2.6 Call stack / debug trace

Materializovaný stack (zásobník) volání. Slouží zejména pro ladění, v PHP lze však použít i pro dynamické chování na základě nadřazených volání. Pozor, při lokální úpravě předaných dat se tyto data změní i v call stacku. Tato optimalizace umožňuje PHP nižší overhead při volání funkcí, zvláště pro často volané "rychlé" funkce jakými jsou např. gettery, settery.

2.7 GC

Garbage collector se stará o uvolňování zdrojů z izolované, již neodkazované paměti. Někdy je potřeba, aby GC uvolnil paměť i při určitých referencích, takové reference je možné realizovat od PHP 7.4 pomocí `WeakReference` třídy.

2.8 Throw výjimky

Koncept jak předávat chyby v aplikaci v rámci call stacku. Call stack však neobsahuje reference na anonymní metody, takže nelze tyto metody znovu zavolat.

2.9 Kontext funkce

Prostředí proměnných. Uvnitř nestatické metody definuje `$this` (referenci na třídu instance s danou metodou) a `static` (1:1 ekvivalent `get_class($this)`).

2.10 Late static binding

Odkazy na třídy pomocí jejich jména je nemožné (bez externího rozšíření) přetížit. PHP proto zavedlo výraz `static`, který odkazuje na třídu z kontextu. Pozor však na specifikum PHP nazývané "forwarding", `static::method()` a `(static::class)::method()` by očekávaně mělo zavolat stejnou metodu, avšak nezavolá - volání metody přes klíčové slovo `static` použije `$this` namísto očekávané třídy z kontextu, která může být rodičovská, a tedy nemusí se rovnat třídě instance. Ačkoli PHP klade velký důraz na AST, některé koncepty OOP, se chovají jinak při použití s vlastností, metodou, a pro hlubší použití je vhodné se seznámit s přesným chováním prostřednictvím oficiální dokumentace/otevřeného kódu[4].

2.11 Trait

"Copy paste" části definice třídy, z pohledu programátora jediná možnost multidědičnosti (možnosti složit definici třídy z více tříd/neduplikovaných částí kódu). Pokud však trait vkládáme na více míst, tj. výsledkem bude několik tříd bez společného předka, je vhodné takto vložené metody vynutit pomocí rozhraní. Pozor, může však obsahovat metody pouze s viditelností public.

Kapitola 3

Analýza problémů a použité technologie

V ideálním světě, perfektním kódu/návrhu je udělat změny snadné. Upraví se jedna vrstva a řešení je hotové. V této části práce je nastíněn výběr problémů, které autor řešil.

Než bude čtenáři předloženo řešení, autor práce považuje za nutné čtenáře seznámit se slepými cestami, které autor prošel, ale do cíle nevedou.

3.1 Slepá cesta - celá vlastní implementace

Kvalitní programátor ví, co je kombinatorická exploze a umí si velmi snadno představit realizaci a reálnou rychlost myšlenky. Programátor se dá do práce a do týdne, měsíce, může mít hotový produkt.

Pomineme-li nepsané pravidlo, že napsání kódu je cca 1/10 práce, další prací jsou testy, podpora - rozšiřování, adaptace novou verzi PHP, db, oprava případných chyb atd., má vlastní řešení jedno výrazné omezení - může být rychlé pro jeden účel, ale nelze jej snadno integrovat. Ano, vlastní implementace může podporovat všechny rozhraní cílových rozhraní - ale proč Doctrine vyvíjí desítky php guruů uctívaných php komunitou? Protože tato implementace je složitá, pokud je cílem právě rychlost, tj. požadavek na lokální cachování bez nutnosti volat/aktualizovat databázi po každé operaci.

Ve zkratce, vlastní řešení úplně od nuly je skvělé na pochopení, jak jednotlivé knihovny, návrhové přístupy fungují, jak využít silných stránek jazyka, ale komplexní řešení není snadné s omezenými zdroji nahradit.

3.2 Slepá cesta - mezivrstva mezi ORM a databází

Před rozhodnutím upravit Doctrine autor vyzkoušel problém řešit na úrovni SQL a to pomocí implementace vlastního SQL/PDO driveru, který by příkazy bufferoval a provedl je později najednou.

Řešení je výhodné ve smyslu další transparentní vrstvy, avšak pro univerzální použití vyžaduje plný SQL parser - který v PHP je cca 20x pomalejší oproti kompilovanému C. Jelikož parsování dotazu stojí výrazný podíl spotřebovaného procesorového času, celkové zrychlení je tudíž výrazně omezeno ještě něž vůbec došlo ke spojení s databází.

Dalším výrazním omezením je, že toto řešení musí insert/update/delete příkazy ihned potvrzovat a v případě jejich selhání (např. chybnou hodnotou, cizím klíčem) je potom téměř nemožné pro aplikaci selhání izolovat na problémový příkaz/záznam a řešit individuálně. Jediným výstupem je provedeno, nebo chyba celé transakce.

Dalším poměrně komplikovaným problémem je správné řazení pro seskupení a sloučení příkazů, kde následující příkazy spoléhaly na již provedené úpravy.

Výstupem této cesty se stal však celkem robustní nástroj na logování provedených změn, kdy stačí aplikaci předat tento proxy driver. Driver 1:1 předává požadavky dál, aplikace tedy funguje naprosto stejně, ale jakékoliv změnové příkazy jsou logovány. Na rozdíl od ostatních řešení, toto řešení umí logovat nejen managed změny (např. pomocí ORM), ale změny i při použití nativního SQL.

Lze také velmi snadno rozšířit o logování select dotazů a tedy využít pro bezpečnostní logy s kompletním přehledem z jakého stroje, kódu, ... byly které záznamy prohlíženy.

Požadované zrychlení však tento přístup nabídnout nedokázal.

Obě dvě slepé cesty vyžadovaly měsíce práce. Aby se čtenář, zabývající se stejnými problémy, vyvaroval podobných chyb, je nutné jej seznámit s dostupnými řešeními a základními přístupy k práci s daty.

3.3 ORM

Zkratka pro anglické spojení "Object-relational mapping". Jedná se o techniku namodelování dat v nativních objektech daného jazyka pro co nejlepší využití jeho výhod a abstrahování specifického přístupu (SQL, API) ke zdrojovým datům.

Pro tuto techniku existují v zásadě dvě možnosti lišící se primárně tím, zda-li každý unikátní záznam znovu použije stejný objekt nebo se pro každý nový/nově načtený záznam vytvoří nová instance s tímto záznamem (nejlépe) pevně svázaná.

3.4 Active pattern

Návrhový vzor active pattern se vyznačuje implementací, která:

1. "přepisuje data" ve stejném objektu/instanci při načtení jiného záznamu
2. (alespoň základní) metody pro práci s ním jsou dostupné/implementovány v tomto datovém objektu

Příklad typického použití:

```
1      class Model {
2          public $id;
3
4          public function loadById(int $id): static
5          {
6              // reload data by ID
7
8              return $this
9          }
10     }
11
12     class User extends Model {
13         public $name;
14     }
```

Z příkladu je zřejmé, že tento přístup trpí, alespoň dle názoru autora, neoptimálním návrhem:

- objekt musí rozšiřovat specifickou třídu
- data k záznamu nelze snadno v rámci aplikace uložit - data svázaná s objektem se při dalším načtení změní

Mezi pozitiva patří možné intuitivnější použití při prvním seznámení a správné našeptávání návratového typu u metod v rámci IDE (PHP nemá podporu generiky).

Active pattern je velmi rozšířený zejména pro nižší výpočetní náročnost (není potřeba vytvářet nové objekty pro každý záznam a trackovat je - viz. data mapper vzor dále) a historicky (kdy se nepoužíval/jazyky nepodporovaly magický lazy loading).

Populárními implementacemi jsou Lavarel/Eloquent ORM, RedBeanPHP či Propel.

Jelikož autor tento přístup nepodporuje, dále se analýzou knihoven založených na tomto návrhovém vzoru nazabýval.

3.5 Data mapper

Návrhový vzor ORM alternativou k active pattern. Tento vzor se vyznačuje:

1. mapováním dat na unikátní objekty pro jednotlivé unikátní záznamy
2. oddělením implementace pro práci s nimi

Příklad:

```
class User {
    public $id;
    public $name;
}

class ModelManager {
    public $objectsById = [];

    public function loadById(int $id): object
    {
        if (isset($this->objectsById[$id])) {
            return $this->objectsById[$id];
        }

        // load data by ID

        return $obj;
    }
}
```

Z příkladu lze vypozorovat:

- datový model definuje pouze "co" (datový popis) a ne "jak" (implementaci)
- 1 instance entity (datového modelu) odpovídá právě jednomu unikátnímu klíči

Autor práce je přesvědčený, zejména na základě popsaných výhod a zkušeností z předchozích implementací, že toto je jediný správný přístup pro větší projekty, a na základě toho padla jednoznačně volba na knihovnu Doctrine ORM.

Doctrine ORM je v současnosti jednoznačně nejpopulárnější ORM knihovnou implementující data mapper vzor. Alternativou je např. Cake ORM, popularita je však, dle GitHubu o více než řád nižší.[1] Autor se jinými knihovnami z tohoto důvodu dále

[1] Dominanci Doctrine ORM také naznačuje Reddit diskuse:

nezabýval.

3.6 Entity manager

Jelikož datový model při použití data mapper vzoru neobsahuje implementaci, je potřeba další třída, která definuje implementaci a nazývá se manager (česky správce). Manager se stará o načtení, uložení, v pokročilejších implementacích také o cachování, implementaci lazy loadingu, dealokování již nepoužívaných/nezměněných objektů.

Krátký příklad pro lepší pochopení lazy loadingu:

```
1   class Address {
2       public $id;
3       public $name;
4   }
5
6   class User {
7       ...
8       /** @var Address */
9       public $address;
10  }
11
12  class ModelManager {
13      ...
14
15      public function loadById(int $id): object
16      {
17          if (isset($this->objectsById[$id])) {
18              return $this->objectsById[$id];
19          }
20
21          // load data by ID
22
23          $obj->address = new class($this, $obj->addressId)
                extends Address {
24              public $manager;
25
26              public function __construct($manager, $id) {
27                  $this->manager = $manager;
28                  $this->id = $id;
29              }
30
31              public function __get($name) {
32                  if ($this->id === null) { // not loaded yet
33                      $obj = $this->manager->loadById($this->id);
34                      // set $this props from $obj
35                  }
36
37                  return $this->{$name};
38              }
```

https://www.reddit.com/r/PHP/comments/3ieffs/looking_for_a_data_mapper_orm/

```
39                };
40
41                return $obj;
42            }
43        }
```

Název adresy uživatele z příkladu lze poté získat pomocí:

```
1     $user = $manager->loadById(10);
2
3     $addressName = $user->address->name;
```

Zdánlivě normálně vypadající přístup k vlasnosti `name` objektu třídy `Address` zajistil načtení tohoto objektu až při samotném přístupu. Data jsou obvykle relačně provázána a lazy loading je nezbytný, pokud vyžadujeme transparentní traverzování a není žádoucí načíst při 1. načtení celou databázi.

3.7 RDB

Relační databáze, někdy také RDBMS (RDB management system), např. MySQL, SQLite, PostgreSQL. RDB je ideálním úložištěm pro uschování trvalých dat a jejich rychlé vydání. Nejčastější jazyk pro manipulaci s daty je jazyk SQL. Ačkoli je jazyk standardizovaný, jsou mezi jednotlivými vendory rozdíly, mezi některými převážně jen v syntaxi, některé databáze se však liší výrazněji.

Pro účel spojování zápisů/změn/mazání řádků je velmi podstatné, kdy databáze validuje cizí klíče - některé databáze integritu ověřují až při potvrzení transakce (commitu), MySQL však integritu ověřuje instantně při každém SQL dotazu. Jelikož podpora MySQL je nezměnitelným požadavkem, musí být toto omezení plně podporováno.

3.8 Současná struktura a typické workflow Doctrine ORM

Doctrine ORM lze rozdělit na tři části:

- datový model, tedy vše související s popisem modelu, tedy např. sloupce, jejich datové typy, relace mezi nimi
- "select logika", implementace zajišťující maximálně efektivní a použitelný přístup k existujícím datům
- "update logika", implementace, zajišťující uložení lokálních změn do databáze

Tato práce se zabývá výhradně "update logikou" a to její úpravou ze zpracování po jenom řádku na úpravu na až stovek záznamů najednou.

Jak již bylo nastíněno ve vysvětlení data mapper vzoru, data mapper vyžaduje managera. V Doctrine ORM se tento manager nazývá `EntityManager`.

EntityManager obsahuje reference na aktivní entity (načtené nebo připojené k manageru pomocí `EntityManager::persist`, aby je manager začal managovat), zároveň má uložená původní data a nabízí možnost entitu uložit/aktualizovat v databázi (pomocí `EntityManager::flush` metody).

Je vhodné upozornit, že před každým uložením jsou nová data porovnána s původními a ukládány jsou jen rozdílné. Z toho vychází důsledky:

- data, která jsou lokálně nastavena v jiném než kanonickém formátu (formát při převedení do DB, uložení do DB v cílovém datového typu a převedení zpět), mohou mít za následek neustálé ukládání (př. buď celočíselný typ s hodnotou `5`, avšak lokálně vždy přepočítaný na `5.2` výsledkem nějakého výpočtu - hodnota po uložení se nezmění, ale standardní Doctrine implementace tohoto datového typu hodnotu (5.2) při ukládání jen převede na string a tuto (ne)změnu nerozpozná)
- mohou vzniknou nepřípustné kombinace i napříč jednotlivými řádky (př. mějme sloupec `item` a `color`, ve dvou aplikacích načteme `< "Skoda", "zluta" >`, v jedné upravíme na `< "Mazda", "modra" >`, uložíme, a ve druhé upravíme na `< "Renault", "zluta" >` a uložíme - vždy se standardně uloží jen diff, v databázi bude následně záznam `< "Renault", "modra" >`, který dohromady nemusí být vůbec platný)

EntityManager "update logiku" implementuje pomocí třídy `UnitOfWork`, která:

- identifikuje změny (aktuální hodnoty nerovnající se původně načteným hodnotám)
- rozdělí je na insert/update/delete
- pomocích nadefinovaných cizích klíčů seřadí jednotlivé změny v rámci insert/update/delete operací tak, aby byla zachována okamžitá integrita
- zahájí transakci
- provede jednotlivé operace v daném pořadí
- pokud se všechny operace provedou bez chyby, transakce se potvrdí a lokální původní hodnoty se aktualizují na právě uložené, v opačném případě se provede rollback

Doctrine ORM však výkonnostní optimalizaci formou slučování SQL příkazů neřeší, autor se domnívá, že je to zejména proto, že všechny persist rozhraní jsou navržené výhradně pro zpracování jednoho záznamu, a oficiální změna by vyžadovala výrazné, zpětně nekompatibilní změny do masově používané knihovny.

Pozor, Doctrine používá pojem "batch processing"[2], který je však používán ve smyslu volání `EntityManager::flush()` po určitém počtu záznamů, to proto, aby se data mohla lokálně uvolnit z paměti. Pojem však vůbec nesouvisí se slučováním SQL dotazů.

[2]https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/batch-processing.html

Kapitola 4

Nutné úpravy a související problémy

4.1 Přímé úpravy Doctrine ORM

Tato podkapitola popisuje přímou úpravu kódu[2]. Podkapitola záměrně není napsána přímo jako návod krok za krokem, protože samotná knihovna Doctrine se neustále vyvíjí a refaktoruje. Cílem této kapitoly je navést čtenáře na správná místa, aby byl schopný Doctrine ORM knihovnu upravit sám. Předpokládá se pokročilá znalost PHP.

Dokud se nezavolá metoda `UnitOfWork::flush`, všechny změny jsou uloženy pouze lokálně. Z této metody se volá metoda `UnitOfWork::commit`, která ukládání řeší po jednotlivých operacích a následně po jednotlivých entity třídách. Spojit různé operace v rámci SQL nelze (např. INSERT a UPDATE, bez použití DB procedur, které se však výrazně liší napříč výrobci). Slučovat operace nad různými třídami entit by výrazně narušilo originální rozhraní Doctrine, proto postupné zpracování/rozdělení na této úrovni zůstane zachováno. S větším počtem záznamů by stejně nepřinášelo výrazné zlepšení.

Operace insert/update/delete mají svá specifika, v popisu níže je však pro jednoduchost popsána úprava pouze pro update. Zejména z důvodu, že zrychlení updatu je hlavním cílem této práce, myšlenky pro ostatní operace jsou podobné a změny pro insert/delete vycházely ze stejné myšlenky.

Metoda `UnitOfWork::commit` volá následně `UnitOfWork::executeUpdates` metodu, která originálně provádí jednotlivé změny pomocí `EntityPersister::update($entity)`.

`BasicEntityPersister::update` resp. `BasicEntityPersister::updateTable` metody jsou původně čistě per jednotlivá entita a musely být kompletně nahrazeny vlastní implementací.

Sloučení SQL dotazů pro select a delete je triviální. Update dotazy však mohou obsahovat různé nové hodnoty nebo změny v různých sloupcích. Autor vysvětlí na příkladu, uvažme tyto SQL dotazy ke sloučení:

```
1         UPDATE t SET colxx = 4 WHERE id = 1;
```

a

```
1         UPDATE t SET colyy = 8 WHERE id = 2;
```

tyto zdánlivě nesloučitelné dotazy však sloučit lze, a to takto:

```
1         UPDATE t SET
2         colxx = CASE WHEN id = 1 THEN 4 ELSE colxx END,
3         colyy = CASE WHEN id = 2 THEN 8 ELSE colyy END
4         WHERE id IN(1, 2)
```

výsledný dotaz je ekvivalentní předchozím dotazům, přiřazení z ELSE nemohou hodnotu, ani při vysoké souběžnosti/konkurenci dotazů, změnit, protože databáze všechny updatované řádky nejprve zamkne a vyřídí atomicky.

Jelikož však původní knihovna neobsahuje potřebné API pro potřebné úpravy, bylo nutné úpravu řešit forkem. Fork původní knihovny však představoval další výzvu a to udržování forku up-too-date s aktualizacemi v originální/upstream knihovně.

V následujících podkapitolách je popsán výběr problémů se kterými se autor setkal v průběhu realizace.

4.2 Co představuje údržba forku?

Údržba forku zdalena není je o zmergování nových změn. S údržbou se pojí další úkoly jako pravidelné testy (novější závislosti mohou projekt rozbít i bez jeho samotné změny) či třeba údržba balíčkovacího systému a jeho konfigurace, aby fork mohl nahrazovat balíček původní.

Autor v souvislosti s touto prací zprovoznil vlastní balíčkovací systém postavený na open source systému Satis. Zároveň vyvinul v PHP nástroj, který pomáhá fork aktualizovat a vydávat, respektive tagovat.

4.3 Jak lokalizovat chybu v anonymní funkci?

PHP je jazyk extrémně flexibilní, avšak toto je nyní jeho omezení - k anonymní funkci v nynější implementaci PHP (7.4 a zřejmě i připravované 8.0) nelze přistoupit, ani

pomocí `debug_backtrace`, ani `(new Exception)->getTrace()` (vrátí identická data).

Feature request veden pod označením PHP bug číslem 62325[1]. Jelikož toto omezení snižovalo rychlost vývoje, autor má v plánu se pokusit změnu do PHP protlačit, problémem je však kdy referenci na anonymní funkci uvolnit a zda-li do backtrace zahrnout i ne zcela zkonstruované objekty (probíhá diskuse napříč komunitou, zatím s nejednoznačným závěrem).

4.4 Omezení interface pro použití s traitem

Trait je v PHP jediné řešení vícedědičnosti. Následnou implementaci (použití traitu) je možné popsat pomocí interface, které má však nyní omezení pouze na public viditelnost a protected metody nyní pomocí interface deklarovat nejdou. Pull request přímo do PHP-src je k dispozici pod číslem 5708[2]. Klíčoví autoři jsou změně nakloněni, avšak změna bude v PHP nejdříve ve verzi 8.1, jde o změnu jazyka, a tudíž je dle komunitních pravidel PHP vyžadováno RFC a projevení zájmu celou komunitou v hlasování s požadavkem na alespoň 2/3 hlasů pro.

4.5 Použití profileru na nízkoúrovňové metody

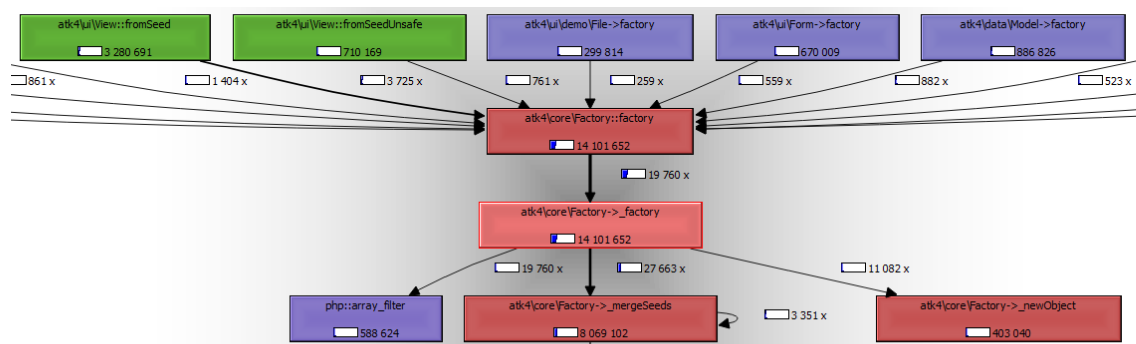
Pro identifikované problémy lze kód pro otestování výkonu obalit např.:

```
1          class Cl {
2              public $array = ['x' => 'xx', 'y' => 'yy'];
3
4              public function &getRef(int $i) {
5                  // two indexes to make sure we always return a
                       different reference
6                  return $this->array[$i % 2 ? 'x' : 'y'];
7              }
8          }
9
10         $cl = new Cl();
11
12         $t = microtime(true);
13         for ($i = 0; $i < 10000000; $i++) {
14             $res = [$cl->getRef($i)];
15         }
16         unset($res);
17         echo 'needed ' . round((microtime(true) - $t) * 10e9 / $i,
               3) . ' ns / iter';
```

tento způsob zaručuje nulový overhead (pomineme-li overhead daný for cyklem)

[1]https://bugs.php.net/bug.php?id=62325
[2]https://github.com/php/php-src/pull/5708

Pro identifikaci neznámého problému lze z kódu postupně odmazávat a hledat velký skok v rychlosti. Ne vždy to ale aplikace umožňuje. Potom je vhodné využít debug rozšíření PHP `xdebug` [5], zapnou tracing a výstup otevřít např. pomocí `QCacheGrind` [3].



Obrázek 4.1: Ukázka Xdebug grafu volání

Je však nutné si dát pozor na xdebug overhead (dost často i více než 10x), který je variabilní a fixní per PHP opkód/instrukce, tedy ne vždy je možno získat úplnou a přesnou představu.

Jelikož profiler zaznamenává čas, je nutné testy opakovat při stejné/minimální zátěži jinými procesy na testovacím počítači. Dále je potřeba zajistit co nejreplikovatelnější výkon hw po dobu a napříč testy. Je tedy vhodné vypnout Intel Thermal Velocity boost a podobné boost/thermal nastavení.

4.6 Xdebug a jeho vylepšení

Vzhledem k zaměření autora na optimalizace a čistý kód[8] se autor rozhodl využít časovou rezervu a xdebug overhead blíže pochopit.

Autor identifikoval hned několik absolutně kritických míst a řadu z nich se rozhodl v jazyce C adresovat.

Z nejdůležitějších lze jmenovat:
- zvětšení rozlišení, více než o řád, při měření času (což je naprosto kritické, pokud hlavní meřešenou veličinou je právě čas)
- optimalizace kódu s výrazně nižší dynamickou alokací paměti a omezení použití `sprintf` (obecně rodiny printf metod), které u takto kritických částí představovaly značný overhead
- implementace a prosazení změny výstupního formátu z 1 mikrosekundy na 10 nanosekund
Pro představu, nejrychlejší PHP funkce trvá okolo 10, 20 nanosekund, triviální operace např. přiřazení (`$a = 5;`) trvá okolo 5 nanosekund (na nejrychlejších procesorech v době psaní této práce).

[3] https://sourceforge.net/projects/qcachegrindwin/

Kapitola 5

Použití v praxi

V této kapitole jsou představeny dvě různé aplikace ukazující možnosti použití v praxi.

5.1 Hromadný import/úprava dat

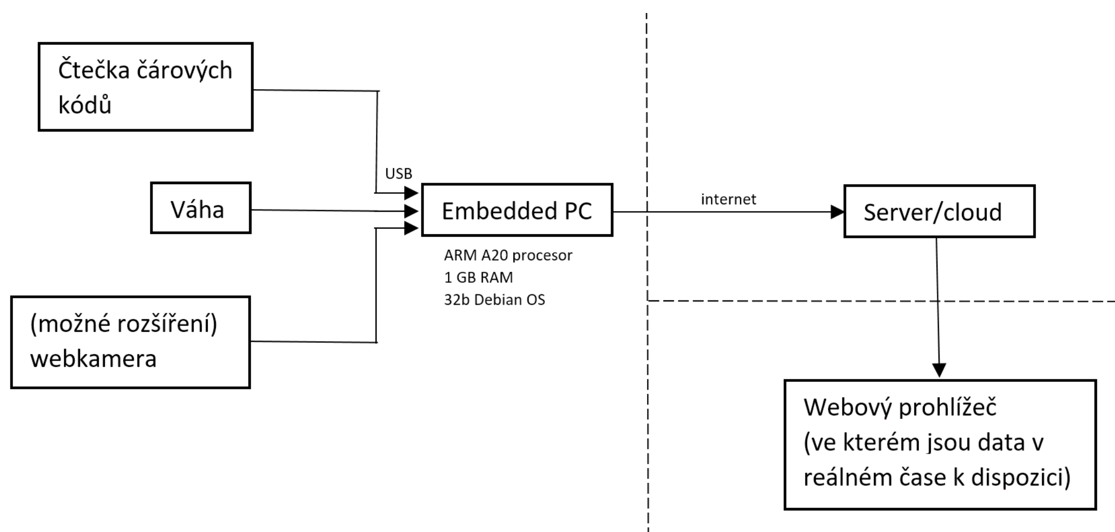
Import dat je ukázkový příkaz použití. Uživatel má větší objem dat, které potřebuje uložit. Při vědomí, že upravená knihovna umí komunikaci s databází slučovat, stačí, aby kód zpracovávající záznam po záznamu byl obalený v transakci (což mimo jiné představuje i správnou praxi), která se potvrdí na samotném záměru, příp. po např. 10k záznamech, pokud je vstup většího rozsahu.

Při úpravě stávajících dat je nutné zajistit načtení všech dat najednou (či větších dávkách). Načtení více než jednoho záznamu najednou je základní vlastnost knihovny Doctrine ORM a stačí ji jen správně využít. Tento problém se také mnohdy označuje 1:1+N (z typického scénáře, kdy načteme nekompletní data, které potom donačítáme ve smyčce).

Dalším velmi silným use casem, při kterém použití upravené knihovny může představovat dramatické zrychlení, je migraci/upgrade při změně struktury. Hodně uživatelů tento use case řeší nepoužitím ORM, což však výrazně narušuje čistotu tohoto procesu a bývá zdrojem chyb, nehledě na náročnost na lidské zdroje/přípravu takové migrace.

5.2 Zpracování dat z embedded hw

Druhou ukázkovou aplikací je realizace real time přenosu lokálních dat na server a (zpět k uživateli, resp. do jeho prohlížeče). Funkci aplikace znázorňuje následující blokové schéma:



Obrázek 5.1: Blokové schéma přenosu dat přes server až do prohlížeče uživatele

Jelikož použitá vstupní zařízení spadají do kategorie USB HID (human interface device), ve Windows nelze vstup z těchto zařízení zachytávat (ve smyslu potlačení jejich standardní funkce, tj. klávesnice v případě čtečky, váha spadá také do HID kategorie, avšak proprietárního typu), a to ani z uživatelského programu s plným oprávněným, natož webového prohlížeče.

Z tohoto omezení vyplynul požadavek, aby vstupní zařízení byla připojena síťově. Nabízela se hotová ethernetová řešení, avšak kompletně uzavřená a drahá. Z těchto důvodů padlo rozhodnutí na vlastní řešení postavené na embedded PC.

Po rychlé analýze vhodných technologií a zkušeností autor zvolil vlastní návrh na embedded linuxu a aplikaci založené na Javě.

Java aplikace, pro maximální univerzálnost, načte všechny připojené USB/BT zařízení a spolu se základními informacemi je pošle/přihlásí se na server. Server následně může embedded PC požádat o streamování dat z jednoho nebo více připojených vstupních zařízení.

Realizovaná aplikace klade důraz na:
- poradit si s jakýmkoli problémem (výpadek spojení, odpojení/připojení zařízení, ...) bez nutnosti uživatelského zásahu, tj. absolutní plug-and-play
- bufferování dat v případě výpadku spojení se serverem, zaroveň však o minimální latenci při dostupném nebo i nestabilním spojení

Pod nestabilním spojením autor myslí ztrátu nebo nenutné prodlení při doručování zprávy na server. Navržená aplikace tento problém řeší několika interními metrikami, a v případě nepotvrzení zprávy serverem, se pokouší data doručit na server poměrně rychle znovu (v řádu desítek až stovek milisekund po prvním pokusu).

Píše se rok 2020 a podobná řešení "klávesnice" na bázi systému, schopného přehrávat HD video, budou čím dál tím běžnější. Realizovaná aplikace touto formou

řeší elegantní propojení hardwaru "připojeného do sítě s internetem" a webovým prohlížečem uživatele, který nutně nemusí být takto zapojený ani do stejné lokální sítě, může jím být třeba i prohlížeč v mobilu připojený přes mobilního operátora. Konfigurace spočívá ve spárování data streamu a uživatele/prohlížeče s ním.

Zde je ukázka vyrobeného vstupního/streamovacího celku.

Obrázek 5.2: Embedded setup - váha a bluetooth čtečka, data streamována přes ethernet

Podrobný popis celého zařízení a řešených problémů by si vyžádal možná i další diplomovou práci, z ne hned zřejmých překážek lze jmenovat:
- embedded linux pro platformy jiné než x86 není úplně snadné nainstalovat, dostupnost ovladačů je výrazně nižší a některé jsou napsány např. pro ARM extra z čehož dost často plyne výrazně jiné chování a náchylnost k chybám - jiná vstupní zařízení (např. váha) mohou streamovat data proprietárním protokolem

Proprietární protokol byl rozkódován pomocí profesionálního disassembleru aplikovaného na software dodávaný výrobcem.

Upravená knihovna Doctrine ORM z této práce byla použita na realizaci serverového API a výrazně zrychlila jeho odezvy, zvlášť u požadavků, které najednou žádají o uložení desítek až stovek nabufferovaných změn.

Kapitola 6

Zhodnocení výsledků a diskuse

6.1 Porovnání stavu před a po

Implementace je plně transparentní a scénáře, kde dotazy nelze slučovat, jsou ošetřeny. Toto tvrzení dokládá možnost spustit kompletní testy původní knihovny na upravené implementaci. Opakovatelný benchmark potvrdil velmi výrazné zrychlení při větších počtech záznamů k uložení.

6.2 Kompatibilita

Doctrine ORM je velmi robustní knihovna s velmi vysokým pokrytím testy. Výsledek OK níže po spuštění originálních unit testů, za použití upravené knihovny, dokládá 100 % náhradu originální implementace.

```
1    PHPUnit 7.5.20 by Sebastian Bergmann and contributors.
2    Runtime:       PHP 7.4.0
3    Configuration: mysql.xml
4    ..........................................   61 / 3335 (  1%)
5    ..
6    .....................                      3335 / 3335 (100%)
7    Time: 22.71 seconds, Memory: 222.50 MB
8    There were 3 incomplete tests:
9    ..
10   --
11   There were 37 skipped tests:
12   ..
13
14   OK, but incomplete, skipped, or risky tests!
15   Tests: 3335, Assertions: 12348, Skipped: 37, Incomplete: 3.
```

Nekompletní/přeskočené testy nesouvisí s upravenou implementací, testy oproti originální knihovně nebyly modifikovány.

6.3 Benchmark - testovací scénář

Zrychlení v unit testech pozorovat nelze. To je očekávaný výsledek, jelikož unit testy testují funkcionalitu s minimálním počtem záznamů.

Pro vyhodnocení v praxi, a možnost opakovatelných výsledků, byl napsat krátký program sestávající se z definice jedné entity nazvané `Record`:

```
namespace DpBenchmark;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity
**/
class Record {
    /**
    * @var int
    * @ORM\Id
    * @ORM\Column(name="id", type="bigint", options={"unsigned"=true})
    * @ORM\GeneratedValue(strategy="CUSTOM")
    * @ORM\CustomIdGenerator(class="DpBenchmark\GlobalIdGenerator")
    **/
    public $id;

    /** @var string @ORM\Column(type="string", length=500) */
    public $name;

    /** @var \DateTime @ORM\Column(type="datetime_immutable") */
    public $createdAt;
}
```

a program:

```
use Doctrine\ORM\EntityManager;
use DpBenchmark\Record;

require_once __DIR__ . '/bootstrap.php';

$h = mt_rand();
for ($i = 0; $i < 1000; $i++) {
    $user = new Record();
    $user->name = 'Example ' . $h;
    $user->createdAt = new \DateTimeImmutable();

    $em->persist($user);
}

$t = microtime(true);
$em->flush();
echo "\n" . 'time taken: ' . round((microtime(true) - $t) * 1000 / $i, 3) . ' ms / record' . "\n";
```

který připraví určitý počet nových záznamů, které následně hromadně ukládá do databáze.

6.4 Benchmark - ukázka generovaných SQL příkazů

SQL příkazy generované testovacím programem s originální knihovnou:

```
1     START TRANSACTION
2     INSERT INTO Record (id, name, createdAt) VALUES (?, ?, ?)
3     INSERT INTO Record (id, name, createdAt) VALUES (?, ?, ?)
4     INSERT INTO Record (id, name, createdAt) VALUES (?, ?, ?)
5     INSERT INTO Record (id, name, createdAt) VALUES (?, ?, ?)
6     INSERT INTO Record (id, name, createdAt) VALUES (?, ?, ?)
7     COMMIT
```

po použití autorem upravené knihovny:

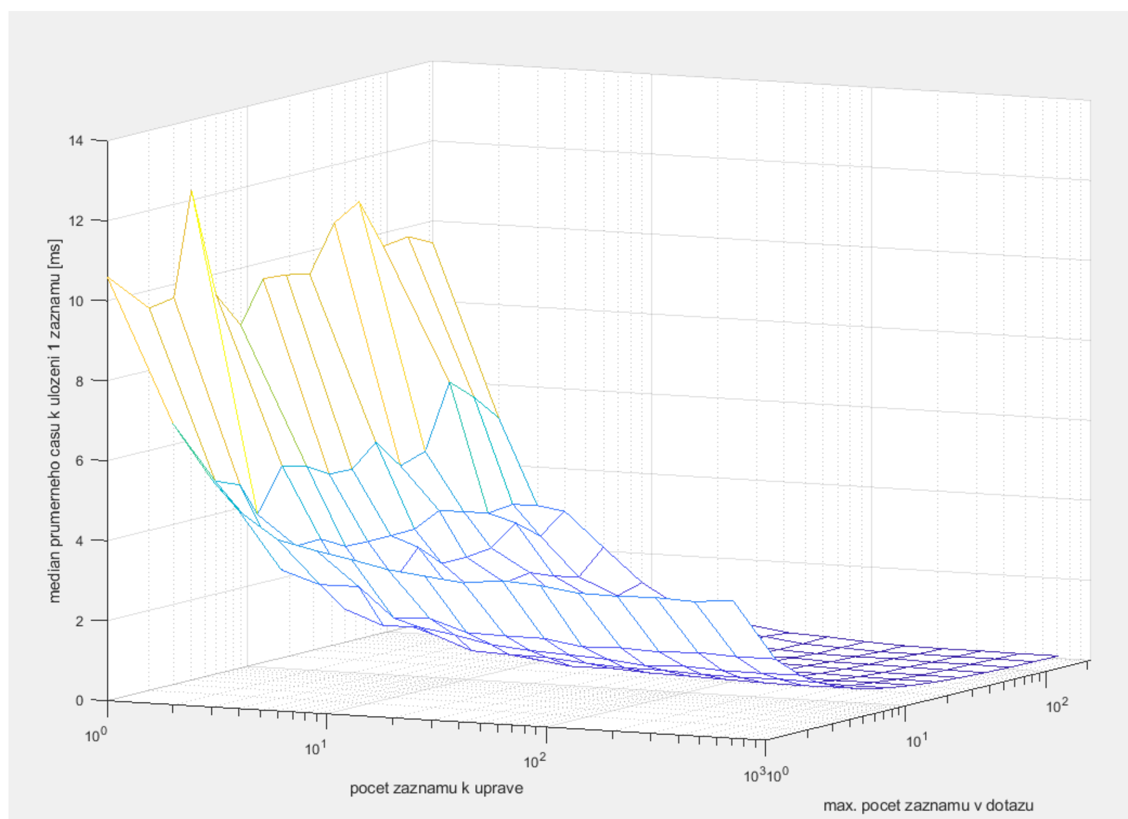
```
1     START TRANSACTION
2     INSERT INTO Record (id, name, createdAt) VALUES
3         (?, ?, ?), (?, ?, ?), (?, ?, ?)
4     INSERT INTO Record (id, name, createdAt) VALUES
5         (?, ?, ?), (?, ?, ?)
6     COMMIT
```

Testovací program pro výše uvedené výstupy byl pro kratší výstup upraven, konkrétně byl změněn celkový počet záznamů pro uložení na 5 a počet záznamů pro sloučení na 3.

Každý SQL dotaz je synchronní a vyžaduje tedy nejméně jeden RTT (čas pro doručení dotazu do databáze a zpět), toto tvrzení a výhodnost navržené optimalizace dokládá výkonnostní vyhodnocení.

6.5 Benchmark - vyhodnocení zrychlení

Pro vyčerpávající představu o výkonnostních vlastnostech upravené knihovny byl testovací program vyhodnocen ve dvou stupních volnosti:
- počtu záznamů k úpravě
- a max. počtu záznamů, které se mohou sloučit v jednom SQL dotazu:



Obrázek 6.1: Naměřené výsledky ve 2 stupních volnosti

Graf reprezentuje medián průměrného času k uložení 1 záznamu vypočtený z 20 měření pro každou/stejnou konfiguraci. Z grafu je jasně patrné zrychlení dané snížením počtu SQL dotazů a tedy snížením násobnosti RTT.

Dále lze vypozorovat, že větší max. počet záznamů v jednom dotazu nemá negativní vliv, avšak výrazně vyšší hodnoty mají již jen omezený přínos. V praxi je vhodné použít hodnotu okolo 20 - 100, větší převážně pro malé záznamy.

Největší testovaný počet odpovídal 711 záznamům k úpravě v rámci jedné transakce. Toto je pořád poměrně malý počet, v praxi není nezvyklé importovat najednou až milióny záznamů. Pokud se použije knihovna bez úpravy, trvá změna jednoho záznamu v průměru 3.4 ms, s upravenou knihovnou 0.061 ms. Jednoduchým výpočtem lze dospět k výsledku - zrychlení je více než 50 násobné!

Byla testována konfigurace virtualizovaný klient i databáze na dvou serverech v Praze, end-to-end latence cca 1.5 ms. Volba na tuto konfiguraci padla dle reálného nasazení. Pokud byla databáze umístěna na stejném stroji, bylo naměřeno zrychlejší zhruba 6 násobné.

Pro úplné vyhodnocení výkonnosti bylo provedeno také měření s databází SQLite ve verzi 3.16. SQLite databáze pro PHP je kompilované rozšíření, které představuje jednoduchou DB komunikující prostřednictvím SQL, avšak veškerá komunikace se děje přímo v PHP procesu, tj. bez jakéhokoliv přepínání procesů či síťového/RTT

overheadu.

Byly otestovány dvě použití - in memory (databáze pouze v paměti) a souborová databáze. Při použití SQLite bylo pozorováno zrychlení 0.3x, respektive 0.5x. Kompletní data jsou k dispozici na přiloženém CD.

Na základě těchto dat lze konstantovat násobné zrychlení při běžném použití.

6.6 Jak moc to bylo náročné?

Samotná implementace není příliš rozsáhlá, ale nejprve bylo potřeba v aplikací adresovat jiné podružné problémy, problematiku pochopit a přemýšlet dlouhodobě - program je potřeba psát, aby jej bylo možné udržovat.

Tento výsledek autor považuje za úspěch a je jasným důkazem, že problém byl adresován správnými rozhodnutími při výběru technologie a její modifikaci.

6.7 Výrazné přínosy pro celou PHP komunitu

Vysokoúrovňové výkonnostní problémy lze typicky snadno identifikovat a velmi často odladit izolováním jednotlivých částí až k samotnému úzkému hrdlu. Tato práce se však vyznačuje naprostým opakem - výkonnostním problémem na nejnižší úrovni.

Při práci bylo zjištěno, že nejpopulárnější profiler xdebug vnáší do měření až 20x průměrný overhead (spotřebovaný procesorový čas samotným profilerem) a ještě vyšší per volání funkce. Nízko úrovňové problémy jako tento tak byly v měření několikanásobně zkreslené, že bylo těžké až nemožné interpretovat reálné produkční výsledky. Autor této práce identifikoval v tomto nástroji hned několik problémů, které vyřešil a nabídl prostřednictvím pull-requestů komunitě. Všechny nabídnuté řešení byly akceptovány a integrovány. Výstupem je zrychlení o zhruba 75 %, a tedy výrazné zlepšení vypovídající hodnoty tohoto nástroje.

6.8 Možné pokračování dalšího vývoje

Mezi databází a programem vždy bude RTT latence. Po minimalizaci počtu databázových dotazů již další zlepšení na úrovni ORM knihovny možné není.

Slučování některých změn (např. definovaných jinými avšak velmi podobnými třídami) je v aktuální implementaci zakázáno. Odstranění tohoto omezení by mohlo ORM knihovnu zrychlit, ale pouze však pro malé sady změn, které jsou již typicky dostatečně rychlé. Další optimalizace na míru specifické aplikaci autor nevylučuje, základem je však správné použití, aby bylo možné již provedené optimalizace použít.

Práce se zabývá výhradně zlepšením aktuální verze Doctrine ORM 2.7. Jelikož je údržba forku poměrně časově náročná, je vhodné s komunitou otevřít diskusi na

provedení změn v samotné knihovně. Jedním ze silných argumentů může být dosažitelné zrychlení a dále již existující implementace v knihovně pro Javu, ze které PHP knihovna silně vychází.

Závěr

Výstupem práce je plné splnění vytyčených cílů a reálné zrychlení o více než 50x. Navrhnutou implementací lze pomocí composer balíčku snadno nahradit standardní Doctrine ORM i v dalších projektech.

Dalším podstatným výstupem, zhodnocením práce a přínos pro PHP komunitu je vyřešení souvisejících subproblémů. Během této práce autor přímo ovlivnil vývoj jazyka PHP pomocí spolupráce na několika RFC[1], nareportování desítek chyb v rámci PHP, MySQL databáze a více než tisíc individuálních přispění do open source projektů formou issue reportů či přímo pull requestů[2].

Práce plně vyřešila prodlevy související s RTT prodlevami. Dalšího zrychlení je možné dále dosáhnout např. nastíněnou striktnější normalizací dat na straně aplikace, přesnější identifikací změn a tedy transparentnímu snížení počtu záznamů k aktualizaci.

Autor se plánuje nastíněnými problémy zabývat i po dokončení magisterského studia.

[1] https://wiki.php.net/rfc/make_ctor_ret_void
[2] https://github.com/mvorisek

Literatura

[1] https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/index.html dokumentace Doctrine ORM

[2] https://github.com/doctrine/orm zdrojový kód Doctrine ORM

[3] https://github.com/doctrine/dbal zdrojový kód Doctrine DBAL

[4] https://github.com/php/php-src zdrojový kód php interpreteru

[5] https://github.com/xdebug/xdebug zdrojový kód (nejen) profileru pro PHP

[6] Persistence in PHP with the Doctrine ORM, Kévin Dunglas, ISBN: 1782164103

[7] Hibernate Tips: More than 70 solutions to common Hibernate problems, Thorben Janssen, ISBN: 1544869177

[8] Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert C. Martin, ISBN: 0134494164

Přílohy

Digitální verze této práce a zdrojové kódy přiloženy na CD.