

Mendelova univerzita v Brně  
Provozně ekonomická fakulta

---

# **Webová architektura v prostředí vysoké zátěže**

**Diplomová práce**

Vedoucí práce:  
Ing. Michael Štencl, Ph.D.

Bc. Jakub Škrha

Brno 2012

Text poděkování

Text prohlášení

místo a datum prohlášení

.....

**Abstract**

Citace práce v anglickém jazyce

abstrakt práce v anglickém jazyce

**Abstrakt**

Citace práce v českém jazyce

abstrakt práce v českém jazyce

## Contents

|           |   |           |
|-----------|---|-----------|
| <b>1</b>  | <b>Úvod</b>   | <b>7</b>  |
| <b>2</b>  | <b>Nežádoucí vlivy a důsledky vysoké zátěže</b>       | <b>8</b>  |
| <b>3</b>  | <b>Zneužití vysoké zátěže</b>                         | <b>9</b>  |
| <b>4</b>  | <b>Tří a vícevrstvá architektura webových systémů</b> | <b>11</b> |
| <b>5</b>  | <b>Aplikační vrstva</b>                               | <b>12</b> |
| 5.1       | Webová aplikační architektura MVC . . . . .           | 12        |
| 5.2       | Optimalizace aplikační vrstvy . . . . .               | 13        |
| 5.3       | Druhy aplikačních vrstev . . . . .                    | 13        |
| 5.4       | Ajax a webové služby . . . . .                        | 13        |
| <b>6</b>  | <b>Dabázová vrstva</b>                                | <b>15</b> |
| 6.1       | Optimalizace SQL dotazů . . . . .                     | 15        |
| 6.2       | Indexace . . . . .                                    | 16        |
| 6.3       | Partitioning . . . . .                                | 16        |
| 6.4       | Replikace . . . . .                                   | 17        |
| 6.5       | Druhy relačních databází . . . . .                    | 18        |
| <b>7</b>  | <b>Webové cache</b>                                   | <b>19</b> |
| 7.1       | Typy obsahu . . . . .                                 | 20        |
| 7.2       | HTTP hlavičky pro ovládání cache . . . . .            | 20        |
| 7.3       | Druhy cache . . . . .                                 | 21        |
| 7.3.1     | Proxy cache a cache prohlížeče . . . . .              | 21        |
| 7.3.2     | Reverzní proxy cache . . . . .                        | 22        |
| 7.3.3     | Aplikační distribuovaná cache . . . . .               | 22        |
| <b>8</b>  | <b>Další vrstvy aplikace</b>                          | <b>24</b> |
| 8.1       | CDN . . . . .   | 24        |
| 8.2       | NoSQL Databáze . . . . .                              | 24        |
| 8.3       | Vyhledávání . . . . .                                 | 24        |
| <b>9</b>  | <b>Virtualizace</b>                                   | <b>25</b> |
| <b>10</b> | <b>Load balancing</b>                                 | <b>26</b> |
| <b>11</b> | <b>Cloud Computing</b>                                | <b>27</b> |
| <b>12</b> | <b>Praktická část s experimenty a výsledky</b>        | <b>28</b> |
| 12.1      | Vrstvy webové architektury . . . . .                  | 28        |
| 12.2      | Testovací a profilovací nástroje . . . . .            | 29        |

---

|           |  |           |
|-----------|--|-----------|
| 12.2.1    | XHPProf . . . . .  | 29        |
| 12.2.2    | Siege . . . . .  | 30        |
| 12.2.3    | PostgreSQL Explain . . . . .                             | 30        |
| 12.3      | Aplikační vrstva PHP . . . . .                           | 31        |
| 12.3.1    | Optimalizace pomocí APC . . . . .                        | 31        |
| 12.3.2    | Dosažené výsledky . . . . .                              | 32        |
| 12.4      | Databázová vrstva . . . . .                              | 34        |
| 12.4.1    | Optimalizace databáze . . . . .                          | 34        |
| 12.4.2    | Dosažené výsledky . . . . .                              | 35        |
| 12.5      | Aplikační cache . . . . .                                | 37        |
| 12.5.1    | Optimalizace aplikace pomocí Memcached . . . . .         | 37        |
| 12.5.2    | Diagram tříd pro aplikaci s podporou Memcached . . . . . | 37        |
| 12.5.3    | Dosažené výsledky . . . . .                              | 37        |
| 12.6      | Reverzní proxy cache . . . . .                           | 37        |
| 12.6.1    | Nasazení a konfigurace NGINX s Memcached . . . . .       | 37        |
| 12.6.2    | Význam Ajax a webových služeb pro NGINX . . . . .        | 37        |
| 12.6.3    | Dosažené výsledky . . . . .                              | 38        |
| <b>13</b> | <b>Diskuze</b>   | <b>39</b> |
| <b>14</b> | <b>Závěr</b>   | <b>40</b> |
| <b>15</b> | <b>Literatura</b>  | <b>41</b> |

## 1 Úvod

Něco na téma jak vzniká vysoká zátěž na internetu, jaký je vliv internetu, rostoucí zájem o internet a stoupající zátěž. Uvidíme, napíšu to jako poslední.

## 2 Nežádoucí vlivy a důsledky vysoké zátěže

Úspěch internetových a webových projektů je přímo úměrný výši návštěvnosti, používání a registracím a samozřejmě výdělku z aplikace. Obecně se dá předpokládat, že čím je větší návštěvnost projektu, tím jsou větší i zisky. A to už díky reklamní činnosti či placených služeb. Ovšem zde se dá velice jasně konstatovat, že tyto výdělky nejsou až tak lehce získané. Nejenom, že si musí aplikace získat své uživatele, ale musí řešit problémy s obrovským počtem uživatelů, čili problémy s vysokou zátěží.

Vysoká zátěž může mít ve své podstatě několik nežádoucích vlivů, které mohou mít až katastrofický scénář. Může docházet k takovému zatížení aplikace, že odpovědi na jednotlivé požadavky mohou trvat velice dlouhou dobu. Tím pádem si uživatel může rozmyslet, zda-li přistě navštíví tuto webovou aplikaci, či zkusí některou z jiných možných konkurenčních alternativ. Takto velice nepříznivý scénář může být ještě horším. A to tak, že díky velkému zatížení dojde dokonce k výpadku celé aplikace, a tím pádem už se uživateli nedostane vůbec žádné odpovědi na jeho požadavek. Při takovém scénáři existuje vysoká pravděpodobnost ztráty a poklesu uživatelů, což může znamenat velký pokles zisků pro firmu či společnost.

Po technické stránce se při velké zátěži vytvoří pro každý požadavek celý samostatný proces či vlákno procesu, které si klade nároky na procesor CPU a operační paměť RAM serveru. Vznikají tak i procesy, které musí čekat na přidělení takových prostředků a tím pádem zatížení, které roste a čeká na vykonání může server přetížít až už nebude provozuschopný. Takovéto zatížení je možné pozorovat při výpisu právě běžících procesů a jejich vytížení na RAM či CPU (například příkazem `top`). Dále je možno pozorovat tzv. "Load Average" (například příkazem `uptime`), které představuje počet procesů čekajících na přidělení prostředků během jedné, pěti či patnácti minut. (Kyle Rankin, 2010) Takto je možné sledovat jaké jsou nežádoucí vlivy vysoké zátěže na technické úrovni.

Je nutné podotknout, že existuje i jeden skrytý a ne tak viditelný důsledek vysoké zátěže. Tím je fakt, že jakmile se začne zvedat návštěvnost uživatelů, a tím pádem i zátěž aplikace, projektové vedení si klade požadavek, aby tento nárůst uživatelů již zůstal, a naopak se dokonce i zvětšoval. A to vše z toho důvodu, že velký počet uživatelů, znamená velkou zátěž pro aplikaci, ovšem velký finanční přínos pro firmu.



### 3 Zneužití vysoké zátěže

Z nežádoucích vlivů a důsledků uvedených v předchozí kapitole vyplývá, že rostoucí zátěž může způsobit katastrofické scénáře, čili může negativně působit na celou aplikaci. Tento poznatek představuje obrovské riziko při jeho zneužití. Zpravidla může být způsobeno úmyslným či neúmyslným chováním nějaké organizace či jedince. Takovéto zneužití má svoji oporu i v zákonech, kde hrozí až odmětí svobody několika let.

U webových projektů je možné z těch neúmyslných zmínit například nějaké klientské chyby programů, či větší míru indexace robotů jednotlivých vyhledávačů. Roboti vyhledávačů pravidelně prochází webovou aplikaci a indexují její jednotlivé části, a tyto výsledky zohledňují následně ve svých výsledcích ve vyhledávání. Tito roboti mohou představovat nežádoucí zátěž.

Tím druhým a daleko nebezpečnějším úmyslným způsobem lze mnohdy způsobit daleko větší škody. Toto úmyslné chování už je klasifikováno jako útok na webovou aplikaci za který hrozí postih podle zákona. Tyto útoky jsou nazývány DoS, neboli "Denial of Service". Jejich cílem je ochromit infrastrukturu celé webové architektury přehlcením požadavků na které aplikace bude vytvářet odpovědi. Tyto útoky využívají chyb, nedostatků či nedokonalostí protokolů ICMP, TCP, UDP a jiných protokolů či samotných webových aplikací. Například útok s názvem "Tcp Syn Flood", kdy útočník vytíží aplikaci SYN pakety pro navázání spojení využívá nedokonalosti TCP protokolu. Dalšími útoky využívající nedokonalosti mohou být "ICMP Flood", "Ping of Death", "Smurf Attack", "IP Spoofing", "Fraggle Attack", "Teardrop", "Application level" aj. Ovšem proti většině těchto útoků už dnes existuje ochrana ve formě servisních instalací jednotlivých systémů či aktualizací programů síťových zařízení a nebo lehkou konfigurací. (Faisal Khan, 2009)

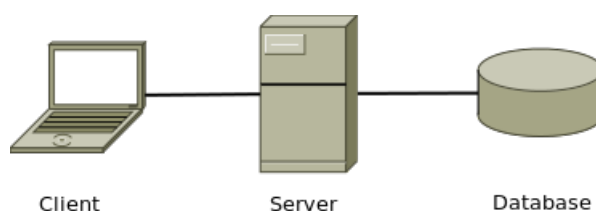
Ovšem co v dnešní době představuje daleko větší nebezpečí, jsou útoky typu DDoS, neboli "Distributed Denial of Service". V tomto případě jako princip obdobný jako u DoS, ovšem s tím, že je úkol distribuovaný. Tedy je spuštěn z několika stanic, několika uživatelů a pomocí různých nástrojů. Tento způsob je tedy daleko více organizovaný a daleko více nebezpečný a účinnější. (Faisal Khan, 2009)

Právě v dnešních dnech se stává symbolem boje za svobodu internetu skupina s názvem Anonymous, která využívá útoků DDoS. Při svých útocích využívají například útoky typu "Slowloris", kdy útočník využívá protokolu HTTP na aplikační úrovni a chce celou odpověď na svůj požadavek. Při navázaném spojení ovšem odesílá HTTP hlavičky co nejpomaleji, aby tak co nejvíce prodloužili dobu spojení a získal prostor pro vytvoření dalšího spojení, čili další zátěže. Tato skupina napadá webové aplikace veřejnosti neoblíbených politických stran, vládních organizací, protipirátských asociací a jiných subjektů. Získávají si tím obrovskou podporu ve společnosti i médiích, která s jejich kroky souhlasí. Dokonce pro své útoky využívají i příznivců z řad veřejnosti, kteří nemusí odborníky informačních technologií. Stačí jim si pouze stáhnout upravený program, v určený čas ho spustit a připojit. Útoky probíhají hlášeně či neohlášeně, organizovaně a distribuovaně.

Otázkou zůstává, kdy jejich konání přeroste z útoků pro dobro společnosti, a stanou se útoky pro vydírání, posílení moci, či za účelem finančního obohacení. V ten moment i společnost, která je v tyto dny podporuje, může pocítit, jak jsou pro ně nebezpeční. I v historii Země nalezneme spoustu skupin, které byly lidmi podporovány a nakonec se z nich stal symbol krutosti, tyranie, úzkosti a neštěstí. Proto je důležité jejich útoky nepodceňovat a umět se bránit. Má práce se nezabývá konkrétním řešením nějakého z typů útoků, ale zabývá se obecně vysokou zátěží, a jak ustát narůst obrovské zátěže a tedy i nějaký útok. (Pavel CČepský, 2012)

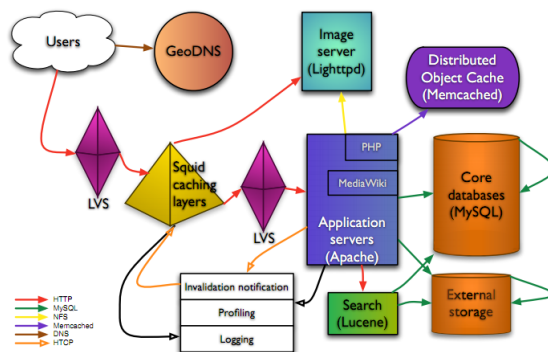
## 4 Tří a vícevrstvá architektura webových systémů

Webová architektura je ve svém základu třívrstvá. První vrstvu tvoří klient, neboli uživatel se svým Hardware a Software, a svými aplikačními požadavky. Druhou vrstvu tvoří aplikační server, který zpracovává požadavky aplikace, tedy požadavky klienta, zpracuje tento požadavek, vytvoří odpověď a zašle zpět klientovi. Ovšem k tomu, aby mohl tuto odpověď vytvořit, potřebuje i data aplikace, která jsou uloženy v perzistentní databázi, která tvoří třetí a poslední vrstvu třívrstvé architektury. Každá z vrstev, tedy prezentační, aplikační i datová má své místo a svou správu v aplikaci. (Jaroslav Zendulka, 2005)



Obr. 1: Tří vrstvá architektura webových aplikací

V architektuře webových aplikací s vysokou zátěží už je potřeba jiného přístupu. V tomto případě se dá říci, že je třívrstvá architektura nedostačující. Je potřeba počítat se síťovými prvky pro load balancing, s více aplikačními stroji, s databázovými replikacemi, s DNS řešením pro geografické rozdělení zátěže, s CDN pro rozdělení zátěže přidělování obsahu, s vrstvami pro cache aplikace a s dalšími vrstvami pro backendové či frontendové aplikace a služby. V tomto případě neexistuje žádné jasně dané a pevné řešení, každá aplikace si s sebou nese své individuální a charakteristické řešení a strategii, i když některé osvědčené postupy se opakují. Tyto strategie už nesou název vícevrstvá architektura.



Obr. 2: Webová architektura společnosti Wikimedia provozující Wikipedia.org

Nutno podoknout, že webové architektury využívají nejčastěji ke své komunikaci mezi klientem a architekturou protokol HTTP, který využívá portu číslo 80 a protokolu TCP pro komunikaci. Proto je celá má odborná studie založena na práci s tímto protokolem.

## 5 Aplikační vrstva

Aplikační vrstva představuje jádro webové architektury. Jejím účelem je přijmout a zpracovat klientův požadavek, vytvořit odpověď a tuto odpověď zaslat nazpět klientovi. Na aplikační vrstvu jsou tak kladeny úkoly celé rezie procesu tvorby odpovědi, a tím pádem má velkou zodpovědnost a mnohdy i největší zátěž.

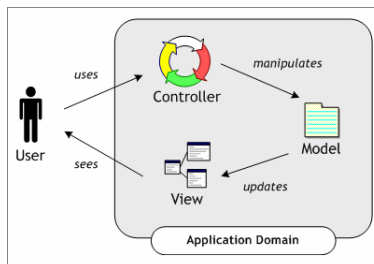
### 5.1 Webová aplikační architektura MVC

V dnešní moderní aplikační vrstvě se používá aplikační architektura návrhového vzoru MVC pro přehlednější a rychlejší způsob tvorby aplikace. Tato zkratka vychází z tří slov Model, View a Controller, které představují tři základní vrstvy aplikační architektury. Často bývá označován i jako MVC framework, který rozděluje aplikaci do tří modulů. (Rudolf Pecinovsky, 2007)

Controller je prvotní inicializační vrstva každého požadavku. Zpracovává příchozí data, parametry a atributy dané akce od uživatele, provádí jejich kontrolu a formátování. Stará se i o zabezpečení dané konkrétní akce vrstvy Controller. Často spolupracuje s vrstvou Model, které předává požadavky na data aplikace, a tyto data dále zpracovává pro předání do vrstvy View. (Rudolf Pecinovsky, 2007)

Model má za úkol přistupovat k datovým úložištím, a to až už k perzistentní databázi nebo souborovému systému, cache či jiným typům úložišť. Zapouzdřuje tak datovou logiku frameworku. Často se jedná o soubor dalších návrhových vzorů, kde se může vyskytnout přeprava (Crate) či jejich kolekce pro přenášení dat, zástupce (Proxy) pro přístup k implementacím nad přepravkami, příkaz (Command) pro vykonání nějaké akce či příkazu, strategie (Strategy) pro určení nějaké konkrétního algoritmu ze skupiny algoritmů nad určitou úlohou, a mnohé další z návrhových vzorů. Modelová vrstva bývá označována za nejsložitější vrstvu, a právě proto je potřeba dodržovat techniky OOP včetně návrhových vzorů pro další možnou rozšiřitelnost a pro přehlednost. (Rudolf Pecinovsky, 2007)

View klade důraz na presentační úroveň, tedy na grafickou a jinou interakci s uživatelem. Zpracovává tak výsledek práce vrstvy Controller nad vrstvou Model a zobrazuje výsledek určitých operací. Často využívá nějakých šablonovacích přístupů. (Rudolf Pecinovsky, 2007)



Obr. 3: Návrhový vzor MVC a jeho životní cyklus

## 5.2 Optimalizace aplikační vrstvy

Optimalizace na úrovni aplikační vrstvy může mít několik způsobů a přístupů. Tato činnost se týká převážně programátorů a softwarových inženýrů, kteří mají za úkol vývoj a údržbu aplikační vrstvy. K tomu, aby se dali identifikovat problematické části pro optimalizaci slouží tzv. profilery. Ty mají obecně za úkol vyprofilovat jednotlivé funkce, metody, procedury, dotazy a příkazy, které se na dané vrstvě, již je profiler určen, vyskytují, a určit jejich dobu trvání, počet volání, čas spuštění, závislosti a další parametry. Profilování, neboli určení kandidátů pro optimalizaci, je prvním a nejdůležitějším krokem pro optimalizaci aplikační vrstvy webové architektury. Další kroky se týkají především těchto oblastí:

- Výběr neoptimálnějšího algoritmu pro danou úlohu
- Výběr nejrychlejšího MVC frameworku
- Vytváření cache souborů aplikace
- Způsob překladu a vykonání zdrojových souborů
- Přidání další vrstvy architektury - aplikační cache

## 5.3 Druhy aplikačních vrstev

Existuje celá škála různých programovacích jazyků a webových serverů pro implementaci aplikace. Každý z nich má své výhody a nevýhody, specifická řešení a přístupy. Uvádím zde krátký seznam těch v praxi nejběžněji se vyskytujících:

- Webový server Apache2 s programovacím jazykem PHP
- Java Servlets, Java Spring Source
- C# s technologií .NET
- Ruby on Rails
- Python a Django
- a mnohé další

## 5.4 Ajax a webové služby

Ajax, neboli Asynchronous JavaScript and XML, se dnes stává nedílnou součástí při vývoji webových aplikací. Aplikace tak dostávají interaktivnější charakter bez nutnosti znovuzaslání celého požadavku webové aplikaci. Celý tento přístup probíhá nejvíce na straně klienta. Je použito javascriptu pro programovou implementaci, který má přístup ke stromu objektů dokumentu zvaného DOM, neboli Document Object Model. Do aplikační vrstvy jsem se rozhodl přidat AJAX z toho důvodu, že používá objekt XMLHttpRequest pro komunikaci s aplikačním serverem. Tyto

---

aplikační požadavky jsou nazývány webovými službami pro Ajax. Tyto požadavky jsou vykonávány na aplikační vrstvě a představují potenciální zátěž, která musí být i v některých případech optimalizována. (Brett McLaughlin, 2005)

## 6 Dabázová vrstva

Úkolem databázové vrstvy ve webové architektuře je zajišťovat datové služby a uchovávat tak aplikační data perzistentní. V oblasti webových architektur se nejčastěji vyskytují relační databázové systémy, a proto i má práce je soustředěna na tento typ databázových systémů. Databázový systém obecně tvoří databáze, jakožto skupina strukturovaných homogenních souborů, a SŘBD, neboli Systém řízení báze dat, jakožto integrovaný softwarový prostředek řídící bázi dat.

### 6.1 Optimalizace SQL dotazů

Optimalizace dotazů SQL je nedílnou součástí procesu práce s databázovým systémem v prostředí vysoké zátěže. Je totiž důležité nejenom si umět získat potřebná data, ale je potřeba zvážit i za jakou cenu tyto data prostřednictvím databázového systému získáváme. Hovoříme-li o webových architekturách s vysokou zátěží, je tento proces optimalizace velice důležitý. Každá operace, každý dotaz, každá akce potřebuje ke své realizaci určité hardwarové a systémové prostředky, a v prostředí vysoké zátěže je důležité ušetřit co nejvíce těchto prostředků.

K tomu, abychom mohli vůbec přistoupit k optimalizaci SQL dotazů, je potřeba určit a identifikovat, které tyto dotazy jsou opravdu náročné na prostředky a čas, neboli mají vysokou cenu. K tomu slouží tzv. profilery (viz. kapitola 5.2). Profilery mohou být určeny pro aplikační vrstvu, kde profilují nejenom zdrojové kódy aplikace, ale samozřejmě i databázové dotazy, které jsou z této aplikační úrovně spuštěny. Tímto způsobem je možné získat přehled všech operací, které probíhají na aplikační i databázové vrstvě, poněvadž tyto vrstvy spolu neúzce souvisí a spolupracují. Další možností je použít profiler určený přímo k databázové vrstvě. Takový profiler pak profiluje pouze databázovou vrstvu, jednotlivé databázové dotazy, jejich cenu, dobu trvání, a jiné další statistiky.

Každý SQL dotaz má nějaký svůj exekuční plán. Databázový systém po obdržení SQL dotazu vybírá z několika možných exekučních plánů ten nejoptimálnější, který je po té v databázi proveden. Při výběru exekučního plánu je brán v potaz výběr indexu a způsob skenu tabulek, vybraná spojení, aj. Exekuční plán je možné zobrazovat v mnoha databázových systémech pomocí EXPLAIN a identifikovat tak místa exekučního plánu, která mohou být kandidátem pro optimalizaci. (Bohdan Blaha, 2007)

Pro optimalizaci SQL dotazů je možné určit několik základních oblastí, na které je možné se zaměřit při konkrétní optimalizace určitého SQL dotazu:

- Normalizovaný databázový návrh
- Vnořené SQL dotazy
- Indexace, výběr indexu a způsob prohledávání
- Výběr druhu a pořadí spojení

- Způsob používání podmínek, klauzulí a operátorů

## 6.2 Indexace

Indexace je důležitá a nejefektivnější optimalizace dotazů SQL. Při průchodu dat tabulkou má databáze na výběr několik možností prohledání. První možností je prohledat všechny řádky tabulky podle sql podmínek. To je nazýváno obecně Full Table Scan, nebo také Sequence Scan, neboli sekvenční prohledávání. Další možností je použití některého z indexů pro přístup k hodnotám namapovaných na jejich ROWID, které ukazuje na fyzické uložení. Toto prohledání bývá nazýváno jako Index Range Scan, nebo jen obecně Index Scan, neboli indexační prohledávání. Samozřejmě prohledávání tabulek pomocí indexace je výrazně rychlejší a tím pádem důležité pro optimalizaci SQL. (Bohdan Blaha, 2007)

Indexy jsou fyzicky i logicky uloženy v asociativních tabulkách, a díky tomu tak i odděleny od datových tabulek. Čili při smazání indexů se nesmíží ani nijak neovlivní datové tabulky. Pouze se může zpomolit přístup k datům, který byl rychlejší pomocí těchto indexů. Tabulky s indexy jsou samozřejmě uloženy na disku, poněvadž jejich velikost je obrovská a nevešly by se do operační paměti RAM. Operační paměť a přístup k ní je daleko rychlejší než přístup k datům uložených na disku, a proto je potřeba volit nějaký vhodný algoritmus prohledání a přístupu k indexům, a od toho se odvíjí i název a druh používaných indexů. V každém databázovém systému samozřejmě naleznete některé typické a některé atypické druhy indexů. (Bohdan Blaha, 2007) Zde je krátký výběr možných indexů:

- B-tree - pro přístup pomocí Root-Node-List
- Bitmap - pro výčtové sloupce
- R-tree - typ indexu optimalizovaný pro geometrická data.
- GiST - zobecněný vyhledávací strom
- a další

## 6.3 Partitioning

Partitioning, který je občas do češtiny překládán jako segmentace, občas jako škálování, slouží v relačních databázových systémech k rozdělování tabulek a indexů do menších částí a komponent. Díky tomu je pak činnost databáze rychlejší a snadnější. Při této segmentaci tak může dojít k rozdělení tabulek i na více pevných disků či serverů. Tyto segmenty jsou na sobě nezávislé, ale přitom je k nim možné přistupovat přes tabulku, pro kterou byla segmentace vytvořena. Databázová tabulka a její vlastnosti, jako například referenční integrita nebo žádná redundance, jsou stále zachovány a fungují přes všechny její segmenty. Dokonce i když dojde k selhání či výpadku jednoho ze segmentů, ostatní jsou stále přístupné a je možné s



nimi pracovat. Partitioning je možné provádět na několika úrovních a podle různých klíčů. U segmentovaných tabulek je tak důležité si rozmyslet jakou strategii si zvolit.

Při vertikální segmentaci dojde k segmentaci podle definovaných sloupců databázové tabulky. Klíčem při tomto rozdělení je určení sloupečků, které se nepoužívají ve where klauzuli, nebo jsou prázdné či zřídka používané.

Častěji používaným přístupem je horizontální segmentace tabulek, čili segmentace podle řádků. Zde se segmentují řádky, podle určité hodnoty databázového sloupce. To do jakého segmentu bude řádek tabulky vložen rozhoduje nějaký interval, či hodnota výčtu a nebo nějaká funkce.

Další možností je aplikační úroveň segmentace, která se ne často objevuje v souvislosti s Partitioning. Nejedná se totiž o segmentaci určité databázové tabulky, ale o segmentaci databáze. Část tabulek je umístěna na jednom serveru, část na dalším serveru, a tak dále.

Partitioning je důležitým nástrojem při optimalizaci databázové vrstvy v architektuře webových systémů v prostředí vysoké zátěže. Dá se totiž předpokládat, že se zvětšující se zátěží roste i počet záznamů tabulek, a tak se doba přístupu zvětšuje a prostředky zatěžují ještě víc. Tyto problémy dokáže vyřešit partitioning. (Eli White, 2009)

## 6.4 Replikace

Replikací rozumíme technologii, kdy je možné nasadit více databázových serverů v rámci jedné databáze. Jedná se tak o sdílení dat mezi více hardwarovými, softwarovými a jinými prostředky a jejich přenositelnost. Účelem replikace je tak dosáhnout vysoké dostupnosti databázového systému a škálování výkonu pro optimalizaci v prostředí vysoké zátěže. Obecně existují dvě základní varianty databázových replikací od kterých se odvyjí jejich další využití. Samozřejmě v závislosti na konkrétním databázovém systému pak existují další členění a nastavení.

Replikace varianty master-slave je podporována ve většině databázových systémech. Jedná se o jednodušší tzv. jednosměrnou replikaci. V této variantě je určen autonomní prvek, jedna replikace, která akceptuje a zpracovává požadavky na změny. Takováto replikace nese název master. Prvek s názvem slave je věrnou kopií autonomního prvku master. Slouží pouze ke čtení a může jich být více pro jeden master. Jakmile master obdrží a zpracuje požadavek na změnu, tak jej po dokončení přenese na ostatní slave replikace.

Replikace typu master-master bývá označována jako obousměrná. To znamená, že jsou v rámci jednoho databázového systému minimálně dvě replikace typu master, které akceptují všechny druhy požadavků na změny i čtení a přenáší je vzájemně mezi sebou. Z této vlastnosti vyplývá, že může dojít ke kolizím, kdy například dvě replikace master zapisují do stejné tabulky. Takovéto kolize jsou nevyhnutelné, a je potřeba je řešit.

Způsob přenosu mezi jednotlivými replikacemi může být synchronní či asynchronní. U synchronního přenosu se čeká až se změny provedou na všechny os-

tatní repliky. Takovýto proces je časově náročný, ovšem na druhou stranu je celý databázový systém konzistentní jako celek. U asynchronního přenosu se nečeká na dokončení přenosu mezi ostatními replikacemi. Díky tomu je celý databázový systém rychlejší, ovšem může dojít k nekonzistenci, kdy na ostatní replikace ještě nejsou přenesena všechna data.

Administrace, nástroje a konfigurace replikací jsou zabudované v téměř každém databázovém systému. Je důležité ale poznamenat, že tyto nástroje nejsou mnohdy dostačujícími řešeními pro architektury v prostředí vysoké zátěže a je proto nutné používání jiných doplňkových nástrojů. Také je více než důležité říct, že v prostředí vysoké zátěže se webová architektura bez databázových replikací jen těžko obejde. (Tomaáš Vondra, 2011)

## 6.5 Druhy relačních databází

V dnešní době existuje několik druhů relačních databázových systémů. Každý z nich má své klady a zápory, ovšem princip a způsob práce těchto databází je v základu podobný. Uvádím zde přehled těch v praxi se běžně vyskytujících:

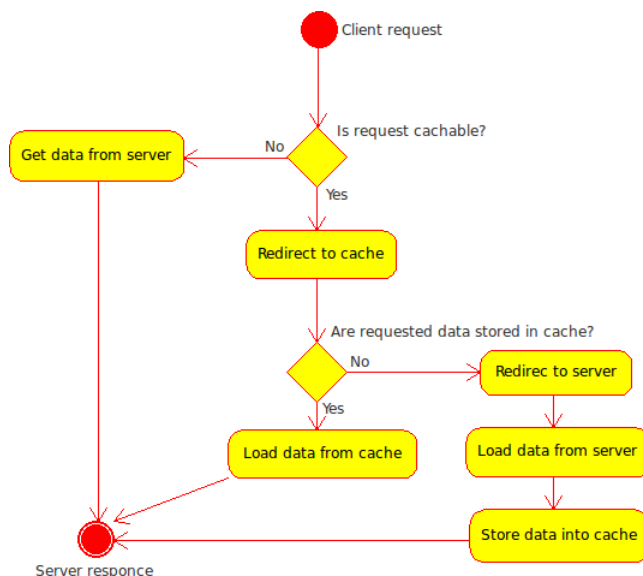
- Oracle
- MySQL
- PostgreSQL
- MSSQL
- Firebird
- a mnoho dalších

## 7 Webové cache

Webové cache jsou důležitou vrstvou pro rozšíření výchozí třívrstvé architektury. Jak už napovídá z názvu cache, jedná se o vyrovnávací paměť jejíž hlavní účel je zrychlit odpověď webové aplikace na požadavek klienta. Zároveň je důležité poznamenat, že nefunkčnost, zánik či pád cache vrstvy nesmí nijak ovlivnit chod aplikace, která musí být funkční i nadále. Účel této vrstvy je pouze zrychlit přístup k datům či již jednou interpretovaných odpovědí webové aplikace.

Webová cache se tak nachází mezi klientem a aplikační vrstvou. Základní komunikace probíhá tak, že klient pošle HTTP požadavek, aplikační server ho přijme a navrátí HTTP odpověď. Při existenci cache vrstvy probíhá komunikace jiným způsobem. Klient pošle HTTP požadavek a webová architektura zjistí, zda-li je tento HTTP požadavek určen pro ukládání do cache paměti. Pokud ano, tak webová architektura zkusí získat odpověď z cache paměti. Když tuto odpověď v paměti nalezne, zašle jej přímo klientovi. V opačném případě je požadavek poslán do aplikační vrstvy architektury, která jej zpracuje, vytvoří odpověď kterou uloží do cache paměti a pošle klientovi. Při příštím stejném požadavku bude odpověď vrácena z paměti cache. Dojde-li ke změně dat, která jsou uložena v cache paměti, dojde k invalidaci uložených dat v cache paměti a celý proces začne od začátku s novým HTTP požadavkem.

Pro webové architektury v prostředí vysoké zátěže je tato vrstva nevyhnutelná. Odpovědi z cache vrstvy jsou daleko rychlejší, dokonce až mnohonásobně, než odpovědi z aplikační vrstvy. Ve své práci se zabývám právě různými druhy cache a z výsledků profilování a testování vyplývá, že jsou tato tvrzení pravdivá.



Obr. 4: Diagram aktivit cache webových systémů

## 7.1 Typy obsahu

Z pohledu ukládání do cache paměti je přístup a druh obsahu webové architektury rozdělen na dva typy. Obsah může být buď statického nebo dynamického charakteru. Jak už vyplývá z obecné teorie systémů, statický obsah je konečný, transparentní a měněn podle předepsaných pravidel, zatím co dynamický obsah se mění v závislosti na čase a jiných nepředvídatelných pravidlech. Například invalidace u statického obsahu proběhne pouze jednou za deset minut. A i když je obsah změněn uprostřed tohoto intervalu, změna se projeví až v přelomu těchto pravidelných desetiminutových intervalů, tedy až po invalidaci. Obsah je tak celých deset minut statický a neměnný. Naopak dynamický obsah se může změnit kdykoli, nepředvídatelně, typicky na nějakou uživatelskou činnost a nebo podle času. Například při vložení komentáře se ihned změní seznam komentářů. U dynamické ukládání dat do cache paměti je tak potřeba se zaměřit především na jejich změnu, tedy na jejich invalidaci.

Z uvedených vlastností vyplývá, že správa cache paměti statického obsahu je jednodušší, kdežto správa cache dynamického přístupu je náročnější. Ve své práci ukazují, analyzují a testují oba dva přístupy a srovnávám jejich přínos jak z pohledu aplikace, tak náročnosti na vybudování.

## 7.2 HTTP hlavičky pro ovládání cache

Komunikace ve webových aplikacích probíhá ve valné většině na úrovni protokolu HTTP. Každý HTTP požadavek obsahuje HTTP hlavičky, kterými určuje požadavek, klienta a obsahuje časové razítko. Odpověď na tento požadavek navrací v hlavičkách návratový kód, informace o serveru, časové razítko, typ obsahu odpovědi, délku odpovědi a informace pro případné ukládání do cache paměti. Právě jednotlivé cache hlavičky jsou důležité pro nastavení ukládání a invalidace do proxy cache paměti nebo do reverzní proxy cache paměti.

Existují dva základní modely pro určení práce s cache vrstvou. Prvním je expirační model. Tento model určuje, do kdy je platná HTTP odpověď, neboli do kdy může tuto odpověď cache vrstva ukládat a označovat ji jako platnou a stále čerstvou. Tento model může být realizován dvěma způsoby. Prvním je hlavička "Expires" udávající časové razítko do kdy je odpověď čerstvá. Druhým, novějším, daleko flexibilnější a konfigurovatelnější způsobem je použití "Cache-control". Díky tomuto způsobu je možné určit další parametry a to nejenom do kdy je daná HTTP odpověď platná. Je možné tím i určit, zda-li je určena jen pro sdílené cache (proxy cache), nebo jen pro uživatelské prohlížečové cache, nebo zda-li je ukládat či nikoli, atd.(RFC:2616, 1999)

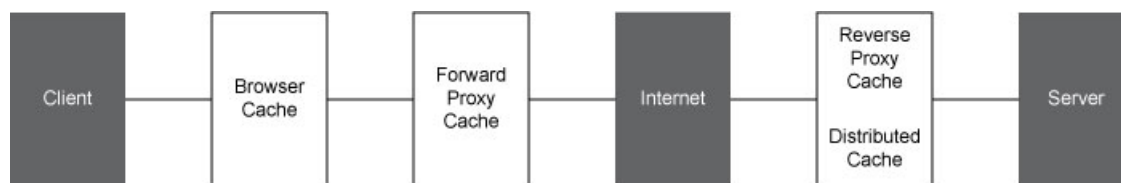
Druhým modelem je model validační. Ten určuje způsob komunikace pro zjištění, zda-li je odpověď uložená v cache paměti stále validní. Základem je, že cache, která má uloženu odpověď se zeptá je-li odpověď stále validní a server odpoví jestli ano či vrátí novou čerstvou odpověď. Realizováno to může být opět dvěma způsoby. Tím prvním je určení tzv. "Last-Modified". Při prvním požadavku cache

získá odpověď s touto hlavičkou. Odpověď si uloží a při dalším požadavku se cache vrstva dotáže serveru pomocí hlavičky "If-Modified-Since" s časovým razítkem, zda-li je odpověď stále validní. Server odpoví návratovým kódem 304, který říká že nedošlo k žádné změně od daného časového razítka, a nebo vrátí novou čerstvou odpověď pokud došlo ke změně. Podobný princip je založen i na způsobu pomocí hlavičky "Etag" neboli "Entity tag". Etag je nějaký unikátní identifikátor vygenerovaný serverem pro danou HTTP odpověď. Cache vrstva si při prvním požadavku uloží odpověď s tímto Etag. Při dalším požadavku pošle na server HTTP požadavek s hlavičkou "If-None-Match" s tímto Etag. Pokud se Etag shodují, vrátí aplikační server odpověď s návratovým kódem 304, a nebo celou čerstvou odpověď. (RFC:2616, 1999)

Ovládání pomocí HTTP hlaviček slouží hlavně k statiskému obsahu. Pro tento účel je pak dobré zvážit jestli je lepší odpovědi měnit v nějakých časových intervalech a používat tak expirační model, či jestli se nevyplatí nasazovat model validační, který nese složitější administraci.

## 7.3 Druhy cache

Ve webových architekturách exstuje několik druhů cache vrstev. Jejich členění se odvíjí od jejich účelu, pozici v architektuře a typu ukládaných dat. Tyto atributy mezi sebou neúzce souvisí. Základním členěním může být jejich pozice z pohledu internetu, a to na klientskou a serverovou část. Za klientskou část považujeme cache systém uživatele prohlížeče, či systémovější proxy cache. Na straně serveru se vyskytují reverse proxy cache či distribuované cache.



Obr. 5: Druhy cache webových systémů

### 7.3.1 Proxy cache a cache prohlížeče

Tyto druhy cache se týkají klientské části webové architektury, neboli části před internetem. Znamená to tedy, že odpověď na klientův požadavek je vrácena z cache paměti lokální sítě, a tento požadavek vůbec nevstoupí do internetu na vzdálený dotazovaný server. Tyto cache vrstvy zde nejsou jen pro to, aby zrychlily odevzvu na požadavek, ale také aby zmenšily odchozí komunikaci z lokální sítě. A to z toho důvodu, aby byla šířka pásma přístupná i jiným službám, a ušetřili se prostředky za odchozí komunikaci.

V dnešních moderních prohlížečích je práce s cache pamětí podporována. Cache prohlížeče může nastavovat, konfigurovat, či vymazat pouze uživatel, neboli klient

sám. Tato paměť ukládá hlavně statický obsah, například obrázky, CSS soubory či JS soubory. Cache paměť najde své uplatnění například když jsou uživatelovi požadavky směřovány jedné webové aplikaci se stejným vzhledem, rozhraním a nebo částí funkcionalit. Takováto data mohou být ukládána do cache paměti na dlouhou dobu, a proto musí aplikace dbát na to, aby byla tato data z cache promazávána tak jak je opravdu potřeba. Jednou z možností je určit jinou url adresu pro stejný soubor, pomocí tzv. "Query string", například takto:

```
<link rel="stylesheet" type="text/css" href="style.css?QUERY_STRING" media="all" />
```

Proxy cache bývají budovány a instalovány providery a poskytovali připojení k internetu. Jejich záměr je jednoduchý. Jde o to snížit míru odchozí komunikace, nezatěžovat tolik šířku pásma a zmenšit tak náklady spojené s touto komunikací. Nejběžnějším nastavením těchto proxy cache pamětí bývá ukládání a invalidace obsahu v závislosti na HTTP hlavičkách přijaté odpovědi. Nainstalujeme-li například takovouto cache ve firmě či nějakém sídlišti s několika desítky, sty či dokonce až tisíci uživateli, je možné, že část z nich má každé ráno stejný požadavek na stejný zpravodajský server, tudíž mají i stejnou odpověď. A tak může být první odpověď uložena do cache paměti a těm dalším uživatelům může být odpověď servírována z proxy cache paměti. (Mark Nottingham, 2012)

### 7.3.2 Reverzní proxy cache

Reverzní proxy cache, někdy označovány jako tzv. "Gateway cache", mají podobný systém činnosti jako proxy cache. Mají za úkol ukládat statický obsah ve formě HTTP odpovědí a tyto odpovědi vracet nazpět odkud přišel jejich požadavek. Úkolem je tak pokud možno nepropustit požadavek dále na aplikační vrstvu, stejně jako úkolem proxy cache je pokud možno nepropouštět požadavek dále do internetu. Rozdíl mezi reverzní proxy cache a proxy cache spočívá v jejich pozici ve webové architektuře, a také v tom, kým jsou instalovány. Pozice reverzní proxy cache je na straně serveru, neboli blíže aplikační vrstvě architektury, tedy na druhé straně internetu, než odkud přichází požadavky klientů a kde jsou instalovány proxy cache. Reverzní proxy cache jsou instalovány samotnými administrátory a programátory webové aplikace. Jejich účel je zajistit rychlejší odezvu webové aplikace na HTTP požadavky klientů. Reverzní proxy cache jsou nedílnou součástí webové architektury v prostředí vysoké zátěže a bez jejich existence by jen těžko mohla být webová aplikace spolehlivá a rychlá. (Mark Nottingham, 2012)

### 7.3.3 Aplikační distribuovaná cache

Doposud byly popisovány cache paměti pro ukládání statického obsahu. Ovšem v praxi se od projektového vedení setkáme s požadavky na dynamické chování aplikace. Příkladem může posloužit opět situace s komentáři, kdy jeden uživatel vloží komentář k některému z článků. Tento komentář se musí okamžitě objevit v seznamu komentářů k článku. V případě statického obsahu, který se mění pouze v

nějakých intervalech, může uživatel nabýt dojmu, že tento komentář se nepodařilo vložit a zkusí ho vložit znovu. Takovéto chování aplikace určitě není v souladu s dobrým chováním interakce mezi uživatelem a aplikací. K tomu, aby se dala data ukládat do cache a aplikace byla stále dynamická, slouží tzv. aplikační distribuovaná cache. HTTP požadavek je tak rozdistribuován mezi menší požadavky a mezi cache paměti.

Jak už vyplývá z názvu druhu této cache, o manipulaci s touto cache pamětí se stará aplikační vrstva. Standartně se ukládají do cache paměti ta data, k nimž je dlouhá doba přístupu, obvykle data z databáze. Účelem je tedy ušetřit spojení a dotazy nad databází. Jeden HTTP požadavek s konkrétní url je často rozdělen na několik požadavků do databáze, v závislosti na tom, o která data se jedná. Například v rámci jedné HTTP stránky může být databáze dotázána o požadovaný článek a seznam komentářů pod článkem. Databáze tak obdrží dva SQL dotazy, na které vrátí příslušná data z databáze. Aby se příště aplikace nemusela na tato data dotazovat, uloží si je do aplikační cache paměti.

Při ukládání takových dat do cache paměti aplikační vrstva určuje jak a kam mají být data uložena, a kdy a za jakých podmínek mají být invalidována. Při aplikaci této cache vrstvy dochází k propuštění požadavku na aplikační servery, což znamená větší zátěž aplikační vrstvy. Reverzní proxy cache jsou v tomto směru nepoužitelné, slouží pouze pro statický obsah a všechny požadavky propouští dále na aplikační vrstvu.

Při práci s aplikační distribuovanou cache je důležité si uvědomit několik základních pravidel. Prvním pravidlem je ukládat data do cache paměti co nejmenší. Účelem je tak distribuovat jednotlivá data v různých kontextech. Dále je důležité ukládat data znovuzkonstruovatelná. Je důležité umět z cache paměti přecíst to, co do ní bylo uloženo. Nejlepší je ukládat data serializovaná. Dále je důležité dodržet jednoznačný a jasně definovaný klíč či jmenný prostor pro tato data. Je totiž důležité vědět kde a jak se ptát na data. Největším problémem je invalidace dat v cache paměti. Stejná data mohou být interpretována v různém kontextu. Například při vložení nového článku se změní data pro seznam všech nejnovějších článků, a také data pro seznam nejnovějších článků v dané rubrice. Aplikace má za požadavek dynamické chování, a tak musí být tento článek vidět ihned po vložení ve všech zmíněných kategoriích. Čili musí dojít k invalidaci všech závislostí. Tento problém je označován jako invalidační kaskády, a proto je důležité konstruovat aplikaci tak, aby se těmito kaskádám vyhýbalo co nejvíce či s nimi počítat už při návrhu aplikační cache vrstvy.

## 8 Další vrstvy aplikace

K čemu jsou další vrstvy

### 8.1 CDN

Content delivery network obrázky a stream.

### 8.2 NoSQL Databáze

K čemu slouží a kde najdou své uplatnění.

### 8.3 Vyhledávání

Z vyhledávání se také dělá další vrstva.



## 9 Virtualizace

Projekty dnes neběží vždy na jednom serveru, ale na více virtualizovaných serverech. Proč tomu tak je.

## 10 Load balancing

Nevím jestli k této kapitole se vůbec dostanu, uvidíme. Každopádně serverů bývá vždy několik a jak zajišťovat toto rozložení zátěže.

## 11 Cloud Computing

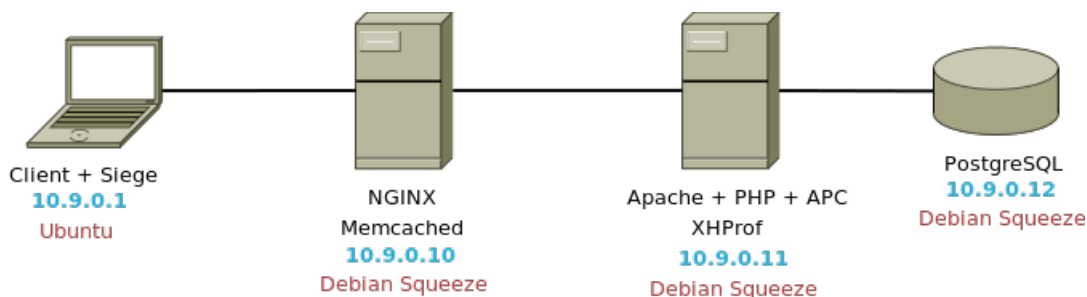
Budoucnost projektů, startupů, vše řešeno cloudem. AWS

## 12 Praktická část s experimenty a výsledky

V praktické části se zaměřuji na testy, optimalizaci jednotlivých vrstev webové architektury v prostředí vysoké zátěže. Rozebírám zde jednotlivé vrstvy architektury, tedy aplikační vrstvu, databázovou vrstvu, reverzni cache vrstvu a aplikační distribuovanou cache. Součástí práce nejsou pouze popis a charakteristika jednotlivých kroků optimalizace, ale i jejich výsledné statistiky a zátěžové testy. Hlavní částí praktické části je ovšem návrh a implementace aplikační úrovně plně podporující práci s aplikační distribuovanou cache. Dále je popsána i výsledná konfigurace a nastavení jednotlivých vrstev architektury. Výsledkem je tedy kompletní použitelná studie řešící problematiku webové architektury v prostředí vysoké zátěže.

### 12.1 Vrstvy webové architektury

Webová architektura sestává z klientské vrstvy, reverzní proxy cache vrstvy, aplikační vrstvy, aplikační distribuované cache a databázové vrstvy. Klientská část je tvořena jedním uživatelem s operačním systémem Linux Ubuntu a zátěžovým testovacím nástrojem Siege. Reverzní proxy cache vrstva je realizována programem Nginx. Aplikační vrstva se skládá z webového serveru Apache2 s programovacím jazykem PHP a s profilovacím nástrojem XHProf. Aplikační distribuovaná cache vrstva je tvořena programem Memcached. Databází je PostgreSQL. Všechny tyto vrstvy serverové části jsou vytvořeny na zvláštních serverech. Pro reverzní proxy cache a aplikační distribuovanou cache byl zvolen jeden server. Každý ze serverů je virtualizován za pomoci virtualizačního nástroje VirtualBox. Celá webová architektura je konfigurována v jedné virtuální lokální síti 10.9.0.0/24 se statickým přidělováním IP adres. Rozhraní pro připojení těchto virtualizovaných serverů do virtuální sítě je realizováno pomocí síťových mostů. Vzhledem k tomu, abych mohl ukázat velké poměrové rozdíly v jednotlivých krocích optimalizace v jednotlivých vrstvách architektury, rozhodl jsem se o velice slabé hardwarové vybavení jednotlivých serverů, která mají 512MB operační paměti RAM a jeden jednojádrový procesor s frekvencí 2GHz.



Obr. 6: Vrstvy implementované webové architektury

## 12.2 Testovací a profilovací nástroje

Proto, abych mohl všechna svá tvrzení a závěry řádně podložit a abych mohl identifikovat místa nutná pro optimalizaci, jsou potřeba určité testovací a profilovací nástroje. Pomocí profilovacího nástroje XHProf jsem vyprofiloval statistiky pro aplikační vrstvu. Udělal jsem rozборы exekučních plánů pro SQL dotazy pro databázovou vrstvu, aby byla vidět míra jejich míra optimalizace. Také bylo potřeba vytvořit prostředí vysoké zátěže.

### 12.2.1 XHProf

K tomu, aby bylo možné profilovat programovací jazyk PHP, jsem zvolil profilovací nástroj XHProf. Tento nástroj byl vyvinut v programovacím jazyce C. Uvolněn byl pod open-source licencí Apache 2.0. Autorem tohoto profilovacího nástroje je společnost Facebook. Tento profilovací nástroj dokáže vyprofilovat nejenom dobu trvání jednotlivých metod a funkcí, ale i jejich procesorové či paměťové nároky. XHProf je možné provozovat na operačních systémech Linux, FreeBSD a Mac OS X.

Jeho instalace a konfigurace je jednoduchá a bezproblémová. Stačilo mi si stáhnout instalační balíček s rozšířením, zkompileovat a nainstalovat. Po instalaci jej bylo nutné zaregistrovat jako rozšíření pro php v konfiguračním souboru `php.ini`. Samotné použití je pak velice snadné. Na začátek php skriptu jsem přidal zdrojový kód s jeho aktivací a na konci php skriptu kód pro jeho vypnutí.

Výsledkem profilování je přehledná statistika jednotlivých metod a funkcí ve formátu HTML. Všechno v přehledné tabulce, ve které je možné měnit pohled výsledků profilování podle specifické metody či funkce. V tabulce je možné pozorovat čas strávený vykonáváním pouze dané funkce, čas strávený vykonáváním i funkcí z ní volané, počet volání funkce, doba trvání, procesorové čas a paměťové nároky. Výsledkem může být i graf vygenerovaný z těchto statistik a ze závislostí mezi jednotlivými funkcemi.

Tento program je nenáročný a může být provozován i v ostrém provozu. Výsledky takového profilování jsou důležité k identifikaci problémových míst k optimalizaci, čili pro webovou architekturu v prostředí vysoké zátěže jsou nevyhnutelné. (Jakub Onderka, 2012)

Příklad použití XHProf profilování:

```
<?php
// Enable profiling
if (extension_loaded('xhprof')) {
    include_once '/usr/local/lib/php/xhprof_lib/utils/xhprof_lib.php';
    include_once '/usr/local/lib/php/xhprof_lib/utils/xhprof_runs.php';
    xhprof_enable(XHPROF_FLAGS_CPU + XHPROF_FLAGS_MEMORY);
}

// Php code for profiling
...

// Disable profiling
if (extension_loaded('xhprof')) {
    $profiler_namespace = 'myapp'; // namespace for your application
    $xhprof_data = xhprof_disable();
    $xhprof_runs = new XHProfRuns_Default();
    $run_id = $xhprof_runs->save_run($xhprof_data, $profiler_namespace);
}
?>
```

### 12.2.2 Siege

Vzhledem k tomu, abych mohl nasimulovat prostředí vysoké zátěže jsem zvolil testovací nástroj Siege. Siege je tedy nástrojem pro testování webových aplikací a jejich doby odezvy v prostředí vysoké zátěže. Tento nástroj má opět lehkou a přívětivou instalaci, nachází se totiž v základních instalačních balíčcích pro operační systém Linux Debian. Siege je programem spouštěným z příkazové řádky. Jako každý program, tak i siege má samozřejmě svůj konfigurační soubor, který je důležité hlavně pro nastavení maximální prováděcí doby jednotlivých testů.

Volby parametrů tohoto příkazového programu jsou velice bohaté, ale přitom jednoduché pro pochopení a použití. Základními parametry jsou:

- c - počet simulovaných konkurentních uživatelů
- d - interval zpoždění mezi jednotlivými uživatelskými požadavky
- r - počet repetice pro zátěžové testy

Ve své práci jsem prováděl zátěžové testy simulováním deseti uživatelů, v intervalu od nuly do jedné sekundy ve třech repeticích. Spuštění vypadá takto:

```
siege -d1 -c10 -r3 -v http://dp-xskrha.local/hello/Mendelu
```

### 12.2.3 PostgreSQL Explain

Pro konkrétní analýzu problémových SQL dotazů je potřeba identifikovat a analyzovat jejich exekuční plán. V databázi PostgreSQL a ve většině databázích k tomu slouží příkaz explain, který takto zobrazí exekuční plán. Ve své práci jsem analyzoval problémová místa pomocí XHProf profilování, ze kterého jsem zjistil, že všechna problémová místa souvisí s dotazi pro databázi. Po této fázi identifikace následovala fáze vysvětlení pomocí SQL dotazu explain.

Pomocí explain je možné vysvětlit všechny kroky exekučního plánu. Zjistíme tak jaký způsob prohledání byl zvolen, zda-li sekvenční nebo indexový. Dále je možné zjistit druh spojení, jestli merge join nebo hash join, apod. U jednotlivých kroků exekučního plánu je zobrazena jejich cenu, předpokládaný počet řádků a předpokládaný počet sloupců. Takto je možné identifikovat slabá místa exekučního plánu a provést tak možnou úspěšnou optimalizaci.

Ukázka analýzy exekučního plánu:

```
explain analyze select * from film f where film_id in (select film_id from film_actor);
                                QUERY PLAN
-----
Hash Join  (cost=117.26..195.78 rows=977 width=390) (actual time=21.067..26.570 rows=997 loops=1)
  Hash Cond: (f.film_id = film_actor.film_id)
    -> Seq Scan on film f  (cost=0.00..65.00 rows=1000 width=390)
        (actual time=0.016..1.699 rows=1000 loops=1)
    -> Hash  (cost=105.05..105.05 rows=977 width=2) (actual time=21.029..21.029 rows=997 loops=1)
        -> HashAggregate  (cost=95.28..105.05 rows=977 width=2)
            (actual time=17.598..19.298 rows=997 loops=1)
            -> Seq Scan on film_actor  (cost=0.00..81.62 rows=5462 width=2)
                (actual time=0.012..8.019 rows=5462 loops=1)
Total runtime: 28.020 ms
```

## 12.3 Aplikační vrstva PHP

Ve své aplikační vrstvě jsem zvolil pro implementaci programovací jazyk PHP běžící na webovém serveru Apache. Webový server Apache je jedním z nejrozšířenějších a nejpopulárnějších webovým serverem na internetu. Byl implementován v roce 1996 v jazyce C++. Jeho instalace, konfigurace a administrace není nikterak složitá. Na spoustě webových hostingů je dostupný v základní konfiguraci. Je to volně použitelný produkt, který obsahuje spoustu různých přídatných módů. Z těchto důvodů jsem ho vybral pro praktickou část své diplomové práce.

Programovací jazyk PHP se stal jedním z nejpoužívanějších programovacích jazyků pro svoji srozumitelnost, přenositelnost a jednoduchost. Je to dynamicky typovaný programovací jazyk, čili i z těchto důvodů je hodně ohebný. Plně podporuje OOP přístup, čili je možné vyžívat těchto technik včetně návrhových vzorů, které jsou pro složité webové aplikace v prostředí vysoké zátěže velice důležité.

### 12.3.1 Optimalizace pomocí APC

Programovací jazyk PHP je sice jednoduchý na vývoj, čitelnost a dynamičnost, ovšem jeho daň je právě náročnost. Tento jazyk je interpretovaný jazyk, čili při jeho interpretaci dochází k překladu do "Opcode" PHP kompilátorem. Vzniká tak jakýsi mezikód, který je následně po skončení překladu vykonán.

K tomu, aby se nemusel pokaždé překládat stejný script PHP kompilátorem, je možné ho uložit do cache pro mezikód, neboli Opcode cache. K tomuto účelu slouží APC, neboli Alternative Opcode Cache. APC je rozšířením pro PHP vytvořené přímo samotnými tvůrci programovacího jazyka PHP. Toto rozšíření není nijak zvláště náročné na instalaci ani konfiguraci. Dokonce je k němu možné zprovoznit i

webové rozhraní ukazující počet úspěšných či neúspěšných dotazů do cache a jiné další statistiky. Prostřednictvím APC programového rozhraní se dají do cache paměti ukládat i jiné další hodnoty, a to právě přímo ze zdrojových kódů. Úložištěm pro APC je operační paměť RAM. Z toho důvodu je tedy důležité, aby webserver Apache2 stále běžel zapnutý. Při restartu web serveru dojde k invalidaci celé APC cache paměti.

Většina interpretovaných jazyků je překládána do nějakého mezikódu. Proto se při žádné změně zdrojových kódů dá využít už jednou zkompileovaných výsledků. Tato forma recyklace zkompileovaného mezikódu dokáže přinést velké výsledky při optimalizaci. U APC jsou tyto výsledky více než viditelné.

### 12.3.2 Dosažené výsledky

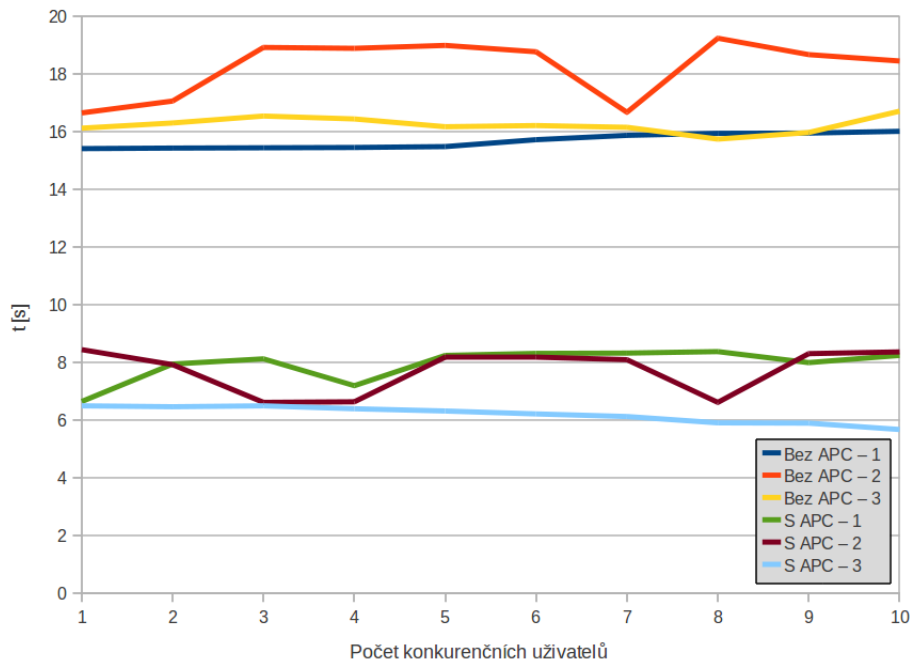
Pro dosažení výsledků optimalizace aplikační vrstvy jsem vygeneroval prostředí vysoké zátěže deseti konkurenčních uživatelů s intervalem požadavků do jedné sekundy ve třech repeticích. Na aplikačním serveru tak vzniklo deset vláken Apache2 s PHP pro každého z uživatelů a celý procesorový výkon byl tak rozdělen rovnoměrně mezi jednotlivé vlákna po deseti procentech. Vytížení procesoru aplikačního serveru tak dosahovalo sta procent, čili maximálního vytížení. Tuto prostředí jsem nasimuloval jak pro neoptimalizované prostředí bez APC, tak i pro optimalizované prostředí s APC. Pro každý požadavek docházelo ke zpracování celého MVC frameworku, čili ke zpracování mnoho funkcí a metod.

Z XHProf profilování jsem vyprofiloval výsledky pro jeden požadavek jak pro APC, tak i bez APC. Výsledky toho profilování jsou takovéto:

- neoptimalované - bez APC - 1,3s
- optimalizované - s APC - 0,4s

Z výsledku vyplývá, že při optimalizaci aplikační vrstvy pomocí APC došlo k zrychlení provedení PHP skriptů o více než padesát procent, a to jak při jediném požadavku, tak i v prostředí vysoké zátěže. Takovýto fakt se dá určitě označit úspěšnou optimalizací. Alternativou může být použití přístupu s webovým serverem Lighttpd a s cache pamětí pro mezikód eAccelerator. Výsledky by měli být hodně podobné. V dnešních dnech se objevují a publikují zprávy o uvolnění HIP-HOP pro PHP od společnosti Facebook. HIP-HOP používá úplně jiného přístupu pro zrychlení běhu PHP skriptů. Tato varianta je dnes ještě hodně čerstvá a ne tak prověřená, ovšem představuje další možnosti pro optimalizaci aplikační vrstvy v prostředí vysoké zátěže v budoucích dnech.





Obr. 7: Graf s výsledky doby trvání jednotlivých požadavků pro repetice

```
top - 17:14:24 up 6 min, 3 users, load average: 1.37, 0.43, 0.18
Tasks: 137 total, 1 running, 136 sleeping, 0 stopped, 0 zombie
Cpu(s): 12.6%us, 47.0%sy, 0.0%ni, 10.3%id, 0.0%wa, 30.1%hi, 0.0%si, 0.0%st
Mem: 514700k total, 361288k used, 153412k free, 11304k buffers
Swap: 392184k total, 0k used, 392184k free, 196388k cached
```

| PID  | USER     | PR | NI | VIRT  | RES  | SHR  | S | %CPU | %MEM | TIME+   | COMMAND         |
|------|----------|----|----|-------|------|------|---|------|------|---------|-----------------|
| 1085 | www-data | 20 | 0  | 92012 | 19m  | 12m  | S | 11.7 | 3.9  | 0:01.78 | apache2         |
| 1088 | www-data | 20 | 0  | 80288 | 21m  | 15m  | S | 11.4 | 4.2  | 0:00.81 | apache2         |
| 1091 | www-data | 20 | 0  | 80780 | 17m  | 11m  | S | 11.1 | 3.5  | 0:02.12 | apache2         |
| 1846 | www-data | 20 | 0  | 82652 | 15m  | 7600 | S | 11.1 | 3.1  | 0:01.27 | apache2         |
| 1086 | www-data | 20 | 0  | 94976 | 37m  | 17m  | S | 10.4 | 7.4  | 0:07.75 | apache2         |
| 1847 | www-data | 20 | 0  | 84100 | 19m  | 10m  | S | 10.4 | 3.9  | 0:01.12 | apache2         |
| 977  | root     | 20 | 0  | 44240 | 18m  | 6460 | S | 6.5  | 3.7  | 0:05.02 | Xorg            |
| 1855 | www-data | 20 | 0  | 82652 | 15m  | 7604 | S | 3.9  | 3.1  | 0:00.86 | apache2         |
| 1089 | www-data | 20 | 0  | 80352 | 13m  | 7700 | S | 3.6  | 2.6  | 0:01.30 | apache2         |
| 1856 | www-data | 20 | 0  | 80688 | 15m  | 9360 | S | 3.6  | 3.0  | 0:00.79 | apache2         |
| 1857 | www-data | 20 | 0  | 82820 | 17m  | 9988 | S | 3.3  | 3.6  | 0:00.92 | apache2         |
| 1890 | serga01  | 20 | 0  | 83304 | 11m  | 9216 | S | 1.0  | 2.3  | 0:00.63 | gnome-terminal  |
| 1786 | serga01  | 20 | 0  | 19940 | 9836 | 8076 | S | 0.7  | 1.9  | 0:00.60 | metacity        |
| 1143 | root     | 20 | 0  | 17408 | 3048 | 2236 | S | 0.3  | 0.6  | 0:00.14 | console-kit-dae |
| 1784 | serga01  | 20 | 0  | 21928 | 9404 | 6728 | S | 0.3  | 1.8  | 0:00.46 | gnome-settings- |
| 1788 | serga01  | 20 | 0  | 87592 | 18m  | 13m  | S | 0.3  | 3.7  | 0:01.37 | gnome-panel     |
| 1930 | root     | 20 | 0  | 2468  | 1192 | 900  | R | 0.3  | 0.2  | 0:00.14 | top             |
| 1    | root     | 20 | 0  | 2036  | 716  | 624  | S | 0.0  | 0.1  | 0:00.94 | init            |
| 2    | root     | 20 | 0  | 0     | 0    | 0    | S | 0.0  | 0.0  | 0:00.00 | kthreadd        |
| 3    | root     | RT | 0  | 0     | 0    | 0    | S | 0.0  | 0.0  | 0:00.00 | migration/0     |
| 4    | root     | 20 | 0  | 0     | 0    | 0    | S | 0.0  | 0.0  | 0:00.01 | ksoftirqd/0     |
| 5    | root     | RT | 0  | 0     | 0    | 0    | S | 0.0  | 0.0  | 0:00.00 | watchdog/0      |

Obr. 8: Výpis systémových procesů aplikačního serveru

## 12.4 Databázová vrstva

Databázová vrstva je důležitou a nezbytnou součástí webové architektury. Její úloha je perzistence dat pomocí databázového systému. Ovšem přístup k datům může být někdy velice zdoluhavý, až v řádech několik minut a hodin. V prostředí vysoké zátěže dochází obvykle k obrovskému růstu dat a je potřeba nastavit optimalizovat databázovou vrstvu tak, aby jejich doba přístupu byla co nejkratší.

Ve své práci jsem vytvořil databázi nad databázovým systémem PostgreSQL. Tento systém je volně dostupný a najde velké uplatnění právě ve webových architekturách. Podporuje všechny standartní databázové operace a přístupy a je oblíben pro svoji spolehlivost.

Databáze modelované aplikace obsahuje záznamy v řádech miliónů řádků. To z toho důvodu, aby bylo nasimulováno prostředí vysoké zátěže i s velkou databází. Datové záznamy byly pro potřeby mé práce vygenerovány a nepředstavují žádnou paralelu s reálnými daty.

### 12.4.1 Optimalizace databáze

Před samotnou optimalizací databázové vrstvy nebyla databáze žádným způsobem optimalizována, ať už za pomoci indexace, optimalizace SQL dotazů, segmentování či vytvoření replikací. Přístup k datům v databázi také nebyl žádným způsobem efektivní. Jednalo se o znovu se opakující dotazy a dokonce místo jednoho SQL dotazu se objevovalo více dotazů pro zjištění stejné inforační hodnoty. Rozhodl jsem se tedy pro aplikování kroku optimalizace databázové vrstvy pomocí optimalizace dotazů SQL.

Při procesu optimalizace jsem si napřed určil problematické sql dotazy pomocí XHPProf profilování. Z tohoto procesu jsem zjistil které sql dotazy je potřeba zoptimalizovat. Dalším krokem následoval rozbor jejich exekučního plánu, vyhledání řešení a jeho aplikace. Řešením optimalizace některých z exukčních plánů byla indexace příslušných sloupců pro nahrazení sekvenčního prohledání za indexové prohledání. Zde je ukázka rozboru neoptimalizovaného exekučního plánu se sekvenčním prohledáním:

```
SELECT * FROM dp_comment WHERE article_id = 15 ORDER BY create_time DESC;
-----
Sort (cost=121201.07..121201.08 rows=3 width=540) (actual time=16300.440..16300.444 rows=3 loops=1)
  Sort Key: create_time
  Sort Method: quicksort Memory: 20kB
  -> Seq Scan on dp_comment (cost=0.00..121201.05 rows=3 width=540)
        (actual time=13.531..16300.393 rows=3 loops=1)
        Filter: (article_id = 15)
Total runtime: 16300.526 ms
```

Po optimalizace je vidět, že sekvenční prohledání bylo odstraněno a nahrazeno indexovým prohledáním, které je daleko efektivnější:

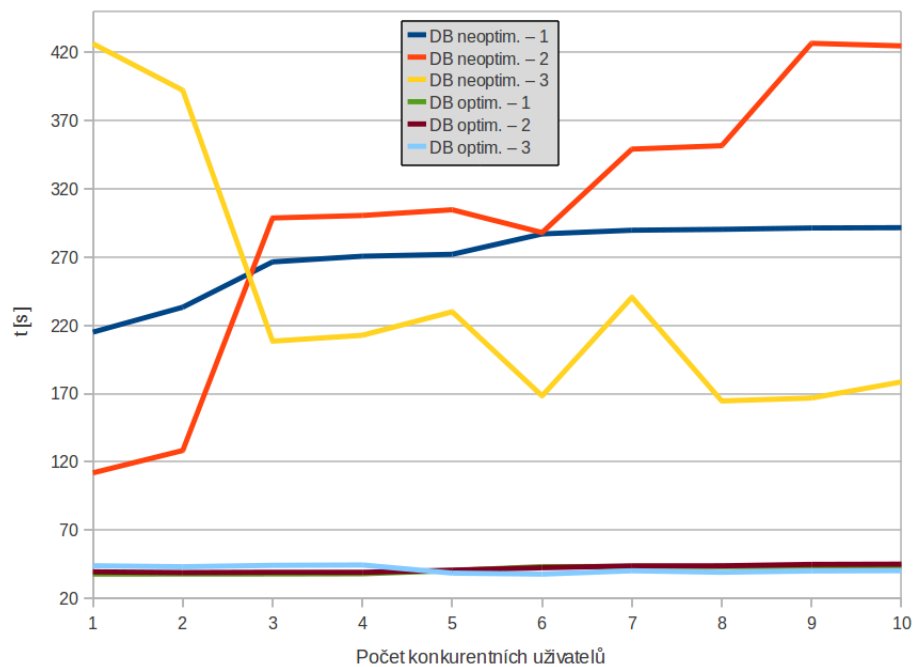
```

SELECT * FROM dp_comment WHERE article_id = 15 ORDER BY create_time DESC;
-----
Sort (cost=8.47..8.48 rows=3 width=540) (actual time=0.119..0.134 rows=4 loops=1)
  Sort Key: create_time
  Sort Method: quicksort Memory: 21kB
  -> Index Scan using dp_comment_article_id_fk_i on dp_comment (cost=0.00..8.45 rows=3 width=540)
      (actual time=0.043..0.068 rows=4 loops=1)
      Index Cond: (article_id = 1441)
Total runtime: 0.303 ms

```

### 12.4.2 Dosažené výsledky

Při optimalizaci dotazů jsem důkladně prozkoumával exekuční plány jednotlivých dlouhotrvajících SQL dotazů. Na spoustě místech byly řešitelné pomocí indexace. Abych tyto kroky potvrdil i pro prostředí vysoké zátěže, rozhodl jsem se nasimulovat prostředí vysoké zátěže opět pomocí programu siege, který vytvořil deset konkurenčních uživatelů s intervalem požadavku do jedné sekundy a ve třech repetících. Test proběhl samozřejmě před i po kroku optimalizace.



Obr. 9: Graf s výsledky doby trvání jednotlivých požadavků pro repetice

| Function Name  | Calls | Calls% | Incl. Wall Time (microsec) | IWall% |
|--|-------|--------|----------------------------|--------|
| <b>Current Function</b>  |       |        |                            |        |
| <a href="#">call_user_func_array@1</a>   | 209   | 49.4%  | 35,558,194                 | 97.1%  |
| Exclusive Metrics for Current Function   |       |        | 1,839                      | 0.0%   |
| <b>Parent functions</b>  |       |        |                            |        |
| <a href="#">Symfony\Component\DependencyInjection\ContainerBuilder::createService</a>        | 5     | 2.4%   | 35,555,786                 | 100.0% |
| <a href="#">Twig_Template::getAttribute</a>  | 204   | 97.6%  | 2,408                      | 0.0%   |
| <b>Child functions</b>   |       |        |                            |        |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount</a>   | 1     | 0.5%   | 15,898,330                 | 44.7%  |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew</a>     | 1     | 0.5%   | 14,046,220                 | 39.5%  |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount</a>   | 1     | 0.5%   | 5,397,372                  | 15.2%  |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount</a>      | 1     | 0.5%   | 186,680                    | 0.5%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories</a> | 1     | 0.5%   | 27,077                     | 0.1%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle</a>         | 31    | 14.8%  | 96                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory</a>        | 30    | 14.4%  | 94                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember</a>          | 30    | 14.4%  | 91                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl</a>        | 15    | 7.2%   | 81                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl</a>           | 23    | 11.0%  | 72                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent</a>         | 15    | 7.2%   | 52                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl</a>             | 15    | 7.2%   | 50                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle</a>           | 15    | 7.2%   | 50                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName</a>         | 15    | 7.2%   | 46                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName</a>          | 15    | 7.2%   | 44                         | 0.0%   |

Obr. 10: XHProf profilování aplikačního modelu databázových dotazů před optimalizací

| Function Name  | Calls | Calls% | Incl. Wall Time (microsec) | IWall% |
|--|-------|--------|----------------------------|--------|
| <b>Current Function</b>  |       |        |                            |        |
| <a href="#">call_user_func_array@1</a>   | 209   | 49.4%  | 24,632,393                 | 84.0%  |
| Exclusive Metrics for Current Function   |       |        | 1,985                      | 0.0%   |
| <b>Parent functions</b>  |       |        |                            |        |
| <a href="#">Symfony\Component\DependencyInjection\ContainerBuilder::createService</a>        | 5     | 2.4%   | 24,629,750                 | 100.0% |
| <a href="#">Twig_Template::getAttribute</a>  | 204   | 97.6%  | 2,643                      | 0.0%   |
| <b>Child functions</b>   |       |        |                            |        |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount</a>   | 1     | 0.5%   | 18,444,038                 | 74.9%  |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount</a>   | 1     | 0.5%   | 5,708,528                  | 23.2%  |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew</a>     | 1     | 0.5%   | 262,783                    | 1.1%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount</a>      | 1     | 0.5%   | 194,205                    | 0.8%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories</a> | 1     | 0.5%   | 20,080                     | 0.1%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle</a>           | 15    | 7.2%   | 116                        | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember</a>          | 30    | 14.4%  | 113                        | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory</a>        | 30    | 14.4%  | 101                        | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle</a>         | 31    | 14.8%  | 92                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl</a>        | 15    | 7.2%   | 85                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl</a>           | 23    | 11.0%  | 74                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl</a>             | 15    | 7.2%   | 56                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent</a>         | 15    | 7.2%   | 48                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName</a>         | 15    | 7.2%   | 47                         | 0.0%   |
| <a href="#">Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName</a>          | 15    | 7.2%   | 42                         | 0.0%   |

Obr. 11: XHProf profilování aplikačního modelu databázových dotazů po optimalizaci

Z výsledků experimentu vyplývá, že SQL dotazy, které trvaly několik řádů sekund se zoptimalizovali až pod dobu jedné sekundy. Z grafu v prostředí vysoké zátěže je vidět, že před optimalizací byla maximální doba jednoho požadavku až 420 sekund. Celý proces byl hodně nestabilní a kolísavý. Po optimalizaci je viditelné ustálení činnosti databázového systému a zmenšení doby odezvy. Zrychlení je vidět až o více než padesát procent. Dalšími budoucími řešeními, která se by se dala aplikovat mohou být segmentace či replikace. Samotná optimalizace SQL dotazů řeší určité problémy, ovšem databáze může přerůst až do takových rozměrů, kdy je potřeba sáhnout k těmto řešením.

Databázová vrstva bývá velice často největším zdrojem zpomalení. Dotazy na tuto vrstvu jsou nejnáročnější ze všech procesů skrze celou webovou architekturu. Proto je velice důležité tento proces nepodceňovat a věnovat tomu patřičné prostředky.

## 12.5 Aplikační cache

Tady se budeme snažit optimalizovat databázy.

### 12.5.1 Optimalizace aplikace pomocí Memcached

Co jsem udělal s Memcached, jednotlivé vrstvy modelu, popis toho čeho chci dosáhnout.

### 12.5.2 Diagram tříd pro aplikaci s podporou Memcached

Diagram tříd mé aplikace

### 12.5.3 Dosažené výsledky

Toto bude vždy na konci každého experimentu, grafy, časy, screeny, apod.

## 12.6 Reverzní proxy cache

Tady se budeme snažit použít reverzní proxy cache.

### 12.6.1 Nasazení a konfigurace NGINX s Memcached

Jak jsem co dělal s NGINX, problémy, řešení, návrh architektury a jak to ovlivňuje aplikační vrstvu

### 12.6.2 Význam Ajax a webových služeb pro NGINX

Ajax komunikuje s NGINX, proč a jak.

### 12.6.3 Dosažené výsledky

Toto bude vždy na konci každého experimentu, grafy, časy, screeny, apod.

## 13 Diskuze

Diskutovaná řešení, jak je možné je kombinovat, apod.

## 14 Závěr

Závěr ve smyslu nákladů a přínosů, kdy je lepší co. Uvidíme, napíšu jako poslední.



## 15 Literatura

- KYLE RANKIN *Linux Journal: Hack and / - Linux Troubleshooting, Part I: High Load* [online]. Dostupné z: <http://www.linuxjournal.com/magazine/hack-and-linux-troubleshooting-part-i-high-load>.
- FAISAL KHAN *DOS ATTACKS: Dos Attacks Overview - What are DoS attacks* [online]. Dostupné z: <http://dos-attacks.com/what-are-dos-attacks/>.
- PAVEL ČEPSKÝ *Lupa.cz: Útoky jménem Anonymous: Jak se rodí hackeri?* [online]. Dostupné z: <http://www.lupa.cz/clanky/utoky-jmenem-anonymous-jak-se-rodí-hackeri/>.
- DOC.ING.JAROSLAV ZENDULKA,CSC. *VUT-FIT: 10. Architektura klient/server a třívrstvá architektura* [online]. Dostupné z: [http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/10\\_clsrv.pdf](http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/10_clsrv.pdf).
- BRETT McLAUGHLIN *Ibm developer works: Mastering Ajax* [online]. Dostupné z: <http://www.ibm.com/developerworks/web/library/wa-ajaxintro1/index.html>.
- RUDOLF PECINOVSKÝ *Návrhové vzory : 33 vzorových postupů pro objektové programování* 1. vyd. Brno: Computer Press, 2007. 527 s. ISBN 978-80-251-1582-4.
- BOHDAN BLAHA *SQL Optimalizace v Oracle* Praha: Unicorn College, 2010. 47 s. Dostupné z: [http://www.unicorncollege.cz/katalog-bakalarskych-praci/bohdan-blaha/attachments/Blaha\\_Bohdan\\_-\\_Optimalizace\\_SQL\\_dotaz%C5%AF\\_v\\_datab%C3%A1zi-Oracle.pdf](http://www.unicorncollege.cz/katalog-bakalarskych-praci/bohdan-blaha/attachments/Blaha_Bohdan_-_Optimalizace_SQL_dotaz%C5%AF_v_datab%C3%A1zi-Oracle.pdf).
- ELI WHITE *Habits of Highly Scalable Web Applications* DCPHP Conference 2009.
- TOMÁŠ VONDRA *Replikace v PostgreSQL* CSPUG Konference 2011.
- R. FIELDING, UC IRVINE, J. GETTYS, J. MOGUL, COMPAQ, H. FRYSTYK, L. MASINTER, XEROX, P. LEACH, MICROSOFT, T. BERNERS-LEE, W3C/MIT *RFC:2616 - Hypertext Transfer Protocol – HTTP/1.1*.
- MARK NOTTINGHAM *CACHING TUTORIAL for Web Authors and Webmasters* [online]. Dostupné z: [http://www.mnot.net/cache\\_docs/](http://www.mnot.net/cache_docs/).
- JAKUB ONDERKA *Zdrojak.cz: Profilování PHP skriptů pomocí XHProf* [online]. Dostupné z: <http://www.zdrojak.cz/clanky/profilovani-php-skriptu-pomoci-xhprof/>.